QuickCheck
An introduction

MORITZ GÖCKEL

Hochschule Karlsruhe Technik und Wirtschaft

November 26, 2018

Contents

1 Introduction

1.1 Structure

At first this work is going to give a brief introduction to software testing, automated testing and the difficulties of testing large systems. To conclude the introduction the idea of random testing is going to be proposed and motivated. The second chapter is aiming to give an overview over the ideas of QuickCheck. The theory is aided by examples in Haskell, as this is the language QuickCheck was originally written in. The third chapter deals with the evaluation on how the ideas of QuickCheck translate to another language of another paradigm. As an example Java was chosen as it is commonly used in the industry and is an imperative language in contrast to Haskell. In the last chapter this work will then give an example on how QuickCheck was used in a commercial setting and how it performed.

1.2 Software testing

Testing is the primary method to evaluate the correctness of software today[1]. Even though exhaustive testing is possible in many cases and the majority of bugs can be triggered by relatively few test cases[9], the required effort to find these test cases can be high. In practice the time spend writing code is not seldom matched by the time spend on finding bugs and creating tests.

Test cases have to be defined to evaluate a systems correctness. Such a test case usually consists of two components: The stimulus and the expected result. A stimulus can be any kind of input that is passed to the system under test (SUT). The expected result is the expected output or state of the SUT after the test.

The execution of such a test case can be divided into three distinct phases. First the SUT has to be stimulated by the defined stimulus. In practice this means for example that a person uses the software in a certain way or that a function is called with certain parameters. The second phase is the observation of the SUT's behavior. In practice the user might just look at the UI of the software or the return value of the invoked function is stored in memory. In the final phase the previously recorded data is compared with the expected results of the test case. This could mean in practice that the tester notices that the formatting of a website is off or that the function returned a incorrect result.

1.3 Automated software testing

One way to evaluate the correctness of software is to keep and eye on irregularities while manually testing it. This way is very time consuming as every test case has to be conducted by a person. It can take months to test a big software system this way[2]. Because of this human element, manual testing is also very error prone and therefore not reliable. As an addition to manual testing automated testing has been widely adopted in the industry.

In automated software testing the three previously described phases are implemented in code to be executed automatically after a certain test case has been defined. The major benefit of this way of testing is that conducting tests becomes very cheap. The downside is that defining such test cases can be labour intensive.

```haskell
testSum :: Bool
testSum =
    sum [1,2] == 3
```

This is an example of a very basic test case. The test case is implemented as a function in Haskell; it's name is ″testSum″. As the name suggests this test case evaluates the correctness of the ″sum″ function. The ″sum″ function receives as input an array of integers and returns a single integer. Therefore the SUT in this example is the ″sum″ function. When we execute the test case (invoke the ″testSum″ function) the three previously described steps are taken: The SUT is stimulated by invoking it with the array of 1 and 2. The behavior of the system is recorded and compared with the expected result. In this case the expected result is ″3″. The test case then returns ″True″ if it holds and ″False″ otherwise.

1.4 The problem of testing large systems

The problem that arises with testing is that the amount of required test cases for a good coverage of a software system is rising exponentially as the system becomes larger and more complex. Imagine for example a software with n features. To test each feature individually, one would need at least n test cases. This is still feasible. But in most systems the features interact with each other. This leads to the necessity to also test every feature in cooperation with every other feature. Testing these pairs of features requires us to write therefore an additional n^2 new test cases. This thought experiment can be continued with three interacting features, ergo n^3 necessary test cases and so forth[5].

We find that in very large complex systems it is often no longer feasible to gain good test coverage by writing test cases manually. Another way of testing needs to be found.

1.5 Random testing

With automated testing bigger test coverage can be achieved and a test runs can be conducted within minutes or hours instead of the months it would take to manually test the software. Unfortunately the testing process is still one of the most expensive parts of software development, even with automated testing. Not seldom the testing efforts account for about half of the development costs[3]. The logical next step to reduce these costs after automated testing is the automated generation of test cases. The utilization of automated generation of test cases may lead to further reduced testing effort and the possibility to gain more test coverage in less time.

