Generative scenario-based testing on a real-world system

Master's thesis in Computer Science - algorithms, languages and logic

DANIEL ANDERSSON & DAVID LINDBOM

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY AND UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017

Generative scenario-based testing on a real-world system

DANIEL ANDERSSON & DAVID LINDBOM

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF
GOTHENBURG

Department of Computer Science and Engineering
Division of Functional Programming
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2017

Generative scenario-based testing on a real-world system
DANIEL ANDERSSON & DAVID LINDBOM

Supervisor: Koen Claessen, Department of Computer Science and Engineering
Examiner: Carlo A. Furia, Department of Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Division of Functional Programming
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Darts in a dartboard. Public domain photo.

Typeset in LaTeX
Gothenburg, Sweden 2017

Generative scenario-based testing on a real-world system
DANIEL ANDERSSON & DAVID LINDBOM
Department of Computer Science
Chalmers University of Technology and University of Gothenburg

Abstract

Testing is a vital part of the software development process and occupies nearly half of the time used for developing a product. Automating the testing process could result in less developing time required for testing; however, test automation brings many challenges. This thesis explores generative scenario-based testing, in a real-world system, using a QuickCheck implementation.

We implement two test-suites adopting different variants of the method, and discuss advantages and disadvantages of each method, design choices we made, limitations of the QuickCheck tool we used, and obstacles we encountered. The implementation produced satisfying results and managed to reveal 11 bugs of varying severity.

From this work we conclude that generative scenario-based testing is especially suitable for testing interfaces and random user scenarios. Moreover, we found that it could be beneficial to adopt the method already in an early phase of development, and that the method could potentially constitute a great complement to an already existing test-suite.

Keywords: testing, real-world system, QuickCheck, generative scenario-based testing

Acknowledgements

We would like to thank, in no particular order:

Emil Kristiansson and Johan Persson, for providing insight into Qmatic's systems and giving us valuable advice.

Koen Claessen, for supervising this project and giving us feedback on the report.

Qmatic AB, for letting us use their system and supplying us with resources and expertise.

The people at Qmatic, for endless fika.

Daniel Andersson & David Lindbom, Gothenburg, June, 2017

Contents

1
Introduction

Anyone involved in software development is aware that the quality ensuring process of a product is a difficult and time consuming task, often estimated to occupy well over 50 percent of development time. Still, testing is of paramount importance in system development as the reputation of a company relies on providing a product with good user experience.

In the industry today the most common testing method is example-based testing, which requires a lot of manual work as developers have to define a huge amount of unique test cases to secure a high-quality product. Automating parts of the testing phase can be of significant benefit, not only to achieve a more efficient development process, but also to provide a better product. However, test automation produces many challenges that may require deeper understanding of the system and the automation techniques.

This thesis explores one way to automatically generate example-based tests, namely generative scenario-based testing. This method for test automation is not widely applied in the industry at the time of writing this paper. We assess the proficiency of the method in a real-world setting by implementing test-suites for two modules of the Qmatic Orchestra system, a customer journey management implementation that provides an advanced queuing-ticket solution (see section 1.3). The project uses the ScalaCheck library, a QuickCheck implementation in the Scala language that also has support for this method of testing (see chapter 3).

1.1 Our contribution

We will develop two test-suites implementing a generative scenario-based testing approach for a complex real-world system, the Orchestra system. The test-suites will operate through the implemented REST API of the system and, therefore, adopt a black-box testing approach on a system-level. Our project will show how these test-suites could be implemented, what problems we encounter during the development, and how we solve them. We will also discuss in what setting we think this method is best suitable and how it compares to traditional, example-based testing.

Many of the most distinguished case studies [1] [2] that are applying generative scenario-based testing in an industrial context are using Quviq QuickCheck. To further separate this case study from the rest, and because the library meets our requirements for a QuickCheck implementation, we choose to use ScalaCheck, a Scala implementation of the original Haskell QuickCheck.

1.2 Related work

Quviq develops a commercial implementation of the QuickCheck library in the Erlang programming language, called Quviq QuickCheck[1]. Quviq has used the tool to conduct a number of technical studies on different systems and specifications, e.g. the Media Proxy interface used by Ericsson [1] and the AUTOSAR standard [2]. While both these projects are carried out in an industrial context, they focus on the technical aspects and target a part of a system with a well-defined specification.

Boberg uses Quviq QuickCheck to test an e-mail gateway implemented in a message gateway product [3]. In contrast to the aforementioned study on the Ericsson Media Proxy interface conducted by Quviq, this paper aims to evaluate model-based testing on a system-level and using QuickCheck in early development. However, the message gateway product still follows a relatively strict specification.

1.3 Orchestra

The system under test (SUT) used for this thesis is Qmatic's Orchestra platform[2]. Orchestra is a customer journey management system that operates the logic of an advanced queuing-ticket solution. Included among the system's components are queuing tickets, booking of appointments, statistics gathering and media presentation.

Orchestra is a large system of about 500,000 lines of Java code. Today, Orchestra deploys an extensive suite of unit-tests, which are automated for ensuring quality. Qmatic also deploys other quality ensuring measures like code reviews, pair programming and manual QA testing but these will not be considered during this thesis.

The project will implement test-suites for two Orchestra modules, the Java Implemented Queuing Logic (JIQL) module and the Calendar module. The modules in themselves can be regarded as large and complex as they consist of about 100,000 and 30,000 lines of code respectively. Our test-suites will operate through the public REST API, designed for the included web interface and for customers to integrate into their own systems.

JIQL module

The JIQL module contains the queuing logic of Orchestra, which includes creating and managing tickets in the system and calling-rules for counters. This module is one of the oldest and most fundamental parts of the system; therefore, it is also the most well-tested module.

[1] `http://quviq.com/`

[2] `http://www.qmatic.com/customer-journey-management/orchestra/`

Calendar module

The Orchestra system also includes a module for handling prebooked appointments and resource management. The module was previously a separate product but was later reworked to be included as a component of the Orchestra system, with a public API to enable integration with existing customer booking systems. As a consequence of the relatively brief lifespan of the module, it is not as well-tested as most other components of the system.

2
Background

There are several approaches to build a satisfactory test-suite for a specific application. In the industry, unit testing followed by integration testing and system testing is a widely applied methodology for testing software [4]. Individual functions or classes are verified to work correctly by using unit-tests in an early phase of the development of a system. Functionality of modules containing several classes is then verified to function as intended with integration tests. System tests are later deployed to test interaction between multiple modules.

