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 in 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 analyses and services to the tool.

Essential 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

A *proof obligation (PO)* 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 this case, an 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 an obligation that must hold at that point. For a vulnerability within an operation, the outermost part of the context stack effectively says “for all possible argument values and state vectors when calling this operation”. Subsequent context items describe the choices that the operation 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 possible 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 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, looking for problems such as the example given above. The tool labels POs 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 has 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 VDMJ command line and VSCode console.

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 strategies, as discussed below.

The built-in strategies use classes in the *visitors* package 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 PO 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.

2.1. The plugin package

Class Summary	
QuickCheckPlugin	The VDMJ plugin
QuickCheckLSPPlugin	The LSP plugin
QuickCheckHandler	An LSP handler for slsp/POG/quickcheck RPC calls
QuickCheckThread	An LSP AsyncThread to run slsp/POG/quickcheck calls

These classes implement the AnalysisPlugin interface needed to load a plugin into VDMJ or the LSP Server. They do not contain any PO checking functionality as such, which is all done in “QuickCheck”.

As well as giving the plugin a unique name, “QC”, they also implement the getCommand and getCommandHelp methods, which allow the plugins to return new instances of the “qc” and “qr” commands, and provide help information for the user about usage.

```
@Override
public AnalysisCommand getCommand(String line)
{
        String[] argv = Utils.toArgv(line);

        switch (argv[0])
        {
                case "quickcheck":
                case "qc":
                        return new QuickCheckLSPCommand(line);

                case "qcrun":
                case "qr":
                        return new QCRunLSPCommand(line);
        }

        return null;
}
```

In an LSP environment, rather than responding to commands on the terminal, JSON RPC messages are sent via an extension of the LSP protocol. The QuickCheckHandler is registered to handle “slsp/POG/quickcheck” messages, directing them to the QuickCheckLSPPlugin’s “quickCheck” method.

Because quickCheck evaluations can take a while for large specifications, the VSCode evaluation is performed by a background CancellableThread called QuickCheckThread. This is similar to the QuickCheckExecutor (below) which is used to background quickcheck commands issued from the VSCode console. This means that long running evaluations can be interrupted via the GUI.

2.1.1. Comments

It is confusing that we have both VDMJ and LSP plugins, and we also have the ability to run QuickCheck from the VSCode console or from the GUI via a dedicated LSP message. Perhaps this could be simplified, but only by losing some feature or other.

2.2. 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” commands 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 remaining 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 proof 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 2.3.

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 2.6. A ConsoleExecTimer thread is also started, which will interrupt the PO 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 LSP 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 at the end 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.

2.2.1. Comments

The pattern of dividing functionality between a "plugin" part and a "common" part works well and allows the same code to be shared between VDMJ and LSP plugins. This is recommended for other plugins that want to offer services in both environments.

2.3. The strategies package

Class Summary	
QCStrategy	An abstract base class of all strategies
FixedQCStrategy	A strategy that returns fixed values of every type
RandomQCStrategy	A strategy that returns random values of every type
FiniteQCStrategy	A strategy that looks for finite types and returns all their values
TrivialQCStrategy	A strategy that looks for common PO patterns that are true
SearchQCStrategy	A strategy that looks for falsifiable subexpressions
DirectQCStrategy	A strategy that looks at the original spec to try to prove POs
StrategyResults	An object to pass back binding/value pairs from a strategy

As discussed in the Overview, QuickCheck uses a number of pluggable strategies for deciding what binding values to try when trying to analyse obligations. Six strategies are provided with the tool, but new ones can be added easily, by extending the QCStrategy class and adding a jar to the classpath. The jar should also contain a resource file called "qc.strategies" which lists the fully qualified class name of the classes within the jar which are strategies. See the QuickCheck jar for examples.

Consider the following recursive factorial function:

```
f: nat -> nat
f(a) ==
      if a = 0
      then 1
      else a * f(a-1)
measure a;
```

It generates the following proof obligation for the recursive argument:

```
Proof Obligation 2: (Unproved)
f: subtype obligation in 'DEFAULT' (test.vdm) at line 6:21
(forall a:nat &
  (not (a = 0) =>
    (a - 1) >= 0))
```

The recursive f(a-1) call must pass a **nat**, and hence a-1 must be >= 0. The “not (a=0)” context is because the f(a-1) call is in the “else” clause, and hence the “if” condition must be false.

So to evaluate this PO, we have to produce a number of a:nat values to try, looking for cases that are counterexamples. Generating these values and associating them with the a:nat type bind is the job of the strategies.