This work focuses on QuickCheck which is one implementation of the idea of automated test generation. QuickCheck relies on the same three steps of testing that have been described earlier and additionally defines three concepts: Properties, generators and "shrinking". These concepts are going to be explained in detail in the next chapter.

```
prop_sum :: Property
prop_sum =
    forAll
        -- defining a generator for arrays of two elements
        (vectorOf 2 (choose (0, 1000)))
        -- applying it to the sum function and comparing it
        -- with the sum of the two elements
        ((\xs -> sum xs == xs!!0 + xs!!1) :: [Integer] -> Bool)

-- executing quickCheck to generate concrete test cases
*Main> quickCheck prop_sum
+++ OK, passed 100 tests.
```

This is example is a reimplementation of the sum test case that we defined in the section "Automated software testing". We use the concepts of QuickCheck to test the SUT not only with one array but with 100. This may seem daunting at first glance, but it is going to get clear after reading the next chapter.

2 The idea of QuickCheck

QuickCheck has three core pillars: Properties which define a general rule about the to be tested code, generators which create controllable random input data to create test cases and ″shrinking″ which reduces failing test cases to an minimal test case to aid diagnosis. These three concepts will be shown in this chapter.

2.1 Properties

The most expensive part of software testing is not to run or to evaluate them, often the most expensive part is writing the test cases. QuickCheck tackles that problem by letting the software developer write generalized test functions of which each covers many test cases. These generalized test functions are called properties in QuickCheck[7]. Because of this generalized approach it is not necessary to write more than one property for each logical property of the to be tested function because many different test cases will be generated anyway[6].

The concept named property in QuickCheck is equivalent to the concepts of invariants and test oracles. Invariants originated in mathematics to describe rules that hold no matter the concrete case. Oracles are functions that determine whether or not a test has passed.

In QuickCheck a property is defined as a function that takes some random test data, provides that test data to the to be tested function and determines afterword whether a general rule about the relation between the input and the to be tested functions output holds.

2.1.1 Defining properties

For example imagine we want to test the ”sum” function. We know that the sum of the elements in an array is always the same, no matter the order of the array. We can formulate this kind of logical assertion as a property in Haskell:

```
prop_sumRev :: [Int] -> Bool
prop_sumRev xs =
    sum xs == sum (reverse xs)
```

Now we can use QuickCheck to validate whether or not this statement about the sum function holds:

```
quickCheck prop_sumRev
-- OK, passed 100 tests.
```

QuickCheck generates a set of inputs for the provided function, in this case arrays of Int. QuickCheck may be able to infer the type of the to be generated parameters, but it is good practice to define the type explicitly. That way it is also certain that the generated input is of type Int and not Double for example. The property holds if the function always returns true for every generated test case[3]. Here QuickCheck generated 100 test cases which all yielded true, it can therefore be assumed that the property holds.

2.1.2 Conditional properties

Many logical properties only hold under certain conditions. Fortunately QuickCheck provides a concise way to define such an conditional property:

```
prop_positivePlusOne :: Int -> Property
prop_positivePlusOne x =
    x > 0 ==> x + 1 > 0
```

```
quickCheck prop_positivePlusOne
-- OK, passed 100 tests; 110 discarded.
```

This property basically states that if x is positive then (x + 1) is also positive. The ==> operator asserts that only if the left side is true, the right side needs to be true as well[3]. Not all test cases are relevant to the test outcome when using this operator. In this example all test cases with x being zero or negative do not contribute to the result and are therefore discarded. That is why QuickCheck actually had to run 210 tests to find 100 valid ones.

2.1.3 Classify & Collect

As seen in the previous example, it is sometimes important to have some information about the generated input data to evaluate how reliable the test actually was. To get more insights into the test cases QuickCheck provides us the functions ”classify” and ”collect”[3]. Here an example:

```
prop_sumRev :: [Int] -> Property
prop_sumRev xs =
    classify (null xs || xs == []) "Array empty" \$
    sum xs == sum (reverse xs)

quickCheck prop_sumRev
-- OK, passed 100 tests (7% Array empty).
```

