U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Property-based testing of a financial market platform

Rui Pedro Fernandes dos Santos Matos Godinho

Dissertação de Mestrado apresentada à
Faculdade de Ciências da Universidade do Porto em
Segurança Informática
2019

MSc
2.º CICLO
FCUP
2019

U.PORTO
Property-based testing of a financial market platform
Rui Pedro Godinho
FC

U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

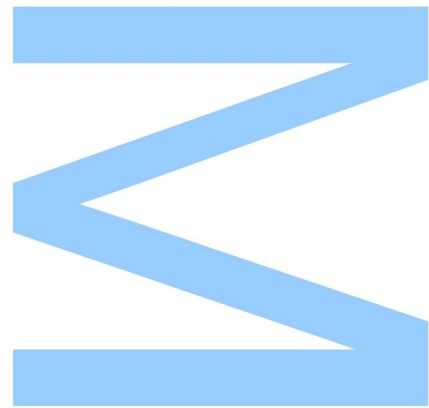
MSc

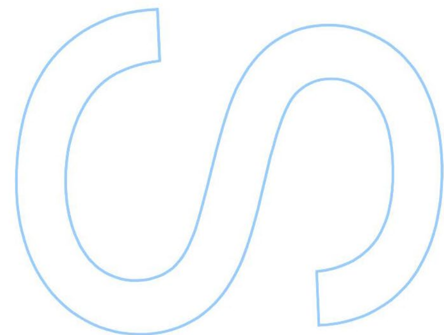
Property-based testing of a financial market platform

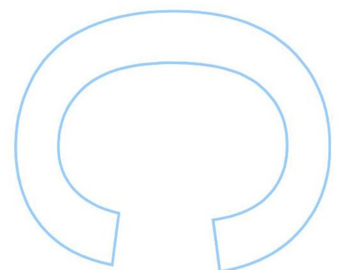
Rui Pedro Fernandes dos Santos Matos Godinho
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
2019

Orientador
Pedro Baltazar Vasconcelos, professor auxiliar, Faculdade de Ciências da Universidade do Porto

Coorientador
Duarte Boucinha Monteiro







Abstract

One of the biggest challenges in the context of software engineering is the avoidance of program crashing. We know for sure that all programs eventually tend to fail, but we can always try to avoid the presence of certain errors in the code that will produce failures on a piece of software. This can be lessen through a continuous testing task, of manually defining testing examples that try to catch the maximum number of failures in the program, before the program reaches the end user. This process is not only tiresome but it is also unmanageable considering the ever growing complex programs appearing in the market.

With this thesis we propose a different alternative, where test cases are automatically generated to test properties of programs. If a test case fails during a test run, an output is returned containing the smallest counter-example proving that the program does not comply with a given property. For this to work, we only have to setup a model that mimics the same part of the software that we want to test.

This methodology is called property based testing and it was used to test the entry gateway of a financial market platform. We decided to test one specific property of the program, and we found, after several test case runs, a failure in the program as it did not comply with the property defined by us.

Acknowledgements

I would like to express my sincere gratitude to my teachers Dr. Pedro Vasconcelos & Dr. Pedro Brandão for guiding me through the course of this project.

I would also like to thank Miss Flávia Natário for helping me understand the mechanics surrounding the trading platform, Mr. Duarte Monteiro for also guiding me and encouraging me to finish this thesis and Mr. Ricardo Gonçalves for helping me debug some annoying problems on the test environment.

Dedicated to my family, who never let me give up on this thesis.

Contents

Abstract **i**

Acknowledgements **iii**

Contents **vii**

List of Tables **ix**

List of Figures **xi**

Listings **xiii**

Acronyms **xv**

1 Introduction **1**

2 Software Testing **3**

2.1 Definition 3

2.2 Terminology 3

2.3 Testing Approaches 4

2.3.1 Unit testing 4

2.3.2 Mutation testing 4

2.3.3 Black-box vs white-box testing 5

2.3.4 Fuzzing 6

2.3.5 Property Based Testing 10

3 Property based testing **11**

3.1 QuickCheck 11

3.1.1 Properties definition 12

3.1.2 Testing stateful programs 14

3.1.3 Practical use of QuickCheck 14

3.2 Hypothesis 19

3.2.1 Generators 19

3.2.2 Shrinking 20

3.2.3 Composite strategies 20

3.2.4 Stateful testing 20

3.3 First use case 22

3.3.1 Testing the C file input/output Python API 22

4 Testing a market platform **25**

4.1 Trading chain platform 25

4.2 Order Entry Gateway 26

4.2.1 Session management 27

4.2.2 Message formatting 28

4.2.3 Structural nuances 28

4.3 Testing the Order Entry Gateway (OEG) with Hypothesis 29

4.3.1 PropertyTesting class 30

4.3.2 TC_Session class 32

4.3.3 Reply parser 34

4.3.4 Rule decorated functions 36

4.3.5 Message building blocks 37

4.3.6 Fields generators 38

5 Experiment and results **41**

5.1 Testing the Heartbeat post-condition 41

6 Conclusions and future work **45**

6.1 Discussion 45

6.2 Future work 46

Bibliography **47**

List of Tables

4.1 Trading chain message specification 30

6.1 Effort metric for each section 46

List of Figures

4.1 High level architecture of the trading platform 26

Listings

2.1 CMD Exploit . . . 8
2.2 Env Var Exploit . . . 8
Code/reverse_law.hs . . . 12
Code/reverse_hask_func.hs . . . 13
3.1 Composite example . . . 20
3.2 Precondition example . . . 22
3.3 File model initialize . . . 23
3.4 Write rule function . . . 23
3.5 Read rule function . . . 24
3.6 Seek rule function . . . 24
3.7 C file API violation . . . 24
4.1 PropertyTesting class . . . 31
4.2 send_first_logon initialize . . . 31
4.3 send_resend_request rule . . . 32
4.4 TC_Session class . . . 32
4.5 Order Entry Gateway (OEG) reconnect function . . . 33
4.6 Messages string to array . . . 35
4.7 Parsing for heartbeat . . . 35
4.8 Receive reply from OEG . . . 36
4.9 Composite functions for generating an header . . . 37
4.10 Optional Header block . . . 37
4.11 Trailer generator . . . 38
4.12 Checksum calculus . . . 38
4.13 TestReqID generator . . . 39
4.14 PossDupFlag generator . . . 40
4.15 Roll the dices . . . 40
4.16 Original sending time . . . 40
5.1 Send Test Request message . . . 42
5.2 Optional fields configuration . . . 43
5.3 Heartbeat assert violation . . . 44

Acronyms

COM Component Object Model

CSS Cascading Style Sheets

DCC Departamento de Ciência de Computadores

FCUP Faculdade de Ciências da Universidade do Porto

FTP File Transfer Protocol

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

SQL Structured Query Language

TCP Transmission Control Protocol

XSS Cross Site Scripting

OEG Order Entry Gateway

ME Matching Engine

MDG Market Data Gateway

FIX Financial Information eXchange

TC Trading Chain

LOC Lines of Code

Chapter 1

Introduction

Over the last 30 years, software has gradually become more complex and difficult to test. Nowadays, it is easy to start developing an application or a program, but it is harder to ensure that these programs do not end up crashing over time, especially, considering the multitude of lines of code required for them to work and the errors that we, human beings, make.

Programs are no longer built from scratch these days. They are a mixture of modules and components, each one built using other software components as well. This adds up to the complexity of how a piece of software is built as well as to the challenge of testing it. So testing has become one of the greatest challenges in the software development area. Tests are now part of most programs, with each developer or tester defining customized tests in the software code, making it more complex than ever. This solution is not the best one since it is not scalable as programs grow, and also difficult for the programmers and testers to maintain.

With this thesis, we propose a suitable alternative to the test generation approach of software testing. The approach followed in this project is based on the concept of property based testing that relies heavily on automatic testing generation and program property assertion. The tests were made on an European trading platform, targeting its gateway of market orders.

Over the next chapters we will cover the most important concepts of software testing as well as the state of the art of automatic test case generation, with approaches like fuzz testing. Then we will delve deep into the concept of property based testing, exploring some relevant frameworks used for this type of testing. In the end we will explain how the tests on our test subject were developed, and also detail the results obtained.

Chapter 2

Software Testing

Before diving into more detail on fuzzing methodologies and property based testing, lets begin with by explaining what software testing is, and why is it so important for the software development life cycle.

2.1 Definition

According to Ammann und Offutt (2016), testing is the most significant way industry has to evaluate software during its development cycle. This necessity for testing software stems from the fact that software itself is produced by humans, who sometimes make mistakes. These human unintentional induced errors can lead to failures along the way of the software life cycle.

In order to catch and resolve these errors, software testing appears as a series of processes which try to ensure that computer code does what it was designed to do and nothing more. These tests help improve the consistency and predictability of software, by mimicking different execution contexts, with various inputs, so as to detect unanticipated behavior early on in the software development life-cycle (Myers u. a., 2004).

2.2 Terminology

In order to better understand the concept of software testing, one needs to grasp three major concepts in the context of software development. These are the basic ideas of *Error*, *Fault* and *Failure*.

In the previous section, we discussed that software testing is required because human beings have a tendency to make mistakes. This idea applies to the concept of Error in software testing. An error is the consequence of software development being made by humans. These errors manifest themselves either in the software code, or even in the software specification. These manifestations are called Faults, and they are commonly referred to as bugs. When a software

fails, i.e produces an unexpected result, as consequence of one or more Faults, we call that a software Failure.

Software testing tries to prevent failures from happening, by attempting to detect the faults that caused them (Ammann und Offutt, 2016). This process is an integral part of the software life cycle, and it is broadly deployed during its different phases (Pan, 1999), consuming up-to 50% of the time spent on the whole software development process (Myers u. a., 2004).

2.3 Testing Approaches

Software testing besides being a time consuming task, is also a complex one, composed of different methodologies, each one with its pros and cons. Pan stated that software tests can be categorized as correctness tests, performance tests, reliability tests and security tests, according to their purpose, life-cycle phase where they are implemented and also according to their scope.

Correctness is the minimum requirement that we expect software to satisfy, thus being the most important category of testing. Correctness testing requires the existence of an oracle in order to differentiate between a right and a wrong behavior. There are two approaches that can be followed depending on the tester's knowledge of the software internals. These are commonly known as "black-box" and "white-box" testing (Pan, 1999).

2.3.1 Unit testing

Unit testing follows the logic of splitting the target software code into small units, in order to individually check each one. These units are usually program functions that are individually tested with predefined inputs, and asserted for expected outputs.

Despite not detecting every single bug on the code, unit testing is most of the times easy to implement and can detect faults in the code early on in the process of software development life-cycle. This is extremely helpful since the effort put on finding and fixing these faults, found during unit testing, is smaller compared to the overhead of fixing them later on in the development process.