Usually the tests in the three previously mentioned test levels are produced by manually creating test cases with fixed input, which results in each test evaluating the exact same functionality in the same way every time it is run. This deterministic behaviour is desirable when developing a test-suite using the example-based testing method, which we choose to call the group of techniques that requires manual definition of each test case. However, it is extremely difficult to cover all possible scenarios with tests while still retaining a test-suite of maintainable size. A test case in example-based testing can for example be a simple unit-test, shown in figure 2.1a, or a scenario-test [5], described in figure 2.1b.

```
cat.getLegs() == 4
```

(a) Unit-test

```
POST(/new, cat)
PUT(/cat/age, 7)
GET(/cat) == {age: 7}
```

(b) Scenario-test

Figure 2.1: Simple example test cases of each example-based testing method. The unit-test only tests a simple function, while a scenario-test checks that a predetermined scenario works as expected.

A possible solution to the test coverage problem is to automatically generate the test cases with random input data. The difficulty of this automation, however, is to predict the result of the random input data. For example, if the two integers 3 and 5 are generated as input to an *add* function there is no way to know the correct result without calculating the addition, which would require a correctly implemented addition function, rendering the function we are testing redundant. However, there are several methods to circumvent this obstacle and still achieve satisfactory automatic generated tests.

2.1 Fuzzing

In 1990, Miller et al. published [6] a method for testing software called fuzz testing, or fuzzing, which aims to produce a random stream of data to a program. The data may be of a different type or size from what the program originally expects, which could raise exceptions in the application or even result in a system crash. This method of applying invalid data as input to a program is also known as negative testing.

While fuzzing may be useful for testing the stability of a program, the method is not very convenient for finding logical errors in the software. Firstly, when using random input data there is a relatively low probability that the fuzzing tool will provide the program with valid data. Secondly, the original iteration of fuzzing does not support any proper way to confirm what caused a logical error. To accommodate both of these concerns we would need a way to generate random data that is still valid and a model of the system.

2.2 Property-based testing

When combining features of fuzz testing and example-based testing we arrive at property-based testing. Instead of applying fixed, predefined values to a function, property-based testing generates input for a function and compares the output of the function with a property which the function is specified to have. Examples of properties that could be considered when testing the previously mentioned *add* function are commutativity, associativity and the identity property. The introduction of properties removes the requirement of knowing the exact return value, instead the result is ensured to uphold the constraints of the properties.

In property-based testing, with each time a test is run a new test case is generated and the probability of finding new bugs is increased. This makes property-based testing, in theory, superior to traditional, example-based testing when it comes to test coverage. For the latter to be comparable, several thousand (unit) tests would be required, which all have to be maintained in case of changes in the specifications of the system or when new functionality is added.

With all the aforementioned advantages of property-based testing, the main drawback of this method is the more complex nature of stating a property about a function instead of just one expected result to a predefined input. Furthermore, a clear specification of the system is essential to produce the correct properties. A tool for generating input data, assert properties, and minimising failing test cases, which all are the main attributes of property-based testing, is QuickCheck [7].

2.2.1 QuickCheck

The QuickCheck library was originally developed to automatically generate tests for functions in the Haskell programming language [8]. However, the library is, nowadays, present in several of the major programming languages. As mentioned in the previous section, QuickCheck uses a testing method called property-based testing.

The general procedure for this method is as follows; a programmer specifies a number of properties which a function should fulfil; QuickCheck then generates random input parameters of a specified type to try to break one or more of these properties; when a property is broken, QuickCheck tries to minimise the input data and provide the least complex example that contradicts the specified property. Listing 2.1 shows a simple example of a property on lists, where a twice reversed list is the same as the original list.

```
prop_revrev :: [a] -> Bool
prop_revrev xs = reverse (reverse xs) == xs
```

Listing 2.1: A QuickCheck property example in Haskell.

While the QuickCheck libraries do contain generators for most basic types they also provide a way to implement custom generators. These could either be completely custom or a combination of other generators.

Since the generated tests in property-based testing could be of considerable size, QuickCheck accommodates a function that tries to minimise a failed test to the smallest possible example that provokes the failed assertion. QuickCheck also provides minimisation functions for the most common data types. However, as in the case of the generators, there are also possibilities to implement custom algorithms for minimisation.

2.3 Model-based testing

In more complex systems, where a multitude of different functions are implemented, it might not be very interesting to apply QuickCheck for testing a single function. Instead, testing a sequence of function calls, where the result depends on the order of the functions called, results in more intriguing test cases. These function calls are represented as transitions in a state-machine. The state-machine also consists of a model which represents a simplified interpretation of the SUT as well as preconditions and postconditions verifying the result for each function call. The postconditions confirms that the state of the model and the state of the SUT matches after each executed state transition, while the preconditions checks which functions are allowed for the next transition at a given state. This approach, called model-based testing [9], creates possibilities to handle testing in complex systems with a state that carry over between function calls. Listing 2.2 demonstrates a state-machine which can be used as a prototype for this set-up.

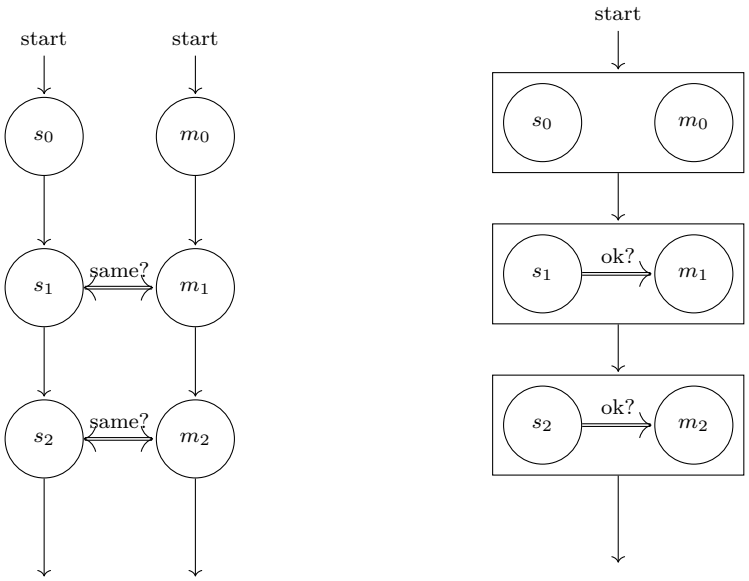
Model-based testing can be split into two variants, oracle-based and verification-based. The oracle-based variant uses a model that is fully distinct from the SUT. Each function is applied on both the model and the SUT, and the returned states are compared to match. This method is best suited when there is an existing implementation which can be used as the model, e.g. when optimising an algorithm. The verification-based approach, on the other hand, revolves around applying the functions on the SUT and verifying against the model that the result of each transition

```
machine :: Model -> [Function] -> IO Model
machine s0 fs = foldM run s0 fs where
  transition :: Model -> Function -> IO Model
  transition s DoA = ...execute, update model, check conditions...
  transition s DoB = ...
```

Listing 2.2: A very simple Haskell state-machine.

follows the specified properties of the system.