As we can see in the output, 7 of the 100 test cases have been conducted with either null or an empty array. This value changes from test run to test run because the generation of input data is random. It might be useful to know these things as these kinds of tests do not really test the sum function like the developer intended. Now lets look at ”collect”:

```haskell
prop_sumRev :: [Int] -> Property
prop_sumRev xs =
    collect (length xs) $
    sum xs == sum (reverse xs)

quickCheck prop_sumRev
-- OK, passed 100 tests:
--   6% 2
--   4% 0
--   4% 1
--   4% 11
-- ...
--   1% 86
```

The output shows the distribution of data that has been passed to the ”collect” function. In this case one gets insights on the distribution of the array lengths. 6% percent of the arrays have been of length 2 followed by 4% of the length 0.

Note that because the output of QuickCheck contains more information about the test runs when using ”classify” or ”collect” the return type ”Property” has to be used. The return type ”Bool” can only encode whether or not the test passed and is therefore no longer valid when using these functions.

2.1.4 Infinite data structures

As QuickCheck was initially implemented in Haskell and Haskell supports infinite data structures, therefore QuickCheck also in theory enables developers to test infinite data structures. Because infinite data structures are only feasible with lazy evaluation and can never be fully evaluated, they of course can also never be fully tested. QuickCheck solves this problem by making the assumption that if a finite amount of elements of an infinite data structure are valid, then the entire data structure is valid. The solution in practice is to convert an infinite data structure into a finite one by taking n elements and then evaluating the correctness of the finite data structure. The number n is thereby also randomly generated by QuickCheck. If the rule holds for the finite data structure we assume that it also holds for the infinite one[3].

2.2 Generators

To generate a concrete test case from a general property some varying input data is required to pass to the to be tested function. This input data is generated randomly in QuickCheck. Even though it might be surprising, it turns out that this random approach competes quite well with systematic methods in practice[3].

QuickCheck has no heuristics or assumptions about the possible input data, it is therefore the task of the user to define its structure. This can be done by implementing generators. This enables the user to define the distribution of the test data to model the expected real world input as close as possible.

2.2.1 Basic generators & forAll

QuickCheck provides a wide variety of combinable generators which can be used very effectively to model the randomly chosen input data. One of these generators is ″choose″ which when provided two arguments, returns a random value between them. This can be used to optimize the previous example ″prop_positivePlusOne″ by generating only positive input data in the first place. That way we avoid generating test cases that do not contribute the evaluation of the function and therefore will speed up computation time.

```
prop_positivePlusOne :: Property
prop_positivePlusOne =
    forAll
        (choose (1, 10000))
        ((\x -> x + 1 > 0) :: Integer -> Bool)

quickCheck prop_positivePlusOne
-- OK, passed 100 tests.
```

The ”forAll” function is central to this implementation: It receives an generator and a testable function (a function that either returns bool or a property) and conducts the testing. Also note that the outer property function no longer takes any input as it generates it itself using the provided generator ”choose”. The result is just the same as in the previous example in the section ”Conditional properties”, just that it avoids generating test cases that have to be discarded.

There are many more predefined generators provided by QuickCheck. The most basic generator named ”arbitrary” for example generates random values for a given type. ”arbitrary :: Int” generates integers for example. If one wants to generate a fixed size list there is the ”vectorOf” generator which generates a list of a given size and populates it with the values from a given generator. If one needs a list of random length the ”listOf” function comes handy. Many more of those generators can be found in the QuickCheck documentation[12].

2.2.2 Frequency generator

For testing functions one usually creates test input, that mimics the later expected real input or creates input that puts the to be tested function under maximum pressure to find bugs fast. To be able to do that one might want to define a generator statistically:

```haskell
mostlyPositive :: Gen Int
mostlyPositive =
  frequency
    [
        (8, choose (1, 10)),
        (1, choose (-10, -1)),
        (1, return 0)
    ]

generate (vectorOf 12 mostlyPositive)
-- [7,10,4,10,7,4,-8,3,9,1,0,7]
```

This example defines a generator which consists of three generators with assigned probabilities: In 80% of the cases it is going to choose an integer between 1 and 10, with 10% probability it will choose an integer between -10 and -1 and the 0 is chosen with also 10% probability. This generator is therefore highly skewed towards positive numbers but also generates negative numbers occasionally. This is achieved by using the frequency function that takes an array of tuples of the frequency and the generator. To test the generator we did not define a property to use it with but instead just created some values with the function "generate".

