QuickCheck Design Specification	
Author	Nick Battle
Date	16/12/24
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

[4] Overture, https://github.com/overturetool/overture

[5] VDMJ Plugin Writer's Guide

[6] LSP Plugin Writer's Guide

0.3. Document History

Issue 0.1	16/12/24	First draft.

0.4. Copyright

1. Overview

This design describes the QuickCheck plugins for the VDMJ tool suite.

Section 1 gives an overview of the Java package structure. Section 2 gives detailed information about each package. Section 3 walks through various common scenarios to describe the operation of the internals.

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 (with arbitrary precision arithmetic), a debugger, a proof obligation generator and a combinatorial test generator with coverage recording, as well as *JUnit* support for automatic testing and user definable annotations.

1.2. VDMJ and LSP Plugins

VDMJ and the VDMJ LSP Server are designed as a collection of *plugins*, each offering new analyses and services to the tool.

Some plugins are built-in and offer parsing, type checking, execution and proof obligation generation. But extra plugins can be added by including them on the Java classpath. See [5] and [6].

QuickCheck offers both VDMJ and LSP plugins. These provides two commands in each environment, “qc” and “qr”, which analyse proof obligations.

1.3. Proof Obligations

In VDM, a *proof obligation* is a short boolean expressions which will always be true if the specification is free from some particular vulnerability. For example, a function which performs an arithmetic division will fail if the denominator is zero. In the case, the obligation would state that the denominator can never be zero in that particular call context.

Proof obligations have two parts: a context stack which describes how an evaluation may reach a particular vulnerability, and the obligation that must hold at that point. For a vulnerability within a function, the outermost part of the context stack effectively says “for all possible argument values to this function”. Subsequent context items describe the choices that the function has to make to reach the vulnerability, such as if/then/else choices, cases clauses and so on. And then lastly, the obligation expression itself is added for form the complete obligation expression.

For example:

```
(forall tr:Trace, aperi:nat1, vdel:nat1, mk_Pacemaker(aperiod, vdelay):Pacemaker &
  (forall i in set (inds (tl tr)) &
    (((i mod aperi) = (vdel + 1)) =>
      i in set inds tr)))
```

Here, the first line is the outermost context, and shows the possible arguments to this operation, as well as the Pacemaker state vector values. The second line is caused by a sequence comprehension that is selecting values to insert. The third line is a check that occurs within the comprehension element evaluation, and lastly, the index value is tested to show that it is within the indices of the “tr” sequence.

You can see that it is possible to falsify this obligation by setting the following argument values:

```
Counterexample: tr = [], vdel = 1, aperi = 1, vdelay = 0, aperiod = 0
Causes Error 4033: Tail sequence is empty at line 2:21
```

This indicates that it is possible to pass arguments to the operation that contains this obligation that will cause the operation to fail. The solution is to add preconditions or other checks in the specification, or change the type of “tr” to make sure that the value passed cannot be empty. With these extra tests in place, the proof obligation generator (POG) would either not generate this obligation, or the same counterexample arguments would no longer produce a false/error result.

QuickCheck is a tool for analysing proof obligations and labelling them as one of three categories:

- FAILED, with counterexample arguments (as above)
- Probably PROVABLE, with reasons why it is believed to be true in all cases
- MAYBE correct – which means neither of the cases above.

1.4. QuickCheck Package Overview

The implementation is divided into the following Java packages.

Packages	
plugin	Classes that provide the VDMJ and LSP plugin interface
commands	The “qc” and “qr” commands to perform PO analysis
strategies	Classes that implement the built-in strategies
visitors	Visitors used by the built-in strategies
annotations	The @QuickCheck annotation
quickcheck	The main QuickCheck class that coordinates the analysis.

The *plugin* package contains classes that implement the VDMJ and LSP AnalysisPlugin interface. These are thin wrappers around the common QuickCheck class which is the common implementation for both plugins.

The *commands* package contains classes that implement the VDMJ and LSP AnalysisCommand interface. This allows the “qc” and “qr” commands to be made available to users on the command line.

There are six built-in *strategies* for searching for argument bindings, in order to categorize the PO as FAILED, PROVABLE or MAYBE. It is possible for users to add more, as discussed below.

The built-in strategies use common *visitors* for processing obligations.

