Testing Graphical User Interfaces with Property-Based Testing

Master of Science Thesis
University of Turku
Department of Computing
Software Engineering
2022
Tiina Willberg-Laine

Supervisor:
Jaakko Järvi

UNIVERSITY OF TURKU
Department of Computing

TIINA WILLBERG-LAINE: Testing Graphical User Interfaces with Property-Based Testing

Master of Science Thesis, 52 p.
Software Engineering
June 2022

Before a software product is released, it has to be verified that the product works as it should. Graphical User Interfaces (GUI) need to be tested like any other software products. The purpose of testing GUIs is to detect defects but also unexpected behaviour of a GUI.
In 2000 John Hughes and Koen Claessen introduced a new software testing technique: *Property-Based Testing* (PBT). In this testing technique the functionality of the system under the test is defined as properties. Properties are like rules for the features under test. A property defines a relation between input and output that should always hold for all inputs. A property is tested by generating a large number of inputs for which the property is tested.
The goal of this thesis is to explore if PBT is applicable to UI testing. We formulate properties that describe the rules that a GUI should follow, then apply PBT and investigate whether defects could be exposed this way. We also explore whether PBT solves any of the challenges of UI testing, in particular whether test coverage can be increased by using PBT.
As its results, this thesis shows that PBT can be applied in GUI testing and that there are defect classes that might not be detectable by traditional testing methods, but can be found using PBT.

Keywords: Software Testing, Property-Based Testing, Graphical User Interface

Contents

1 Introduction

A user interface (UI) is the way how human users interact and command a computer, website or an application. The simplest UIs are command line interfaces (CLI), where a user writes commands to a command prompter. Modern user interfaces are graphical user interfaces (GUI). That means the UI consists of graphical elements like icons and buttons and communication is performed by interacting with these elements. Through a GUI, anyone can use a computer without having technical knowledge. Compared to CLIs, of course GUIs are much more complex to program and they require more resources too like memory. The focus in this thesis is on GUIs.

Software testing is the process of evaluating, verifying and validating that a software product does what it is supposed to do. The goal is to indetify errors, gaps or missing requirements in contrast to actual requirements. Of course UIs need to be tested like any other software products and the purpose is to detect unexpected behaviour of a UI.

One action in testing process is to consider an adequate test coverage. To increase confidence that a software product works as intended one usually tries to ensure that every piece of code gets executed by some test during the testing process. This is usually laborous and sometimes too ambitious goal in practice. In the context of UI testing obtaining an adequate test coverage is particulally challenging.

Usually test coverage is defined in terms of which portion of code (lines, branches, paths) gets executed by tests. In UI testing we are also interested of coverage in terms of event sequences: do tests execute all possible different sequences of events user may perform. Even in a simple UI the number of possible event sequence is enourmous.

In 2000 John Hughes and Koen Claessen introduced a new software testing technique: *Property-Based Testing* (PBT). [1]. In this testing technique the functionality of the system under the test is defined as properties. Properties are like rules for the features under test. A property defines a relation between input and output that should always hold for all inputs. A property is tested by generating a large number of inputs for which the property is tested.

The goal of this thesis is to explore if PBT is applicable to UI testing. We formulate properties that describe the rules that a GUI should follow, then apply PBT and investigate whether defects could be exposed this way. We also explore whether PBT solve any of the challenges of UI testing, in particular whether the testing coverage can be increased by using PBT, and whether it provides any other advantages to UI testing.

The thesis first gives a literature review that concerns how software products in general and UIs in particular are tested and what are the main challenges of UI testing. The thesis also describes the concept of PBT. Then the thesis describes examples of PBT being applied to testing of UI elements and functionality. Our example cases are implemented with React JS and the tests use technologies and tools which are suitable for testing React and JavaScript.

2 Software Testing

This chapter describes the basic concepts and methodologies of software testing relevant for this thesis. It is intended as an overview about software testing in general to set the context for the thesis.

2.1 The Basic Concepts of Software Testing

The purpose of software testing is to evaluate that a software product does what it is supposed to do. To evaluate the product a definion of the behaviour of the software is needed. This definition is called the *specification*. The Specification defines what the program should do and what users should be able to expect from the program. It describes the properties that the program must have and the input/output behaviour of the program. The focus is on *what* the program does, not on *how* it does it. [2]

A program is said to be *correct* if it behaves as defined in specification. Correctness can be ensured in many ways, for instance using formal verification. Correctness of a program is proved using formal methods of mathematics like propositional logic. Another way is to use *model checking*. A model of a program, like *finite state machine*, is created to model all possible different states of the program. This model is compared with specification and evaluated whether the model meets the specification in all its states. [2]

Attaining a formal proof of a program's correctness is often not feasible with the resources available for typical software projects. More practical way to gain

assurance of the correctness of a program is by executing the program with *test cases*. A test case defines a single test to be executed. A test case usually emulates some action a user might perform, so in this sense test case is a subset of the specification. A test case defines the input, testing procedure and expected results. Often individual test cases are organized in *test suites*. [3]

To determine the result of a test the result has to be defined. That is, it has to be determined when the test has passed and when it has failed. So testing requires a *test oracle*. A test oracle is a tool which defines when a test has passed. A test oracle is based on the specification and it checks the correctness of test output. Usually the oracle is an input/output oracle, which defines the output for a given input. An oracle may not be able to define the output for an arbitrary input. It is often programmed for specific set of inputs. Often the set of inputs that the oracle can handle can be enlarged, it instead of having to know this exact output, the oracle is programmed to know some constraints specified about output. [4]

There are many kinds of test oracles, but usually they are separate from *the system under the test*. A human who determines the expected behaviour is called a *human oracle*. A human oracle can for example consult the program's specification to determine the expected behaviour or calculate the correct output by hand. Using a human oracle is of course expensive, but usually humans have system-specific knowledge that helps to determine the excpected behaviour in even cases that are difficult to formally specify. Oracles can also be automated. If the specification is formal enough, the oracle can be generated automatically. It is also possible to write a separate program, which takes the same input as the system under the test. Also oracles can be generated from previous test results or operations of previous versions of the software. [5]

Even though the concept of test oracle seems simple, it is not. It is not easy to define precisely the expected behaviour of a software system, because the system

usually interacts with the real world and reality is often difficult to capture in a test suite. Also there is no quarantee that the oracle has defined the expected behaviour correctly. The challenge to determine the correct output or the correct behaviour of the system for a given input is called *the oracle problem.* [5] One should also keep in mind when the behaviour of a program is not what is expected it is not always because of a bug. Users might use a piece of software in an unintended way or its behaviour may not be interpreted correctly. Also users and developers may have a different point of view about how the software should function. What users think to be a bug, may not necessarily be a bug from the developer's point of view. [6]

Of course, one test case does not exercise all the program code. *Test Coverage* describes how much of the program code particular test cases cover, that is how much of the code is exercised when running the test case. Test coverage is measured as a percentage and there are several different coverage criteria. A *coverage criterion* specifies the requirements that a test suite needs to satisfy.

A program usually contains many branches. Ideally a test suite should exercise each branch of each control structure. This is called *branch coverage* and it is one often used coverage criterion. If the program contains one conditional statement, exercising all the braches is trivial: It is enough to create one test case where the given if statement's condition is true and another where it is false. But if there are many conditional statements and choosing one option leads to another decision, there can be a large number possible routes through the code. Even if there are only two options to choose every time, n decisions lead to 2^n options.

Another coverage criterion is *path coverage.* It describes how many of all possible paths of control flow are executed. Achieving full path coverage is harder than achieving full branch coverage. Usually excercising all paths is not feasible. Even in a simple program there can be millions of possible paths since loops increase the number of paths rapidly. [7]

Complete testing, that is testing a program with all possible inputs is highly impractical and often impossible. For instance even if the input was as simple as a six-character string that includes only letters, it would probably take days to test all the possible options in and typical computer and simpe test suite. So complete testing is highly impractical and often impossible. This is why considering adequate test coverage and the number of test cases is an important part of test design. [4]

Testing should always be a carefully planned process. A *test plan* is a document specifying the target test coverage and the testing methods and strategies that are used. A test plan defines which features are tested and which are not and the pass/fail criteria. The set of used test cases are also introduced in the test plan. Because tests cannot be executed with all possible inputs, the set of used cases must be chosen carefully. The goal is that the used test cases which are a subset of all possible cases have the highest probability of detecting most errors. That is, a well-selected test case reveals something about the presence of error when using a specific set of inputs and reduces the number of other test case needed to achieve a reasonable coverage.