Each strategy extends an abstract QCStrategy class, and must implement some simple methods, the most important being “init” and “getValues”. The init method is called at the start (see 2.2) and allows the strategy to take options from the “qc” command line. The bulk of the work is done by the getValues method:

```
public StrategyResults getValues(ProofObligation po,
                  List<INBindingOverride> binds, Context ctxt);
```

The method is passed the PO to process, a list of its type binds, and an evaluation Context. The INBindingOverride interface is provided by VDMJ and allows you to provide a specific list of bind values to either a *forall* or an *exists* quantifier. The Context is provided to allow type invariants to be calculated as values are generated.

For example, the “fixed” strategy does the following in its getValue method:

```
Map<String, ValueList> values = new HashMap<String, ValueList>();

for (INBindingOverride bind: binds)
{
      String key = bind.toString();
...
      verbose("Generating fixed values for %s\n", bind);
      ValueSet set = bind.getType().apply(
            new FixedRangeCreator(ctxt), expansionLimit);
      ValueList list = new ValueList();
      list.addAll(set);
      values.put(key, list);
}

return new StrategyResults(values, false, ...)
```

So note that the StrategyResults includes a map from the string form of each type bind to a ValueList of candidate values. It uses the FixedRangeCreator visitor to do this (see 2.4). The *expansionLimit* is set from the plugin options passed to the init method.

In general, a strategy can use any information from the PO and its bindings to try to guess or calculate or prove which values to try. The StrategyResults also include a *hasAllValues* argument that indicates whether the ValueList passed back contains all values of the types (here it’s false).

- The “random” strategy uses a PRNG to generate simple type values. Complex types are then composed using combinations of values from their more primitive components.
- The “finite” strategy uses the INGetAllValuesVisitor to generate all of the values of its binds, assuming the types are finite and not too large.
- The “trivial” strategy looks for common patterns that indicate that a PO is true.
- The “search” strategy looks for falsifiable subexpressions within the PO. For example, if it sees “x <> 0” then it will return a binding of x=0 to try to provoke a failure.
- The “direct” strategy looks at the specification and the type of the PO, and tries to prove the same thing that the PO is trying to achieve, but by direct means. For example, total functions raise a PO that says that the result of the function must be defined for every argument value. So the direct strategy looks at the function to determine whether it has any partial operators within its body. If it does not, then it must be a total function and hence the PO is labelled as

PROVABLE.

2.3.1. Comments

Strategies are plugins with a plugin, and use the same GetResource method of VDMJ to achieve this – see the “loadStrategies” method of QuickCheck.

2.4. The visitors package

Class Summary	
***TypeBindFinder	Parts of a VisitorSet to locate type binds
***RangeCreator	Visitors to create “fixed” and “random” ranges of values
SearchQCVisitor	A visitor used by the “search” strategy
TotalExpressionVisitor	A visitor used by the “direct” strategy
TrivialQCVisitor	A visitor used by the “trivial” strategy

This package contains a few visitors used either by the strategies, or used in order to locate type binds within PO expressions.

2.4.1. Comments

Arguably these ought to reside closer to the strategies or classes that use them, rather than being in a separate package. If strategies were loaded as plugins, they would have to include their own visitors (if they were new) so that would make more sense.

2.5. The annotations package

Class Summary	
**QuickCheckAnnotation	The AST, TC and PO classes for @QuickCheck
IterableContext	A collection of evaluation Contexts, one for each @T binding

The @QuickCheck annotation is provided to allow a specifier to give a list of types to use for polymorphic type parameters when instantiating a function within QuickCheck.

The syntax for the annotation is one of two possibilities, for VDM-SL and other dialects:

```
-- @QuickCheck @T = <type> [,<type>*];
-- @QuickCheck @T = new C(<args>);
```

When a PO is created, any annotations on the definition that contains the obligation are reflected in the PO itself. These annotations are then used in the main QuickCheck evaluation, creating an IterableContext that includes each of the polymorphic bindings specified in the annoation(s) – several annotations can be given for the same @T parameter, if that is clearer.

The IterableContext is then used in the main QuickCheck processing to effectively repeat the evaluation with @T parameters bound to alternative types.

2.5.1. Comments

This is a fairly crude way to provide the testing of polymorphic functions. It works for now, but it probably should be improved somehow, especially for highly polymorphic specifications.

2.6. The quickcheck package

Class Summary	

2.6.1. 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. Simple Usage of “qc”

3.2. Simple Usage of “qr”

3.2.1. Comments