This type of testing comprises the first battery of tests to be performed against the developed software. It precedes the integration testing approach, that checks the combination of the individual bits of tested code.

2.3.2 Mutation testing

Mutation testing is a testing approach that focus on assessing the quality of test cases as well as generating new ones. This is done through a process called mutation. The tester will create several different programs, called mutants, based on the insertion of small faults on the original

program. Each mutant will then be tested against the test cases, already developed by the programmer. This way, we can have a better understanding of whether or not our test cases are effective in detecting the small errors introduced in each mutant. The more tests fail with each mutant the better the input test data is.

This method of testing can be categorized as a type of white-box testing, as it requires access to the original source code in order to derive new mutants. One of the major downsides of this type of testing is the fact that it is very time-consuming.

2.3.3 Black-box vs white-box testing

The black-box testing methodology (as the name hints) entirely disregards the internal workings of the target component (Pan, 1999). The tester treats the software under test as a black box, such that only the requirements, as well as the inputs and outputs are observable, and it is only by observing these that we can determine certain functionality traits (Howden, 1980).

Considering that in this type of approach the software testers do not have access to the source code of the program under test, the only way of being sure that all the faults are detected is to test the program with all the possible inputs [1]. If we think that even simple programs have a large input space, the idea of exhaustive input testing of more complex programs becomes impractical, as it is both extremely difficult as well as costly to implement (Myers u. a., 2004). To illustrate this problem, lets consider a 6 argument function, with each argument being a 32 bit integer. In order to be able to black-box exhaust this function it would be necessary to generate 2^{6*32} different inputs. A number that is way bigger than the number of atoms on the Earth [2].

Partitioning comes in hand as one common technique for facing this problem. This approach partitions the input test domain in such a way that the input values in each sub-domain are equivalent, thus enabling the exhaustive testing of each partition by selecting representative value(s) in each domain (Sutton u. a., 2007). Another valid alternative is to have access to the interior parts of the program and use that extra knowledge to guide the generation of input test cases (Myers u. a., 2004). This is something the white-box testing approach does.

Contrary to the black-box testing approach, the white-box testing philosophy lets the tester explore the building blocks of the software under test. Most of the times, this is achieved by giving the tester [3] access to the source code of the software under test. Test cases will be then derived from the examination of the program's logic (Myers u. a., 2004).

The main goal of white-box testing is to cause every statement in the code to execute at least once. This is considered to be a good heuristic of test completeness since a program is considered to be completely tested, when all the possible control flow paths are executed [4] (Myers

[1] this idea is called exhaustive input testing

[2] approximately 10^{50}

[3] here we call tester, the person who will execute the software tests. The person itself can be a dedicated tester or even the software developer

[4] this is the concept of exhaustive path testing

u. a., 2004). This concept is extremely useful not only for stressing all the different features of a program but also for discovering the so called *dead code*, which is the code that never gets executed (Pan, 1999).

But, as it happens with all testing methodologies, this one has also its downsides. The drawback of the exhaustive path testing is centered on the considerably large number of unique logic paths throughout the program's execution, making it impractical, much like the exhaustive input testing approach, to test every single one of them (Myers u. a., 2004). Furthermore, the fact that every logic path is covered, does not necessarily removes the possibility of the program containing faults. Exhaustive path test does not guarantee that a program complies with its specification nor does it detect the absence of a necessary path in the program. Additionally, this testing approach might also not uncover data-sensitive errors, where detection is dependent on the input values used [5] and not on the execution of every logical path (Myers u. a., 2004).

2.3.4 Fuzzing

Fuzzing is a testing methodology whose sole purpose is to test the reliability of a program by injecting it with random input data. According to Takanen u. a., the purpose of fuzzing is to send anomalous data to a system in order to trigger undefined or erratic behaviour on it, therefore revealing reliability problems. The only program specification that fuzz testing checks for is whether or not a crash is provoked when a program is presented with a series of non-sense data, as opposed to approaches like property based testing, where specifications are written by the programmer/tester, and tests are generated to check whether those specifications are obeyed by the software under test.

The term *fuzzing* was first used in a research project from the University of Wisconsin-Madison, and has since been adopted to describe an entire methodology of software testing that is implemented by the so called fuzzers (Sutton u. a., 2007).

We can categorize fuzzing software or fuzzing tools (as they are commonly called), depending on how they generate input testing data. Fuzzers are either labelled as mutation-based or generation-based fuzzers. The mutation-based ones, as the name suggests, apply mutations on predefined data samples to create new test cases. On the other hand, the generation-based fuzzers create test cases from scratch (Sutton u. a., 2007). These general fuzzing philosophies have inspired other fuzzing methods such as the pre-generated fuzzing test cases or the automatic protocol generation testing.

2.3.4.1 Pre-generated Test Cases

The first phase of this method tries to define all data structures and corresponding value ranges of the specification under study. Consequently, all test cases are defined as hard-coded packets

[5] commonly known as input coverage

or files (depending on the program's input interface), that will try to test boundary conditions. Fuzz testing in this manner is inherently limited since there is no randomness on the tests being executed (Sutton u. a., 2007).

2.3.4.2 Random Test Cases

With the random testing approach, each test iteration will simply throw random data at the target, while observing its behaviour. This method, although being the least effective, it can sometimes pay off when detecting (with little complexity [6]) vulnerabilities associated with faulty input validation, by just sending rubbish input test data (Sutton u. a., 2007). One of the downsides of this method is the cumbersome task of manually detecting the fault that caused the failure that the fuzzer detected. This sometimes requires tracking back how, for instance, 500000 random bytes of random input sent by the fuzzer caused a server to crash (Sutton u. a., 2007).

2.3.4.3 Automatic Protocol Generation Testing

Fuzz testing [7] can be used to test communication protocols, relying heavily on the protocol being implemented by the target program. It therefore requires a description of the type of input that is sent, pinpointing the portions of the data (either a packet, a file or any other form of input) that are to be fuzzed, as well as the ones that are to remain static (Sutton u. a., 2007). This description is made through a grammar which the fuzzer parses to generate fuzzed data, that is then sent to the respective target (Sutton u. a., 2007).
The quality of the fuzzer will greatly depend on the tester's ability to describe the protocol specification, with special relevance to the portions of it that are most likely to trigger a failure on the target application (Sutton u. a., 2007).

2.3.4.4 Types of Fuzzers

Fuzzers can also be categorized based on the location of the software being tested. Thus, they can either be defined as local, remote or in-memory fuzzers (Sutton u. a., 2007).

Local fuzzers are mainly comprised by the Command-Line Fuzzers and the Environment variable Fuzzers (Sutton u. a., 2007). The Command-Line fuzzers, just like the name suggests, pass malformed arguments to the target application, through the command line. One of the key features this type of fuzzers try to exploit is the *setuid* file permission, present in Unix based operating systems, which temporarily elevates the privileges of a normal user. Any vulnerability on a *setuid* application will allow an user to permanently escalate privileges and execute arbitrary code (Sutton u. a., 2007).

[6] the only complexity associated with this type of testing is associated with the random data generation
[7] or just fuzzing

```
#include <string.h>

int main(int argc, char **argv){
    char buffer[10];

    strcpy(buffer, argv[1]);
}
```

Listing 2.1: Command Line Exploit

The snippet of code present in 2.1 is a good example of a vulnerable program that would be of interest to a Command-Line Fuzzer, like the *clfuzz* [8], created by *warl0ck*, which tries to exploit both buffer overflows and format string vulnerabilities in command line programs.

An additional type of local fuzzing methodology, that tries to exploit *setuid* applications, is the Environment variable fuzzing approach. Lets consider the following example taken from (Sutton u. a., 2007):

```
#include <string.h>

int main(int argc, char **argv){
    char buffer[10];

    strcpy(buffer, getenv("HOME"));
}
```

Listing 2.2: Environment Variable Exploit

The code snippet of 2.2 shows an attractive example of a program vulnerable to the Environment variable attack vector.

Additionally, we can also include in this list of local fuzzers, the file format fuzzer. The idea behind this type of fuzzing is based on the fact that several applications need (at some point) to parse input files. These applications can be therefore susceptible to file parsing vulnerabilities, which can be exploited via malicious files (Sutton u. a., 2007). The file format fuzzer will then dynamically generate malformed files that are then passed on to the target application, monitoring for any eventual crash (Sutton u. a., 2007).

Web browser fuzzing is a special type of file format fuzzing. The logic behind these fuzzers is based on an Hypertext Markup Language (HTML) functionality that enables the automation of all the fuzzing process. The fuzzer exploits the functionality of the <Meta Refresh> tag to continuously parse the HTML of each test case. This type of fuzzing is not limited to HTML parsing, as there are other fuzzing tools specialized in the test of other components of a web page, such as Cascading Style Sheets (CSS) or the Component Object Model (COM) objects that can be loaded into a web browser (Sutton u. a., 2007).

[8] https://github.com/tuwid/darkc0de-old-stuff/tree/master/clfuzz

One of the most appealing targets for remote fuzzing is any application that listens on a network interface. Most applications nowadays are web-based, which means they communicate with other applications (either remotely or internally) through well known network protocols like the HTTP, DNS and other application layer protocols. Remote fuzzers, also known as network protocol fuzzers, can be split into two major categories: those whose targets are simple protocols and the others that fuzz complex protocols (Sutton u. a., 2007).

Simple protocols are characterized by having a simple or even lacking an authentication mechanism. Most of the times their communications are based on printable characters, like ASCII text, and sometimes they do not even include control fields like length and check-sums. An example of this type of protocols is the File Transfer Protocol (FTP) whose communications are all made in clear text. On the other hand, the complex protocols use binary data as well as encryption or obfuscation methods to enforce data confidentiality and integrity on the established communications.

One key target of remote fuzzers is any web-based application. These fuzzers tend to explore some of the well known vulnerabilities inherent to web-based applications, like SQL injections, Cross Site Scripting (XSS) . . . . These vulnerabilities exist at the application layer and are triggered by data sent on HTTP communications, meaning that the remote fuzzer must be able to communicate with the target applications through this application protocol. Some useful tools for web application fuzzing are (Sutton u. a., 2007):

– *WebScarab* [9] developed by OWASP. It is an open source web application auditing suit with some fuzzing capabilities;

– *Codenomicon* [10] HTTP Test Tools.

In order to test a specific program's process, one can make use of an in-memory fuzzer (a very specific type of local fuzzer). This type of tools will freeze the process under test (by making a snapshot of it) in order to inject faulty data into the process's address space, so as to force any type of crash on the program's in-memory process (Sutton u. a., 2007).

This fuzzing process has the advantage of being fast in execution, as it runs at the memory level, avoiding the overhead of data parsing. However, most of the times, when the process crashes, it is hard to reproduce the same error via an outside source (Sutton u. a., 2007).