2.3 Shrinking

QuickCheck provides very powerful tools to evaluate the presence of bugs with its generators and properties. Imagine a just defined property states a logical rule, an appropriate generator creates test inputs and now it is found that a property does not hold. QuickCheck informs the test engineer about the input data that breaks the property. Finding the mistake in the code can be quite challenging, especially if the violating test case is very complex and patterns are therefore hard to find.

To help the software developer to understand the bug more easily QuickCheck ”shrinks” the violating test case systematically to iteratively find violating test cases that are smaller. QuickCheck shrinks the violating test case until it can not be shrunken further. This minimal violating test case is then given to the user. This method makes finding the error in the code easier after a violating test case has been discovered by QuickCheck[4].

Shrinking consists of removing method calls and simplifying numbers in violating test cases and evaluating whether or not the new and simpler test case still violates the property. This can be a very computational expensive endeavor and is reported to take up to 80% of the computation time. This can have quite an impact on waiting time for the testing engineers, especially when violating test cases are easy to find but hard to simplify[8].

Imagine we have a function that breaks sometimes, but unfortunately very seldom. This is hard to debug. The next example is such a function. Most of the time it receives an array of integers and returns the sum. Except for when the array contains a seven and an thirty: Then it returns a sum that is off by one.

```haskell
buggySum :: [Int] -> Int
buggySum xs =
    if not ((elem 7 xs) && (elem 30 xs))
        then sum xs
        else (sum xs) - 1
```

We define a property to test the function. This is simple because in this example we know that buggySum should behave just as sum. Therefore we can just compare the results of the two functions.

```haskell
prop_sum :: [Int] -> Bool
prop_sum xs =
    sum xs == buggySum xs
```

No imagine we execute QuickCheck and it tells us that it found a violating test case. Something along these lines:

```
*Main> quickCheck prop_sum
-- Failed! Falsifiable (after 93 tests)
-- [30, 21 , 52, 199, 41, 85, 1, 0, 7, 23, 49, 19, 30, ...]
```

This report is not useful. QuickCheck told us that the test breaks but we have no idea why. There is no pattern obvious in the input data that could help us locate the bug. Fortunately this is not the actual output of QuickCheck. Instead the output looks like this:

```
*Main> quickCheck prop_sum
-- Failed! Falsifiable (after 93 tests and 9 shrinks):
-- [30, 7]
```

QuickCheck found a failing test case within 93 tries and then removed iteratively more and more elements from the list that made the test fail. After 9 shrinks QuickCheck found the simplest failing test case. Now the bug is obvious: ”buggySum” always breaks when both 7 and 30 are in the input array. QuickChecks shrinking is a very useful feature for finding bugs.

3 QuickCheck in Java

The section "The idea of QuickCheck" utilized exclusively Haskell to show the main ideas behind QuickCheck. This is because QuickCheck has been originally implemented in Haskell. But even though QuickCheck is rooted in functional programming, its main ideas still have been translated into many other languages and paradigms. As functional programming avoids uncontrolled side effects, it is known for being better testable. Therefore the question whether or not the ideas of QuickCheck can be applied to imperative languages arises.

Java is at the time of writing the most popular programming language according to the TIOBE Index[13]. Even though such rankings should always be "taken with a grain of salt" it is still agreed upon that Java is a quite widely used language today. This is why it is interesting to evaluate whether or not the ideas of QuickCheck can be applied to modern Java today.

3.1 Jqwik

In the research for this work already seven implementations of QuickCheck for Java have been found. Some ar no longer under development and the amount of provided functionalities varies. But anyhow its surprising to find such a variety of implementations of the QuickCheck idea. For the sake of providing some insights how the QuickCheck ideas might look in a Java environment the Jqwik[10] has been chosen. Jqwik is a convenient to use and feature rich software package with an exceptional documentation that makes good use of the Java features to implement the ideas of QuickCheck. These are the reasons why Jqwik has been chosen as an example.