The verification-based method results in a less complex model as it does not need to replicate the logic of the SUT to produce a result, but rather merely have to verify that the server response is acceptable. This property is desirable especially when testing non-deterministic systems as information from the response can be saved in the model, which is not possible with the oracle-based approach as the state of the SUT and the state of the model is separated. A concept of the differences between the two variants is illustrated in figure 2.2.



Oracle-based approach.

Verification-based approach.

Figure 2.2: A comparison of the oracle and verification method. Annotation s_i is used for numeration of the state of the SUT and m_i for the state of the model.

2.4 Generative scenario-based testing

By applying the generators of property-based testing to the previously described model-based testing method we obtain generative scenario-based testing, which provides automatic generation of state transitions, in contrast to the regular model-

based version where a developer is required to manually specify a sequence of transitions. The method can be classified as a subset of property-based testing, and the only property basically is to confirm with the model that every state transition of the state-machine is valid. Alternatively, the postcondition of each transition can be seen as an individual property.

Generative scenario-based testing gives us the tool to generate scenarios of random length and order, consisting of random function calls. Compared to other testing methods, property-based testing in general, and generative scenario-based testing in particular, is fairly unknown and typically not applied in an industrial context.

3
Implementation

In the initial phase of the project we assessed the effectiveness of a number of different QuickCheck libraries for several different programming languages. Attributes we considered in the evaluation were state-machine support, maturity of implementation, documentation and compatibility with the software system at Qmatic. Libraries taken into consideration were: Haskell QuickCheck[1], Hypothesis[2] which is implemented in Python, JUnit-QuickCheck[3] which is written in Java, and ScalaCheck[4] for the Scala programming language.

Our conclusion in this matter, after researching and prototyping in the mentioned libraries, was that ScalaCheck fulfilled all of our requirements for a QuickCheck implementation. ScalaCheck is a rather mature implementation of the original Haskell QuickCheck with adequate documentation. Since development of Orchestra is primarily done with Java and other JVM-based languages, using Scala for our project was logical as existing client libraries for parsing data structures could be reused while still having the advanced type system of Scala, similar to that of Haskell.

Unlike the implementation of other QuickCheck libraries that were considered, ScalaCheck promotes the use of the oracle-based approach to model-based testing. This became a limitation for us which is discussed in section 3.5.

For the implementation of our test-suites we utilised the state-machine support in ScalaCheck. Each function transition in the state-machine was represented by the `Command` trait, which is shown in listing 3.1. First, a precondition was checked to determine if the current command was allowed to execute in the current state, otherwise, another command was considered for execution; second, the function was executed on the system under test; third, the next state of the model was calculated; finally, a postcondition was used to confirm that the result of the execution corresponded to the state of the model. An example of a command using this type declaration is shown in appendix A.

The test-suites that were implemented in this project follow a black-box testing approach. This implied that the only way for a test-suite to communicate with the SUT was through an external interface, giving the test-suite no knowledge of the current state of the system. We utilised the REST API implemented in Orchestra as the interface for our test-suites, consequently, the `run` function of each implemented `Command` corresponded to one REST endpoint.

[1] `https://hackage.haskell.org/package/QuickCheck`
[2] `http://hypothesis.works/`
[3] `https://github.com/pholser/junit-quickcheck`
[4] `https://www.scalacheck.org/`

```
trait Command {
  type Result
  def preCondition(State): Boolean
  def run(Sut): Result
  def nextState(State): State
  def postCondition(State, Try[Result]): Boolean
}
```

Listing 3.1: Representation of a command in ScalaCheck.

Instead of letting each command handle the connection to the SUT directly through the REST interface we soon found that having an abstraction layer of the SUT in-between the `Command` and the SUT would be beneficial. The abstraction layer handled, amongst others, authentication, required headers, and the serialisation and deserialisation of JSON objects.

3.1 Defining a model

Defining a suitable model for a SUT proved to be complex. The model had to imitate the system well enough to produce states that adequately described the state of the SUT, but the model should not become another implementation of the system. By simplifying the functionality of the system, treating it as a black box that took an input and returned a result, a satisfactory model was achieved.

Orchestra is on closer inspection a sophisticated system with intricate functionality, but from a black box perspective, using the REST API as our interface, the main functionality of the JIQL module could initially be interpreted as a simple queue. From that simplification we iteratively extended the model with sufficient details, like identifiers for services, to cover more advanced features of the module. The Calendar module, on the other hand, is essentially comprised of one or more week schedules, which each is connected to a number of bookable resources. The model for the Calendar module was, therefore, designed to verify that a booking which was carried out on the SUT followed the rules of the system and that there existed a week schedule with a resource that could accommodate the booking.

The model was defined as an immutable algebraic type. As it was just a simplified view of the system and also not production grade code, the model did not need complicated data structures and could rely fully on simple, standard structures like maps, sets, and lists.

3.2 Parallel tests

Using parallel execution in software always proves to be tricky. The ScalaCheck implementation supports two kinds of parallel execution, either running multiple tests at the same time against multiple SUTs or running commands for one test in parallel.

The first way was straightforward and only needed a way to set up a new instance of the SUT that did not interfere with other systems under test.

The second approach was more complicated. Instead of sequentially executing the commands in order, a pool of threads executed them in parallel. This induced concurrency bugs such as race conditions and deadlocks. The drawback here was that, since some commands were executed concurrently, we could not know which command executed first and what postcondition should hold at any given moment. Instead we had to calculate all possible end states and check if at least one of them held, which proved to be very resource inefficient.

For this project both these options was experimented with. The first was quickly abandoned as Orchestra was a very large system to setup and start, and it was not reasonable to create a new instance for each test. The second approach was also tried but with the exponential growth in both CPU power and memory the method was abandoned as it was not providing enough value for the increase in execution time to be justified.

3.3 Generating commands

In the generation of a sequence of `Command`s, we had the choice of creating valid or invalid input, producing either positive or negative testing. Negative testing aims to generate unexpected data that rarely would represent a real scenario, while positive testing tries to follow the specification of the system. In this project we chose to adopt the positive testing approach as we wanted to produce real scenarios, and only using valid input would produce more complex states.

Our test-suites accommodated two kinds of command generators. The first type was specified for each unique `Command` that was included in a test-suite and generated the parameters for that `Command`. An example of such a command generator is shown in listing 3.2. The second generator type generated the order in which the commands should be executed on the model and the SUT. The latter generator was also used to prevent the generation of a specific `Command`, should it not be desirable at the current state of the system, and there was also the possibility to apply weights to each `Command` to control the rate of generation. An example of how the general command generator may look like is shown in appendix B.

```
object GetBookableDays {
  def gen(state: Model): Gen[Command] = for {
    nServices <- Gen.choose(1, state.services.size)
    services <- Gen.pick(nServices, state.services)
  } yield GetBookableDays(services)
}
```

Listing 3.2: Generator for the GetBookableDay `Command`, which fetches all days that contain at least one bookable slot where the randomly picked services are available.