If test inputs are chosen randomly test coverage is usually low. So some reasoning for choosing what test cases to write is needed. Usually test cases are written by using some kind of error-guessing: Usually experienced programmers and testers have a hunch about what kind of errors are the most probable and what kind of inputs cause errors. For instance, empty inputs usually cause unexpected behaviour. Then based on intuition and experience, developers try to construct good and well-covering test suites [7]

2.2 Software testing methodologies

There are numerous software testing methodologies, they are widely used during the development process, to make sure that software products really meet their requirements. Software testing methodologies can be roughly divided in two categories: *functional* and *non-functional testing*. Functional tests verify that the system behaves as described in its specification, for example, that it produces the expected output for a given intput. Non-functional tests evaluate operational requirements like performance or security. The focus in this thesis is in functional testing: non-functional testing is out of scope.

2.2.1 Levels of testing

Functional tests can be executed at different *levels*. The first level is unit testing. That is, an individual component like one class or method, is tested. Especially if the system under the test is large, it is reasonable to focus testing on smaller building blocks. Then it is easier to locate errors and understand their causes. It is also possible to test multiple modules simultaneuosly, so constructing test suites for smaller building blocks saves time and makes testing more manageable. [7]

Of course it is not enough to test a system in isolated modules; these components must work also when they are combined. When the units are combined, it has to be tested that they work together properly, for instance that data is correctly transferred between components. This is called *integration testing*.

There are different types of strategies to combine modules. When using a bottom-up approach lowest level modules are tested first and then the high-level modules. Finally the integration between the low-level and high-level modules is tested. Top-down approach is of course the opposite: testing starts from the high-level modules. *The Big Bang approach* means that all the modules are combined once. The Big Bang approach has some disadvantage: It can only be used if all the modules are

ready. Further the defects are not easy to locate from large test that involve many modules.

System testing or *end-to-end testing* tests system as a whole. End-to-end tests ensure that an application flow from start to end works as intended from the perspective of the user and the product fulfills business objectives. The focus of end-to-end tests is on how an end user would use the system. One type of system testing is regression testing. That is evaluating that a system still works as it should. after it has been modificated.

Acceptance Testing is the last and highest level of testing. It ensures that a product is ready for delivery.

2.2.2 Mocking

The idea of unit tests is to test just the module under the test, not anything else. The code that is tested usually has dependencies to other components or internal or external services. Dependencies can be replaced with fake versions that imitate the behaviour of the real ones. This is called *mocking*. Mocks can be used to replace API calls, to make some time consuming tasks simpler or to simulate a database by providing fixed responses. Mocks can be created manually or by some library. [8], [9]

Mocks are particularly useful in unit testing where just a particular piece of code like one component or one method is under the test. In such cases it is sometimes necessary to isolate the code under the test. Sometimes erroneous behaviour of a dependency can make the test fail even if the code under the test works correctly so mocking can make tests run more reliably. [10]

The purpose of mocking is to substitute something that is not controllable with something that is. Mocking provides the code we can control, for instance for producing a fake but expected database response, that is suitable for the test case. That

is why mocks are useful also in *negative testing*. Negative testing means testing how the application under test handles errors. Mocks can easily simulate errors from dependencies for instance erroneous responses from a database.

Sometimes mocking may make test suites simplier. If the focus is on testing that some partcular API or method is called, it is not necessary to use a real dependency. A mock is enough to verify the code calls the API or method, because the focus of the test is not on responses. Simirlarly mocks can be used if a test should verify that an API or a method is not called. [11]

Mocking is also useful when dependencies would slow down the whole testing process. For example if the code to be tested is run after authentication, testing may be very slow, if an authentication protocol is completed before each test. A mocked authentication may be a good solution. Mocking dependencies is also useful in that it allows tests to be written even before all dependencies are implemented. [10]

Mocks can also reduce maintenance works of test suites. If there are only a few methods to call, mocking them requires less configuration than setting up an entire real class. On the other hand, if there are a lot of mocks in a test suite, maintaining them may be tedious. If refactoring of the code takes place and mocks are not properly updated, the test suite may allow bugs that should be caught by unit tests.

A common depency to mock is a database connection. Using a real database connection can make testing inconvenient for many reasons. Connection to the database can be slow or there may be no connection at all. If a test writes something to a database, it may be expensive to undo after the test, because there is something to manually delete every time. Also, some states of the database are difficult to create or reproduce for instance some errors. Further, if the tested functionality is some action triggered by a database update and the database is updated rarely, testing would include much waiting.

Mocking is a useful method when performing unit testing, but sometimes it can be applied also when testing integrations. Of course, if the integration under test is between the application and a connection to a server, mocking cannot be utilized. On the other hand, if the purpose of a test case is to test how an application handles erroneous responses from a server, mocking is an easy way to produce them. Further if the integration under the test is the integration between components, mocking may be beneficial. If the test focus is, for instance, on how different parts of a form work together, isolating the form from server integration helps to stay focused on how the form works, not on server errors. When performing end-to-end tests, mocking is not applicable, because the idea of end-to-end testing is to test the system as a whole. [10]

Even if mocking has many advantages, it is important to understand when not to apply mocking. Mocking has to be used in a proper contex and in a proper way. A mock is always a compromise from a real dependency - with misguided mocking a test may exercise an entirely different system than it should. Also misplaced mocking can lead to testing of implementation details. The purpose of unit testing is to test a component's external behavior rather than its internal implementation. Considering the limitations, mocking should always be a conscious choice rather than a tool to be used in all testing.

3 Testing User Interfaces

Modern User Interfaces (UI) are complicated, hierarchical systems. They consist of elements like buttons and scroll bars that enable the user to interact with the system. These elements are called *widgets*. The functioning of UIs is based on *events*, like a user pushing a button and on handling the events. Widgets are associated with event-handlers that define how the system anwers to an event. Usually a user performs multiple actions which create *event sequences*. Even very simple UIs usually generate dozens of possible event types. [12]

The GUI responds to events and event sequences asynchronously. That means the code runs separately and independently. The event code runs only when event is triggered and for instance it does not wait another piece of code to be ready. So UI must be able to handle multiple events and event sequences at the same time. [13]

Programming a modern UI is based on *declarative paradigm*. Traditionally computer programs are based on *imperative paradigm*. That means a program consists of statements that change the program's state. The focus is on how a program operates and a detailed control flow is defined. The statements are commands that a program executes. When using declarative paradigm the focus is on what the program should accomplish, but it is not defined how to accomplish it. The program logic is defined, but not a detailed control flow.

UI has a state, that means what widgets are visible and what operations a user is able to perform at the moment. The behaviour of the event-handlers may depend

on the state. Also the state may change whenever an event occurs and the new resulting state may depend on the preceeding state. This means that differently ordered sequences of the same set of events may lead to different UI states. Even the same sequences of events can lead to different states, e.g., if the timings of the events differ. Even small and simple UIs can have thousands of possible states. Because the behaviour of an event handler depends on the current UI state, it is not enough to test event handlers in one state only. Some errors may occur only in one state. [12]

So testing UI is not trivial and the developer has to construct a large suite of different event sequences to expose defects. Also obtaining an adequate test coverage is laborous, because in the context of UI testing test coverage is not only about catching all execution paths in the code but also test suite exercising all possibile event sequences that might behave differently. Also complex event sequences expose defects more probably than simple ones, but of course constructing and running such tests is more time-consuming. [3], [14]

Two main categories of UI testing are *usability testing* and *functional testing*. Usability testing asses how easy a UI is to use. This type of testing is based on principles from user interface design; in this thesis usability testing is out of scope. Functional testing asseses if the UI works as intented. There are four types of functional testing. *GUI system testing* is testing the system through the UI. *Regression testing* is performed when new features are added to make sure that modifying the UI did not cause any bugs. *Input validation* testing asseses how the software responds different kind of inputs, specially invalid ones. Finally *UI testing* is checking that all the controls works as intented,navigation works as intenteded and the error messages occur when they should. [3]

The UI testing is maybe the most critical part of testing the software, because every action a user performs is done on the UI. The user does not see the source code

or does not have any knowledge of the application. The quality of the UI decides the user's impression of the applications. That is why the focus of UI testing is on what user sees.

3.1 Manual User Interface Testing

The simplest way to test a user interface is just using it, e.g. by clicking the buttons and filling text boxes and then observing if the behaviour is what it should be. This is called *manual testing*. Like testing in general manual UI testing should be a well-planned process. The test cases should be planned beforehand and the expected behaviour of the UI defined.

The error guessing is a technique that can be utilized to choose the test cases when testing UIs and applicable especially when testing manually. Usually testers have an intuition about error-prone operations, for instance user performing actions in unusual order or submitting an empty form, based on their experience. The testers use this intuition to guess the problematic areas of software. The main purpose is to guess possible defects in the areas where formal testing would not work.