3.2 Properties

In Java first a class has to be created in order to define a property inside it as a method, just like in JUnit, one of the most commonly used testing libraries for Java. To mark the method as a property annotations are used. In contrast to Haskell we no longer run the tests by calling quickCheck with the method as parameter, we just run the test suite.

```
@Property
boolean prop_positivePlusOne(@ForAll @IntRange(min = 0, max = 100) int num) {
    Assume.that(num > 0);
    Statistics.collect(num);

    return num + 1 > 0;
}
```

This first example is similar to the first QuickCheck example in Haskell in the beginning of this work. It is assumed that the input parameter is greater than one or else we discard the test case. This is equivalent to the conditional property in Haskell. Also shown in this example is the previously presented collect method that generates some statistics about the test run that are going to be printed after completion. The generation of input data is manipulated by providing optional annotations. In this example the IntRage annotation is used to specify the min and max value of the input. This example shows how the ideas and also the nomenclature of QuickCheck is implemented quite similar and intuitively by Jqwik in Java.

3.3 Generators

The next example[11] makes use of some more generator annotations to create a quite complex specification of the input data:

```java
@Property
void uniqueInList(@ForAll @Size(5)
                  List<@IntRange(min = 0, max = 10) @Unique Integer> aList) {

    Assertions.assertThat(aList).doesNotHaveDuplicates();
    Assertions.assertThat(aList).allMatch(anInt -> anInt >= 0 && anInt <= 10);
}
```

The input of this property will always be a list with unique integer values between zero and ten of size 5. This shows the power and expressiveness of this approach. Also this example shows that in Jqwik it is common to use assertions to signal a failing test instead of returning a boolean.

The following example shows how to create more custom generators in Jqwik. Generator functions in Jqwik always return an arbitrary of the to be generated type and is marked with the "@Provide" annotation. To use the generator in an property one just needs to provide the name of the method to the "@ForAll" annotation next to the parameter[11].

```java
@Property
boolean testingGenerators(@ForAll("names") String name,
                          @ForAll("oddNumbers") int num) {

    Statistics.collect(name);
    return num % 2 == 1;
}

@Provide
Arbitrary<String> names() {
    return Arbitraries.frequency(
            Tuple.of(1, "John"),
            Tuple.of(5, "Jack"),
            Tuple.of(10, "Jordan")
    );
}
```

```java
@Provide
Arbitrary<Integer> oddNumbers() {
    return Arbitraries.integers().filter(i -> i % 2 == 1);
}
```

Note that this example uses the previously presented frequency method that works just like the one in Haskell and that the Arbitraries class utilizes Java streams for some of its predefined generators.

3.4 Stateful testing

As Java is a imperative and object oriented language, Java applications are inherently stateful. Jqwik supports testing stateful methods in a quite similar way like described in the section ”Stateful systems” in Haskell. Just to recap: In essence Actions need to be defined so that they can be executed on the to be tested object. One test case consists of a list of random length containing a random sequence of these actions, instantiated with random parameters. After the execution of each action the state of the to be tested object has to be evaluated to determine whether irregularities occurred. Here an example:

```java
class MyList {
    private ArrayList<String> list = new ArrayList<>();

    public void add(String element) { list.add(element); }
    public int size() { return list.size(); }

    public void remove() {
        if (size() > 0 && !list.contains("10"))
            list.remove(0);
    }
}
```

Here we have an simple data structure with an obvious bug, ”remove()” does not work if ”10” has been added before. Now we create the actions to test this data structure:

```
class AddAction implements Action<MyList> {
    private String str;

    AddAction(String str) {
        this.str = str;
    }

    @Override
    public MyList run(MyList list) {
        int beforeSize = list.size();
        list.add(str);
        Assertions.assertThat(list.size()).isEqualTo(beforeSize + 1);
        return list;
    }

    @Override
    public String toString() { return String.format("add(%s)", str); }
}

class RemoveAction implements Action<MyList> {
    @Override
    public MyList run(MyList list) {
        int beforeSize = list.size();
        list.remove();
        Assertions.assertThat(list.size()).isEqualTo(Math.max(beforeSize - 1, 0));
        return list;
    }

    @Override
    public String toString() { return String.format("remove"); }
}
```