3.4 Generating configurations

The configuration of a system could be set up using static parameters, resulting in identical starting states for every time a test was run. However, each additionally implemented configuration parameter results in an exponential growth of possible system configurations. Ideally, every such configuration of the system should also be covered by the test-suite. Randomising the input of the function that was setting up the initial state would get us closer to satisfying this requirement. Thus, a test iteration in ScalaCheck was initialised through generation of a configuration, which was used as the starting state of the model, and the SUT was set up to mirror the configuration using the REST API.

While Orchestra is a flexible system, supporting different databases and application servers, these configurations would require a restart of the full application. Ignoring that part of the configuration process, we focused on the internal configuration of the SUT, e.g. number of counters, number of queues, and calling rules.

We could include configuration changes as commands in the test sequence, but as the test-run often was fairly short, executing a test sequence on each generated setup made more sense.

3.5 Non-determinism

ScalaCheck implements a structure where the model and the SUT are completely distinct, i.e. no information is passed from the system to the model. This gives ScalaCheck the advantage of allowing the programmer to access the model in the command generator without having to execute the command on the SUT. Other QuickCheck libraries that we researched could do this as well by not generating the next command before the entirety of the current command is executed. However, if the generator gets stuck and has no commands to choose from, ScalaCheck could just discard this test and generate a new sequence, while other libraries may also have to teardown and setup a new SUT since the changes on the SUT may be irreversible.

One problem that could occur from the separation of the model and the SUT is non-determinism, i.e. in the event that the SUT produces an arbitrary value, then there is no possibility for the model to obtain or reproduce that value. Thus, the model state transition following the non-determinism in the SUT becomes ambiguous, resulting in the model failing to represent future states of the SUT. This problem is exclusively present when using the oracle-based method as the separation of the model from the call to the SUT is forced.

This would not always be a problem in ScalaCheck. Even if information could not be transferred from the SUT into the model, sending data the opposite way was still possible in the generation step of `Command` objects. For example, when creating a new visit on the server a unique identifier for this visit had to be returned in the response body. This information, however, was never returned back to the model, instead we let the model generate an identifier by itself and let the client hold a mapping between this generated identifier and the real identifier given to us

by the SUT. This worked well for many kinds of identifiers and possibly reduced the number of identifiers to keep track of, as one identifier in the model could be mapped to multiple different kinds of identifiers in the SUT, which is shown in appendix A.

Unfortunately, it was not always possible to remedy this problem using the proposed mapping solution. Another example of a problem that we encountered, which emerged from non-determinism when developing the Calendar module test-suite, was when an appointment was booked and there were more than one resource available at the requested time. The SUT would choose the resource for the appointment randomly (first element in a hash table) from the available resources and the model would not know which. This became a problem as a resource could be a member of multiple resource groups and a booking of a resource in one resource group could at the same time deplete all the resources of another resource group. The problem occurred later when the model needed this information to make decisions if another booking should be accepted or not. Figure 3.1 illustrates this scenario where two resource groups, X and Y, contains two resources each and one of the resources, resource B, is shared between X and Y. If an appointment is booked from resource group X followed by two appointments from resource group Y, the third appointment may not have a resource available depending on which resource was chosen for the first appointment. To solve this we found and tried two different approaches, which are described in the two following subsections.

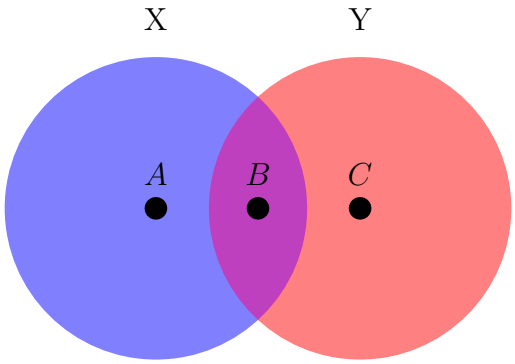


Figure 3.1: Resource group X and Y containing resource A and B, and B and C respectively.

3.5.1 Parallel universes

The first, and most naïve, solution to the issue with non-determinism was to create a new model for each possible model state transition resulting from the non-determinism. In the case where several resources could be selected for an appointment, a new model was created for each possible resource. We added some extensions to the `Command` trait and made general implementations of the old methods to acquire a nice interface with this approach. This is shown in listing 3.3.

For any following case of non-determinism we had to repeat this procedure for each and every model we had created so far. Consequently, the number of models grew exponentially and we quickly ended up with an unmanageable set of models

```
trait ExtendedCommand extends Command {
  def nextStates(State): List[State]
  def postConditions(Model, Try[Result]): Prop
  // General implementations of trait
  def nextState(states) = states.flatMap(nextStates)
  def postCondition(states,result) =
    states.exists(s => postConditions(s,result))
}
```

Listing 3.3: Extension of the `Command` trait, which provides helper-functions for handling parallel states.

which resulted in our test-suite exhausting CPU and memory. Our testing system could handle a few thousand concurrent models and, therefore, a restriction was set for how many states would spawn. This was solved by not generating commands that could induce non-deterministic behaviour after the threshold was reached in each test iteration. The threshold occasionally was reached very quickly, resulting in the subsequently generated commands only utilising a subset of the originally defined commands. Moreover, as there were a few thousand models to process for each generated command, all which require calculation of the postcondition and the transition to the next state, the rest of the test-run progressed at an unworkable pace.

3.5.2 Let the model interact with the SUT

The second approach was to move the model into the client, though this defeats the purpose of having the model and the SUT fully distinct. We earlier found that having the model and client distinct was unusual among other implementations that were considered.

Avoiding the need to rearchitect the ScalaCheck library, the model was split in half, leaving the parts necessary for command generation in the state and moving the rest to the client. Consequently, the new model was calculated in the `run` method instead of in `nextState`.

While this seemed like a small change to the structure of the test-suite, it completely changed the framing of the problem. Previously, the idea was to execute the command on both the model and the SUT and comparing the results. However, following the change, a command was executed on the SUT and the result was confirmed to be valid according to the model.

The new approach resulted in a simplification of the model as fewer system details had to be implemented. Thus, we chose to implement the Calendar module test-suite using the verification-based approach discussed in section 2.3.

3.6 Legacy and handling known faults

In this project we implemented two new test-suites for an existing product. A known complication for automatic testing of software is the handling of bugs that have already been found but still not fixed [10] and, as priorities are set, may not be fixed in any near future. Consequently, the possibility that ScalaCheck would find the same, most probable bug over and over again was high. Since these bugs were already known we did not want the test-suites to find them; therefore, exceptions had to be made to sort out these problematic sequences. To deal with this we explored a few options:

Extend the model to include this behaviour

The obvious way for preventing the test-suites from provoking a known bug was to add the behaviour to the model. However, this required more developer time, which could be better spent on fixing the actual bug rather than, informally, adding it to the specification. Moreover, adding these exceptions to the model was not reasonable as it would eventually make the model too complex.