In agile development *exploratory testing* is a widely used method to structure the testing process. Exporatory testing is based on the tester exploring the application. The tester learns and understands better the system while testing it. Exploratory testing is not planned accurately beforehand. The tester plans the testing process simultaneuosly while learning more about the application during the testing process. The tester makes observations about the application and plans the next steps based on these observations. [15]

Manual testing is performed by a human so it is error-prone and manual tests are less reliable. It is hard to stay concentraded for hours. Also human can concentrate on one or two verification point only so the scope of the test cases is very limited. Testing actions may be repetitive. Filling out the same form again and again is boring so it understandable that testers have a hard time staying engaged in this process and errors are more likely to occur. Of course, manual testing is highly time-consuming and the coverage is low.

3.2 Automating User Interface tests

Some testing activities can be automated. That means for instance developing and executing test scripts or using automated test tools. Automated test tools are ,for instance, record and play -tools where the tester performs some actions. These actions are recorded and they can be repeated. Record and play tools are useful when performing regression testing. The recorded use of a GUI is executed after the changes have been made and the tool reports differences.

UI events can be generated programmatically, so test scripts can be programmed using some programming language or domain specific language. Record and play tools also create test scripts based on the actions.

When test scripts are created on a way or another the test case is designed first. When designing a test case it has to be considered what is the input and what is the expected behaviour. Then the script is created.

Automating testing activities has many advantages. Automated tests are more efficient, so time and money is saved. Also tests are reusable and repeatable. Once the test is created it can be used over and over again for instance when performing regression testing. Automation also reduces errors. Human errors are common when testing manually. It is hard to keep focus for hours when testing a web site.

Of course creating automated tests for a large amount of test cases is time-consuming too, because it is still manual work. Obtaining a high coverage still requires much work. Also automated tests require some maintenance. When the tested program is updated, the tests have to be also updated and completely new tests will likely be needed.

3.2.1 Generative testing

Tests can be automated by using generators. Testing needs *test data*. That means all the test cases. When testing manually the test data set is very limited, also when using record and play-tools, they exercise a program with very limited amount of cases. Sometimes testing needs a large set of data and using generators that set is created quickly. Larger data set can increase the test coverage. Generators are useful for instance when testing input validation. They create test data and the data is the input. Scripting the input manually is timeconsuming and laborous while generators create large set of input quickly.

The test data is not generated in an ad hoc manner. Of course, there are requirements that the generated data has to satisfy. The program is always tested against a specification and the test data is always considered inside the domain of the specification. When the requirements to the input is defined, the output has to be also defined.

The requirements depend on the goal of the test so there are three strategies for designing the test data set. The data can concentrate on valid test cases, which the program should accept and be able to process. Or the data can include corner cases. Corner cases are data sets which are at the beginning or at the end of the input range, e.g. an empty list or extremely long list. The data can also be invalid, that means data that the program should not accept.

The most effective tests are tests that have the highest coverage. When considering the requirements of the test data, it has to be noticed that the test data should exercise as much code as possible. The data has to be generated so that the coverage is as high as possible.

When testing UI, events and event sequences can also be generated. One way to generate event sequences is Model-Based testing. Model-Based testing is like model checking applied to UI testing. The expected behaviour of system is described as a model. Model is a precise description of the system and it is usually an abstraction of the real world behaviour. The model can be for instance UML chart or decision table. In case of UI testing the event sequences are usually modeled as a graph or finite state machine. The event sequences are generared based on these models and the behaviour is compared to the model. [3]

The challenge of generating events is the same as UI testing challenge in general: the large amount of possible events and possible states. There are several techniques to reduce the number of states. One approach is to imitate novice users. Usually expert users take short, direct paths through the UI. These paths are not useful for testing. Novice users usually take indirect paths that exercise the UI in different ways. [3]

There are many diffenrent kinds of generators and many ways to build them. The simpest examples of generators are random generators. They create test data randomly. For example, to test the operation of a text field random generator can create random strings. The custom generators can also be implemented, but of course building a custom generator takes more time than just using a random generator.

Usually a test data generation system consists of three parts: program analyzer, path selector and test data generator. Program analyzer examines the program code and produces the data needed by path selector and the test data generator.

For instance control flow graph can be used to analyze the code. The Path Selector indentifies the exercised paths from the code using some coverage criterion. Finally the test data generator generates the input that will traverse the paths selected by path selector. Path-Oriented Test data generation selects just one path and generates input for the path. Test data generator can use the goal-oriented approach, where generated data exercises a program point rather than a program path. [16]

The challenge of UI testing is UI's interaction with the real world. Also generated test data must be realistic. If the generated data is not valid and realistic, the generation of test data does not however increase coverage. Logging in to a service with an e-mail address is an example. Generating hundred random strings without the '@'-character does not really increase coverage. [16]

There are two aspects to consider when analysing if the data is realistic or not. First the data has to be structurally valid. For example Finnish postal codes are five digit numbers, so if postal codes are generated they have to be five digit numbers. Then realistic data is also semantically valid. That means it represents a real world entity. Like postal codes; every five digit number is not an existing postal code. E.g. the number 20782 is not a postal code while 20780 is. If postal codes where generated it would not be enough to pick five digit numbers at random, especially if we want to test with valid postal codes only. So designing a generator producing realistic data is not trivial. [17]

Also generated UI event sequences must be realistic to make testing effective. In this case realistic means event sequences the user might most likely perform. Unfortunately generators usually fail to produce event sequences representing realistic user interactions especially in the case of more complex event sequences. Generators tend to spend effort on testing behaviour that is not relevant in practice, like filling data into a form without ever submitting it. Of course effective testing needs also unusual event sequences. Generators may also produce infeasible event sequences.

That means one or more events are disabled by a previously executed event. Of course valuable time is vasted when testing infeasible test sequences. [17], [18]

To summarize, manual testing usually produces realistic test data, but it is laborous. Generators produce data easily, but data is not necessarily realistic.

4 Property-Based Testing

Property Based Testing (PBT) is a software testing technique first introduced by Koen Claessen and John Hughes in 2000. In this chapter first the concept of PBT is introduced followed by insights about PBT tools for JavaScript.

4.1 The concept of Property-Based Testing

In Property-Based testing the system or function under test is defined as properties. *A Property* is a feature or a functionality of the system or an aspect of the function, so it is like a partial specification. Properties define the preconditions of a function, such as the range of input parameters, and postconditions, such as the results of a computation. A Property is always true regardless of the exact input, so the properties are like rules for feature under the test [1] A simple example of a property can be given in the context of a function for sorting a list. In this case we can define a property that the length of the input list is the same as the length of the output list.

Traditional testing methods introduced previously in this thesis can be called *example-based testing*. When using these methods the exact input is defined and the exact output or behaviour is known. The test asserts the actual result against the designated output. When using PBT just some conditions for input is defined for instance the input's data type, but not the exact input. Also for output just conditions are defined not the exact output. For example, when testing a function

for sorting a list, an example-based test would define that when the input is a list 1,2,5,4,3, the output is 1,2,3,4,5. The property for the test can be defined "for every list, the length of the input list is the same as the length of the output list". So the focus is not on specific use cases, but on general properties, that must hold for all input-output pairs. Such properties are a bit more abstract than use cases.

Properties define the expected behaviour in a compact way, so they tend to be easier to understand than a full specification of the whole system. Of course several properties are needed to cover the desired functionality. Properties are expressed mathematical functions or logical laws and usually they are written in a formal way using a domain specific language created to express properties. Using mathematical logic enables defining expected behaviour accurately. Formally expressed properties also make it precise what is tested and when a test fails. Properties are therefore test oracles.

A property is tested by generating a large number of input. For instance the previous example about the function for sorting a list is tested by generating lists, then sorting them and then comparing if the length of the output list is the same as that of the input list. Claessen and Hughes used random generation. They conclude that if properties are well-defined, random generation quarantees an adequate test coverage and there is no need to define custom generators. Many PBT frameworks, however, support defining custom test data generators.

So assuming that the property is always true, the aim is to find a counterexample which proves that the property does not hold. That is why the function is tested by generating a large amount of inputs. The generated input may produce large and complex counterexamples, pinpointing a bug from such counterexamples may be difficult.

PBT frameworks often use *shrinking* to make it easier to understand reasons of defects. When a test fails it means that a property does not hold for some input.

The framework then tries to shrink the failing input to be as simple as possible. The framework can for example remove elements from a list given as input to find a minimal counterexample. This of course helps finding and fixing bugs, because a simple counterexample is usully more understandable for humans. So the first found counterexample is not necessarily used for finding bugs. The framework explores if a simpler counterexample exists and always returns the minimal case. There are different strategies for finding the minimal counterexample, the used strategy depends on the framework. [19], [20]

Even though test input is randomly generated, PBT tests are always reproducable and test runs can be replayed. The testing framework creates a seed for every executed test case. The seed makes possible to replay the test with the same values and reproduce the failing case.