In this code two actions are defined: The AddAction and the Remove-Action. Both retrieve the size of the data structure before conducting the operation on it. The actions make a prediction about the expected size after the operation and then check whether that assertion is true or not. The code for the AddAction is a bit longer than the RemoveAction as it needs to store the random value that then gets used as parameter for the ”add” operation. Now lets have a look on how the list of actions gets created:

```
@Provide
Arbitrary<ActionSequence<MyList>> listActionSequences() {
    return Arbitraries.sequences(
            Arbitraries.oneOf(
                    Arbitraries.constant(new RemoveAction()),
                    Arbitraries.integers().between(0, 100)
                                          .map(Object::toString)
                                          .map(AddAction::new)
            )
    );
}
```

This piece of code may look a bit daunting at first, but its actually quite straight forward as soon as one gets comfortable with generators. Here the goal is to define a generator for sequences of the two previously defined actions. To that end predefined generators have been combined to fulfill that exact need. First (from innermost) two action generators are defined. The first one is trivial: Its a constant generator that always returns a new instance of a ”RemoveAction”. The second one is more complex: Integers from the predefined generator are retrieved, filtered for the ones between 0 and 100, converted to a string and then for each of these a new ”AddAction” gets instantiated. The ”AddAction” gets the strings provided via its constructor. On the next level a random one of these two just defined generators gets chosen repeatedly as elements for a sequence of random length. Now the sequence of actions is defined and it just needs to be applied to the data structure in a property:

```
@Property
void checkList(@ForAll("listActionSequences") ActionSequence<MyList> actions) {
    actions.run(new MyList());
}
```

This one is simple. A property gets defined that retrieves action sequences from the previously defined generator ”listActionSequences” and runs it against a new instance of the to be tested data structure. The following output informs us about a found bug:

```
timestamp = 2018-10-29T20:56:05.363094
    tries = 13
    checks = 13
    generation-mode = RANDOMIZED
    seed = -5749229686075369685
    originalSample = [SequentialActionSequence (after run):[ ... ]]
    sample = [SequentialActionSequence (after run):[add(10), remove]]

org.opentest4j.AssertionFailedError: Run failed after following actions:
    add(10)
    remove

Expecting:  <1>
to be equal to: <0>
but was not.
```

The implementation of the data structure had a bug that Jqwik found after 13 tests. The value of ”originalSample” shows how complex that first failing test case was, it contained 32 operations. Fortunately Jqwik ”shrunk” the failing test case to the minimized failing text case that can be seen as the value of ”sample” and under the headline ”Run failed after following action”. To recall: The bug that was deliberately introduced, was that ”remove” does not work when ”10” is in the list. Jqwik managed to find that exact and minimal case where it just adds the ”10” and then calls

″remove″. After the ″remove″ ″size″ should be zero again but its not. That is what the last five lines inform us about.

3.5 Differences between Jqwik and QuickCheck

3.5.1 Assertions

In the original QuickCheck properties are always defined as functions that are testable and therefore either return a boolean or a property. Jqwik on the other hand encourages the user to utilize a rich library of assertions that do not return anything but throw an error on runtime when violated. Because of this heavy use of assertions most properties in Jqwik actually return void.

3.5.2 Class & Annotations

The entry point for QuickCheck in Haskell is the quickCheck method. To start the testing procedure this method is called with a to be tested property. This is not necessary in Jqwik, as Jqwik finds all the properties by itself and executes them automatically.

Because in Java no method can live outside a class, it is necessary to create a class or use an existing one to contain the property methods. Good practice is to group the properties and generators that are highly related together in one class. Such a construct is not necessary in Haskell.

To enable Jqwik to automatically find the property functions, even though they are not identifiable by their method signature, Java annotations are used. Such annotations are also utilized by Jqwik to mark generators and to map them to certain parameters of properties. Because of their heavy use, annotations are very powerful and vital to Jqwik. QuickCheck in Haskell does not need such a mechanism because every needed generator for a property is either given by the user or inferred by type.