Control the command generation with preconditions or generator functions

There were two possibilities to control the generation of the `Command` sequence, either by altering the `precondition` method or by including exceptions in the generator function (see appendix B). This would enable the exclusion of specific problematic sequences. The problem with this option was that the exception code could become too limiting. The test-suite could generate a smaller set of tests than desirable, and even have difficulties finding a valid test fulfilling all of the exceptions.

Ignore broken assertions for some commands

Another simple approach was to ignore any failed postconditions or assertions for known faults. However, this could also ignore valid failing tests that were demonstrating an unknown issue.

Introducing a bug database

A more advanced method, which would extend the testing tool significantly, was to include a bug database in the test-suites. This would compare new bugs against a database with already found bugs to see if the bug was of the same type. The method is described by Hughes et al. [10] but was deemed to be outside of the scope of this project.

For this project we chose to control the command generation with preconditions and ignoring a few assertions to cope with some of the bugs that were found. These techniques required some extra overhead for each discovered bug, but the extra work was regarded to be considerably lower than that of the others.

4
Results and discussion

The two implemented test-suites for the JIQL module and the Calendar module respectively gave some promising results in terms of bugs found. Altogether the test-suites uncovered 11 bugs of varying severity and type, whereof four in the JIQL module and seven in the Calendar module. Mainly, the bugs are of an elusive nature, implying that it is very difficult to cover them with traditional, example-based tests.

The main difference between the JIQL test-suite and the Calendar test-suite is that the former primarily relies on a larger amount of endpoints covered with tests, while the success of the latter heavily depends on a more sophisticated generation of unique configurations and, thus, requires significantly fewer implemented endpoints to produce results.

Another notable difference in the two test-suites is the variant of generative scenario-based testing that is used. For the JIQL module the oracle-based approach was used, while the test-suite for the Calendar module implements the verification-based method. The primary reason for using different methods for the test-suites is the non-determinism complication that is described in section 3.5. The main effect of deploying the verification-based approach in the Calendar module is that a less complicated model is required compared to if the oracle-based method would be used, making the test-suite less complex.

Figure 4.1 shows that even though the Calendar test-suite model is simplified as a result of the change to the verification-based approach, it still occupies a larger part of the total test-suite size than that of the JIQL model. This is a result of the additional code required for the randomisation of configurations in the Calendar model. However, if we implement the JIQL test-suite to randomise configurations as well, the JIQL model will probably occupy a larger part of the total test-suite. We can also see that the code needed for each `Command` is much smaller compared to the Calendar test-suite when considering the number of REST-endpoints implemented (see table 4.1). The main reason for this probably is the static configuration used in the JIQL test-suite, which results in a simpler implementation of the `Command`s as values for the configuration parameters can be assumed to be the same for every test-run.

4.1 Notes on ScalaCheck

The execution of a ScalaCheck test-suite consists of a number of tests, which in turn comprise a number of generated commands. Both these numbers are configurable and prove to have a significant impact on how the test-suite behaves and what results

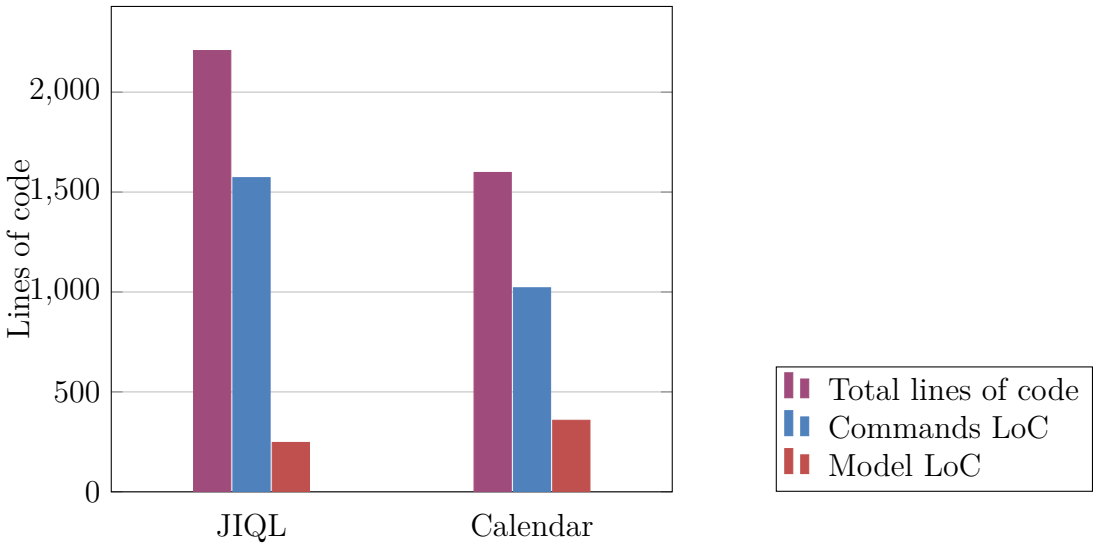


Figure 4.1: The amount of code for different parts of the test-suites.

are obtained from the tests. Specifying the test-suite to have a higher number of tests and utilise a lower amount of commands per test leads to more configurations being tested. While the opposite, i.e. fewer tests with many commands, results in testing more complex scenarios. As the configuration of the SUT occupies a relatively large part of the execution time of a test-run, we opted for tests with longer command sequences instead of a larger amount of smaller tests. We ran the test-suites with test sequences containing up to 500 commands, making one test execute in about one to two minutes, which means that a full test-suite execution of 100 tests is done in about two hours.

Whenever we encountered a failing test when running a test-suite, the failed-test minimising functionality provided by ScalaCheck proved to yield satisfactory minimal test cases. However, we found that the minimising tool used a relatively inefficient mechanism for shrinking our command sequences. A test in ScalaCheck consists of three separate phases: First, a sequence of commands is generated; Second, each command in the sequence is executed sequentially on the SUT and the model; Finally, the results from the SUT and the model are compared with the specified postconditions of each executed command. This entails that the test run will not know that a postcondition is broken until after all the commands have executed, which slows down the minimisation of a test case significantly since all commands subsequent to the broken postcondition still will be executed. Solving this requires a modification to the order in which the minimising step is executed and, therefore, in the ScalaCheck library itself. When it comes to the actual minimising process, ScalaCheck does provide the possibility for a user to specify a custom `shrink` method, giving some opportunity to tailor the minimising process to fit a specific application. However, this custom method does not provide any functionality to interrupt a test-run prematurely. Thus, customising the `shrink` method was not explored further.