There is a @QuickCheck annotation that can be used for functions with polymorphic type parameters. This is defined in the *annotations* package.

And lastly the main QuickCheck class is in the *quickcheck* package. This performs the analysis and is common to the VDMJ and LSP plugins.

2. Package Detail

This section gives more detail about the Java classes in the packages within QuickCheck.plugin

2.1. The commands package

Class Summary	
QuickCheckCommand	The VDMJ “qc” command
QuickCheckLSPCommand	The LSP “qc” command
QCRunCommand	The VDMJ “qr” command
QCRunLSPCommand	The LSP “qr” command
QuickCheckExecutor	An LSP AsyncExecutor for QuickCheck runs
QCConsole	A QuickCheck console that allows “quiet” processing

The commands package contains classes that implement the “qc” and “qr” command for both VDMJ and LSP environments.

The QuickCheckCommand class extends the abstract VDMJ AnalysisCommand, and implements the “run” method. It starts by creating a QuickCheck instance to actually perform the analysis, and asking it to load its pluggable strategies. The strategies are passed the command line and are responsible for processing and removing any flags that they require.

The run method is passed the command line that the user entered, and it processes the various common flags that the user can pass.

After that, the command obtains the POPlugin from the Registry and calls its getProofObligations method, which returns a list of unchecked obligations. This list is cut down by any options that the user has passed which would select a subset of the obligations.

Next the strategies are initialized. At this point, a strategy may request that the run is aborted. This can be used by strategies that have options which do not want to analyse obligations but rather just prepare for them. For example, the “fixed” strategy has a “-fixed:create” option which creates a configuration file that can be edited and used in a subsequent run.

If no strategies abort the run, the command loops through the chosen proof obligations.

Firstly, it calls the getValues method on the QuickCheck instance. This is responsible for calling the getValues method of each of the configured plugins, and these in turn return lists of values for each type binding in the obligation. See 2.2.

If a list of binding values has been obtained, the QuickCheck instance’s checkObligation method is called, passing the PO and the results from the getValues call. This method attempts to evaluate the obligation expression using the bindings passed. See 2.5. A ConsoleExecTimer thread is also started, which will interrupt the evaluation if it takes too long.

The console output is produced by the checkObligation method. So when the command has looped through each obligation selected, it terminates with no extra output. For example:

```
> qc
PO #1, MAYBE in 0.004s
PO #2, PROVABLE by direct (patterns match all type values) in 0.002s
PO #3, PROVABLE by witness map_ = <ManyMany>, prset = {} in 0.001s
PO #4, PROVABLE by direct (body is total) in 0.002s
PO #5, PROVABLE by witness rels = {|->}, esets = {|->} in 0.001s
>
```

The processing performed by the QuickCheckLSPCommand is very similar, except it uses an AsyncExecutor called QuickCheckExecutor to actually run the evaluation. This is because VSCode is a more asynchronous environment, and the user is allowed to do other operations while the background QuickCheck evaluation is in progress. But apart from that difference, the execution in the QuickCheckExecutor's "exec" method is very similar to the VDMJ command.

The output on the VSCode debug console (below) is virtually identical in content to the VDMJ command line because the output is coming from the same QuickCheck processor, but an "OK" is added to let the user know that the background evaluation has completed. If you attempt to do something while "qc" is still running, you will be told "*Still running qc*".

```
qc
PO #1, MAYBE in 0.005s
PO #2, PROVABLE by direct (patterns match all type values) in 0.003s
PO #3, PROVABLE by witness map_ = <ManyMany>, prset = {} in 0.001s
PO #4, PROVABLE by direct (body is total) in 0.001s
PO #5, PROVABLE by witness rels = {|->}, esets = {|->} in 0.001s
OK
```

The QCConsole class extends the VDMJ standard PluginConsole class. This is because the "qc" command has its own verbose/quiet flags, which override the VDMJ settings.

2.1.1. Comments

2.2. The strategies package

2.2.1. Comments

2.3. The visitors package

2.3.1. Comments

2.4. The annotations package

2.4.1. Comments

2.5. The quickcheck package

2.5.1. Comments

3. Scenarios

This section describes the sequence of actions that occur during common tasks with QuickCheck. The intention is to provide a more tangible description of how the classes described in section 2 work together in practice.

3.1. Server Creation and Lifecycle

3.1.1. Comments