When writing a test it is important to consider if the test really tests the feature that is wanted to be under the test. When writing example-based tests this is usually easy, but the *input scope* is of course low. Input scope means how much of all possible inputs the test case covers. The input scope of course increases with generation of inputs, but especially when generating the examples randomly, random input is not necessarily be testing the feature under the test. PBT combins the benefits. By generating input, we achieve a high input scope and properties still keep the focus on the feature under the test.

4.2 Property-based testing tools for JavaScript

When Hughes and Claessen first introduced the concept of property-based testing, they had developed a property-based testing tool called QuickCheck. [1] QuickCheck was originally developed for testing Haskell programs, but nowadays there are property-based testing tools available for almost every programming language. For instance, for Java is jqwik [21], for Python is Hypothesis [22] and for

C# is FsCheck [23].

JavaScript programmig language is used in this thesis, so this chapter introduces proberty-based testing tools for JavaScript. The first subsection defines the basic concepts of property-based testing tools. They are common for all tools. The second section is a more detailed introduction to a tool called Fast-Check, the tool used in this thesis. Rest of the sections are short overviews of other property-based testing tools for JavaScript.

4.2.1 Basic concepts of property-based testing tools

Property-Based testing frameworks consist of two main building blocks, *runners* and *arbitraries*. Runners are functions that execute the tests and check that properties stay true. A Runner function receives a property to verify and it runs the property several times. Frameworks define by default how many times a property is tested. Typically frameworks run tests 100 times. Of course it is possible to configure the number of times tests are executed.

Arbitraries are generated datatypes. Usually property-based testing frameworks have an abstract class called `Arbitrary` which is able to generate values. Arbitrary defines a default test data generator for each type. Frameworks usually have some built-in arbitratries for primitive types but it is also possible to create custom ones. While testing frameworks usually define by a default for how many inputs will be generated, for many arbitraries it is also possible to set a minimum or maximum number of generated inputs.

Of course testing tools have to define the properties under the test. Properties bind the arbitraries and a test function. The test function is a predicate and it tells what should be observed during the test execution. A predicate should always stay true.

Especially when testing JavaScript programs, it is important that testing framework is able to test promises and other asynchronous methods. All common testing frameworks for JavaScript support asynchronous testing.

According npm trends[1] the most popular tools are JSVerify and Fast-Check.

4.2.2 Fast-check

Fast-Check is a property-based testing tool for JavaScript developed by Nicolas Dubienn since 2017 [24]. The library is written in TypeScript because of TypeScript's static typing. Fast-Check was developed relatively late compared to many other property-based testing tools. Older tools had some known limitations and Fast-check was designed without these limitations. One extension compared to other tools is Fast-Check's verbose mode. That is, in case of the failure, it is possible to get a more detailed log which includes all the counterexamples encountered during shrinking. This sometimes makes debugging easier. Another improvement is that generated values are biased by default. This means that Fast-check generates both small and large values. In addition to these advantages Fast-Check was chosen for this thesis because of a clear documentation and because it is actively maintained.

Fast-Check testing framework has two options for runner functions, `fc.assert()` and `fc.check()`. Both of these functions run a property, but `fc.assert()` doesn't return anything but if the property fails it throws. Function `fc.check()` returns an object containing useful details like test status, seed and of course in case of failure the minimal counterexample. Both of these functions have to be awaited if calling in an asynchronous property.

Properties are defined in the function `fc.property()`. Arguments of the function are an arbitrary and a predicate. Fast-Check enables testing asynchronous properties in which case properties are defined in the function `fc.asyncProperty()`.

[1] https://www.npmtrends.com/fast-check-vs-jsverify-vs-testcheck

Listing 1 The structure of FastCheck test

```
fc.assert( //run the property several times
  fc.property( //define the property: arbitrary and
    // what should be observed (predicate)
    arb1, arb2, ..., // 1 to +infinity arbitraries
    (valueGeneratedByArb1, valueGeneratedByArb2, ...) => {
       // predicate receives generated values
      // In case of success: No return, return undefined
      // or return true
      // In case of failure: Throw or return false
    }
  )
)
}
```

Fast-Check provides some built-in arbitraries. There are of course arbitraries for primitive datatypes like Integer. The function `fc.integer()` generates integers between -2147483648 and 2147483647, but it is also possible to set a range for generated integers. For instance `fc.integer(-20,20)` generates integers between -20 and 20. Fast-check of course offers a generator for strings, `fc.string()`, but also more specific generators like generator for e-mail addresses, `fc.emailAddress()`.

It is possible to create arbitraries by combining built-in arbitraries. Fast-check provides combinators for creating custom arbitraries. One of these combinators is `fc.oneOf()`. It chooses randomly one of the generated values. For instance the function call `fc.oneof(fc.integer(1,5), fc.integer(11,15))` generates a pair of integers. The first of the pairs is an integer between 1 and 5 and the other is an integer between 11 and 15. One of those two integers is chosen randomly. It is also possible to write an arbitrary from scratch. In FastCheck this is done by extending the abstract class `Arbitrary` and implementing a method `generate`.

So as a summary, the structure of test written using Fast-check is shown in Listing 1.

Listing 2 Sorting tested with Fast Check

```
it('should have the same length as input list'), () => {
    fc.assert(
        fc.property( fc.array(fc.integer()), (input) => {
            expect(sort(input).toHaveLength(input.length))
            })
    )
}
```

In the previous chapter testing an algorithm for sorting a list was mentioned as an example of property-based testing. The length of the list was defined as a property, that is the length of the output list should be the same as the length of the input list. This test written using Fast-check is shown in Listing 2.

This test generates arrays of integers, then sorts them and compares the length of sorting results to the length of the input array.

4.2.3 JSVerify

JSVerify is a property-based testing tool for JavaScript developed by Oleg Grenrus since 2014 [25]. Like in Fast-Check there are two options for runner functions in JSVerify, `jsc.check` and `jsc.assert()`. Both functions get the property and options as an argument. Options include for instance a number for maximum size of the generated values, which is 50 by default.

The difference of the runner functions is the same as in Fast-Check. The function `jsc.check()` returns true if the test is passed and in case of failure it returns an object with details of the shrinking operation, like number of shrinks performed, and of course the minimal counterexample. The function `jsc.assert()` doesn't return anything in case of success, but it throws an exception if the property doesn't hold.

The property constructor of JSVerify is the function `jsc.forall()`. As an argument it expects an arbitrary and a property function. JSVerify enables also testing promise based properties so property function can return a promise.

Listing 3 Sorting tested with JSVerify

```
it('should have the same length as input list'), () => {
    jsc.assert(
        jsc.forall( jsc.array(jsc.integer()), (input) => {
            expect(sort(input).toHaveLength(input.length))
            })
    )
}
```

JSVerify has built-in arbitratries for primitive data types such as `jsc.integer`. There are also arbitraries for strings. Custom arbitratries are built with combinators like `jsc.oneof` or restricting arbiratries with `suchthat()`.

The test for sorting lists written using JSVerify, is shown in Listing 3.

4.2.4 TestCheck

TestCheck is a property-based testing tool for JavaScript developed by Lee Byron since 2014 [26]. It is based on Clojure's test.check. [27] It seems that TestCheck is not mantained anymore. The last update is from 2018, so it is not used in this thesis.

TestCheck includes only one runner function `check()`. This function expects a property created via the `property()` function and an optional Object. The object includes properties like the seed for rerunning the test. The function returns the results. The properties like boolean value `result` and in case of fail, `shrunk` , which is an object including the smallest argument with failing the test.

The function `property()` creates a property. It receives the value generators that generate the arguments of the predicate and of course the predicate itself. Generating values is not implemented like in Fast-Check or JSVerify. TestCheck doesn't include a class like `Arbitrary`. Values are generated with function `gen()`. The function is called a generator builder.

Listing 4 Sorting tested with TestCheck

```
check.it('should have the same length as input list',
gen.array(gen.int), (input) => {
    expect(sort(input).toHaveLength(input.length))
    })
```

Like other tools, TestChek has some built-in generators, for instance, `gen.int()` generates integers. Custom generators can be built by combining generators like `gen.array(gen.int)`, because TestCheck includes combinators like `gen.oneOf`. Custom generators can be defined by using `ValueGenerator` instances, like `ValueGenerator#suchThat()`. For instance a generator for even numbers is created like this: `var genEvenNumbers = gen.int.suchThat(n => n % 2 === 0)`

By default TestCheck generates 200 values. The function `check()` has an optional object with property `maxSize` that defines the number of generated values. TestCheck generators have one limitation. They begin by generating small test cases. If there is a need to test input including very large numbers or extremely long arrays, running test with default options isn't enough.