Typically used to fuzz a variety of targets, the frameworks used for fuzzing simplify the work of the tester by including libraries for data representation on different targets. These include libraries which can produce fuzzed data (strings, values, etc) in order to trigger exceptions on a variety of targets (Sutton u. a., 2007).

These fuzzing frameworks also facilitate the fuzzing in networked environments, by providing a set of routines for networked communication. One of the main advantages, of a fuzzing framework

[9] https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
[10] https://github.com/Codenomicon/defensics-contrib

is the re-usability in different targets, provided that the respective framework is a generic one and not tailored specifically for a target. The paradoxical downside of the fuzzing frameworks is also related with the fact of them being generic, and thus not covering more specific targets (Sutton u. a., 2007).

A very well known fuzzing framework is the *Boofuzz* [11] fuzzer, a successor to the also praised *Sulley* framework [12], that provides its users with data generation, failure detection, target reset after a failure as well as the recording of test data, all in one package.

2.3.5 Property Based Testing

Property based testing is a testing methodology that uses program property definition as well as test data generation in order to find inconsistencies in software. This testing approach is well known for its automatic generation of test cases that can only prove that a program does not comply with certain properties, by showing counter-examples for when tests fail. It is important to highlight that these properties are defined by the tester, either in the same language of the property based testing tool or in any other language. Afterwards, they are passed on to the main testing framework to generate test cases that will stress a certain program against those same properties.

In the next chapter, several relevant experiences (made in the context of property based testing) are introduced. These examples will help understand this complex concept, as well as demonstrate the usefulness of frameworks like QuickCheck and Hypothesis, which implement the property based testing methodology.

[11] https://boofuzz.readthedocs.io/en/latest/
[12] https://github.com/OpenRCE/sulley

Chapter 3

Property based testing

Property-based testing is the concept of testing programs, modules or even functions, based on properties. The goal here is to define properties and then generate test cases that test those properties on the target software. We can define properties as characteristics that we want our program to comply with. For instance, we can design and develop a small program that only prints out even pseudo-random numbers. One major property that we can define for this program might be that all printed output must be even. After defining this property, we can start generating tests that will stress this specification. If for any reason the output return is not even (but odd), the property fails and we can therefore conclude that our program is wrongfully built.

In this chapter we will discuss some relevant work made in the field of property-based testing. The examples given, try to illustrate the basic principals of property testing, so that in the next chapter we can further detail the work done in the context of this thesis.

3.1 QuickCheck

QuickCheck was initially a library for testing properties of `Haskell` programs (Claessen und Hughes, 2011). Nowadays, libraries similar to QuickCheck have been developed for other programming languages (e.g. Erlang, Scala, Python), all of them implementing the concept of property based testing. The properties to be tested are defined as Haskell functions and can either be tested with random input data or with data produced by a custom test data generator.

One interesting aspect of the QuickCheck framework is that the properties for which the programs are tested against, as well as the test case generators, are written in the host language [1], using a small set of library functions of the programming language. This approach is also followed by other such frameworks such as the Hypothesis tool, for *Python*.

[1] in this case, in Haskell

According to its official documentation web page [2], Hypothesis is a modern implementation of property based testing, designed for mainstream languages. Hypothesis generates and runs a much wider range of tests cases than a human tester could, finding edge cases in the program that a programmer alone would otherwise miss. It also applies a shrinking process on the examples that make the program fail, making them as simple and short as possible in order to save time and money during the testing phase.

The main objective of property based testing is to test universal properties, i.e properties in the form *forall x, y ... P(x, y ...) = True*. If the property holds for all the inputs, then the we can say that our program complies with the property. But in order for the test case generation to end we have to choose a partition in the input space, and that can be achieved with random sampling, which, much like Hypothesis, QuickCheck is able to do. Considering this and because random test generation can most of the times stress unknown edge cases, the tests are not generated following a distribution similar to the one of the real input data [3]. Because of this, QuickCheck is ideal for testing reusable code, such as base libraries present in a variety of larger systems, each one having its own distribution of input data [4]. Regardless of which distribution the generated test cases follow, it is the tester's job to define a suitable test case distribution, through a test data generation language, available within the Haskell syntax (Claessen und Hughes, 2011).

3.1.1 Properties definition

In order for QuickCheck to work properly it is necessary for the tester to define properties that the program must comply with in order to function correctly. These properties are most of the times defined in the same programming language as the one used for developing the testing framework.

3.1.1.1 Reverse example

As an example of property definition, lets have a look at the `reverse` Haskell standard function, used to reverse lists. Ideally this function has to satisfy the following law [5] (Claessen und Hughes, 2011):

```
reverse (xs++ys) = reverse ys++reverse xs
```

In order to be possible to compute the aforementioned property, we consider that it holds true, provided that it passes a finite set of tests (Claessen und Hughes, 2011). This law is then defined as a Boolean Haskell function:

[2] https://hypothesis.works/
[3] the one that is passed on to the target system when it is on the production environment
[4] One good alternative would be to use a uniform distribution in the generation process.
[5] the *++* operator is used to concatenate lists

```
prop_RevApp xs ys = reverse (xs++ys) == reverse ys++reverse xs
```

This function is implicitly quantified, i.e the specification associated with it only holds considering all possible lists. However, in order to be tested, it is only checked for a limited set of lists. Thus, if the function returns `True` for every generated list of the finite input space, then testing succeeds (Claessen und Hughes, 2011).

We can ask QuickCheck to test this property in the Haskell interpreter, through the following call (Claessen und Hughes, 2011):

```
Main quickCheck prop_RevApp

OK: passed 100 tests.
```

The *quickCheck* function receives a property and runs it against a significant volume of randomly generated arguments [6], returning 'OK' if for every test case the returned result is the True value.

On the other hand, if for any reason the property fails for a given argument, QuickCheck reports back the respective counter-example where the property *prop_RevApp* is incorrectly defined (Claessen und Hughes, 2011). Take the below property definition as an example:

prop_RevApp xs ys = reverse (xs++ys) == reverse xs++reverse ys

Checking the (incorrectly) defined property the output returned is the following:

Main> quickCheck prop_RevApp Falsifiable, after 1 tests: [0] [1]

A possible counter example is choosing '[0]' as *xs* and '[1]' as *ys*. As we can see, this is the smallest counter-example encountered by QuickCheck, meaning that the shrinking process was applied on the test case generation.

In the QuickCheck framework, the values generated for testing each property are type driven, meaning that the tester needs to declare the type of each property (just like in the below example). This is only possible because the function *quickCheck* is already overloaded to handle properties with a variety of parameters (Claessen und Hughes, 2011).

prop_RevApp :: [Int] -> [Int] -> Bool

[6] By default 100 test examples.

3.1.2 Testing stateful programs

quickcheck-state-machine is an Haskell library for testing stateful programs (Andjelkovic, 2019). As the name hints, it is based on the Erlang's proprietary QuickCheck framework, however, what sets it apart from the later is that it specifies the correctness of a program through pre- and post-contiditions, much like a state machine based model.

The state machine mechanics are defined in such a way that the tester/developer is required to model the internal states of the system as well as the transitions between states. A transition is defined by a function that modifies the current state. To test the state machine we will need a model that abstractly represents the internal state of the target program, pre- and post-conditions that compare the model against the output of the target program, a state transition function that advances the model to the next state, given the model itself and a transition [7], a mechanism to generate a sequence of actions and another one to shrink them, whenever a failure is found.

The test run is composed of a list of transitions and corresponding arguments (generated by the *quickcheck-state-machine*) and an abstract model representing the program under test. For each generated action, the library checks if the pre-condition holds. If so, the action gets executed and the post-condition is also checked. In the end, the model gets updated using the transition function (Andjelkovic, 2019). If, for some reason, one of the conditions is not satisfied, the initial list of actions is shrunk until the minimal counter-example is found.

3.1.2.1 Shrinking process

The shrinking process takes effect when one of the conditions does not hold for a given action. When QuickCheck detects this incongruity, it stops the generation of new actions, looks at the list of actions/transitions leading to a failure and checks if the failure can still be obtained by a structurally smaller list of actions. The reduction process includes removing actions from the list as well as considering smaller action parameters, e.g. instead of testing with the an integer value of 2, the test is made with the value 1. The process is repeated until the smallest example that produces the failure is found.

3.1.3 Practical use of QuickCheck

As we could see in the previous two subsections, QuickCheck was developed to ease the process of testing. Rather than having to design and develop individual tests, which can become a tiresome task, QuickCheck automatically generates the tests, based on certain properties. This tool is extremely helpful considering the complexity of most of the target applications. These programs usually have a large amount of features, which is something that adds to the complexity of the software, and consequently to the testing process.

[7] A transition is nothing more than the execution of the program under test

3.1.3.1 Testing queue operations with Erlang's QuickCheck

A good test subject for the Erlang's QuickCheck tool, described in (Hughes, 2016), is the circular integer queue structure, and its manipulation methods, implemented in C. Lets consider a structure definition of a queue for storing integer data, placed in a C file (called *q.c*). This queue structure is composed of a pointer to an integer buffer (`int* buff`), two index (int inp, outp), holding the queue's head and tail respectively, and an integer for storing the queue's maximum size (`int size`). The operations to test for are: the `Queue* new(int n)` function, which returns a pointer to a new queue structure, containing an integer buffer of size *n*. It does this by allocating the required amount of space for an integer buffer of size *n*, and then adding it to a newly created Queue structure (with the two indexes starting at zero and the size variable equal to *n*). In the end, it returns a pointer to the newly created Queue structure; the `void put(Queue* q, int n)` function, that inserts an integer *n* into the integer buffer on the *inp* index, adding one unit module *size* units to the same index afterwards; the `int get(Queue* q)`, which returns the value of the integer buffer stored in the *outp* index, increasing afterwards its corresponding value by one unit module *size*; and lastly, the `int size(Queue* q)` function that returns the difference between the *inp* and *outp* variables, modulo *size*.

Before using QuickCheck to automatically test the queue implementation, we can manually test it by making them callable on the Erlang shell. This is done by compiling the C code through the command `eqc_c:start(q)`. After that, it is possible to call each function with any given argument, just like the following examples (Hughes, 2016):

```
2> Q = q:new(5).
ptr, "Queue", 6696712
3> q:put(Q, 1).
ok
4> q:put(Q, 2).
ok
5> q:size(Q).
2
6> q:get(Q).
1
7> q:get(Q).
2
```

In this example, we just created a queue structure and executed a couple of actions, namely the insertion and retrieval of a few values, as well as the calculation of the queue's size, in order to test (although poorly) if the functions work as expected. By looking at the results returned we can see that the functions are working properly, however we cannot assume that they are bug free, just by running them a few times. Instead of manually executing them and look at the returned output, so as to check for any implementation mistakes, we can use QuickCheck to test for certain properties that we want our C code to be compliant with.

QuickCheck tests the code by randomly executing the defined functions and comparing each result of each run, with the result of a parallel abstract model. This model will be defined in other programming language, other than C. If both results (the result from the function under test and also from the abstract model) are equal, the test passes (failing if not).