ScalaCheck also supports the possibility to minimise the initial configuration that

is used in a failed test, which could be very useful for making the representation of a failed test easier to interpret by a human reader. We chose not to take advantage of this optimisation as we considered our generated configurations to be comprehensible enough and since the shrinking process would become significantly slower for our application. Instead, we relied on manual shrinking of the model for each time we discovered a failed test, which proved to be sufficient for the cases that we encountered.

One of the bugs we found, but chose not to demonstrate further in this report, was of a non-deterministic nature and was only provoked haphazardly by the same test sequence. This proved to cause some problems for the minimisation function, which struggled to find the smallest sequence of commands that could induce the bug. We could not find any suitable solution to this problem, but had to manually minimise the partly shrunk command sequence provided by ScalaCheck. We recognise that this is a problem, but without a solution as this method of testing relies on deterministic system behaviour. However, discovering this bug could still be considered progress as we are able to show that there is a problem, but the methodology cannot produce a minimal, deterministically reproducible test case.

As observed before, the generative scenario-based testing method used in this project is a special case of property-based testing, which encourages our use of a QuickCheck library. From the considerations mentioned earlier, we can conclude that there are both advantages and disadvantages concerning the use of ScalaCheck in this project and that another QuickCheck library probably would have worked just as well for our application. However, as in most cases the use of a specific library does limit the flexibility of a test-suite. Therefore, when applying generative scenario-based testing in a larger project it could be beneficial to build upon the basic constructs of a QuickCheck library, i.e. the generators and property specification, and from that foundation reimplement a minimal library to fit the specific application. Such an implementation could be based on the simple state-machine shown in listing 2.2.

Some of the problems we encountered regarding the ScalaCheck library, e.g. optimisation of the shrinking process, required changes to the library itself. Some of these changes were explored and tested, but were later reverted as we considered them not to belong in the scope of this project.

4.2 Results from test-suites

As mentioned previously, we chose to create two different test-suites covering different system functionality, and as Orchestra is divided into modules it was easy to partition the test-suites into fitting sizes. We ended up with two test-suites which are completely separated, one for the JIQL module and one for the Calendar module. To ensure that the test-suites found what was expected we also ran the test-suites against older versions of the software to see if the new test-suites would find already known and fixed bugs.

Presentation of a discovered fault was facilitated by exportation of the minimised test sequence to a Scala script file followed by some manual tweaks to the config-

uration for extra simplification. The scripts also simplified the reproduction of the error, as ScalaCheck itself does not support any easy way to repeat a sequence of commands which breaks an assertion. The exported scripts still use the functionality provided by the test-suite, specifically the implemented `Command` trait, but are run manually. An example of such a script can be found in appendix C.

Table 4.1 shows a summary of the implemented test-suites and their results. The JIQL test-suite resulted in four reported bugs, together with a few more errors which were already addressed in a patched version, while the number of bugs reported for the Calendar module was eight. In the following subsections we will elaborate on the test-suites we implemented and show a few bugs of different types that were discovered.

Module	REST-endpoints	Configuration parameters	Bugs found
JIQL	15	0	3
Calendar	4	15	8

Table 4.1: Summary of the two produced test-suites.

4.2.1 Test of JIQL module

The first test-suite covers the JIQL module and consists of 15 implemented REST endpoints. While this is only a handful of the available endpoints they are, in our opinion, the most interesting. Among the implemented endpoints are: creating visits, calling visits, opening and closing counters, and switching queues. Some of the properties used to test the module were response code from the server, service point status, queue length, and user status.

Using the test-suite we found three low impact bugs, and one medium impact bug. All these errors were provoked by using the REST API in a way which is not possible to reproduce with the provided web GUI.

Service point status bug

A bug was provoked by first having a simulated cashier log in to one counter and then requesting a logout from another service point. According to the server response acquired from the request the logout was a success, but an inspection of the cashier user status reveals that the cashier remains logged in at the counter. The bug also produced some dubious statuses in the web interface.

This is a bug induced by using the API in a legal but unexpected way. Note that the error could not be provoked from the web interface as a logout from the current counter is triggered automatically if you try to access another counter when using the application as intended. This bug is a clear example of how the coverage of generative scenario-based testing is significantly larger compared to that of a manual method as subtle scenarios can be produced.

4.2.2 Test of Calendar module

For the second test-suite we used the Calendar module as the SUT. The main functionality of the module is to book appointments, leading to relatively few endpoints that alter the SUT compared to that of the JIQL module. As mentioned in section 3.4, generation of configurations increases the coverage of the test-suite and that was taken advantage of in the Calendar model. Examples of high impact parameters that were randomised in the configuration of the initial state of the module were: week schedule opening hours, appointment length and resource group size.

Several of the bugs found in this module would probably not be discovered had we not generated the configurations randomly. Furthermore, only four endpoints had to be implemented as commands and this still resulted in a test-suite that uncovered several high impact bugs. In total, the Calendar module test-suite uncovered seven new bugs and additionally one that was already reported.

Cache bug

One of the bugs we found with the Calendar test-suite turned out to already have been reported by a customer around the same time we found it. The bug was of a time-zone and cache related nature and had eluded the current test-suite. It manifested when an appointment was booked in a time-zone other than UTC+0 and the Datetime of the appointment was converted to UTC+0. As the server converts and stores all received Datetime in UTC+0, the bug only occurs when the conversion yields a Datetime with a different date than the original date. Following such a conversion, the cache containing bookable slots on the previous or the next day was updated, but not the cache of the actual date. The server response to subsequent customer requests for available slots on that date would contain the booked slot as bookable until a command which requires a cache update on the affected date was executed.

This kind of problem is very hard to cover with a traditional test-suite. It requires forethought that the system will behave differently when setup in different time-zones and deep knowledge of how the specific cache implementation works and what limitations it has. The combination of these makes this fault very hard to find.

Daylight saving time bug

We encountered an interesting bug related to the switch to daylight saving time. Provoking the bug required a system configuration with opening hours which overlaps the skipped hour resulting from the switch. Any booked appointment on a system using such a configuration for opening hours will be rejected. The explanation for this behaviour is that Orchestra calculates the number of bookable slots for each appointment request by looping through the opening hours, resulting in an exception in the Datetime class when the function reaches the non-existing hour.

The daylight saving time bug is a prime example of the corner cases which can be produced by generative scenario-based testing, and which would be extremely hard to cover with traditional, manual testing. Since the switch to daylight saving time does only occur once every year, covering this particular bug with unit-tests

would require a developer to produce a test case that have a configuration with the required opening hours as well as at least one appointment on the particular day of the daylight saving time adjustment, which we consider highly unlikely.

