VDMJ Library Writer's Guide

Author	Nick Battle
Date	29/04/25
Issue	0.1

0 Document Control

0.1 Table of Contents

0.2 References

[1] Wikipedia entry for The Vienna Development Method, http://en.wikipedia.org/wiki/Vienna_Development_Method

[2] Wikipedia entry for Specification Languages, http://en.wikipedia.org/wiki/Specification_language

[3] VDMJ, https://github.com/nickbattle/vdmj

0.3 Document History

Issue 0.1	29/04/25	First draft.

0.4 Copyright

1 Overview

This document describes how to write *libraries* for VDMJ.

Section 1 gives an overview of the architecture into which libraries fit. Section 2 gives detailed information about how to implement libraries. Section 3 gives some examples of what would be possible with libraries and how to achieve it.

1.1 VDMJ

VDMJ provides basic tool support for the VDM-SL, VDM++ and VDM-RT specification languages, written in Java [1][2][3]. It includes a parser, a type checker, an interpreter and debugger with coverage recording, a proof obligation generator, user definable annotations and a combinatorial test generator, as well as *JUnit* support for automatic testing.

1.2 VDMJ Libraries

The core language support in VDMJ is capable of parsing, checking and evaluating specifications that contain any of the standard language features. But there are common utilities that many specifications need. These are conveniently provided via *libraries*, to avoid the need for each specification to repeat them.

Libraries may define functions or operations that perform complex processing written purely in VDM, or they may provide a link from VDMJ to a native Java environment, where external Java libraries can be used to provide useful functionality.

VDMJ is distributed with a *standard library* that provides IO, MATH, CSV and VDMUtil utilities.

1.3 Library Linking

A VDMJ library is a separate Java archive, included on the classpath. The top level of the archive includes VDM source files (possibly in multiple dialects) that define the VDM interface to the library.

If the library is written in pure VDM, the top level source files are enough. But if the library also provides native Java support, the archive includes Java class files in the usual manner. To link from VDM to native code, VDMJ uses the *is not yet specified* expression/statement. At that point, the interpreter will try to find a Java class/method that matches the VDM parameters and result.

To use a library in VDMJ, you include the name of the VDM source files at the top level of the archive, as though they were your own source files. If the file is found in one of the jars, the source is extracted and written to a “lib” subfolder in the current directory.

So for example, the following will work in an empty folder:

```
> java -cp vdmj.jar:stdlib.jar VDMJ -vdmsl -e "sin(1)" MATH.vdmsl
Parsed 1 module in 0.094 secs. No syntax errors
Type checked 1 module in 0.252 secs. No type errors
Initialized 1 module in 0.217 secs.
0.8414709848078965
Bye
```

Note that the classpath includes VDMJ and the standard library jar. The *-e* option identifies an expression to evaluate and finally the *MATH.vdmsl* file is given, which is found in the stdlib.jar and extracted. The *sin* function uses Java native support to evaluate sines. Afterwards, the *lib/MATH.vdmsl* file will remain.

2 Writing Libraries

2.1 Libraries for VDMJ

As described in [1.3], a VDMJ library is a Java archive with VDM source files at the top level. When library jars are on the classpath, VDMJ searches them for file names that you give on the command line. Files are then extracted into your project as though you had written them yourself. Extracted files are written to a “lib” subfolder, which is created if it does not exist.

A single library jar can include multiple VDM files, either because the functionality provided is complex enough to warrant multiple files, or because the functionality is provided for multiple dialects (so *.vdmsl, *.vdmpp and *.vdmrt files). Note that you must name all of the source files that you need when using a library.

Extracted library sources remain in the “lib” subfolder after the execution is complete. On subsequent executions, you can either specify the same source files, which will harmlessly re-extract them, or you can include the “lib” folder by name, which will use all of the sources within there. This extract-and-retain design is to make VDMJ command line operate in a similar way to the VDM VSCode extension[1], covered below [2.2].

2.1.1 Source File Encoding and Translation

A library contains common functions or operations that are intended to be used by many specifications. But specifications may use a range of different character encodings, and all of the files within a VDMJ project must share a common encoding.

For this reason, source files extracted from a library are automatically translated into the character set currently defined for the project (i.e. the current *-c* setting). The encoding of files in the library itself must be UTF-8.

2.1.2 Linking to Java Native Code

Any function or operation in VDMJ can link to native Java code to implement it. This is conveniently done inside libraries, since they already provide Java jar files into which the native classes can be packaged. The linkage involves two conventions: one in the VDM source, and one in the Java class supporting the functionality.