This model is just a clear representation of the testing target: a queue and its three major operations/actions. Therefore we can model the queue as an Erlang's list of integers, with the C `put` and `get` functions modeled by the Erlang's insertion and retrieval operations on the integers list respectively. The `size` action is modeled by the size of the model's list.

In addition to the creation of an abstract model, we have also to define the conditions that we want to hold true before (preconditions), during and after (postconditions) each run of a C function. These conditions are defined in the Erlang language, and QuickCheck will use them during its tests. The properties checked on a `get` action are defined (in QuickCheck) in the following way (Hughes, 2016):

```
get_pre(S) -> S#state.ptr /= undefined andalso
                        S#state.contents /= [].

get_next(S, _Value, _Args) -> S#state{contents=tl(S#state.contents)}.

get_post(S, _Args, Res) -> eq(Res, hd(S#state.contents)).
```

The *get_pre* statement represents the pre-condition of the model's state before the `get` action gets executed. This condition defines that the pointer to the queue cannot be undefined and the queue must not be empty. this is followed by the `get_next` clause, which is the state transition function, where the next model state contents will be the tail [8] (represented by the *tl* keyword) of the contents of the current model state. Lastly, the *get_post* property, representing the post-condition, checks whether the value returned from the `get` action is the same as the head [9] (represented by the *hd* operator) of the model's current list.

When running QuickCheck from the Erlang shell, there is a failure in the test run after four successful tests. The test that fails contains the following calls and returned results (Hughes, 2016):

```
q:new(1) -> ptr, "Queue", 4003800
q:put(ptr, "Queue", 4003800, 1) -> ok
q:get(ptr, "Queue", 4003800) -> 1
q:put(ptr, "Queue", 4003800, -1) -> ok
q:put(ptr, "Queue", 4003800, -1) -> ok
```

[8] A tail corresponds to the values in the queue except the one that is retrieved

[9] The head is the first value on the queue

q:put(ptr, "Queue", 4003800, 0) -> ok
q:get(ptr, "Queue", 4003800) -> 0
Reason: Post-condition failed: 0 /= -1

After this failure, QuickCheck will then initiate the shrinking process, trying to find the smallest group of calls that will provoke the same failure. This process will remove unnecessary calls and also simplify the arguments passed on to the calls (Hughes, 2016).

q:new(1) -> ptr, "Queue", 4027504
q:put(ptr, "Queue", 4027504, 0) -> ok
q:put(ptr, "Queue", 4027504, 1) -> ok
q:get(ptr, "Queue", 4027504) -> 1
Reason: Post-condition failed: 1 /= 0

Failures can happen either due to faulty implementations or flawed requirements specifications of the target application or even, in some cases, as a consequence of a flawed model design. In this particular case of the C queue, the defined model did not account for the case of adding more elements when the queue is already full. This is solved by adding to the precondition the following Boolean statement (Hughes, 2016):

$$length(S\#state.contents) < S\#state.size$$

After this tweak in the model, the default 100 tests from QuickCheck are completed successfully without an error. However, when modeling the `size` C function, QuickCheck starts to fail in the tests, only this time the problem resides on the C code, when computing the length of the queue (in this case 1 unit, modulo 1), it returns 0 instead of 1 (according to the abstract model) (Hughes, 2016).

q:new(1) -> ptr, "Queue", 4033488
q:put(ptr, "Queue", 4033488, 0) -> ok
q:size(ptr, "Queue", 4033488) -> 0
Reason: Post-condition failed: 0 /= 1

The test failures are caused by an algorithmic constraint, as we cannot store `n` values in a circular array of `n` elements since the queue cannot be empty. One possible fix to this problem would be to alter the `new` C function, making it increase (by one unit) the size variable of the Queue structure, as well as giving one extra unit to the allocated space of the integer queue, each time an `n` size queue is created. This will make the previous test successful, but eventually QuickCheck will fail on the next test cases (Hughes, 2016).

q:new(1) -> ptr, "Queue", 5844232
q:put(ptr, "Queue", 5844232, 0) -> ok

q:get(ptr, "Queue", 5844232) -> 0
q:put(ptr, "Queue", 5844232, 0) -> ok
q:size(ptr, "Queue", 5844232) -> -1
Reason: Post-condition failed: -1 /= 1

This time the error appears to be on the `size` function, when the *q->outp* index is subtracted to the *q->inp*, resulting in a negative value, after the remainder (% q->size) is calculated . Once again this can be fixed with a minor tweak of adding to the subtraction, the value of *q->size*, returning the following result: *(q->inp - q->outp + q->size) % q->size*. With this change, the code becomes correct in the eyes of QuickCheck.

This simple example, showed not only that sometimes the faults occur as a consequence of a bad model definition, but also that one property can find different bugs in the code, provided that the model is well defined. We can also praise the advantage of the test case shrinking done by QuickCheck, as it takes away unnecessary steps that produce overhead during the debugging task.

3.1.3.2 A case study: testing AUTOSAR software for Volvo

One of the biggest projects to use the features of Erlang's QuickCheck was the acceptance testing, made by QuviQ [10], of the AUTOSAR Basic Software for Volvo Cars (Hughes, 2016). Since modern cars have a lot of software components from different providers, which need to comply with a standard (the AUTOSAR standard) in order to prevent any system integration problems, there is the need to test these suppliers code against the compliance specifications.

One of the major errors found by QuickCheck in this project was related with the Controller Area Network (CAN) bus and its message priority system. The CAN bus is used to integrate almost every known component in a car (the brakes, the stereo, etc.), through a messaging system that uses a CAN identifier in each message, which is also interpreted as a priority. The smaller the identifier the higher the priority assigned to a message.

The original CAN standard used 11 bits to identify each message, meaning that there was a total of 2048 different messages (as well as priorities) that could be sent through the CAN bus. Most cars nowadays require a bigger variety of messages to be sent through the bus, forcing the number of bits to be increased from 11 to 29, with the addition of the extended CAN identifier. This upgrade on the number of bits did not restrict the message format used on the bus, meaning that the older (11 bits) messages could also continue to be sent, as long as the bus knew which format was being used prior to sending the message. Consequently, the CAN bus implementation stored the messages' identifier on a 32-bit unsigned integer, using the higher order bit to differentiate between the extended and not extended messages.

When this protocol was tested by QuickCheck a failing test case was found, with the following

[10] A company specialized in software testing, that first developed and marketed the Erlang version of QuickCheck

sequence of messages:

> *send a message with CAN identifier 1 and check whether it was inserted on the bus;*
> *send a message with CAN identifier 2 which should not be inserted yet as the bus is busy;*
> *send message with CAN identifier 3 which should also not be sent;*
> *confirm the transmission of message 1 and the insertion of message 2 to be delivered next*

This test failed because message 3 was sent before message 2. This happened because message 2 used an extended CAN id whilst message 3 used the 11 bit format. When calculating the priorities, there was a bug in the code that made the priority of message 2 to be $2^{31} + 2$. This bug appeared because bit 31 (set to 1 for extended identifiers) was not being masked off during the priority calculation, affecting the prioritization of the messages sent through the CAN bus.

This bug had not yet been detected by the software supplier, even though several manual tests had already been made by the same supplier. The problem was that those tests did not consider this scenario of mixing together extended IDs with old IDs. This example clearly demonstrates the advantages of using a property based testing methodology, through libraries like QuickCheck which significantly help on the automatic generation of test cases, for testing the correctness of complex programs.

In the next chapter we are going to discuss another property based testing framework, called Hypothesis, that was used in the context of this project.

3.2 Hypothesis

In this project we will be using the Python framework called Hypothesis, that lets developers and testers define properties, as well as automatically generate data for testing those properties, following a property based testing approach.

One important concept in Hypothesis is the idea of a *strategy*. Strategies can be used to describe the sort of data that we want to generate in order to test our programs. Hypothesis has a rich strategy library that avoids the manual built of generators of data, thus saving the tester a lot of time (MacIver, 2016a).

3.2.1 Generators

The framework provides generators, also known as strategies, for most built-in types, e.g. if you want to generate integer data, you just use the strategy `integers(min_value=X, max_value)`, specifying the range of values through the *min_* and *max_* arguments. These generators can be customized through parameters in order to constrain or adjust the output

returned by the strategy. For more complex generators/strategies, Hypothesis enables the composition of strategies to generate more intricate types. For instance, to generate a list of floats, the strategy `lists(floats())` can be used (MacIver, 2016a).

3.2.2 Shrinking

One major point in using *Hypothesis* as a property based testing framework is the fact that it incorporates the shrinking process in its strategies. This means that when Hypothesis finds a failure, it will not immediately print out the randomly-generated counter example where the target program fails to comply with a given property. Instead, it will try to reduce the group of examples generated in order to find the smallest sequence that will also produce a failure [11] (MacIver, 2016b).

3.2.3 Composite strategies

In order to combine strategies so as to create one complex custom generator, *Hypothesis* provides the *@composite* decorator. With this decorator one can create a function that returns a strategy for generating custom data (MacIver, 2016b).

The composite functions receive as the first argument, a special function named `draw`. This function is intended to be used inside the composite function in order to generate different types of data which can be mixed together, so as to create a complex type returned by the function. The following example makes use of a composite function to return a tuple, containing a list and an index (MacIver, 2016b):

```
@composite
def list_and_index(draw, elements=integers()):
        xs = draw(lists(elements, min_size=1))
        i = draw(integers(min_value=0, max_value=len(xs) - 1))
        return (xs, i)
```

Listing 3.1: Composite example

3.2.4 Stateful testing

Besides being able to randomly generate different types of data, the framework also enables the random automatic execution of tests. In order to implement that, a tester can make use of the Hypothesis's stateful testing, by defining a number of primitive actions, which are then combined together to find a sequence of those same actions that will produce a failure in the system [12] (MacIver, 2016b).

[11] Any failure, regardless whether it is the same one found initially

[12] A failure in this type of testing is any erroneous state of the system that does not comply with the defined requirements

The logic behind stateful testing in *Hypothesis* is based on the same principles followed by the *quickcheck-state-machine* Haskell library, already discussed in the later chapter.

3.2.4.1 Rule based state machines

Rule based state machines is the most common mechanism, offered by Hypothesis, to construct state machines (MacIver, 2016b). A rule is nothing more than a state transition, with each test case being specified as a post-condition evaluation inside a rule based function. A single test run can be comprised of several rule invocations that may or may not have inter dependency between each other, e.g. one rule cannot be executed until another rule triggers. Each rule based function can be executed once, twice or even multiple times [13] in a randomly manner, depending on how Hypothesis generates each sequence of test cases.