Multiple-day-appointment bug

Yet another interesting bug was found using the Calendar test-suite. This time the initial configuration of the system had a great part in revealing the bug. The defect was induced by creating an appointment stretching past the closing time of the system and into the opening hours of the next day. The server disregarded the opening hours of the system and the appointment was accepted by the system.

The bug proves the advantage of randomising the initial state of the system. In this case the closing time and opening time had to be close enough and there also had to exist a sufficient number of services in the system for an appointment to become long enough to bridge the time from closing time to opening time.

4.3 Test coverage

From the amount of bugs that were found we can make the conclusion that our implemented test-suites had larger test coverage than that of the currently utilised test-suite. To verify the coverage we ran the test-suites against older versions of the system to confirm that it would find already known faults. We do however assert that it is probably more correct to assume that the test coverage of our test-suites instead is different from that of the original test-suite, which also tests other aspects like upgrade paths, data persistence, and performance. This shows the value of using both test techniques as complements to each other.

The complementary nature of these methods is even more apparent when considering failed test cases provided by the generative scenario-based test-suites. As mentioned before we did export these failed tests as Scala scripts, which possibly could be included in a regression test-suite of scenario-tests. Since the execution of a full ScalaCheck test-suite may take several hours to run, this set of failed tests could provide a more time efficient measure for developers to ensure product quality during development. The full test-suite can instead be run either locally or centrally by integration services when the developer is finished with a larger patch.

4.4 Development cost

Currently, Qmatic deploys a wide range of software quality measures. Most notable is a large test-suite of unit-tests, which is amended each time a developer is committing new features and bug fixes. These tests are also automatically run with continuous integration software to test different platforms, back-ends and upgrade paths. Moreover, code reviews and manual QA testers further improves the quality of the system.

Typically, developing a example-based test-suite, and managing a traditional testing style in general, requires relatively constant effort independent of what stage of

development is considered. Writing a unit-test or a scenario-test does not require any particular planning or special consideration and is quite straightforward. When it comes to generative scenario-based testing, however, we found that the major part of development had to be focused on the planning and initial setup of a test-suite. That is, defining the base model and formulating the underlying operations of the test-suite. The remaining elements of the test-suite, which essentially consist of commands for the proposed REST endpoints and some subsequent tweaking of the model, are of relatively small impact on the overall test-suite development time. Consequently, a generative scenario-based test-suite, in contrast to a unit-test focused one, has a much larger start-up cost but smaller continuous development cost.

Software is rarely considered to be completely finished as there is constantly new functionality being added to a product, which has to be covered with tests. Therefore, extensibility of a test-suite may be of utmost importance when contemplating what testing method should be used for a project. Traditional testing using unit-tests brings a fairly constant cost for extending a test-suite to cover new functionality in software. Logically, new software modules result in the requirement of extra integration and system tests in addition to the basic unit-tests.

We choose to use an iterative approach when developing our test-suites, systematically adding commands for new REST endpoints and updating the model to correlate with the new command. This proved to work very well with the generative scenario-based approach and resulted in a relatively constant cost for extending a test-suite.

Our test-suites test a public interface to Orchestra. Since any alteration to the interface would also impact the functionality of customer integrations the interface has a higher resistance to change, which results in lower maintenance cost for our test-suites. Moreover, having each REST-endpoint be completely separate means that any alteration to a single endpoint may only require changing the associated `Command` implementation. In contrast, an example-based test-suite would require modifications to all tests using the changed endpoint. However, observe that introducing more detail into the model for one `Command` could require changes to some other `Command` implementation.

We can approximate the total implementation time for each of our test-suites to about 300 hours. However, comparing the cost efficiency of our test-suites to that of traditional, example-based testing may be difficult, as the definition of a test case is very different for the two methods. A test case in example-based testing will test the same functionality every time it is run, while property-based testing, and its derivatives, will produce different tests each time it is executed. This is an important part to consider as automation possibly could bring compelling cost savings in the long run.

We are taking a closer look at one of the errors found with the test-suites, namely the cache bug in section 4.2.2. This bug was reported by a customer at about the same time that we found it using our test-suites. What makes an error like this costly is not the bug itself and to remedy it, but the amount of time required for the bug report to traverse different support personnel and managerial entities of the company and finally reaching the developer teams. In this case the bug was reported

though a subsidiary company that sold the system to the customer. Additionally, bugs found by customers during post-release of a product may require extra resources and effort to avoid the company reputation being affected in a negative way. Studies from both NIST [11] and NASA [12] show that bugs of this nature could entail in between a tenfold and a hundredfold increase in cost compared to if the bug is found during early development. This justifies building a complementary generative scenario-based test-suite, and thereby extending the coverage and possibly catching errors earlier.

5
Conclusion

In this project we have implemented test-suites using the generative scenario-based testing method. While the testing approach is not a universal solution to all testing, we consider the method to be effective for testing interfaces such as a REST API. Among its strength are the generation of complex scenarios and breadth of configurations. We regard this method to be especially suitable for user simulations.

The concern that property-based testing can become overly complex may hold, as more work proves to be required in the initial parts of development. However, we find that our implementation of the generative scenario-based test-suites, which can be argued to be somewhat of a simplification to property-based testing, produced results much earlier than expected. Separating the implementation for each command is considered to be very favourable as it makes an iterative developing style possible, allowing test-suite extension for the most interesting and valuable system functionality at the moment.

Switching testing method in an old product may also bring the problem where the new test-suite encounters known, low priority bugs. A decision has to be made if the bugs are worth fixing, or if an exception should be included in the test-suite. We experienced that adding exceptions for every encountered bug required extra effort and resulted in the test-suite becoming cluttered. Adopting a generative scenario-based test-suite already from the beginning of a project may solve this problem as errors found early may give a higher incentive to fix the problems right away.

While the test-suite is highly configurable in what size each test should be and the number of tests to generate, it may be a hindrance that a full test-round could occupy roughly two hours. Applying such a test-suite in actual development is not viable as a developer needs to validate their work quite often. Instead we propose that the generative scenario-based method works best as a complement to traditional, example-based testing methodologies. As an example, a generative test-suite could be used in the integration step on a server node, while the developer may use a smaller example-based test-suite on their own computer. Moreover, failed test-cases generated by the generative scenario-based test-suite may be exported to the example-based test-suite, extending the over-all coverage and thereby potentially lowering the costs of development.

5.1 Future work

The generative model-based methodology is well researched but the available tooling is still somewhat lacking. There are several libraries supporting the method, but

they often tend to be either overly general or created for a specific use. Looking forward, what we want to see is an open tool that is providing some features which we think is lacking in most of the current QuickCheck implementations, e.g. automatic export of failed test-cases, continuous testing and simplified handling of known bugs, while still having a simple implementation path like that of ScalaCheck.