In TestCheck asynchronous code is tested by using the runner function `checkAsync()` instead of `check()`. When using `checkAsync()` the predicate function needs to return a promise.

As an example the test for sorting list is implemented using TestCheck. The test code is shown in listing 4

5 Background: Technology stack

This chapter introduces the tools and technologies used in this thesis. When testing applications it is common to use a combinations of several frameworks to achieve the most flexible set functionality.

5.1 React JS and testing React applications

The UIs under the test in this thesis are React applications, because perhaps it is the most popular and also the most modern way to build UIs at the moment.

React is a Javascript library for building user interfaces. React applications consists of encapsulated components. The idea is to build encapsulated components and then compose them to make complex UIs. Each component manages its own state. The components are reusable and combining them in many different ways builds different UIs. A component can include other components. Included components are called children and a component, that includes other components is called a parent. A parent component, the children of the parent component, ant the children of the child components and so on, are called a component tree.

Like modern UIs, usually React applications are built using the declarative paradigm. The data is stored using state of the application. The data is transformed from a component to another using props. React code uses JSX which stands for JavaScript XML. At first glance it looks like HTML, but JSX is just a another way to write JavaScript code. The JSX code is compiled to vanilla JavaScript using

Listing 5 A Promise chain

```
promise
  .then(value => {
    // use value...
  }, error => {
    // check error...
  })
```

Babel [28]. Babel is a JavaScript compiler that is used to convert a source code into versions of JavaScript that can be run by older JavaScript engines. This is how new JavaScript features can be used right away even if web browsers do not have support for them. The reason why React components are programmed with JSX is that it is easier to write and understand than vanilla JavaScript.

The HTLM elements present on a webpage or web application are represented in the Document Object Model (DOM). It is a tree data structure that contains a node for each UI element. The content can be modified through JavaScript. [29]

JavaScript code in React applications runs asynchronously. This is a matter of a user experience. It would be highly impractical if the whole application would freeze to wait some operation to be completed. Also fetching data to a UI usually happens asynchronously. Otherwise loading the application would freeze while data is being fetched. When using asynchronous programming, loading the application can be done at the same time as the data is fetched.

React applications use promises for fetching data. A promise is like a placeholder for some value which is not yet available. Promises have three possible states. Pending is the initial state of the promise. The task that will eventually produce a value is still waiting to be completed. When the state is fullfilled or resolved the task is completed successfully and if not, the state is rejected. To handle the fullfill value of the promise, the method `.then()` is used and the method `.catch()` to access the rejection error. Usually these two methods are chained to handle all possible states of the promise. The structure of a promise chain is shown in Listing 5. [30], [31]

Listing 6 Async-await promise

```
async function myFunction() {
  // ...
  try {
    const value = await promise;
  } catch (error) {
    // check error
    error;
  }
}
```

Using keywords `async` and `await` makes working with promises more comfortable. It enables to handling multiple promises as if the code was asynchronous. When the keyword `async` is used before a function it means that the function always returns a promise. If a non-Promise object is returned it is automatically wrapped in a resolved promise. The keyword `await` works only inside `async` functions. It makes the code wait until the promise is fullfilled, but only in that function's context. The keyword `await` won't work at the top-level. To handle the possible errors, try-catch syntax is used. The structure of async-await syntax is shown in Listing 6.

A common way for fetching data in React applications is to use Axios. It is a JavaScript library for HTTP requests and it is based on promises. Axios is widely used because, it is easy to use and it allows for instance cancelling requests and request timeout. It also has protection against most common attacks. [32]

The testing of React applications can be separated into two levels. Testing one component or component tree and testing a complete app. Usually tests for a component or component tree run in simplified testing environment and usually the component is isolated from other components, in other words dependencies are mocked. So the nature of component testing is like unit testing, but the difference between unit testing and integration testing is not that clear when talking about React components. Some components can be tested only as part of a bigger process. So it is worth considering what depencies are mocked. [33]

When using a simplified testing environment, tests run quicker, because there is no waiting for a browser to start up for each test. So the feedback loop and iteration is fast. Of course a simplified environment is an approximation of the behaviour of the browser, but it is often still good enough for testing React components.

There are different techniques to test components. The simplest test is a smoke test that sees that a component renders correctly. When using shallow rendering, the individual component is rendered, not the child components. Of course the component can be full-rendered and the lifecycle and state changes can be tested. [34]

The tests of a complete app usually run in a realistic browser environment and they are called end-to-end tests. End-to-end tests are used for testing longer workflows like signing up to a service. Also they are useful when fetching data from a real API or the side effects in the backend are under test. While end-to-end tests show how React components behave in a real browser, the iteration speed in such tests is much slower. Also, end-to-end tests can be flaky. That means the test sometimes passes and sometimes fails without any code changes. This is a challenge for pinpointing errors because it is unclear if the error is in the code or in the test. [35]

5.2 Jest

Jest is a JavaScript testing framework developed and maintained by Facebook. It is compatible with Babel-based projects like NodeJs, React, Angular ad VueJS. Jest is used in this thesis because it is the most popular and recommended way to test React applications. The documentation is clear and using Jest requires little configuration.

Jest is a test runner. It accesses the DOM via jsdom, which is a browser implementation running inside Node.js. Usually jsdom is good enough to simulate a real browser even if some features are missing, like layout and navigation. It models user

interactions and the side effects of these actions can be observed and asserted. Jest provides good control over the code under the test. Jest has support for mocking modules and timers. [33], [36]

Testing frameworks for testing React application has to support asynchronous testing and Jest does. Jest only needs to know when it can continue to another test, i.e, when the code under the test has completed. When dealing with promises, this is easy. A promise can be returned from a test and Jest waits for the promise to resolve. A test will fail automatically, if the promise is rejected. Another way is to use `.resolves` matcher in the expected statement. Asynchronous code can be also tested with keywords `async` and `await`. [37]

5.3 React Testing Library

React Testing library is created by Kent C. Dodds. [38] It is a library for testing React components, and it encourages better testing practices. Tests have to be maintanable. That means if the implementation of the component is changed, not the functionality, this refactoring does not break the tests. The best practise in testing guide not to test implementation details.

The guiding principle of React Testing library is: "The more your tests resemble the way you software is used, the more confidence they can give you", so it imitates the way the components are used by end users. Of course some comporomises are done because a simulated browser enviroment is used, not a real one. The tests work with actual DOM nodes, not the instances of rendered React components. [38]

5.4 Puppeteer

Puppeteer [39] is Node Library which is used when performing end-to-end tests to React applications. Puppeteer is a browser controller. That is it simulates user

actions like clicking and typing. Puppeteer uses a headless browser, a web browser without a graphical user interface. Headless browser is still able to understand HTML the same way a normal browser would. This includes also styling elements like layout. A headless browser also provides a automated control of a web application. Because of using a headless browser end-to-end tests are fast to execute when comparing with normal browser.

Puppeteer is used in this thesis because it is easy to configure and tests are easy to execute. It also integrates nicely with Jest.

Fundamentally Puppeteer is an automation tool. It does not have a rec and play -feature and the tests and user actions have to be scripted. [39]

6 Catching defects with generated event sequences and Property-Based testing

This chapter introduces four example UI tests. The main idea is to test UI components with generated event sequences and property-based testing. The goal is to investigate whether Property-Based testing is applicable to UI tests and what advantages using PBT provides comparing to other UI testing methods. Of course, the goal is to see if it is possible to catch defects that other testing methods would not catch, or to catch defects with less effort in writing tests.

6.1 Throttled search field

Modern search fields have an autocomplete functionality. That means that when a user starts to type something into the search field, the application suggests search terms with a matching prefix. The suggested terms depend on what is being searched. A search field might suggest terms in alphabetical order. For instance if there is a field where you search countries, usually the suggestions are in alphabetical order. In some GUIs the field suggest the terms that are the most popular search terms; this is the behavior in Google search, for example.

Sometimes using autocompletion is a challenge when considering the performance of the application. If the searched data is fetched from the server and there is large amount of data to fetch, autocomplete causes a heavy load. Every time a user types a letter, new data must be fetched from the server and this of course slows down the application. A common solution is to use throttling. Throttling is a technique to limit the number of events the system needs to handle. Throttling delays the execution of a handler function, so that more events will be collected and then processed in one go.

In this chapter an autocomplete throttled search field is under the test. A simple search field is implemented with React JS and it is used as an example of autocomplete throttled search field. The field allows for searching cities and towns. The database includes all the cities and towns in the world whose population is over 1000. Every city has also a number that tells how many times the city has been searched. The autocomplete search field suggests the most popular cities.