The rules are defined inside a class derived from the superclass *RuleBasedStateMachine* (MacIver, 2016b).

```
class hypothesis.stateful.RuleBasedStateMachine
```

In order to mark a function as a rule, the tester must apply the `@rule()` decorator on the functions that represent a single test case (MacIver, 2016b).

```
@rule(data = strategies.data())
def function_test_case(self, data):
```

3.2.4.2 Initializes

Initializes is another feature of the *Hypothesis* stateful testing. With the *@initialize()* decorator, a function will run at most once at the beginning of a test set run, before any normal rule is invoked. If more than one *initialize* function is defined, all the functions will be run at the beginning, but randomly (MacIver, 2016b).

```
@initialize(data = st.data())
def send_first_logon(self, data):
```

3.2.4.3 Preconditions

Preconditions is a mechanism of the *Hypothesis* framework to filter out rule decorated functions based on a boolean function. The *@precondition()* decorator receives as an argument, a Boolean function. If the function passed on to the decorator returns *True* the rule decorated function

[13] It can also not trigger during a test run

below will execute, if not, the below function is neglected and will not run.

```
1 class MyTestMachine(RuleBasedStateMachine):
2     state = 1
3
4     @precondition(lambda self: self.state != 0)
5     @rule(numerator=integers())
6     def divide_with(self, numerator):
7         self.state = numerator / self.state
```

Listing 3.2: Precondition example

3.3 First use case

An experimental test was developed to experiment with the stateful testing functionality of the Hypothesis framework. This first example (adapted from (Hughes, 2016)) will test a subset of the POSIX file manipulation API as a state machine in Hypothesis. This experiment illustrates how can we test a small but realistic software component, by using an abstract model, rules, input data generators and assertions.

3.3.1 Testing the C file input/output Python API

The purpose of the *TestCFileAPI* rule based class is to test the `read`, `write` and `seek` functions from the C standard file library for Python, to manipulate local files. In order to simulate the same changes happening to a real file when one these functions are evoked, a model is defined inside of the class. We model a single file as a string of bytes (characters), initialized empty in every run test, and an index offset of the current position on the real file.

Each time there is a write action over the real file [14], the model is modified accordingly, thus the string is appended with the characters that were also added to the file, and the variable representing the cursor is incremented by the amount of characters added. Before writing any bytes to the file, the C function appends a certain amount of zero's to the file, provided that the cursor position exceeds the EOF. Because of this, the `write` rule function has to mimic such behavior by appending as much zeros as the number of bytes the cursor position exceeds over the EOF. Only after the zeros are added can the random bytes be inserted both in the file and in the model.

After each action, we assert that the number of characters written to the real file are the same as the ones written to the model. Moreover it is also asserted that the positions on both cursors (real and model based ones) are also the same. All this actions occur inside the `write` rule based function which is randomly called in each run test.

[14] represented by the execution of the rule based function `write`

```
1 def __init__(self):
2     super(TestCFileAPI, self).__init__()
3     open('foo.txt', 'w').close()
4     self.fo = os.open("foo.txt", os.O_RDWR | os.O_CREAT)
5     self.pos = 0
6     self.model = ''
```

Listing 3.3: File model initialize

```
1  @rule(rand_bytes=st.text(alphabet=st.characters(whitelist_characters=
       string.ascii_letters, whitelist_categories=()),min_size=0, max_size=
       None))
2  def write(self, rand_bytes):
3     zeros_to_write = max(0, (self.pos - len(self.model)))
4     self.model += '\0' * zeros_to_write
5
6     bytes_written = os.write(self.fo, rand_bytes.encode('ascii'))
7     self.model = self.model[0:self.pos] + rand_bytes + self.model[self.pos
          + len(rand_bytes):]
8
9     self.pos += len(rand_bytes)
10    assert len(rand_bytes) == bytes_written
11    assert self.pos == os.lseek(self.fo, 0, 1)
```

Listing 3.4: Write rule function

Besides the rule based `write` function, there are also the `read` and the `seek` functions, that simulate the corresponding actions of the related C functions. The `read` function will read a random number of bytes from the file, mirroring this on the model. In order to simplify the tests, the `read` function considers that nothing is read after the EOF character. This is reflected on the *actual_read* variable [15] that will either store zero, if the cursor is beyond the EOF, or the minimum between the random number of bytes to be read and the remaining number of bytes to read until the EOF is reached.

Before terminating, this function asserts that the number of bytes read, both in the real file and in the model, are the same, and that the cursor positions (both real and simulated) are also equal. Lastly, the `seek` function only proceeds with the action of moving the cursor a certain random number of positions in the file, and also in the model. This function does not assert any post-conditions.

By testing (with Hypothesis) these properties of the C file API for Python, the final result is an assertion error caused by a sequence of actions. Apparently, if an empty file is sought by 1 position, written 0 bytes, sought back to the original position and then read by one byte, the number of bytes read in the end are not the same in the real file (1 byte) as in the model (0 bytes). This happens because the API specification was not complete. Apparently this specification lacked the zero padding of the `write` function, when the cursor exceeds the length of the file.

[15] this variable stores the value that will be added to the model cursor position, representation the change of cursor when a read is made

```
@rule(rand_num=st.integers(min_value=0, max_value=1000))
def read(self, rand_num):
    current_pos = self.pos
    bytes_read = os.read(self.fo, rand_num)
    size = len(self.model)

    actual_read = min(rand_num, size - self.pos) # actual_read becomes
        negative if self.pos is bigger than size
    actual_read = max(0, actual_read) # in order to avoid negative numbers

    self.pos += actual_read

    assert len(bytes_read) == actual_read
    assert self.pos == os.lseek(self.fo, 0, 1)
```

Listing 3.5: Read rule function

```
@rule(offset=st.integers(min_value=0, max_value=1000))
def seek(self, offset):
    os.lseek(self.fo, offset, 0)
    self.pos = offset
```

Listing 3.6: Seek rule function

For that reason, Hypothesis found an error not in the API itself, but in our specification. If we correct the specification to consider the zero padding, the tests run successfully.

```
TestCapi02.py:38: AssertionError
-------------- Hypothesis ---------------
Falsifying example: run_state_machine(factory=TesCFileAPI, data=data(...))
state = TestCFileAPI()
state.seek(offset=1)
state.write(rand_bytes='')
state.seek(offset=0)
state.read(rand_num=1)
state.teardown()
```

Listing 3.7: C file API violation

Chapter 4

Testing a market platform

In this chapter we will be exploring how can we apply property based testing to validate certain properties of an European stock exchange market platform, mostly referred to as trading chain.

4.1 Trading chain platform

The Trading Chain (TC) is a complex multi-market platform, composed of three main modules:

- the Order Entry Gateway (OEG), responsible for receiving and parsing the messages sent by the clients that connect to the platform;
- the Matching Engine that (as the name implies) is the engine responsible for matching orders, e.g. buying and selling orders;
- the Market Data Gateway (MDG), a component in the trading platform that provides real-time public market data.

The image in figure 4.1, extracted from (trading company, 2019), presents a diagram of the components and the inter-dependency between each one of them.

According to the figure 4.1, each client, also known as broker, that connects to the trading chain, can send several private [1] inbound [2] messages to OEG. Depending on whether it is an application or an administration message, the order gateway will then process [3] those messages and deliver the ones that are destined to the Matching Engine.

Whenever two orders are matched, the Matching engine will send a public message to the MDG so as to notify the market that a specific match has happened. Each broker can also individually query (through a public application message) the MDG for a specific public

[1] A private message is a message that is sent from one broker or component to another broker or component.
[2] An inbound message is a message that enters the trading platform. An outbound one is the opposite.
[3] The process includes (but is not limited to) the parsing and consequent discarding of malformed messages.

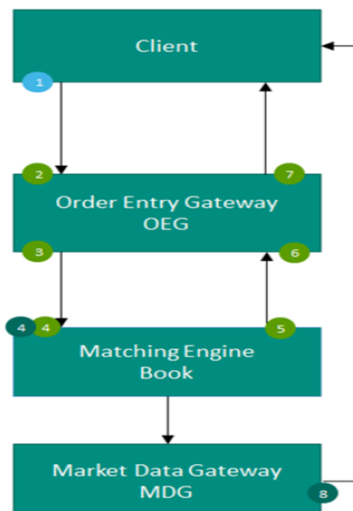


Figure 4.1: High level architecture of the trading platform (trading company, 2019)

information, for which the MDG will reply back with that specific information. Brokers and TC components can also send and receive private messages to and from each other respectively. For instance, a broker can send a private administrative message to OEG in order to check its session status with it, or it can also send a private message to terminate the current session. In the next session we will be explaining in more detail the OEG module of the TC, which is by itself a convoluted system.

4.2 Order Entry Gateway

The trading chain OEG module provides an entry point to the markets platform, through a high-speed and real-time connection (trading company, 2019). The OEG is the software component that manages the access to the markets by acting as the private interface between the clients and the TC matching engine.

The communication between clients and the TC's OEG is based on sending and receiving of messages. A message is a discrete unit of communication, with a predefined format (based on the protocol used), containing information for the trading inside the platform.

A message can either be of the administration or application type. An administration type message is an instruction from a client or a response from the OEG itself, containing non-trade related information, i.e. used to setup and maintain connectivity between both parties (the broker and the OEG). An application message is also an instruction, used to exchange order and trade related information (trading company, 2019). An order is used by a firm to buy or sell an instrument. These orders are then matched upon arrival or placed in an order book, waiting for a match. A trade is an electronic agreement between the clients that submit matching orders.

4.2.1 Session management

In order to establish a session with the TC, clients/brokers have to initiate a TCP/IP connection with OEG, followed by a Logon administrative message. The logon session is always initiated by the client, meaning that the Logon message must be the first message sent by the client, otherwise the OEG will instantly terminate the connection.

After a session is successfully established between both parties, application messages can start being exchanged between the client and the TC.

4.2.1.1 Heartbeats and TestRequests

The OEG uses the administrative Heartbeat and TestRequest messages to ensure that the connection between the client and the TC is functioning properly. OEG sends an Heartbeat message after a given delay of inactivity on its side, i.e. after it has not sent out any messages for a fixed number of seconds. This ensures the client that the OEG is up and running properly. On the other hand, OEG sends a TestRequest message to the client, after a given period of inactivity on the client side. The client, after receiving the TestRequest message, has another equivalent time window to reply, by sending back to OEG an Heartbeat message. If the client does not reply to OEG within the given period, the connection is immediately closed by the OEG. The TestRequest message can also be sent (at any moment) by the client to OEG, which will then have to reply with an Heartbeat message.

4.2.1.2 Logout

The Logout message is used to proceed with the normal termination of the message exchange session between the client and OEG.

4.2.1.3 Message sequence numbers

TC messages are identified by a unique sequence number. The sequence numbers are initialized at the start of each session, starting at 1 and increasing (one unit) throughout the session. Sequence numbers enable each party to identify missed messages and are also used for synchronization when re-connections occur.