In the VDM source, the link to native code is implemented via the *is not yet specified* expression or statement. This is a standard part of the VDM language and is normally used to indicate that a particular definition has not yet been written (though the calling interface is defined). If the interpreter encounters this, it would normally stop execution at that point. But before doing this, it checks to see whether there is a Java class and method defined that matches the VDM parameters and result. Strictly, the *is not yet specified* does not need to be the only thing in the body of the function or operation, but this is usually the case. If you put the linkage in a larger expression or block, it must not be within any local variable contexts (e.g. not within the scope of a “let”).

The Java class name to use is taken from the VDM-SL module name, or the VDM++/VDM-RT class name, with underscores replaced with “dot” package separators. So for example, a VDM module called “Matrix” would look for methods in a class called “Matrix” in the default Java package; a class name called “Support_Vectors” would look in a Java class called “Vectors” in the “Support” Java package. A flat VDM-SL specification (i.e. no modules) would look for a Java class called “DEFAULT” in the default Java package.

Within the Java class, the linkage looks for a Java method that matches the VDM function or operation name, which has the right number of parameters and a return value, all of which must be of

[1] See https://github.com/overturetool/vdm-vscode

type *com.fujitsu.vdmj.values.Value*, with an optional *com.fujitsu.vdmj.runtime.Context* parameter. Finally, methods must be public, static if the VDM definition is static, and include a @VDMFunction or @VDMOperation annotation, to indicate whether they support a function or an operation in VDM. These annotations can optionally include a “params” argument, to identify the subtype of Value for each positional parameter.

If a native implementation uses any third party libraries, these must be included in the classpath when VDMJ or VSCode is running, but they do not need to be included in the library jar.

To give a simple example:

```
module Matrix
...
types
    M = seq of seq of real;
...
functions
    invert: M +> M   -- implicitly static
    invert(m) ==
        is not yet specified;

end M
```

The *invert* function is in a module called “Matrix”, so native support for this would be in a class called “Matrix” in the default Java package, using a public static method called invert, like this:

```
// no package clause – use default package

import com.fujitsu.vdmj.values.SeqValue;
import com.fujitsu.vdmj.values.Value;
import com.fujitsu.vdmj.runtime.Context;

class Matrix
{
    @VDMFunction(params = {SeqValue.class})
    public static Value invert(Value m, Context ctxt)
    {
        ... calculate an inverse matrix Value
        return inv;
    }
}
```

If the Java class above is on the classpath, and the VDMJ interpreter encounters the “invert” method, it will delegate the call to the Java implementation which (presumably) calculates a *Value* representing the inverse of the matrix passed in. The result will then be returned to the specification, which will continue as normal.

If the example required the use of a third party library to do matrix manipulation, this would be added to the classpath at runtime, but it does not affect the design of the native method, apart from the Java import statements that it requires.

The *Context* parameter enables the native code to throw meaningful *ContextExceptions*, which will be caught by the debugger to allow sensible stack traces to be displayed. But this is optional, and older native methods do not use this parameter. If you provide both methods, the one with the *Context* parameter is selected. This should not be included in the “params” list, if any.

A *ValueFactory* class is available in VDMJ which can help with the construction of *Values* from raw Java data, though this is sometimes difficult. Support here should improve in future versions.

2.1.3 Delegate instances for VDM++

Delegation to a native method in VDM-SL is comparatively simple, in that a single Java object instance (called a *delegate*) is created for the module concerned. All subsequent calls to native methods in that module will use the same delegate instance. This means that the object can include state data (Java class fields) which is retained between calls. In some cases this is useful, especially for native operation calls. But in the case of native *function* calls, this should be used with extreme care since it breaks the referential transparency contract of VDM functions (they might return different results for the same arguments passed).

Delegation for VDM++ and VDM-RT creates a delegate instance for each VDM object instance, unless the VDM method is “static” or a function, in which case the delegate is associated with the VDM class (i.e. similar to VDM-SL delegation). The native method supporting a static function or operation should itself be static in Java.

2.1.4 Multiple Libraries per jar

The description above says that a library is a Java archive with sources and classes to support its functionality. It makes sense to keep libraries with unrelated functionality in separate jars, but it is also sometimes convenient to put related functionality in a single jar to simplify packaging. This is how the VDMJ standard library is packaged: the IO, MATH, CSV and VDMUtil files are all within stdlib.jar.

2.1.5 Multi-threading and Libraries

The VDM++ and VDM-RT specifications can create multiple threads, and these can execute library calls that may be native methods. Library functions or operations that have concurrent permission guards (i.e. a *sync* section in the class definition) will behave as expected, so the body of a library function (say) will not be entered until its guard has been satisfied, and this is true even if it is a native function.

The Java method of a native function or operation will execute on the same Java thread that the interpreter uses for the VDM thread. But because of the strict scheduling policy of the VDM interpreter, Java methods will never actually execute concurrently in real time. So they may use *synchronize* primitives if you wish, but only one VDM thread is ever executing at once.