3.5.3 Type classes and generics

QuickCheck in Haskell relies strongly on type classes. Such a thing does not exist in Java. Instead interfaces in combination with generics are used. This difference can be observed when looking at for example the different definitions of arbitrary:

The definition of Arbitrary in Haskell:

```
class Arbitrary a where
    arbitrary :: Gen a
```

The definition of Arbitrary in Java:

```
interface Arbitrary<T>{
    RandomGenerator<T> generator(int var);
    ...
}
```

Despite the very different technologies, the semantics of these two definitions are quite similar. Both declare a type named Arbitrary that provides a generator for a generic type a in Haskell and T in Java.

4 Commercial use

A commercial version named Quviq QuickCheck has been created by Claessen and Hughes, the former authors of QuickCheck. Quviq QuickCheck was first used for testing the Magaco protocol in a case study in 2006 at Ericson. The stated goal was to evaluate Quviq QuickCheck regarding its effectiveness in an industrial setting. The telecommunication sector was chosen because communication protocols are commonly both very complex and also quite well specified. The assumption was that this kind of environment is perfect for a QuickCheck-like testing approach. This chapter is going to be about that case study[2].

4.1 Erlang

In contrast to QuickCheck, which was written in Haskell, Quviq QuickCheck was written in Erlang as this language was wider spread in the industry and supported more natural state based testing. Even though some parts of the to be tested software in the Ericson case study where also written in Erlang, it is important to note that a black-box approach was used, that only generated messages and analysed replies. Therefore the language of the to be tested software was actually not relevant for the project[2].

4.2 Positive & negative testing

To conduct the testing both positive and negative test cases have been generated by Quviq QuickCheck. Positive test cases are ones where valid messages have been generated and the reply of the to be tested system had to be correct to pass the test. Negative test cases were created by generating and sending invalid messages to the system and then expecting a negative response from the system without seeing the system crashing or entering an invalid state. The distinction between negative and positive test cases has to be made quite clear in QuickCheck as they have different criteria for passing. The challenging part of the testing with QuickCheck in such a setting is to be able to randomly generate messages and to predict whether or not it is valid in order to evaluate the response of the system. This requires the test engineer to know and implement the specification of the to be tested system quite precisely[2].

4.3 Stateful systems

Stateless systems are usually assumed to be better testable. But even though the system in the Ericson case study was mostly stateless, some bugs still only showed when executing a certain combination of commands in sequence. Quviq QuickCheck therefore had to be able to test systems that also relay on internal state. Fortunately this is possible in QuickCheck: Instead of generating the input parameters for a function call, an array of to be called functions and the appropriate input parameters were generated. This yields a sequence of commands (with parameters) that can be executed on the to be tested system. The response of the system can then be analysed either after every command or at the end of the sequence. The elegance of this approach lies in the ability to use QuickChecks ″shrinking″ on the command sequence after finding a violating test case[2].

The problem with testing stateful systems on the other hand is that it is hard to evaluate whether or not the answers of the system are correct and whether or not the systems state is still a legal one. Often the solution for this problem is to only approximate if the answer and the state of the system are ″reasonable″ and no crashes occur.

Testing stateful systems was also discussed with an concrete example in the section ″Stateful testing″ earlier in this work.

4.4 Results

The authors of Quviq QuickCheck claim that their software can be used effectively in practice. They reported on finding very subtile bugs that manual testing and automated testing were unable to find. One of those bugs required seven specific commands with specific parameters to be executed in sequence to show. This again illustrates the usefulness of shrinking in QuickCheck, because the first found sequence that caused that bug was 160 commands long but got shrunken down to only seven essential commands. Such a bug would be very hard to find using other testing methods[2].

On the other hand it turns out that after QuickCheck found a bug, it tends to always find it again. Therefore the bug either has to be fixed or the test case has to be adapted to continue testing. Before this change, testing with QuickCheck can hardly continue. The creators of QuickCheck state ″QuickCheck is very likely to report the most common bug on every run″[2].

It is also stated that the specification of the to be tested system needs to be consistent, unambiguous and the testing team has to understand it very well. Every ambiguity or inconsistency in the specification will eventually be found by QuickCheck and will look like a bug. This however should be the case with every thorough testing system and should therefore not been seen as a downside[2].