Each session establishes an independent incoming and outgoing sequence series. Both the client and the OEG maintain two separate sequence series: one assigned to outgoing messages and the other to incoming messages, so as to detect sequence gaps. If an incorrect sequence number is sent to OEG, the associated message can either be ignored or may trigger a Logout message, stating that the message sequence number is incorrect, while at the same time the TCP connection is automatically terminated.

4.2.2 Message formatting

Every message that is sent to OEG is composed of an header, followed by a body and a message trailer. The message formatting follows the convention of the Financial Information eXchange (FIX) protocol (Lamoureux, Robert and Morstatt, Chris, 2019) (version 5.0), where a collection of *"<Field tag>=<Field value>"*, separated by a well defined delimiter, defines the structure of data of a single message. Each *FIX* field has an associated data type that limits the range of values to fill in the field. According to *FIX 5.0* all tags (if present on the message) must have a value specified.

4.2.2.1 Data types

The FIX protocol specifies the following data types (trading company, 2019):

Alphanumerical: with the authorized characters being: '0'..'9' 'a'..'z' 'A'..'Z' '"' '#' '$' '&' '(' ')' '+' '-' '.' ',' '/' ';' '<' '=' '>' '@' '*' '^' '_' '`' '~' ' '

Numerical: ASCII characters '0'..'9' and the decimal separator '.';

String: Any character or punctuation, except the delimiter character. All String fields are case sensitive. Certain message fields have type String, but are restricted to numerical values ('0'... '9'), meaning that OEG will reject the message in case other character appears on the field;

Float: Sequence of digits with optional decimal point and sign character (ASCII characters "-", "0".."9" and "."); the absence of the decimal point will be interpreted as the float representation of an integer value. All float type fields must accommodate up to fifteen significant digits;

MultipleCharValue: A string field built in such a way that allows the sending of multiple values in just one field. The string solely contains one or more space delimited '1's and '0's, representing different values and/or flags (e.g. 18="0 1 0"). The represented values are identified by the index number on the string, with each value having an odd index (provided that the index count starts at 1) and either a '1' or a '0' character, representing the active and inactive states of the value respectively.

4.2.3 Structural nuances

Some messages may contain a subset of consecutive fields, called a repeating group, that can be repeated a variable number of times. The repeating number is specified by the numerical field preceding the group. Additionally, there can be groups of fields repeated within another repeating group. Such groups are called nested repeating components (trading company, 2019).

Moreover, optional and conditional fields can be set to null as defined by the FIX standard (Lamoureux, Robert and Morstatt, Chris, 2019).

4.3 Testing the OEG with Hypothesis

In order to properly test this specific part of the trading platform, one cannot simply try to crash it by blindly sending arbitrary data at it. Though it can provoke an erroneous behavior on the program side, the TC will most likely ignore the randomly input sent, terminating most of the communications established. Instead of following a fuzzing approach (like the one just described), we propose to test the OEG using the state-machine approach discussed in chapter 3. Because of time constraints, we will not try to cover all the specification of the OEG software, but rather focus on a specific set of properties that can be checked against a property based test framework.

The properties that we want to check are intrinsically related with the messages exchanged between a broker and the trading gateway. Each interaction between each party must follow certain rules, defined by a specification that tells how each message should be composed, considering different trading scenarios. The purpose of the property based tests is to automatically explore some of these different scenarios, by sending a random sequence of messages, automatically generated by the test data generators, and assert relevant properties that the program under test must comply with. The table below 4.1 presents a brief description of each message type used for testing the trading gateway.

When the messages are modeled in the testing framework we can start defining property checks for the test runs. One of the most useful checks that we defined is the validation of whether or not the OEG sends back an Heartbeat message as a reply to a well formed TestRequest message. Other properties can also be checked, such as validating if the message sequence numbers returned on the OEG messages are higher than the last sequence number received. Another example is checking if the checksums returned from OEG were correctly calculated. These validations do not cover all the FIX specification of the OEG component but are a good starting point for testing a complex solution such as this one.

The next subsections will detail the testing process of the Order Entry Gateway component of the TC. We tested this interface using the stateful testing library of Hypothesis, which implements a rule based state machine, already explained in previous sections.

Each message that is sent to the OEG in each run test is built with composite functions, which are complex data generators made of simpler ones. In each test run, several rules are randomly called to send and receive messages to and from the TC's entry point (OEG). Certain rules assert post-conditions that are expected to hold. If the assertion holds, the program continues the battery of tests, whilst if not, it stops the test generation and tries to shrink the sequence of transitions to a minimal failing example (cf. Shrinking section on 3.1.2.1).

Message type	Description
Logon	Used by the market members and also the order gateway to establish a session with the TC. It must be the first message sent by the client, otherwise the OEG will terminate the connection.
ResendRequest	Can be issued either by the Client or the OEG when a gap in the messages sequence numbers is detected by one of the parties.
SequenceReset	This message can either be sent by the Client or OEG, containing the sequence number of the next message that will be sent by one of the parties.
Logout	This message is used to terminate the connection between a client and the OEG. It can also be sent by either one.
Heartbeat	Sent after n seconds of inactivity, to notify the opposite side that the session is still active. It can also be sent in response to a TestRequest sent by either party, containing the same ID present in the TestRequest message.
TestRequest	Used to check whether the other party is still connected, after n seconds of inactivity on the opposite side.
NewOrderSingle	Used by the brokers to create new market orders
MassQuote	This message is used to send several quotes on different instruments in an unique message.
OrderCancelRequest	A message solely sent by the broker to cancel the remaining quantity of an active order in the order book.

Table 4.1: Trading chain message specification

4.3.1 PropertyTesting class

This is the main class, derived from the super class *RuleBasedStateMachine*, where the rule functions are defined. There are a total of 10 rule functions, with 7 being used for sending administrative messages and the remaining ones for sending application messages.

The `__init__` function of the class is used to setup the testing world, i.e. the trading chain where the messages are sent to and processed by the OEG. This is done by spawning a child process that will run a setup script to activate the trading chain. Once the trading chain is up and running in a dedicated process, a TCP session is established through the `reconnect()` method. In the end, when the test run is teared down on the `teardown()` function, the trading chain process is killed with a *SIGKILL* signal sent by the parent process. This mechanism of setting up the target environment as well as destroying it after each battery of tests is made to eliminate the inter-dependency between test-runs, i.e. when a battery of test cases is executed and the environment destroyed, we ensure that the state machine starts with a clean state, that is without having received any messages nor communication attempts. This is also important for the shrinking process since we want to test each smaller sequence of actions, after a failure is detected, starting the state machine with a clean slate.

After the TCP session is successfully established with OEG, the first and only message that OEG will process (according to the FIX specification), in order to setup a trading session, is the Logon message. Every other message will be ignored until this session is established.

```
1  class PropertyTesting(RuleBasedStateMachine):
2
3      def __init__(self):
4          self.session = TradingChainSession()
5
6          #invoke the trading chain process
7          self.trading_chain_arguments = [SCRIPT_PATH, TRADING_CHAIN_BIN,
               TRADING_CHAIN_ENV]
8
9          with open(os.devnull, 'w') as fp:
10             self.trading_chain_process = subprocess.Popen(self.
                   trading_chain_arguments, preexec_fn=os.setsid, stdout=fp)
11
12         time.sleep(3)
13         self.session.reconnect()
14
15         super(PropertyTesting, self).__init__()
```

Listing 4.1: PropertyTesting class

In code terms, this is achieved through the use of the *@initialize* decorator on the `send_first_logon()` function. This function generates a logon message, sends it to the OEG and parses its reply. According to the FIX specification, if the OEG replies back with a logon message, the session is successfully established. The OEG can also reply with a logout message, which means that the Logon message did not have any effect and that the TCP session is now closed.

```
1      @initialize(data = st.data())
2      def send_first_logon(self, data):
3          logon = data.draw(gen_logon(self.session))
4
5          try:
6              self.session.oeg_connection.sendall(logon.encode('utf-8'))
7              self.session.update_msg_sent(logon)
8              print("First Logon sent: ", logon)
9
10             # Parse the OEG response
11             reply = receive_stream(self.session)
12
13             self.session.update_oeg_reply(reply)
14             self.session = fix_parse_reply(self.session)
15
16         except socket.error as exc:
17             sys.exit("Caught a Socket Exception during First Logon : ",
                   exc)
```

Listing 4.2: First logon of the run test

When the OEG replies back with a Logon message, an internal session flag (*logon_status*) is set to True, meaning that messages besides the Logon can now be sent. This is achieved with the *@precondition* decorator and the *logon_status* session object attribute. For instance, a ResendRequest message can now be sent after a successful logon.

```
    @precondition(lambda self: self.session.logon_status is True)
    @rule(data = st.data())
    def send_resend_request(self, data):
        resend_request = data.draw(gen_resend_request(self.session))
        try:
            self.session.oeg_connection.sendall(resend_request.encode('utf
                -8'))
            self.session.update_msg_sent(resend_request)
            print("Resend Request sent: ", resend_request)

            # Parse the OEG reply
            reply = receive_stream(self.session)

            self.session.update_oeg_reply(reply)
            self.session = fix_parse_reply(self.session)

        except socket.error as exc:
            sys.exit("Caught a Socket Exception during Resend Request : ",
                 exc)
```

Listing 4.3: Resend request rule

On the other hand, if the OEG replies back with a Reject message, or even a Logout message, a new Logon message must be sent so as to attempt to establish a new trading session. If a Logout message is received, the TCP connection must be re-established, using the function `reconnect()`, which opens a new socket connection with OEG.

The function `reconnect()` belongs to the *TC_Session* class that establishes the necessary attributes and functions to manage a stable OEG session. This class represents the abstract model required for the property based testing of OEG.

4.3.2 TC_Session class

```
class TC_Session:

    def __init__(self):
        self.oeg_connection = socket.socket(socket.AF_INET, socket.
            SOCK_STREAM) # socket to communicate with OEG
        self.oeg_connection.settimeout(1.0) # Sets the socket to timeout
            after 2 second of no activity

        self.seq_num = "1"
        self.oeg_seq_num = "0" # last OEG message sequence number
            processed
        self.first_seq_num = 0 # first sequence number to be sent at the
            beginning of each run test

        self.logon_status = False
        self.socket_status = False
```

Listing 4.4: TC_Session class

The class *TC_Session* contains the necessary constructs to save and manage a session with the trading chain. The main (`__init__`) constructor initializes the socket for the OEG TCP connection (with a timeout of 1 second) as well as the sequence numbers: the one to be sent to OEG and the one expected to be received from OEG. Besides those attributes, the `__init__` constructor also initializes some important flags, like the flag for controlling the session status with OEG [4], named *logon_status*, as well as the flag that mirrors the TCP session status, i.e. the socket connection state [5], named *socket_status*..

