QuickCheck Design Specification

Author	Nick Battle
Date	29/08/25
Issue	0.1

0. Document Control

0.1. Table of Contents

0. Document Control....2
0.1. Table of Contents....2
0.2. References....2
0.3. Document History....3
0.4. Copyright....3
1. Overview....4
1.1. VDMJ....4
1.2. VDMJ and VDM-VSCode Plugins....4
1.3. Proof Obligations....4
1.4. Package Overview....5
2. Package Detail....7
2.1. quickcheck.plugin....7
2.2. quickcheck.commands....8
2.3. quickcheck.strategies....11
2.4. quickcheck.visitors....13
2.5. quickcheck.annotations....13
2.6. quickcheck....14
3. Using QuickCheck....16
3.1. Usage of “qc”....16
3.2. Usage of “qr”....17
4. Writing New Strategies....19
4.1. Overview....19
4.2. The Example Strategy Explained....19
4.3. The Random Strategy Explained....20
4.4. Returning a StrategyUpdater....21
4.5. The “qc.strategies” Resource....21

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 VDM-VSCode

[5] VDMJ Plugin Writer's Guide

[6] LSP Plugin Writer's Guide

0.3. Document History

Issue 0.1	16/08/25	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 VDM-VSCode Plugins

VDMJ and the VDM-VSCode backend are designed as a collection of *plugins*, each offering analyses and services to the tool. The VDM-VSCode backend is accessed via the Language Server Protocol (LSP) and so its plugins are referred to as LSP plugins.

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 [4], [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 falsification of the *obligation* indicates that it is possible to pass arguments to the *operation* that will cause it 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 falsify the obligation.

QuickCheck provides a “qc” command for analysing proof obligations, looking for problems such as the example given above. The tool divides POs into 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.

If an obligation fails with a counterexample, the “qr” command attempts to construct a call to the enclosing function or operation, passing arguments from the counterexample in an attempt to cause the failure that the PO predicts. This is to allow the specification to be debugged in the failure situation.

1.4. Package Overview

The implementation is divided into the following Java packages.

Packages	
quickcheck.plugin	Classes that provide the VDMJ and LSP plugin interface
quickcheck.commands	The “qc” and “qr” commands to perform PO analysis
quickcheck.strategies	Classes that implement the built-in strategies
quickcheck.visitors	Visitors used by the built-in strategies
quickcheck.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 VDM-VSCode console.

There are several 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 base *quickcheck* package. This performs the PO analysis and is common to the VDMJ and LSP plugins.

1.4.1. Key Classes

There are three ways to invoke the core QuickCheck functionality: from the VDMJ command line; from the VDM-VSCode debug console and via the VDM-VSCode GUI. The key classes for these are shown in the diagram below. Orange classes are for VDMJ, blue are for VDM-VSCode. The common QuickCheck class is shared by both environments:

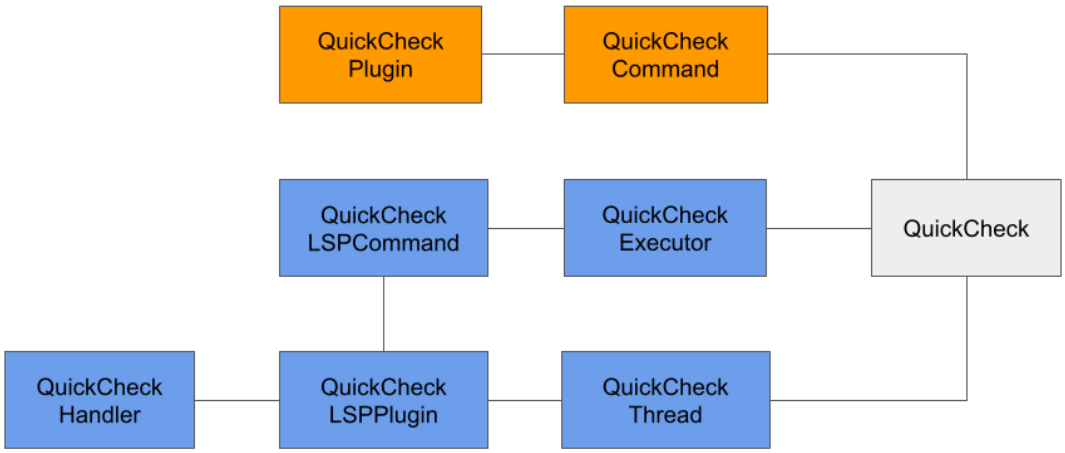

Figure 1: Key Classes in the QuickCheck Architecture

2. Package Detail

This section gives more detail about the Java classes in each package.