Search results are fetched from a database by using React's effect hook. Without throttling the hook is executed every time the search term changes, that is, every time the user types a letter. When the search term is throttled, the hook is executed at most once every second. If the user types three letters during that second, those letters are the next search term. The code of the effect hook is shown in Listing 7.

When autocomplete and throttling are combined carelessly, the search functionality may not behave as intended. When a user types the first letters of the search term in the search field, then takes a little break and then rapidly types the last letters and presses enter, a wrong result may be shown. This happens because of throttling. During the break, the search term and the suggestions are updated, but because the user types the last letters faster than the throttling time, the search term is not updated because throttling prevents it. Autocomplete chooses the first of the suggestions and when enter is pressed, the first suggestions is chosen for the

Listing 7 Effect hook for fetching results.

```
//effect hook for fetching results
    useEffect(() => {
        //preventing the app from setting state after the component
        //has unmounted
        let isSubscribed = true
        //variable deciding if "no results"-text is shown,
        //setting it false
        setNoResults(false)
            //promise for fetching data
            axios
            .get('http://localhost:3001/cities/' + throttledSearchTerm)
            .then(response => {
                if (isSubscribed) {
                //save response data in a variable
                setCities(response.data)
            }
        })
        return () => isSubscribed = false
    },[ throttledSearchTerm])
```

search term. So the chosen search term is not what the user has typed into the search field. This is why the search is done with the wrong term.

When the search field is tested useing a traditional, example-based unit test or component this defect will not be detected. The test takes just one city as an input, searches with it, finds the right result and the test is passed. Coverage can be increased by generating inputs with FastCheck. There the interesting cases are those where a search result is found so a generated number is an index in the JSON-table of cities. Even in these cases, where the cities are generated the tests still pass because the error is not in the code that is being tested. The code works just like it should work. Instead, the cause of the unintended behaviour is on how the user behaves. When testing UIs it is important to use the UI the same way real users would use it. This is why Puppeteer tool was used in this test. In this case the application behaves incorrectly because humans do not always type at the same pace. To imitate this behaviour, a little break is added in the middle of typing the

Listing 8 The test code

```
it('should search data about a given city', async () => {
  await page.waitForFunction('document.getElementById("searchForm")')

  await fc.assert(
    fc.asyncProperty( fc.integer({
      min: 0,
      max: 137520
    }) , fc.integer({min: 500, max: 1500}) ,
    async (city, delay) => {
      const testcity=testCities[city].Name
      const split=Math.round(0.66*testcity.length)
      const firstpart=testcity.substring(0,split)
      const secondpart=testcity.substring(split)
      await page.type("input[name='search']", firstpart)
      await page.waitForTimeout(delay)
      .then(() => console.log('Waited a moment!'))
      await page.type("input[name='search']", secondpart)
      await page.click("button[type='submit']")
      await page.click("button[type='button']")
      expect(await page.$eval('#result', element => element.innerText))
        .toEqual(testCities[city].Name)
    }), {verbose:true})
})
```

search term.

When applying Property-Based testing the property has to be formulated. Here the input is those city names for which data is found; the behavior under the test is not how the search field behaves if the input is something that does not find anything from the database. Of course, the expected behaviour is that the information of the typed city is shown. So the property is: for all cities found from the database, application shows information from the city that the user types.

The test that combines the generation of inputs and imitation of a real user, is shown in Listing 8. In the test, a little break is generated before the last letters are typed.

As seen from the test result in Listing 9, the test fails. Fast-Check has used

Listing 9 Test result

```
Property failed after 15 tests
  { seed: 249612799,
  path: "14:12:13:3:3:1:0:15:14:9:4:9:14:10:6:1:11:
  14:14:14:6:1:1:0:9:1:11:11:11:11",
  endOnFailure: true }
  Counterexample: [774711,501]
  Shrunk 29 time(s)
  Got error: Error: expect(received).toEqual(expected) // deep equality

  Expected: "Bystra"
  Received: "Bystrom"

  Stack trace: Error: expect(received).toEqual(expected) // deep equality

  Expected: "Bystra"
  Received: "Bystrom"
```

shrinking to find a counterexample. In this case the counterexample is that the user tries to find information about a city called Bystra. The user types "Bystr", then takes a little break. Autocomplete suggests a search term "Bystrom" because it has been searched more times than Bystra. When the user types the last letter and presses enter quicky, the search term is not updated because of throttling. Throttling delays the update. That is why the results from Bystrom are shown and the application does not behave as was expected.

Next, more generation is added to the test. In addition to generating the duration of the break, the point of the break is also generated. The test is shown in Listing 10. This test also fails like shown in Listing 11. The reasons for this test failing are the same as in the previous test. This test is even more imitating the real user, because the break it is taking occurs at a random point.

Also when the searched city name is very short and the user types it really fast, the search term is not updated. Throttling prevents it and and again wrong result is shown. A test which reveals this defect is shown in Listing 12. To make the test's typing simulation as realistic as possible, we have added a delay after every press.

Listing 10 The Test Code

```
  it('should search a given place', async () => {
  await page.waitForFunction('document.getElementById("searchForm")')

  await fc.assert(
    fc.asyncProperty( fc.integer({
      min: 0,
      max: 137520
    }) , fc.integer({min: 500, max: 1500}) ,
    fc.integer({min: 1, max: 30}) ,
    async (city, delay, burst) => {
      const testcity=testCities[city].Name
      const split = burst%testcity.length
      const firstpart=testcity.substring(0,split)
      const secondpart=testcity.substring(split)
      await page.type("input[name='search']", firstpart)
      await page.waitForTimeout(delay)
      .then(() => console.log('Waited a moment!'))
      await page.type("input[name='search']", secondpart)
      await page.click("button[type='submit']")
      await page.click("button[type='button']")
      expect(await page.$eval('#result', element => element.innerText)).
      toEqual(testCities[city].Name)
    }), {verbose:true})
})
```

Listing 11 Test result

```
    Property failed after 4 tests
      { seed: -623420858, path: "3:1:7:4:4:5:14", endOnFailure: true }
      Counterexample: [28198,500,30]
      Shrunk 6 time(s)
      Got error: Error: expect(received).toEqual(expected) // deep equality

      Expected: "Nanjin"
      Received: "Nanjie"

      Stack trace: Error: expect(received).toEqual(expected) // deep equality

      Expected: "Nanjin"
      Received: "Nanjie"
```

Listing 12 The Test Code

```
    it('should search data about a given city', async () => {
  await page.waitForFunction('document.getElementById("searchForm")')

  await fc.assert(
    fc.asyncProperty( fc.integer({
      min: 0,
      max: 137520
    }), fc.array(fc.integer({
      min: 100,
      max: 1000
    }), {
      minLength: 50,
      maxLength: 100
    }) , async (city, delays) => {
      const testcity=testCities[city].Name
      for (let i ; i < testcity.length ; i++){
        const char = testcity.chartAt(i)
        await page.type("input[name='search']", char)
        const delay = delays[i]
        await page.waitForTimeout(delay)
        .then(() => console.log('Waited a moment!'))
      }
      await page.click("button[type='submit']")
      await page.click("button[type='button']")
      expect(await page.$eval('#result', element => element.innerText))
      .toEqual(testCities[city].Name)
    }), {verbose:true})
})
```

The delay is again generated, but now the generated values are less than the delay of throttling to imitate really fast typing.

As seen from the test result in Listing 13, the test fails. Fast-Check has used shrinking to find a counterexample. In this case the counterexample is that the user tries to find information about a city called Mirny. The user types "Mirny" really fast, but the search term is not updated because of throttling. The search term is now "Mir" because throttling delays the update. Only a part of the search term is updated and the last letters are not included in the search term. That is why

Listing 13 Test result

```
Property failed after 6 tests
  { seed: -409630996, path: "5:16:16", endOnFailure: true }
  Counterexample: [58484,[20,105,156,396,260,134,179,329,124,
  157,94,209,342,390,102,211,447,134,262,205,425,33,359,375,302,
  320,247,144,120,369,132,116,285,420,219,431,295,428,443,362,394,
  300,308,326,417,423,56,85,374,292]]
Shrunk 2 time(s)
Got error: Error: expect(received).toEqual(expected) // deep equality

Expected: "Mirny"
Received: "Miramar"

Stack trace: Error: expect(received).toEqual(expected) // deep equality

Expected: "Mirny"
Received: "Miramar"
```

the results from "Miramar" are shown and the application does not behave as was expected.

So to reveal this errourneous behaviour, we had to use Property-Based testing and feed the generated inputs to test in a way that emulates realistic use of application. Trying the search with just one city would likely not reveal the error, especially if the city is the most searched city. By generating a large number of search terms the probability to find suitable terms increases.