5 Conclusion

At the beginning of this work the importance of testing has been stated: It is the most common way for evaluating the correctness of software. It was shown that the testing effort grows exponentially the bigger the to be tested system becomes. Random testing is the logical next step after automated testing and as an example for such a random testing approach QuickCheck was presented.

QuickCheck rests on three pillars: Properties, generators and shrinking. Properties define a general assumption about the to be tested system, generators create random input for properties. This generated input together with a property form a concrete test case. When an failing test case is found, it is ″shrunken″ down to the minimal failing test case by QuickCheck. This aids diagnosis of the failing code and helps find the reason of the bug.

The chapter ″QuickCheck in Java″ showed that the ideas of QuickCheck translate surprisingly well to Java and can also be intuitively understood because it is using the same vocabulary as the original QuickCheck. Even though QuickCheck was developed under the functional programming paradigm, it is still very suitable to test software of imperative languages as well. Testing of stateful systems can be conducted with Jqwik quite elegantly, even though it is harder and requires more effort than testing stateless functions.

The last chapter was about how QuickCheck has been used in practice. It was found that QuickCheck helps finding very subtile bugs but on the other hand requires an very consistent, complete and unambiguous specification.

6 References

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, Dec. 2016. DOI: 10.1017/9781316771273. URL: https://doi.org/10.1017/9781316771273.

[2] Thomas Arts et al. "Testing Telecoms Software with Quviq QuickCheck". In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*. ERLANG '06. Portland, Oregon, USA: ACM, 2006, pp. 2–10. ISBN: 1-59593-490-1. DOI: 10.1145/1159789.1159792. URL: http://doi.acm.org/10.1145/1159789.1159792.

[3] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming - ICFP '00*. ACM Press, 2000. DOI: 10.1145/351240.351266. URL: https://doi.org/10.1145/351240.351266.

[4] Koen Claessen et al. "Finding Race Conditions in Erlang with QuickCheck and PULSE". In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ICFP '09. Edinburgh, Scotland: ACM, 2009, pp. 149–160. ISBN: 978-1-60558-332-7. DOI: 10.1145/1596550.1596574. URL: http://doi.acm.org/10.1145/1596550.1596574.

[5] John Hughes. "Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane". In: *A List of Successes That Can Change the World*. Springer International Publishing, 2016, pp. 169–186. DOI: 10.1007/978-3-319-30936-1_9. URL: https://doi.org/10.1007/978-3-319-30936-1_9.

[6] John Hughes. “QuickCheck Testing for Fun and Profit”. In: *Practical Aspects of Declarative Languages*. Springer Berlin Heidelberg, 2006, pp. 1–32. DOI: 10.1007/978-3-540-69611-7_1. URL: https://doi.org/10.1007/978-3-540-69611-7_1.

[7] John Hughes. “Software Testing with QuickCheck”. In: *Central European Functional Programming School*. Springer Berlin Heidelberg, 2010, pp. 183–223. DOI: 10.1007/978-3-642-17685-2_6. URL: https://doi.org/10.1007/978-3-642-17685-2_6.

[8] John Hughes et al. “Find More Bugs with QuickCheck!” In: *Proceedings of the 11th International Workshop on Automation of Software Test*. AST ’16. Austin, Texas: ACM, 2016, pp. 71–77. ISBN: 978-1-4503-4151-6. DOI: 10.1145/2896921.2896928. URL: http://doi.acm.org/10.1145/2896921.2896928.

[9] D.R. Kuhn, D.R. Wallace, and A.M. Gallo. “Software fault interactions and implications for software testing”. In: *IEEE Transactions on Software Engineering* 30.6 (June 2004), pp. 418–421. DOI: 10.1109/tse.2004.24. URL: https://doi.org/10.1109/tse.2004.24.

[10] Johannes Link. *Jqwik*. URL: https://jqwik.net/.

[11] Johannes Link. *Jqwik documentation*. URL: https://jqwik.net/user-guide.html.

[12] *QuickCheck Documentation*. URL: http://hackage.haskell.org/package/QuickCheck-2.6/docs/Test-QuickCheck.html.

[13] *TIOBE Index*. URL: https://www.tiobe.com/tiobe-index/.