2.1. quickcheck.plugin

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 backend. They do not contain any PO checking functionality as such, which is all done in the QuickCheck class (see 2.6).

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” RPC messages, directing them to the QuickCheckLSPPlugin’s quickCheck method.

Because QuickCheck runs can take a while for large specifications, the LSP backend evaluation is performed by a background CancellableThread called QuickCheckThread. This is similar to the QuickCheckExecutor (below) which is used to background “qc” commands issued from the VDM-VSCode console. This means that long running evaluations can be interrupted via the GUI.

The LSP equivalent of command line options for “qc” are sent with the “slsp/POG/quickcheck” message in the “params” field. This is processed by the QuickCheckLSPPlugin. The values are taken from the .vscode/quickcheck.json file, which has the following structure:

```
{
	"config":
	{
		"timeout":		1000,
		"obligations":	[ 1, 2, 3... ]
```

```
        },
        "strategies":
        [
                {
                        "name":     "fixed",
                        "enabled":  true,
                        "size":     100,
                        ...
                },
                {
                        "name":     "finite",
                        "enabled":  false
                },
                ...
        ]
}
```

The “config” section has settings that apply to the overall execution. A fixed list of obligations can be configured, but this is usually set dynamically via the “Filter” option in the POG GUI. This is the equivalent of the pattern matching options in the command line (see 3.1).

The “strategies” section has an array of entries, one for every strategy that you want to apply. Each strategy entry has a “name” and “enabled” flag, plus arbitrary other flags and settings that are specific to the strategy (each equivalent to a `-<strategy>:<type>` option in the command line).

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 VDM-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. quickcheck.commands

Class Summary	
QuickCheckCommand	The VDMJ “qc” command
QCRunCommand	The VDMJ “qr” command
QuickCheckLSPCommand	The LSP “qc” command
QCRunLSPCommand	The LSP “qr” command
QuickCheckExecutor	An LSP AsyncExecutor for QuickCheck background 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.

2.2.1. “quickcheck” (abbr. “qc”)

The QuickCheckCommand class implements the “quickcheck” or “qc” command for the VDMJ plugin. So it extends the abstract VDMJ AnalysisCommand and implements the “run” method.

It starts by creating a QuickCheck instance (see 2.6) to actually perform the analysis, asking it to load its pluggable strategies. The loaded strategies are passed the command line and are responsible for processing and removing any flags that they require. The run method then 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 complete list of proof obligations. This list is cut down by any options that the user has passed to select a subset of the obligations.

Next the QuickCheck instance's 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, doing the following:

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 can 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 checkObligation evaluation if it takes too long.

The console output is written directly by the checkObligation method. So the command adds no extra output. For example:

```
> 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 in 0.002s
Witness: prset = {}, map_ = <ManyMany>
PO #4, PROVABLE by direct (body is total) in 0.001s
PO #5, PROVABLE by witness in 0.001s
Witness: rels = {|->}, esets = {|->}, ents = {|->}
>
```

The processing performed by the QuickCheckLSPCommand is very similar, except it uses an LSP AsyncExecutor called QuickCheckExecutor to run the checkObligation evaluation. This is because VDM-VSCode is an 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 "printQuickCheckResults" method of the QuickCheck instance is used to actually print the results. The output on the VDM-VSCode debug console (below) is virtually identical in content to the VDMJ command line because the output is coming from the same QuickCheck method, but an "OK" is added at the end to let the user know that the background evaluation has completed. If you attempt to run something on the console 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 in 0.002s
Witness: prset = {}, map_ = <ManyMany>
PO #4, PROVABLE by direct (body is total) in 0.001s
PO #5, PROVABLE by witness in 0.001s
Witness: rels = {|->}, esets = {|->}, ents = {|->}
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.

2.2.2. "qcrun" (abbr. "qr")

The QCRunCommand class implements the "qcrun" or "qr" command for the VDMJ plugin. So it extends the abstract VDMJ AnalysisCommand and implements the "run" method.