6.2 Fetching data to UI using Promises

The asynchronous behaviour of promises causes sometimes unexpected behaviour. The exectution of code continues even if there is a promise pending. The code can create a new promise before a previous promise is fullfilled or rejected. Because of asynchronity the second promise can be resolved before the first one and this may cause unexpected behaviour. The next two example tests are designed to reveal

problems in these kind of situations.

Property-based testing can be used to reveal the behaviour described. Fast-check provides an interface called `Scheduler`, which is a built-in asynchronous scheduler. The purpose of the interface is to reschedule promises. The interface provides three scheduling methods, `schedule`, `scheduleFunction` and `scheduleSequence`. The method `schedule` creates wrapped promises. The lifecycle of the promise stays the same, but the promise resolves only when the scheduler decides so. Before scheduling another promise the scheduler waits the first promise to resolve. The method can be used to test functions whose input is a raw promise. The method `scheduleFunction` creates a producer of scheduled promises. If the function makes asynchronous API calls, this method can reorder the order of promises. For instance function `fetch` is this kind of function. If there is sequence of asynchronous calls that must be run in a precise order, the method `scheduleSequence` can be used. Like the next two examples show, the scheduler is applicable to both mocked tests and to tests that use a real database. [40]

6.2.1 Mocked test: search field

The first example is a search field, where a user types the name of the city in the search field and pushes the search button. Then the data from the city is fetched from the database. In this case the erroneous behaviour of promises appears when user types the name of city A, then pushes a button and promise A is pending. Then the user types the name of the city B, then pushes the button and promise B is pending. If promise B is fulfilled first, the data from city B is shown. When promise A is fullfilled the data from the city A is shown, even if the user wanted to see data from city B. This sequence of events is described as a list below just to be clear.

1. User types the name of city A and pushes the search button

2. Promise A is created and is pending
3. User types the name of city B and pushes the search button
4. Promise B is created and is pending
5. Promise B is fullfilled
6. Data from city B is shown
7. Promise A is fullfilled
8. Data from city A is shown

To apply Property-Based testing we again need to formulate a property. Again, the set of interesting inputs is now the cities for which the data is stored. The interesting case is not how the search field behaves if the input is something that does not find anything from the database. Of course, the expected behaviour is that information about the typed city is shown. So the property is much like in the previous example: for all cities found from the database, the application shows information from the city the user types.

In this example test Axios is mocked using Jest. Jest provides mocking out of the box and mocking an API call is easy. First the mocked module, in this case Axios, is imported into the test file. To mock a module Jest uses `jest.mock`. The necessary imports are shown in Listing 14. [37]

Listing 14 Mocking Axios

```
import axios from 'axios'

jest.mock('axios')
```

Later in the test code the method `.mockResolvedValue()` is used to mock the response. Responses are generated from a JSON file. The file includes 137 521 cities, so an integer in the range from 0 to 137 520 is generated. The integer is used as an index of the city to be fetched. The `Search` component is rendered using this mocked response. The promises are reordered using `scheduler`. The test code is shown in Listing 15.

When the test is executed, it fails. The result is shown in Listing 16. In this case the scheduler delays the first promise and the second is fulfilled first.

Listing 15 Mocked search test

```
it('should search data about the city user types', async () => {
  await fc.assert(
    fc.asyncProperty(fc.scheduler(), fc.integer(0,137520),
    fc.integer(0, 137520), async (scheduler, city1, city2) => {

      const resp =[testCities[city1]]
      const resp2 =[testCities[city2]]

      axios.get.mockResolvedValue({ data: resp })
      const {rerender} = render(<Search />)

      scheduler.scheduleSequence([ async () => {
        axios.get.mockResolvedValue({ data: resp2 })
        rerender(<Search />) },])

      await scheduler.waitAll()

      const result= await waitFor(() => screen.getByTestId("result"));

      expect(result).toHaveTextContent(resp2[0].Name)

  }).beforeEach(async () => {
    jest.resetAllMocks();
    await cleanup();
  })
  )
  })
  })
```

Listing 16 Mocked search test

```
Property failed after 1 tests
   { seed: 1578744751, path: "0:0:1:0:0:0:0:0:0:0:0:0:0:0:0:0:0",
   endOnFailure: true }
   Counterexample: [schedulerFor()`
   -> [task${1}] sequence resolved`,0,1]
   Shrunk 17 time(s)
   Got error: Error: expect(element).toHaveTextContent()

   Expected element to have text content:
     Namtsy
   Received:
     Result: Priargunsk
```

6.2.2 React Application for showing RSS-feed

The second example is about testing an application that shows news titles from RSS feeds. The code of the application is introduced in the article by Madsen, Lhoták and Tip [41]. The application sends a new promise every five seconds to update feeds shown by the application. In this application the errorneous behaviour of promises can be noticed when the feed is updated between two promises. The application creates a promise and after five seconds a new one. If the second promise is fullfilled first and there is a feed update between these promises, a new title shows up to the application. But when the first promise is fullfilled the title disappears because the response of the first promise does not include the latest title. Of course the title finally shows up when the next promise is fullfilled, but it can be very confusing to user when the title shows up, disappears and shows up again. The behaviour is also described as a list below just to be clear.

1. Application creates promise A for updating
2. Application sends a request A to RSS feed
3. Five seconds passes

4. Application creates promise B for updating
5. Application sends a request A to RSS feed
6. RSS feed sends a response for request A
7. The feed is updated and new titles are added
8. RSS feed sends a response for request B
9. Promise B is fullfilled
10. User sees new titles in the application
11. Promise A is fullflled
12. New titles disappear

What the user sees depends on how often the feed is updated. If the feed is updated regurlarly and more often than promises are created, reordering of promises can also cause two titles appearing at the same time and then another one disappearing. If the second promise is fullfilled first, the feed has updated twice before the promise is resolved. That is why two titles appears. When the the first sent promise resolves, another disappears. Of course, the next resolved promise includes the title, but it still can be very confusing to the user.

Again property-based testing and Fast-Check's interface `scheduler` is utilized to reorder the promises. Because PBT is utilized the property has to be formulated. Now the observed behaviour is when the feed is updated. The new titles show up always at the top of the list of titles. So the updated list includes the same titles in the same order than before the update. The only change is the new titles at the top. So the property is: for every updated list, the list includes the titles from the previous list in the same order.

Listing 17 Test for RSS feed application

```
it('should keep the rest of the list ', async () => {
   await fc.assert(
      fc.asyncProperty(fc.scheduler() , async (scheduler) => {
        let feed
        let expectedresult
        let result
        let slicedresult

        scheduler.scheduleSequence([ async () => {
        feed= await parser.parseURL('http://localhost:4050/examplefeed')
        expectedresult = feed.items.map(item => item.title)
        await new Promise(res => setTimeout(res, 5000))} ,
        async ()=>  {
        feed = await parser.parseURL('http://localhost:4050/examplefeed')
        result=feed.items.map(item => item.title)
        slicedresult= result.slice(1)
        await new Promise(res => setTimeout(res,5000))
        }, ])

        await scheduler.waitAll()

        expect(slicedresult).toEqual(expectedresult)

      }),)
    })
  })
```

For testing purposes an ad-hoc solution for simulating the feed is impelemented. It is an example feed which is updated every four seconds. In this test mocking is not used, because this example is about to demonstrate that we can use `scheduler` interface also when executing end-to-end tests and because there are many other solutions than to mock, e.g., to build an example feed which is updated more often than a real one.

In the test code the async-await syntax is used. This syntax can be used also in tests just as in the program code. The test code is shown in Listing 17.

The test fails because the second promise is resolved first. The feed has updated

Listing 18 Test result

```
Property failed after 8 tests
    { seed: -364263465, path: "7", endOnFailure: true }
    Counterexample: [schedulerFor()`
    -> [task${1}] sequence resolved
    -> [task${2}] sequence resolved`]
    Shrunk 0 time(s)
    Got error: Error: expect(received).toEqual(expected)
    - Expected  - 0
    + Received  + 1

    @@ -1,6 +1,7 @@
      Array [
    +   "Title 7253",
        "Title 7247",
        "Title 7243",
        "Title 7237",
        "Title 7229",
        "Title 7219",
```

twice after the application has sent a request and that is why two titles are added instead of just one. Of course executing the test stops, but if it will continue, the first title would disappear. The test result is shown in Listing 18.

7 Conclusions

In this thesis we investigated Property-Based Testing for testing user interfaces. Some simple tests were implemented using PBT and some bugs were found using these tests. Based on these tests Property-Based testing is applicable also in UI tests. But it is not enough that some testing technique is applicable. There has to be some advantages comparing to other techniques. One goal of this thesis was also to study what advantages PBT provides. In this thesis was also introduced some challenges of UI testing and discussed if PBT can solve these challenges.