As mentioned in the previous section, the method `reconnect()` is defined inside this class. Here a socket connection with OEG is attempted, having into account the configurations posted on a json file, whose path is saved inside the global variable *JSON_PATH*. This file contains the hostname and network port where the trading chain is listening [6]. If for some reason, the connection cannot be established (returning any particular error) the run test is aborted.

```
1  def reconnect(self):
2      self.oeg_connection = socket.socket(socket.AF_INET, socket.SOCK_STREAM
           )
3      self.oeg_connection.settimeout(1.0) # Sets the socket to timeout after
            1 second of no activity
4
5      try:
6          with open(JSON_PATH) as json_file:
7              data = json.load(json_file)
8              if data['ip'] and data['port']:
9                  try:
10                     # socket connection
11                     self.oeg_connection.connect((data['ip'], data['port'])
                           )
12                     self.socket_status = True
13                     print("Socket connection established")
14                 except socket.error as exc:
15                     print("caught exception socket.error : ", exc)
16                     exit(1)
17     except IOError:
18         exit(1)
```

Listing 4.5: OEG reconnect function

Additionally, the class defines two more flags: the *heartbeat_rcv* and the *reject_rcv* flags. These flags are particularly useful during the assertion of whether the OEG returns an heartbeat or not, after a test request message is sent. If the OEG replies with a message type other than an Heartbeat or a Reject message [7], a violation on the FIX specification of the OEG module is found.

Besides the flags declaration and definition, the instance of the *TC_Session* class also saves, in two specific attributes, the last message sent to OEG as well as the last reply received from it.

[4] Representing the state of being logged on or logged out.
[5] Either established or terminated.
[6] In this particular case, it is listening on the localhost, port 11002.
[7] OEG automatically rejects a Test Request message when it is badly formatted.

These attributes are the *msg_sent* and *oeg_reply*, immediately defined after the heartbeat and reject flags, inside the `__init__` constructor.

Finally, in order to update all the flags and attributes, some auxiliary functions are defined within the *TC_Session* class:

update_seq_num: to update the new sequence number to be sent to OEG;

update_oeg_seq_num: to update the last sequence number received by OEG;

update_msg_sent: to update the last message sent to OEG;

update_oeg_reply: to update the last message received by OEG;

logon: update the *logon_status* to True, meaning that a successful session was established with OEG;

logout: update the *logon_status* to False, meaning that the session established with OEG has been terminated;

4.3.3 Reply parser

One important set of auxiliary functions is defined on the *parser.py* file. These functions were defined to assist the parsing of the OEG replies, after a message is sent to it. The core parsing activity is defined inside the `fix_parse_reply` function. This function starts by checking whether or not there was any reply from the OEG, stored on the *oeg_reply* attribute of the *session* argument. If there was not, the session sequence number is increased by one, so as to synchronize with the sequence number expected by the OEG. Otherwise, the function proceeds with the following steps: starts by retrieving all the messages (sequentially concatenated) and placing each one inside of an array, for further processing. This is all achieved with the help of the `retrieve_messages` function. Inside it, the concatenated reply messages are split by the pattern "8="+FIX_VERSION+"\x01", which is the beginning of every message header, with the global variable *FIX_VERSION* storing the version of the FIX protocol being used; afterwards, it is only necessary to remove the empty string present at the beginning of the messages array. When all the reply messages are all placed inside the array, each one will be transformed into an ordered dictionary, with each key being a tag of the original message, as well as with each value. This rearrange on the data structures is performed inside the function `message_to_dict`; when the message dictionary is created, the value of the key representing the type of reply given by the OEG is checked against a list of possible replies. Once the reply type is found, some processing is made depending on the type of message returned by the OEG. For instance, if the reply is of type 'A' (Logon type message), the logon status flag is set to True by calling the session `logon` method. At the end of each message processing cycle, the session sequence numbers are all updated.

```python
def retrieve_messages(messages):
    message_array = messages.split("8="+FIX_VERSION+"\x01")
    message_array = message_array[1:] # remove the empty string (
        message_array[0])

    return message_array
```

Listing 4.6: Messsages string to array

Another important function inside the parser python file is the `heartbeat_assert`. This function is very similar to the core parsing function `fix_parse_reply` but it is used solely for parsing the OEG replies of the Test Request messages sent. The function differs from the main parsing one in that it parses the reply, searching for any Heartbeat message. If the OEG reply happens to be an Heartbeat, the heartbeat session flag is set to True, otherwise it will remain with the False value. The same applies to the reject session flag, whether or not a reject message is received. In the end, equally to what happens in the `fix_parse_reply` function, both sequence numbers are updated.

```python
def heartbeat_assert(session):
    if not session.oeg_reply:
        session.seq_num = str(int(session.seq_num) + 1)
        session.heartbeat_rcv = False
    else:
        reply_messages = retrieve_messages(session.oeg_reply)
        heartbeat_flag = False
        reject_flag = False

        for reply in reply_messages:
            reply_blocks = message_to_dict(reply)
            reply_type = reply_blocks['35']

            if reply_type == '0':
                print("Heartbeat message received")
                msg_sent_blocks = message_to_dict(session.msg_sent)
                if reply_blocks['112'] == msg_sent_blocks['112']:
                    heartbeat_flag = True

            elif reply_type == '3':
                print("Reject message received")
                reject_flag = True

            session.update_seq_num(str(int(reply_blocks['369']) + 1))
            session.update_oeg_seq_num(reply_blocks['34'])

        #assert heartbeat_flag is True
        session.heartbeat_rcv = heartbeat_flag
        session.reject_rcv = reject_flag

    return session
```

Listing 4.7: Parsing for an heartbeat reply

4.3.4 Rule decorated functions

After establishing a successful session with OEG, a new sequence of messages (both administrative and application related) can now be sent to OEG. Each new message corresponds to a state transition implemented by a *@rule-decorated* method. There are nine rule based functions defined inside the *PropertyTesting* class. Each one of these functions has two decorators prefixed [8], one *@precondition* decorator for checking the status of the session flag *logon_status* and a *@rule* decorator, representing a rule state machine.

Parameters for each rule are generated using custom composite strategies that respect the required message fields. The build of each message is done through the use of composite strategies, which are complex generators made up of simpler ones. Each composite strategy is evoked when passed as an argument of the `draw` function, which is a method of the *data* strategy. When the message is generated, it is then sent to OEG, through the open socket connection already established in the beginning of each run test. After the message is sent, the client tries to obtain a reply from the trading chain by evoking the `receive_stream` function with the *session* object passed as an argument. The `receive_stream` function collects every reply from OEG and concatenates them into a string that returns in the end. The function attempts to read a maximum of 1024 characters in each read attempt [9], saving them in a temporary variable. The temporary variable, storing the whole reply [10], is returned and the function finishes.

```
1  def receive_stream(session):
2      reply = ""
3      aux = ""
4
5      try:
6          aux = (session.oeg_connection.recv(1024)).decode('utf-8')
7          while len(aux) > 0:
8              reply += aux
9              aux = (session.oeg_connection.recv(1024)).decode('utf-8')
10
11     except socket.timeout:
12         return reply
13     except socket.error as exc:
14         sys.exit("Caught a Socket Exception while reading the OEG reply :
               ", exc)
15
16     return reply
```

Listing 4.8: Receive reply from OEG

When the reply is received it is then stored in the *session* object attribute *oeg_reply* and passed on to the parser of the OEG replies.

[8] With the exception of the function `send_extra_logon`, which has only one decorator
[9] Until a socket timeout is raised.
[10] Sometimes an empty string because OEG ignored the message sent by the client

4.3.5 Message building blocks

As for what is defined by the TC OEG client specification, each message sent to the OEG needs to have a header, a body and a trailer. These message components are themselves built by composite functions, with each function using several different composite functions to construct the building blocks, i.e. the "*<tag>:value*" blocks present in each message.

The `gen_header` composite function, as the name implies, is used to construct a message header for every message that is sent to OEG. This composite function starts by inserting the message type, passed on as a parameter, followed by the message sequence number block (returned by the composite function `gen_msg_seq_num`). Afterwards, the function uses a couple of other composite functions to construct the rest of the header building blocks.

Some of the header fields are labelled as optional or conditional, meaning that OEG will not reject the whole message if one them does not appear, as they are not critical for the processing of the rest of the message. A json file, stored in the global variable *JSON_PATH*, declares the fields that are either optional or conditional and maps them with a value between 0 and 1. Each value is then compared against a pseudo-random number, in the range of 0 and 1, generated by a composite function (called `roll_the_dices`). If the generated number is less than or equal to the one in the json file, the optional/conditional value is inserted on the message, otherwise it is not. Some conditional values are not only dependent on the presence of another fields, but also on the values contained by them, thus they are only generated provided that a specific field, or a specific value of a field, is also present in the message.

```
1  from messages.skeleton.fields_gen import roll_the_dices
2  from messages.skeleton.fields_gen import gen_msg_seq_num
3  from messages.skeleton.fields_gen import gen_sender_comp_id
4  from messages.skeleton.fields_gen import gen_target_comp_id
5  from messages.skeleton.fields_gen import gen_on_behalf_of_comp_id
6  from messages.skeleton.fields_gen import gen_deliver_to_comp_id
7  from messages.skeleton.fields_gen import gen_poss_dup_flag
8  from messages.skeleton.fields_gen import gen_poss_resend
9  from messages.skeleton.fields_gen import gen_sending_time
10 from messages.skeleton.fields_gen import gen_orig_sending_time
11 from messages.skeleton.fields_gen import gen_last_msg_seq_num
```

Listing 4.9: Composite functions for generating an header

```
1  if data['last_msg_seq_num'] and draw(roll_the_dices()) <= data['
       last_msg_seq_num']:
2      last_msg_seq_num = gen_last_msg_seq_num(session)
```

Listing 4.10: Optional Header block

The `gen_header` composite function will be used on every composite function that generates a complete message [11] to be sent to OEG. Much like the `gen_header` function, the

[11] A message containing an header, a body and a trailer.

gen_trailer will also appear on every composite function that assembles a whole administrative or application message. The gen_trailer function receives the message without the *BeginString* and the *BodyLength* fields, calculates its length, prefixes the message with the missing fields, calculates the message checksum and appends it to the message. The checksum calculus is made on the auxiliary function fix_checksum in the following way: first, the message string is transformed into an array of bytes; after that, all the bytes of the array are summed and stored on a variable; in the end the function will return the string representation of the variable modulo 256.

```
def gen_trailer(msg):
    message_length = len(msg)
    message = "8=FIXT.1.1" + DELIMITER + "9=" + str(message_length) +
        DELIMITER + msg  # message without the checksum
    message = message + "10=" + fix_checksum(message) + DELIMITER

    return message
```

Listing 4.11: Trailer generator

```
def fix_checksum(data):
    bytes = bytearray(data, 'utf-8')

    i = 0
    sum = 0
    while i < len(bytes):
        sum = sum + bytes[i]
        i += 1

    return (str(format(sum % 256, "03")))
```