The objective of the “qr” command is, given a PO number which has a counterexample, to create the equivalent of a “print” command to evaluate the enclosing function or operation, passing arguments and state values from the counterexample. This is done in the following steps:

Firstly, the command checks that the PO number passed has a counterexample.

Then, a “launch” string is constructed by calling the getCexLaunch or getWitnessLaunch methods on the ProofObligation. These in turn use POLaunchFactory to construct a call string. The result is what you would type in a “print” command to launch the enclosing definition.

Then, if the counterexample has state (ie. it is for an operation call in a module with state), the current module state is adjusted by calling the “set” method on the UpdatableValues corresponding to the state vector. Note this change is not undone, so running “qr” can affect the current module state.

Lastly, a PrintCommand is built, including the launch string. This is executed as if the user had typed it themselves, allowing it to be debugged as usual. The actual print command is printed for information. For example:

```
> qc 2
PO #2, FAILED in 0.001s
Counterexample: i = 1, s = [1.25]
----
f: subtype obligation in 'DEFAULT' (test.vdm) at line 2:5
(forall i:nat, s:seq of real & pre_f(i, s) =>
  is_nat(s(i)))

> qr 2
=> print f(1, [1.25])
Error 4065: Value 1.25 is not a nat in 'DEFAULT' (console) at line 1:1
```

The QCRunLSPCommand is similar, except that an ExpressionExecutor is used to run the evaluation in the background, since the GUI is asynchronous.

The VDM-VSCode GUI can effectively run “qr” commands as well, via two routes. Firstly, the “Proof Obligation Generation” view has a “Debug example” button which provokes a JSON “launch” request, using data that is sent from the LSP backend to the GUI when the PO list is displayed. This creates all the counterexample call strings ahead of time, including a “params” section for state updates.

Secondly, obligations with counterexamples also create code lenses using POLaunchDebugLens. These appear above the obligation point, with a label like “PO #123” and the PO itself will have a warning message, giving the counterexample arguments. Clicking the code lens does the same as clicking the “Debug example”, leaving a new launch configuration in the project’s launch.json file:

```
{
	"name": "Lens config: Debug DEFAULT`op",
	"type": "vdm",
	"request": "launch",
	"noDebug": false,
	"defaultName": "DEFAULT",
	"params": {
		"type": "PO_LENS",
		"state": {
			"s1": "0",
			"s2": "0"
		}
	},
	"command": "p op(0)"
}
```

2.2.3. 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.

The PO code lens is a much better way of launching “qr” behaviour than the “Debug example” button. We may phase out the latter in future.

2.3. quickcheck.strategies

Class Summary	
QCStrategy	The 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
ReasonsQCStrategy	A strategy that checks whether variables are reasoned about
ConstantQCStrategy	A strategy that looks for constants in a PO to infer bindings
StrategyResults	Binding/value pairs and flags passed back from a strategy
StrategyUpdater	A class allowing a strategy to make arbitrary PO status updates

As discussed in the Overview, QuickCheck uses a number of pluggable strategies for deciding what binding values to try when trying to analyse obligations. Several 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 as an example.

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 (simplified):

```
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);
```

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.

A strategy may also choose to return a StrategyUpdater sub-object in the StrategyResults. If set, this is used to *directly* update the ProofObligation, rather than using binding values to try to find counterexamples. So this should only be done if the strategy designer is confident that they know the status of the obligation. See the “trivial” and “direct” strategies as an example of this.

A strategy may also implement a “maybeHeuristic” method, which is called when the strategies have failed to conclusively prove or disprove the PO, resulting in a MAYBE status. This is an opportunity to qualify the MAYBE status rather than changing it. See the “reasons” strategy below.

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

The built-in strategies are as follows:

- The “fixed” strategy generates fixed values for the basic types (eg. 100 ints would generate {-50, …, 49}). Complex types are then composed using combinations of values from their more primitive components.
- 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. This sets the *hasAllValues* flag.
- The “trivial” strategy looks for common patterns that indicate that a PO is true. Rather than returning possible bindings, this returns a StrategyUpdater to say that the PO is *PROVABLE*.
- The “search” strategy looks for falsifiable sub-expressions 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*, via a StrategyUpdater.