2.1.6 Debugging VDMJ Native Libraries

You cannot use VDMJ’s inbuilt debugger to step into native Java methods, though you can “step over” the *is not yet specified* linkage and your native method will be called at that point.

The easiest way to debug the native calls themselves is to use a Java IDE that is able to execute the VDMJ interpreter as a Java program. You can then use the IDE to add breakpoints in your native code, just like any other Java program.

2.2 Libraries for VDM VSCode Extension

Libraries for the VDM VSCode extension are structured in the same way as for VDMJ, but with an additional *library.json* file in the META-INF section of the jar. This extra meta-data allows the extension to display a friendly selection dialog for libraries to be added to a project. When libraries are selected from this dialog, they are extracted and copied into a “lib” project subfolder, as with VDMJ. The library jars are also added to the classpath automatically.

The *library.json* file is structured as follows (an extract from the stdlib file):

```
{
        "vdmsl":
        [
```

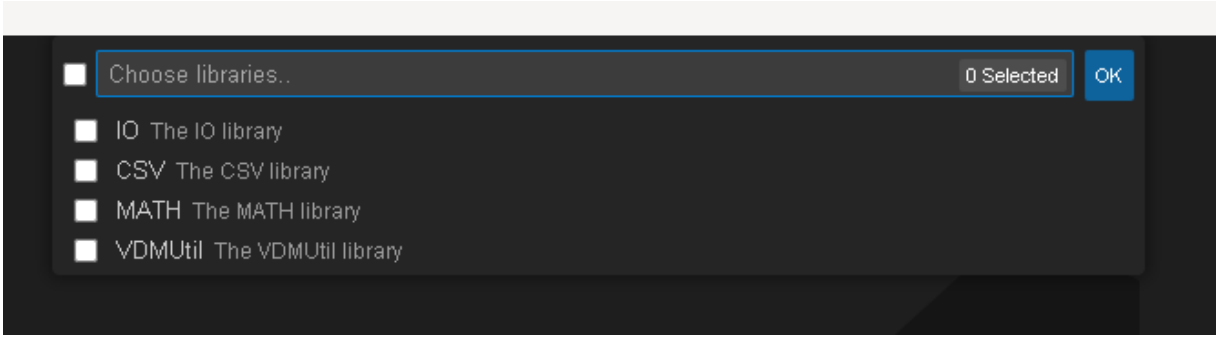


```
                {
                        "name": "IO",
                        "description": "The IO library",
                        "files": ["IO.vdmsl"],
                        "depends": []
                },
                {
                        "name": "CSV",
                        "description": "The CSV library",
                        "files": ["CSV.vdmsl"],
                        "depends": []
                },
                ...
        ],
        "vdmpp":
        [
                {
                        "name": "VDMUnit",
                        "description": "The VDMUnit library",
                        "files": ["VDMUnit.vdmpp"],
                        "depends": ["IO"]
                },
                ...
        ],
        "vdmrt":
        [
                ...
        ]
}
```

The file defines a single JSON object, which includes up to three language fields, “vdmsl” etc. Each entry contains an array of objects that define the name, description, VDM source files and dependencies of a single library. Note that there can be multiple source files. Note also that there may be multiple dependencies, which is a list of names of libraries that must also be included.

The *library.json* file is ignored by VDMJ, so the same library jars can be used by both tools.

The VDM VSCode “Add VDM Library” interface looks like this, by default. This list would include your own libraries if they are added to the configuration:



3 Example Library Functionality

3.1 High Performance Algorithms

A key advantage of linking to native code is to make use of very complex functionality within a specification without the need to re-implement that functionality in the VDM interpreter. The native execution may be many times faster than an equivalent implementation in VDM.

In this use case, VDM interfaces are declared for the external operations required, and a thin native Java layer converts VDMJ's *Value* classes to and from those used by the third party library.

3.2 External Database Access

Native code can easily link to JDBC enabled databases, allowing either specific SQL queries to be made and the results converted to VDM *Values*, or enabling a general SQL interface in VDM.

As with high performance libraries, this would be implemented via a thin native Java layer, converting arguments and results from and to VDMJ *Values*.

3.3 Physical Device Access

Some models benefit from direct access to physical devices that enable "hardware in the loop" simulations. An abstraction of the device interface is required in VDM, with key functions or operations being implemented as native methods.

Once again, this would be implemented via a thin native Java layer, converting arguments and results from the device from and to VDMJ *Values*.

A. VDMJ Standard Library

The VDMJ Standard Library is comprised of four separate groups:

Name	Description
IO	The IO library provides functions to read and write VDM values from files or from a stdio console.
MATH	The MATH library provides basic mathematical functions for trigonometry, logarithms, random number generation etc.
CSV	The CSV library allows basic data to be read from a CSV file.
VDMUtils	The VDMUtils library includes functions to do useful processing on VDM data types, such as turning a set into a sequence, or generating string values for VDM types.