Bibliography

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

[2] T. Arts et al. "Testing AUTOSAR software with QuickCheck". In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2015 - Proceedings*. 2015. ISBN: 978-1-4799-1885-0.

[3] Jonas Boberg. "Early Fault Detection with Model-based Testing". In: *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*. ERLANG '08. Victoria, BC, Canada: ACM, 2008, pp. 9–20. ISBN: 978-1-60558-065-4. DOI: 10.1145/1411273.1411276. URL: http://doi.acm.org/10.1145/1411273.1411276.

[4] Alain Abra et al. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. Piscataway, NJ, USA: IEEE Press, 2001. Chap. 5, p. 7. ISBN: 0769510000.

[5] Cem Kaner. *An introduction to scenario testing*. 2003.

[6] Barton P. Miller, Louis Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities". In: *Communications of the ACM* 33 (12 Dec. 1992), pp. 32–44. DOI: 10.1145/96267.96279. URL: https://fuzzinginfo.files.wordpress.com/2012/05/fuzz.pdf.

[7] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *International Conference on Functional Programming*. ACM. 2000.

[8] Paul Hudak et al. "Report on the Programming Language Haskell: A Non-strict, Purely Functional Language Version 1.2". In: *SIGPLAN Not.* 27.5 (May 1992), pp. 1–164. ISSN: 0362-1340. DOI: 10.1145/130697.130699. URL: http://doi.acm.org/10.1145/130697.130699.

[9] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.

[10] John Hughes et al. "Find more bugs with QuickCheck!" In: *AST '16 Proceedings of the 11th International Workshop on Automation of Software Test*. New York, NY, USA: ACM, May 2016, pp. 71–77. ISBN: 978-1-4503-4151-6. DOI: 10.1145/2896921.2896928. URL: http://www.cse.chalmers.se/~nicsma/papers/more-bugs.pdf.

[11] G. Tassey. *The economic impacts of inadequate infrastructure for software testing*. National Institute of Standards and Technology. 2002.

[12] JM Stecklein. *Error Cost Escalation Through the Project Life Cycle*. 2004.

Glossary

client A layer between the test-suite and the system handling authentication, the mapping between generated identifiers stored in the model and real identifiers from the SUT, and setup and teardown of the system.

Datetime A Java class used for representing the date and time.

example-based testing A method of testing where each test is defined with a fixed input and should have the same result each time. Umbrella term for unit testing, integration testing and system testing among others.

JIQL Java Implemented Queuing Logic.

model-based testing A method of testing where a test is defined as a sequence of transitions, representing function calls, in an abstract state-machine. A transition is defined together with a pre- and postcondition to verify that the state-machine is in the the correct state.

oracle-based A subclass of model-based testing where the model to test against is another implementation of the system. Especially useful when doing optimisation too make sure the new implementation should always return exactly the same thing as the old.

Orchestra A customer journey management system developed by Qmatic AB. The system under test used for this project.

property-based testing This testing method generates input data for a function and assesses the result with specified properties. Examples of common properties that can be used are: commutativity, associativity and the identity property.

SUT system under test.

verification-based A subclass of model-based testing. A function is executed on the SUT and the response is verified through the model to follow the specified properties. Can in contrast to oracle-based transfer data from the system to the model.

A

Command trait example implementation

```
case class CreateVisit(stateVisitId: Int) extends Command {

  type Result = Boolean

  def preCondition(state: State): Boolean = true

  def run(sut: Sut): Result = {
    val response = sut.post(VISIT_URL)
    maybeAddVisitIdMapping(response, stateVisitId)
    response.code == 200
  }

  def nextState(state: State): State =
    if (canCreateVisit) state.visits += stateVisitId else state

  def postCondition(state: State, result: Try[Result]): Boolean =
    hasCreatedVisit(result) bothOrNeither state.visits.contain(stateVisitId)
}

object CreateVisit {
  // Generator for this command
  def gen(state: State): Gen[Commnad] = for {
    id <- genNewId(state)
  } yield CreateVisit(id)
}
```

Listing A.1: Example implementation of a command in ScalaCheck. Each command that is generated takes a number of parameters which can either be data already present in the system and, therefore, stored in the model, or newly generated data. This is shown with the `stateVisitId` variable.

B

Command generator implementation

```
def genCommand(state: State): Gen[Command] = {
  val high = 50
  val medium = 20
  val low = 5

  var dependentGenerators: ListBuffer[(Int, Gen[Command])] = ListBuffer.empty

  // These need to have an id to make the call
  // it could still be an old, deleted visit
  if (state.createdVisitIds.nonEmpty)
    dependentGenerators ++= Seq(
      (high, DeleteVisit.gen(state)),
      (high, PutEndVisit.gen(state)),
      (high, PostCherrypick.gen(state))
    )

  // ...

  val alwaysPossibleGenerators: List[(Int, Gen[Command])] = List(
    (high, PostVisit.gen(state)),
    (high, PostNext.gen(state)),
    (medium, GetQueues.gen(state)),
    (medium, GetBranch.gen(state)),
    // ...
    (medium, GetVisit.gen(state)),
    (low, PutWorkProfile.gen(state)),
    (low, ModelSanityCheck.gen(state))
  )

  Gen.frequency(alwaysPossibleGenerators ++ dependentGenerators : _*)
}
```

Listing B.1: `genCommand` implementation from the JIQL test-suite.

C

Script example

```
// Set up configuration
val sut = new Client
val s0: State = CalendarModel(/*define configuration*/)
sut.teardown()
sut.setup(s0)

// Define commands
val cmd1 = BookAppointment(
  BranchId(1),
  AppointmentId(1),
  DateTime.parse("2017-04-16 22:00", DateTimeFormat.forPattern("YYYY-MM-dd
      HH:mm")),
  List(services(0), services(1), services(2)),
  List(customers(0), customers(1))
)

// Run and compare.
// If the bug is present the postcondition will fail
val r1 = Try(cmd1.run(sut))
cmd1.postCondition(s0, r1).check
```

Listing C.1: An example of a script reproducing a bug where it was possible to book an appointment over closing time in some specific configurations.

D

Postcondition example

```
def postCondition(state: State, result: Try[Result]): Prop = {
  result match {
    // An appointment was booked
    case Success(Right((newAppointment, appointments))) =>
      // Filter out the new appointment
      val prevAppointments: List[Appointment] = appointments.filter(_ !=
          newAppointment)

      // Check that the appointment should be bookable
      log(isBookable(state, prevAppointments), "An appointment was booked but
          according to the model it should not be")

    // An appointment wasn't booked
    case Success(Left((_,appointments))) =>

      // Check that the appointment should not be bookable
      log(!isBookable(state, appointments), "An appointment wasn't booked but
          according to the model it should be")

    case e =>
      log(false, s"unexpected: $e")
    }
```

Listing D.1: A postcondtion from the Calendar module for when booking an appointment.