- The “reasons” strategy looks at the variables in the last line of the PO and checks whether these are reasoned about in the context leading up to the obligation. If not, there is a potential problem, since the PO places obligations on the variables. This is a *heuristic* strategy, since it does not know for certain that there is a problem. It directly updates the PO passed to the maybeHeuristic method.
- The “constant” strategy looks for constant sub-expressions in the PO using a ConstantExpressionFinder. Then it matches the types of these sub-expressions to the bindings of the PO, guessing that the values might be relevant. It then uses the ConstantNudger to produce slight variations of each constant, which might also be able to provoke a counterexample.

Note that a strategy can affect whether it is enabled by default or only enabled when explicitly requested. This is set via the useByDefault method. The “random” strategy is the only built-in strategy that isn’t enabled by default.

2.3.1. Comments

Strategies are plugins within a plugin, and use the same GetResource method of VDMJ to achieve this – see the “loadStrategies” method of QuickCheck. This pattern is very flexible.

2.4. quickcheck.visitors

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
ConstantExpressionFinder	A visitor used by the “constant” strategy
ConstantNudger	A Value visitor used by the “constant” strategy

This package contains a few visitors used either by the strategies, or used in order to locate type binds and constants within PO expressions. They are fairly straightforward.

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. quickcheck.annotations

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, assuming there is a polymorphic type parameter called @T:

```
-- @QuickCheck @T = <type> [,<type>*];
-- @QuickCheck @T = new <class>(<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 annotation(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.

Note that the standard @NoPOG annotation is useful when working with QuickCheck, since you may well find parts of your specification that you do not want to cover. For example, the standard libraries all include @NoPOG annotations at the top, so that they do not produce POs.

2.5.1. Comments

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

2.6. quickcheck

Class Summary	
QuickCheck	The main processing class for the tool.

QuickCheck is the only class in the top level quickcheck package, and it contains the common processing used by both the VDMJ and LSP plugins. A new instance of this class is created by every “qc” command execution (see 2.2.1).

The methods of QuickCheck are called by the command execution in the following order:

Before QuickCheck can do anything, it must load the strategy plugins used to create binding values (see 2.3). This is done by the loadStrategies method, which is passed an *argv* string list of command options. The arguments are passed on to the strategies as they are loaded, allowing them to use settings of their own. The core loading is performed by the GetResource class from VDMJ, which is given a resource name of “qc.strategies”. To enable a strategy to be loaded, its jar must define a resource file of that name which contains the fully qualified classname of the plugin(s) it defines. The resource file for the built-in strategies is in src/main/resources.

The QuickCheck class maintains an error count, which can be queried at any stage with hasErrors.

After the strategies have been loaded, they are initialized by calling initStrategies. This calls the init method of every plugin loaded, passing them the QuickCheck instance.

The getPOs method is called next, which is given the global list of POs, plus a list of selected PO numbers and PO name patterns. The result is a “chosen” list of POs to work on, which is saved in the object and is accessible with getChosen, as well as being returned from getPOs.

The next step is a call to getValues. This is passed one PO at a time and calls each of the enabled strategy getValue methods to generate binding values. Before it can do this, it has to convert the PO’s AST into an executable “IN” form and search for the type bindings in the PO expression. There are private methods for doing this. The getValue also expands any @QuickCheck annotations (see 2.5)