One challenge in software testing is the so called oracle problem. The oracle problem is the challenge to determine wheter a test is passed or failed, i.e. what is the correct behaviour of the system for given input. Properties define the expected behaviour accurately. They also make it precise what is tested and when the test fails, so properties are test oracles and they make it easier to define the correct behaviour. On the other hand, defining properties is not trivial and it is also a challenge to formulate invariants that should always stay the same and yet be likely to be broken in case of defects.

Another challenge of UI testing is to obtain an adequate test coverage. The behaviour of an UI is dependent on the state of the UI. Even a simple UI can have thousands of possible states. Events and event sequences change the state and the state is usually dependent on former events. So testing the behaviour in every possible state and testing the effect of every possible event is generally impossible.

Consequently, obtaining an adequate test coverage is difficult. Further, if an UI has some input fields that accept complex data, such as text testing for different inputs is also a daunting task.

Generating input data is one way to increase coverage. By generating a large amount of test data, tests can be created quickly, but using generators in UI testing may have challenges. UIs interact with the real world and with humans, so generating a realistic test data is not easy. Generators must be specifically programmed to emulate humans' data and event production.

Hughes and Claessen state [1] that well-defined properties quarantee an adequate test coverage, so random generators are suitable for generating test data and custom test data generators are not needed. In example tests introduced in this thesis we were able to find a defect by using random generation of events. The test for throttled search field exposed a defect and the test generated realistic event sequence to uncover a bug. The breaks added to some point of typing imitated a real user much better than typing imitated by Puppeteer. This defect would not have necessarily been found by using example-based tests. Of course it would be interenting to test more complex UIs and to generate more complex event sequences to explore if we find defects.

In example tests of this thesis a PBT testing tool called FastCheck was used. Some observations about the tool were also made. The tool provides an interface called Scheduler and it is designed to reschedule the ordering of promises. This interface is very useful for testing promises and how the UI behaves if the ordering of promises is rescheduled.

When implementing and executing the example tests, some limitations of the tool FastCheck were noticed. Especially when testing the city search for promise errors, the generator started many times with the same integers 0 and 1. Also we noticed that the generator usually produced small test cases, e.g. small integers. To

really increase the coverage, large numbers or long arrays are also needed.

The test cases we used in our examples were rather simple, but they nevertheless demonstared that PBT can be applied in GUI testing and that there are defect classes that might not be detectable by example-based testing, but can be found using PBT.

References

[1] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of Haskell programs", *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, vol. 46, Jan. 2000. DOI: 10.1145/1988042.1988046.

[2] A. Mili, *Software testing : concepts and operations.* Hoboken, New Jersey: John Wiley and Sons, Inc., ISBN: 1-119-06559-3.

[3] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. USA: Cambridge University Press, 2008, ISBN: 0521880386.

[4] B. Beizer, *Software Testing Techniques (2nd Ed.)* USA: Van Nostrand Reinhold Co., 1990, ISBN: 0442206720.

[5] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey", *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015. DOI: 10.1109/TSE.2014.2372785.

[6] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction", in *35th International Conference on Software Engineering (ICSE)*, IEEE Press, 2013, pp. 392–401, ISBN: 1467330760.

[7] G. J. Myers, *The art of software testing*, 3rd ed. Hoboken, N.J: John Wiley and Sons, ISBN: 9781118031964.

[8] M. Fowler, *Testdouble*. [Online]. Available: https://martinfowler.com/bliki/TestDouble.html (visited on 09/26/2021).

[9] M. Fowler, *Mocks aren't stubs*. [Online]. Available: https://www.martinfowler.com/articles/mocksArentStubs.html (visited on 09/26/2021).

[10] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "To mock or not to mock? an empirical study on mocking practices", in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 402–412. DOI: 10.1109/MSR.2017.61.

[11] R. Osherove, *The Art of Unit Testing: With Examples in .Net*, 1st. USA: Manning Publications Co., 2009, ISBN: 1933988274.

[12] Q. Xie and A. M. Memon, "Using a pilot study to derive a gui model for automated testing", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 18, no. 2, pp. 1–35, 2008.

[13] R. F. Jorge, M. Delamaro, C. G. Camilo-Junior, and A. M. Vincenzi, "Test data generation based on gui: A systematic mapping", in *Proceedings of the 9th International Conference on Software Engineering Advances (ICSEA)*, 2014, pp. 240–246.

[14] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for GUI testing", in *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, New York, NY, USA: ACM Press, 2001, pp. 256–267, ISBN: 1-58113-390-1.

[15] A. N. Ghazi, K. Petersen, E. Bjarnason, and P. Runeson, "Levels of exploration in exploratory testing: From freestyle to fully scripted", *IEEE Access*, vol. 6, pp. 26 416–26 423, 2018. DOI: 10.1109/ACCESS.2018.2834957.

[16] J. Edvardsson, "A survey on automatic test data generation", in *Proceedings of the Second Conference onComputer Science and Engineering in Linköping*, ECSEL, 1999, pp. 21–28.

[17] M. Bozkurt and M. Harman, "Automatically generating realistic test input from web services", in *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*, 2011, pp. 13–24. DOI: 10.1109/SOSE.2011.6139088.

[18] M. Ermuth and M. Pradel, "Monkey see, monkey do: Effective generation of GUI tests with inferred macro events", in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, Saarbrücken, Germany: Association for Computing Machinery, 2016, pp. 82–93, ISBN: 9781450343909. DOI: 10.1145/2931037.2931053. [Online]. Available: https://doi.org/10.1145/2931037.2931053.

[19] J. Hughes, "How to specify it!", in *Trends in Functional Programming*, W. J. Bowman and R. Garcia, Eds., Cham: Springer International Publishing, 2020, pp. 58–83, ISBN: 978-3-030-47147-7.

[20] F.-Y. Lo, C.-H. Chen, and Y.-p. Chen, "Shrinking counterexamples in property-based testing with genetic algorithms", in *2020 IEEE Congress on Evolutionary Computation (CEC)*, 2020, pp. 1–8. DOI: 10.1109/CEC48606.2020.9185807.

[21] *Jqwik*. [Online]. Available: https://jqwik.net/ (visited on 05/11/2022).

[22] *Hypothesis*. [Online]. Available: https://hypothesis.readthedocs.io/en/latest/ (visited on 05/11/2022).

[23] *Fscheck*. [Online]. Available: https://fscheck.github.io/FsCheck/ (visited on 05/11/2022).

[24] N. Dubienn, *Fastcheck*. [Online]. Available: https://github.com/dubzzz/fast-check (visited on 05/01/2022).

[25] O. Grenrus, *Jsverify*. [Online]. Available: http://jsverify.github.io/ (visited on 11/13/2021).

[26] L. Byron, *Testcheck.js*. [Online]. Available: http://leebyron.com/testcheck-js/ (visited on 11/13/2021).

[27] *Test.check*. [Online]. Available: https://github.com/clojure/test.check (visited on 05/11/2022).

[28] *Babel*. [Online]. Available: https://babeljs.io/ (visited on 05/18/2022).

[29] *Document object model*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction (visited on 02/18/2022).

[30] *Promises*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise (visited on 09/04/2021).

[31] *Using promises*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises (visited on 09/04/2021).

[32] *Axios*. [Online]. Available: https://axios-http.com/ (visited on 09/18/2021).

[33] *Testing React*. [Online]. Available: https://reactjs.org/docs/testing.html (visited on 01/23/2021).

[34] *Create React app*. [Online]. Available: https://create-react-app.dev/docs/running-tests/#docsNav (visited on 01/15/2022).

[35] R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino, "Know you neighbor: Fast static prediction of test flakiness", *IEEE Access*, vol. 9, pp. 76 119–76 134, 2021. DOI: 10.1109/ACCESS.2021.3082424.

[36] *React Testing environments*. [Online]. Available: https://reactjs.org/docs/testing-environments.html (visited on 01/23/2021).

[37] *Jest*. [Online]. Available: https://jestjs.io/docs/ (visited on 05/28/2021).

[38] *React testing library*. [Online]. Available: https://testing-library.com/docs/react-testing-library/intro (visited on 01/23/2021).

[39] *Puppeteer*. [Online]. Available: https://pptr.dev/ (visited on 05/01/2022).

[40] N. Dubienn, *Fast-check scheduler-interface*. [Online]. Available: https://github.com/dubzzz/fast-check/blob/main/documentation/RaceConditions.md (visited on 05/01/2022).

[41] M. Madsen, O. Lhoták, and F. Tip, "A Semantics for the Essence of React", in *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, R. Hirschfeld and T. Pape, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 166, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 12:1–12:26, ISBN: 978-3-95977-154-2. DOI: 10.4230/LIPIcs.ECOOP.2020.12. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/13169.