to create an IterableContext that is passed to the strategies, once for each type iteration.

After all of the strategies have been applied, if there are any binds that still don't have values, these are filled in using a standard method – essentially the same as the "fixed" strategy. This is so that there will always be *some* values for every bind, even if a limited number of strategies are enabled.

The final result of the getValues call is a StrategyResults object, which usually contains a map of bindings to their possible values, plus a flag indicating whether the result *hasAllValues* of all bindings. Alternatively, a StrategyResults may just contain a StrategyUpdater.

Then the PO and StrategyResults are passed to the checkObligation method. Before attempting to evaluate the PO, checkObligation checks whether the StrategyResults contains a StrategyUpdater. If so, the updateProofObligation method of the object is called and the PO check is considered complete. Otherwise it prepares to check all of the possibleValues passed in the StatusResults.

To start an evaluation, the bindings in the StrategyResults map must be used to instrument the corresponding parts of the IN tree of the obligation. This is done via the INBindingOverride interface. Then a new IterableContext is created for any @QuickCheck annotations, and annotation processing is temporarily suspended – since there may be thousands of executions and any @Printf output would otherwise spool to the console. A *finally* block resets the annotation processing in all circumstances after evaluation.

Then the PO expression is evaluated. Normally the result is a boolean, and ideally *true*. But a PO evaluation may also throw exceptions, which are caught and effectively treated as *false.* A timer thread limits the evaluation duration, sending a user cancel signal if it takes too long, so this case is checked for explicitly and is recorded as a *TIMEOUT*.

A boolean expression may return an *undefined* value. This occurs when, say, a *forall* expression is instrumented to have a subset of the possible bind values, and all of those values pass the predicate of the expression. This cannot return *true* because it is possible that other values would falsify the expression; but the result is not *false* either, because none of the bindings has failed. So the *undefined* value is returned and this is interpreted as MAYBE by QuickCheck. Similarly, an *exists* expression may return *undefined* if a subset of bind values are available and none of them passes the predicate. It is possible that the expression would return *true* if more values were available, but it cannot return *false* because it hasn't tried everything. Here again, an *undefined* result is interpreted as MAYBE. A *forall* with a *false* predicate will immediately return *false*; an *exists* with a *true* predicate will immediately return true. This logic is consistent with LPF[1].

The result of the evaluation has to be analysed to determine what category of result this it – either *FAILED, PROVABLE, MAYBE* or *TIMEOUT.* The analyseResult method looks at the overall true/false result, as well as the flags set regarding *execCompleted, execException, hasAllValues* and *timedOut,* and whether the PO is existential or universal (ie. it starts with an *exists* or a *forall*).

The analysis updates various fields in the PO.

If the final result is *MAYBE*, the maybeHeuristic method is called to try to qualify the outcome. This may also update fields in the PO (typically the qualifier field).

There is a printQuickCheckResult method which is called from the VDMJ/LSP commands to actually print the output of a QC run, using the fields in the PO that have been set by the analysis above.

2.6.1. Comments

QuickCheck is perhaps trying to do too much. It might be better to implement a general "proof support" framework in VDMJ, into which a simpler QuickCheck would fit along with other proof extensions.

[1] The Logic of Partial Functions, which underlies the theory of VDM. See https://eprints.ncl.ac.uk/3872

3. Using QuickCheck

3.1. Usage of “qc”

You can get help on the usage of “qc” by using the “-help” option:

```
> qc -help
Usage: quickcheck [-?|-help][-q|-v|-n][-t <msecs>]
      [-i <status>]* [-s <strategy>]*
      [-<strategy:option>]* [<PO numbers/ranges/patterns>]

  -?|-help           - show command help
  -q|-v|-n           - run with minimal, verbose, basic output
  -t <msecs>         - timeout in millisecs
  -i <status>        - only show this result status
  -s <strategy>      - enable this strategy (below)
  -<strategy:option> - pass option to strategy
  PO# numbers        - only process these POs
  PO# - PO#          - process a range of POs
  <pattern>          - process PO names or modules matching

Enabled strategies:
  fixed [-fixed:file <file> | -fixed:create <file>][-fixed:size <size>]
  search (no options)
  finite [-finite:size <size>]
  trivial (no options)
  direct (no options)
  reasons (no options)
  constant (no options)

Disabled strategies (add with -s <name>):
  random [-random:size <size>][-random:seed <seed>]
>
```

The help output lists the common options to start with, then lists the enabled strategies followed by the disabled strategies. The strategy lists include their own command options. By convention, strategy options include the name of the strategy, to avoid clashes. For example “-fixed:size” sets the default number of values generated by the fixed strategy.

The simplest use of “qc” is to use it with no options. In this case, the POG is called to generate obligations if they have not already been generated, and then all of the POs *in the current module or class* are processed. For very small specifications, this is how you will usually use it.

The next simplest usage is probably to test one particular PO by number. So that would be “qc 123” for PO number 123. But note that obligation numbers can change if you modify the specification and cause new obligations to be created or old ones removed before the one you are looking at. You can run a range of obligations, either by listing them all, or by using “qc <from> - <to>” (with spaces around the hyphen, to make it a separate argument).

You can sometimes get a more stable selection of POs by using a name *pattern* rather than a number. If you use “qc <pattern>” that will be matched against the name of the PO and the name of the module/class that the PO appears in. The patterns are *java.util.regex.Pattern* strings. The name of the PO is usually the function/operation that contains it, though obligations in measures also have that in the name, like “f; measure_f”. If you want to select all POs in all modules/classes, you can use “qc .*”. You can use multiple patterns, if you wish, meaning any of them.

You can limit the amount of output you see with the -q/-v/-n flags. The -q option is as quiet as possible; the -v option adds extra verbose output (see the *verbose* method in QCConsole); and the -n option causes nominal output, which is very regular and just shows the status and the duration.

You can limit the time of any one PO check by using the -t option, which takes a millisecond argument. This will try to cancel the evaluation after this time, though it is only accurate to within

~10ms or so, because of Java scheduling delays. The default timeout is 5000ms (5 seconds). By using a small timeout value, you can get a very quick idea of which POs are trivially (dis)provable. By setting it to a larger value, you may produce fewer *MAYBEs* but the overall run will take longer. If you set the timeout to zero, there is no time limit.

To increase the effort that qc spends on each PO, you can sometimes set a “size” option for strategies. For example, setting “-fixed:size 500” you will set the default number of values generated by the fixed strategy to 500, rather than the default, which is 20. You may need to increase the -t timeout value as well, if you start to see more *TIMEOUT* results.

The -i option will only include results for a particular status, like “qc -i failed” (case insensitive). You can use multiple -i options. This is useful because often you want to focus on just failures, or to know that there are none. Any status can be passed; see POStatus for a complete list. Note that this option still evaluates all of the POs that you specify (to find their status!), but it will only *show* you the ones with the status(es) you asked for.

By default, strategies are enabled or disabled according to their useByDefault method. But this can be overridden using the -s option. The option matches one (case *sensitive*) strategy name, though multiple -s options can be used. As soon as one -s is used, *all other strategies are disabled*. So adding “-s search” will just use the search strategy. This can be useful if you have added your own strategies and want to test them in isolation.

3.2. Usage of “qr”

The “qr” command is very simple. It can only be used with a single PO at a time, and that PO must previously have been processed with “qc” to produce a counterexample. So you may see responses like this:

```
> qr
Usage: qcrun <PO number>
> qr 1
Obligation does not have a counterexample/witness. Run qc?
>
```

If you have run “qc” and a PO has resulted in a *FAILED* status with a counterexample, “qr” will attempt to use the counterexample bindings to match the parameter values in the enclosing function or operation. It naively does this by name. So if you have a function which hides names[2] it may not be possible to guess the argument value to set. Similarly, if you have a complex recursive (especially mutually recursive) specification, or you have complex operations, it may not be possible to unpick the outermost arguments to use from the counterexample. But it will try, and it works for most straightforward cases.

For example:

```
functions
      f: nat * seq of nat -> nat
      f(i, s) == s(i);

> qc
PO #1, FAILED in 0.003s
Counterexample: i = 0, s = []
----
f: sequence apply obligation in 'DEFAULT' (test.vdm) at line 3:16
(forall i:nat, s:seq of nat &
  i in set inds s)

> qr 1
=> print f(0, [])
Error 4064: Value 0 is not a nat1 in 'DEFAULT' (test.vdm) at line 3:16
3:    f(i, s) == s(i);
MainThread> -- start debugging here!
```

[2] For example with a “let” expression that re-defines a parameter variable. Please don’t do this!

So in this trivial example, the nat passed is not always a valid index of the sequence. That is probably obvious, but the “qr” drops you into the debugger at the point where the function fails, and so you can look around and do the normal things that you do to debug what happened.

4. Writing New Strategies

4.1. Overview

As mentioned in section 2.3, QuickCheck is designed to use a set of pluggable strategies for finding counterexamples, or analysing POs to decide whether they are probably *PROVABLE*. The tool comes with a number of strategies built-in, but users can create new strategies by following the same design and adding a jar to the VDMJ classpath.

A strategy is passed:

- A list of command line arguments for its settings
- A *ProofObligation* to analyse
- A list of *INBindOverrides* in the PO
- An execution *Context*

And a strategy returns:

- Its name and help strings
- Whether it has encountered any errors
- Whether it should be configured for use by default, or only via “-s”
- One of:
  - A *StrategyResults* list of possible values for each *INBindingOverride*
  - A *StrategyUpdater* that directly changes the PO as required.
- Messages for any *maybe heuristics* that it provides.

4.2. The Example Strategy Explained

QuickCheck includes the source of an example for how to write a strategy:

```
src/test/java/quickcheck/example.java
```

The example is only to illustrate what methods can be implemented and does not do anything sensible. To run the example, add the *quickcheck-<version>-tests.jar* to the classpath. Then “qc -help” lists a new *disabled* strategy at the end, called “example”:

```
> qc -help
...
Disabled strategies (add with -s <name>):
  random [-random:size <size>][-random:seed <seed>]
  example [-example:proved <bool>]
>
```

Notice that the “example” strategy is listed and shows a command line option, “-example:proved”. The strategy can be tested using the “qc -s” and “-example:proved” options as follows, assuming the specification has a *MAYBE* obligation at #5:

```
> qc 5  # using default enabled strategies
PO #5, MAYBE in 0.002s

> qc -s example -example:proved true 5
PO #5, PROVABLE Just an example in 0.001s

> qc -s example -example:proved false 5
PO #5, MAYBE in 0.0s
>
```

The “proved” option passed to the strategy causes it to claim proof if it is set “true”, or to do nothing if its is “false” – obviously this is just an illustration!

Looking at the Java source, the *ExampleQCStrategy* extends an abstract *QCStrategy*. This has an abstract *getName* method, and defines sensible defaults for the remaining methods.

The constructor is passed a *List<String>* containing all of the command options passed to “qc”. It goes through them, looking for “-example:proved”, removing it if found. Other “-example:” options cause an error. An *errorCount* is inherited from the parent class and is reported by its *hasErrors* method, so this is incremented if any errors are encountered. Error messages can either be printed unconditionally with *PluginConsole.println (*to stdout*)* or *errorln* (to stderr), or they can be printed via the equivalent *QCConsole* methods. The latter will suppress output if the “-q” flag was passed to “qc”.

The example *getName* method simply returns the name “example”. This is case sensitive. By convention, strategy names are all lower case.

The *init* method is called once for each “qc” execution and is passed the QuickCheck instance that is operating. That can provide useful information for a strategy, though not in this example. The return value from *init* is a boolean, which will cause “qc” to stop if it is false – for example for a fatal error, or if your strategy has options that change permanent settings (like “-fixed:create <file>”).

The *getValues* method is the core of the strategy and is responsible for either returning a list of counterexamples to try, or returning a *StrategyUpdater* which can manipulate the obligation directly. The example populates a *Map<String, ValueList>* with empty value lists, one for each binding passed in. The key of the map is the *toString* of the binding. A real strategy would populate these lists with values matching each binding which it hopes will cause counterexamples.

If the “-example:proved” option is “false”, the example returns the map with no values and a flag to indicate that it does not contain all possible values of the bindings. If the “-example:proved” option is “true”, it returns a *StrategyUpdater* which directly alters the *ProofObligation* via its *updateProofObligation* method. The updater is a private *ExampleUpdater* class at the end, extending *StrategyUpdater.*

The *help* method should provide a one-line help description, which is listed by “qc -help”.

The *useByDefault* method returns false, which causes the strategy to be disabled unless it is invoked by “-s”. Returning true enables a strategy by default.

The jar containing the strategy includes a resource file called “qc.strategies” (see more below). In the example, this just lists the classname of the strategy it contains:

```
quickcheck.example.ExampleQCStrategy
```

4.3. The Random Strategy Explained

The *random* strategy is built into the QuickCheck tool, but it is a good example that illustrates the typical design of a real QC strategy plugin. The Java source is in RandomQCStrategy.java.

Like all strategy plugins, the class extends an abstract *QCStrategy.*

As above, the constructor is passed a *List<String> argv* and the arguments which apply to *random* start with “-random:”. This is a convention to keep the arguments for different strategies separate. As the constructor processes the list of arguments, it removes any which it recognises, and any which start “-random:” that are not recognised raise an error – these are usually user typos.

The plugin implements the *getName* method, which returns “random”, as you would expect.

An *init* method is passed the QuickCheck object, which some strategies can use to query settings, but *random* does not use it. The *verbose* method is like *println* but only gives output if the “-v” flag has been passed to “qc”. Built-in strategies use this to print their argument values in *init*.

The *useByDefault* method returns whether this strategy should be included in “qc” checks every time, or whether it should only be used when explicitly named with “-s”. If a strategy is expensive, or only applicable in rare cases, it may be sensible to return false here. The *random* strategy is not enabled by default, because it is very similar to the *fixed* strategy.

The bulk of the strategy's work is done in the *getValues* method. This is passed the PO, a list of its bindings, and an execution *Context*. The objective of *getValues* is to return a set of values for each binding that, in some combination, may hopefully provoke a counterexample. The values are returned in a *StrategyResults* object. If a strategy *knows* that it has a counterexample that fails or a witness that succeeds, it can return this via a *StrategyUpdater* (below).

The random strategy uses a PO flag called *hasCorrelatedBinds* to decide whether the PO should generate fixed values or random ones. This is very unusual, and only required for *CasesExhaustiveObligations*.

For most POs, the random strategy creates a *RandomRangeCreator* for each binding, passing in the *Context* and seed, and adding the returned *ValueSet* to a list, which is inserted into a map keyed by the binding's *toString*. This map is finally used to create the *StrategyResults* to return. A flag to *StrategyResults* indicates that the generated values, being random, do not include all values of all bind types (the "finite" strategy would set this to true, for example).

If a strategy omits values for one or more of the bindings, QuickCheck will fill in some values using the *FixedRangeCreator,* since all bindings need at least one value.

The *RandomRangeCreator* is a VDMJ *TCTypeVisitor,* which considers the type of the binding passed to it, and generates random values of that type, progressively combining primitive types into more complex ones. It uses the execution *Context* to create values that include a type invariant. This visitor, and *FixedRangeCreator,* can be re-used in other strategies.

A strategy can implement the *maybeHeuristic* method from *QCStrategy*, which allows it to add extra information to any PO that results in a *MAYBE* status after all strategies have run. If implemented, this method should directly update the message field of the *ProofObligation* passed to it. This is used by the "reasons" strategy, for example.

Finally, the *help* method returns a line to add to the "qc -help" output for the strategy. It should list the options available, if any, on one line.

4.4. Returning a StrategyUpdater

Instead of returning a list of possible counterexamples for a PO's bindings, a strategy may return a *StrategyResults* that just contains a *StrategyUpdater* object.

A *StrategyUpdater* is used to directly manipulate the PO and to stop the rest of the strategy search process, because your strategy is confident that it can prove or disprove the PO. For example, a *TrivialUpdater* is used by the "trivial" strategy, and does the following:

```
private class TrivialUpdater extends StrategyUpdater
{
	private final String qualifier;

	public TrivialUpdater(String qualifier)
	{
		this.qualifier = qualifier;
	}

	@Override
	public void updateProofObligation(ProofObligation po)
	{
		po.setStatus(POStatus.PROVABLE);
		po.setProvedBy(getName());
		po.setQualifier("by " + getName() + " " + qualifier);
	}
}
```

As mentioned above, if a strategy is sure that it has found a counterexample or a witness, it can set those values directly on the *ProofObligation* using a *StrategyUpdater*, rather than returning them via the bindings as usual.

4.5. The “qc.strategies” Resource

If the code above is packaged into a Java jar and added to the classpath, QuickCheck will not know that the jar contains new strategies unless a resource file called “qc.strategies” is added which lists the full classnames of the classes in the jar that are QC strategies.

The same resource name is used for built-in strategies. So the built-in “qc.strategies” file is a good example. It is in src/main/resources and contains:

```
quickcheck.strategies.FixedQCStrategy
quickcheck.strategies.SearchQCStrategy
quickcheck.strategies.RandomQCStrategy
quickcheck.strategies.FiniteQCStrategy
quickcheck.strategies.TrivialQCStrategy
quickcheck.strategies.DirectQCStrategy
quickcheck.strategies.ReasonsQCStrategy
quickcheck.strategies.ConstantQCStrategy
```

A similar file (with one line for each new strategy) should be added to your plugin jar.