Listing 4.12: Checksum calculus

4.3.6 Fields generators

The composite functions for generating message blocks like the Header and the body of the message, and ultimately the whole message, require the use of simpler generators in order to define the basic elements common to every message sent, i.e. the "<tag>=value" units. These basic blocks are defined in a different file (called *fields_gen.py*) and are then imported for helping construct more complex parts of a message. These simpler generators make use of simpler strategies of the Hypothesis framework, namely:

strategies for generating text (strategies.text()), based on an alphabet range defined by the characters strategy (strategies.characters(min_codepoint=X, max_codepoint=Y)), and with a minimum and maximum size (*min_size*, *max_size*);

strategies for generating integers (strategies.integers()), with an upper and lower limit (*min_value* and *max_value* respectively);

strategies for drawing from a specific sample (`strategies.sampled_from(<sample>)`), where *<sample>* is a tuple of values;

strategies for generating floats (`strategies.floats()`), with a minimum and maximum range (*min_value* and *max_value*), an option to generate NaN numbers (*allow_nan*), infinite floats (*allow_infinity*). Besides that, it is also possible to specify the number of bits used to define the float values (*width*=16, 32 or 64) as well as the inclusion or exclusion of the upper and lower limits (*exclude_min*=False or True, *exclude_max*=False or True);

strategies for generating dates between a time window (in between a start and end date). For this it is used the `strategies.datetimes()` with a minimum and maximum dates (*min_value* and *max_value*), as well as a specification of the timezones to be used (*timezones*);

Based on these strategies, each tag value pair can be defined as well with the use of composite functions. Some of the more relevant composite functions used to generate individual pairs are:

`gen_test_req_id` , used to generate the TestReqID field for the administrative messages Heartbeat and TestRequest. This function makes use of the `strategies.text()` strategy to generate an identifier containing the characters defined on the strategy `strategies.characters(min_codepoint=X, max_codepoint=Y);`

`gen_poss_dup_flag` that returns either the character 'N' or 'Y' (through the use of the `strategies.sampled_from(('Y', 'N'))` strategy), for filling the PossDupFlag field of the Header section of each message;

`roll_the_dices` for determining whether or not an optional/conditional value will be included in the final generated message. This composite function makes use of the `strategies.floats(min_value=0, max_value=1, allow_nan=False, allow_infinity=False, width=32, exclude_min=False, exclude_max=False)` strategy to generate a not *NaN* nor Infinite float value between zero and one;

`gen_orig_sending_time` used for the OrigSendingTime conditional field of the Header, returns a date generated by the `strategies.datetimes(min_value=datetime.datetime(1, 1, 1, 0, 0, 0), max_value=datetime.datetime(9999, 12, 31, 23, 59, 59), timezones=none()))` strategy;

```
1 @composite
2 def gen_test_req_id(draw):
3     return "112=" + draw(st.text(alphabet=st.characters(min_codepoint=48,
        max_codepoint=57), min_size=1, max_size=24)) + DELIMITER
```

Listing 4.13: TestReqID generator

```
1 @composite
2 def gen_poss_dup_flag(draw):
3     return "43=" + draw(st.sampled_from(('Y', 'N'))) + DELIMITER
```

Listing 4.14: PossDupFlag generator

```
1 @composite
2 def roll_the_dices(draw):
3     return draw(st.floats(min_value=0, max_value=1, allow_nan=False,
          allow_infinity=False, width=32, exclude_min=False, exclude_max=
          False))
```

Listing 4.15: Roll the dices

```
1 @composite
2 def gen_orig_sending_time(draw):
3     time = draw(st.datetimes(min_value=datetime.datetime(1, 1, 1, 0, 0, 0)
          , max_value=datetime.datetime(9999, 12, 31, 23, 59, 59), timezones
          =none())).strftime("%Y%m%d-%H:%M:%S")
4     time += "." + draw(st.datetimes(min_value=datetime.datetime(1, 1, 1,
          0, 0, 0, 0), max_value=datetime.datetime(9999, 12, 31, 23, 59, 59,
           999999), timezones=none())).strftime("%f").zfill(9)
5
6     return "122=" + time + DELIMITER
```

Listing 4.16: Original sending time

Chapter 5

Experiment and results

Our tests to the OEG were conducted on a remote machine, inside the company's domain, through the execution of the main application (*PropertyTest.py*) using the *pytest* framework. Only one application needs to be executed since the property based testing tool takes care of setting up and tearing down the target program, as explained in the last chapter.

The abstract model was developed based on the FIX specification of the OEG and with the help of a business analyst, specialized in the mechanics of the trading chain. This analyst helped on the modeling of the interaction between the client and the OEG, specifically on the logic behind the message sequence numbers processing and the checksum calculation. The analyst helped as well in the understanding of each interaction between the broker and the OEG, by explaining what is expected from each part when a given message is sent by one of the parties.

5.1 Testing the Heartbeat post-condition

When the model is correctly defined, one post-condition we can assert for is whether OEG returns an Heartbeat message as a reply to a TestRequest message. Moreover, we can also check if the Test Request ID returned by OEG on the Heartbeat reply, is the same as the one sent on the TestRequest message.

This property test is executed inside the rule based function `send_test_request`, using the auxiliary `heartbeat_assert` function to detect if an Heartbeat message was sent as a reply to the TestRequest message previously sent. If an Heartbeat is detected [1], another validation is made against the *TestReqID*, in order to validate if the one that was sent and the one that was received are equal. If they are, the session flag for the Heartbeat reply is set to *True*, otherwise is set to *False*. The same logic applies to the detection of a Reject message from the OEG, only this time it is solely required to check if the MsgType of the reply is equal to '3'. We check if the OEG reply is a Reject message because the initial Test Request sent to OEG can have a field

[1] This is done by checking if the *MsgType* field of the OEG reply message is equal to '0'

with malformed data (due to the randomness of the fields generators), leading OEG to reject it.

```
1  @precondition(lambda self: self.session.logon_status is True)
2     @rule(data = st.data())
3     def send_test_request(self, data):
4         test_request = data.draw(gen_test_request(self.session))
5         try:
6             self.session.oeg_connection.sendall(test_request.encode('utf-8
                 '))
7             self.session.update_msg_sent(test_request)
8             print("Test request sent: ", test_request)
9
10            # Parse the OEG reply
11            reply = receive_stream(self.session)
12
13            self.session.update_oeg_reply(reply)
14
15            #self.session = fix_parse_reply(self.session)
16            self.session = heartbeat_assert(self.session)
17
18            if not self.session.reject_rcv:
19                assert self.session.heartbeat_rcv == True
20
21            self.session.heartbeat_rcv = False
22            self.session.reject_rcv = False
23        """
24        except AssertionError as error:
25            print("Caught an assertion error: ", error)
26            sys.exit(1)
27        """
28        except socket.error as exc:
29            sys.exit("Caught a Socket Exception during Test Request : ",
                 exc)
```

Listing 5.1: Send Test Request message

The initial tests took too much time, requiring to force stop the tests through a *SIGKILL* signal. We were unable to verify, but we think that this was happening due to the shrinking process, together with the randomness of the field data generation. There are some messages that have a significant number of fields, e.g. the New Order message. Furthermore, some of the fields are not mandatory, meaning that they are not required to be on the message. The message generation process randomly determines which optional (and conditional) fields are going to appear on the message to be sent to OEG. The field values have also a random factor in their generation, despite being restrained by the FIX specification, e.g. the *PossDupFlag* Header field, representing the duplication status of a given message, can either have the value 'N'(not a duplicated message) or 'Y' (a duplicated message). This means that one message can either be rejected or accepted by OEG provided that it contains or not a specific field or a specific value. This factor has an impact on the shrinking process because each time a smaller sequence of messages is defined, there is a possibility of a given message not containing the right fields as well as the right values to provoke another failure.

One possible alternative to this problem is to eliminate the randomness of the fields appearance on each message. We thus manually define which optional and conditional fields are going to appear on each message, letting the randomness stay solely on the values generation. This can be achieved by changing the *json* configuration files for the message Header as well as for the body of each message type defined. Since each configuration file contains the probabilities for each optional/conditional field of a specific message, we can modify the values to be either '0', if we do not want the field to appear on the message, or '1' if we want the field to appear every time the message is generated. 5.2 is an example of the configuration file for the optional fields on the Header section.

```
1 {
2     "firm_id": "00000101",
3     "on_behalf_of_comp_id": 1,
4     "deliver_to_comp_id": 0,
5     "poss_dup_flag":  0,
6     "poss_resend": 0,
7     "last_msg_seq_num": 0
8 }
```

Listing 5.2: Optional fields configuration

Another optimization tweak to decrease the tests duration, is to comment out some of the defined rule based functions [2]. This process despite filtering out some message sequences that might lead to a failure during tests, it will definitely reduce the duration of the test runs.

After these changes were implemented, Hypothesis detected a violation of the post-condition defined on the `send_test_request` rule based function. The failure is detected approximately after two hours of test runs, with the framework returning a sequence of messages sent to OEG. This failure is triggered on the assertion that the Heartbeat flag of the session is set to True (and the Reject flag set to False), after a Test Request message is sent to OEG.

The returned falsifying example, as it can be seen on listing 5.3, is a counter-example for the specification that OEG always sends an Heartbeat message whenever a well formatted Test Request message is sent. In this case, by checking the logs produced by the run tests, we can see that after the New Order message is sent (through the `send_new_order()` rule based function invocation) with a message sequence number equal to '1', the OEG replies back with a Reject message, containing the LastMsgSeqNumProcessed field [3] equal to '0'. Afterwards, when the TestRequest is sent with a message sequence number equal to the last one processed by OEG plus one, the OEG replies back with a Logout message, containing the field SessionStatus with the value '9', meaning that the received message sequence number was too low. The TCP connection is immediately closed afterwards.

According to the business analyst of the European trading company, the OEG should never return the value '0' on the LastMsgSeqNumProcessed Header field. This test demonstrated, with

[2]We have to be aware to not comment the rule based function where the post-condition is evaluated
[3]Last message sequence number processed.

```
PropertyTesting.py:192: AssertionError
-------------------- Hypothesis ----------------------
Falsifying example: run_state_machine(factory=PropertyTesting, data(...))
state = PropertyTesting()
state.send_first_logon(data=data(...))
Draw 1: FIXT.1.1\x019=127\x0135=A\x0134=1\x0149=00000101\x0156=Market\
    x01115=00000101\x0152=20190907-21:09:22.000476133\x01108=60\x0198=0\
    x0121019=1\x0121021=3\x01789=1\x0121020=0\x011137=9\x0110=120\x01
state.send_new_order(data=data(...))
Draw 2: FIXT.1.1\x019=218\x0135=D\x0134=2\x01\x0149=00000101\x0156=Market\
    x01115=00000101\x0152=20190907-21:09:23.000479423\x0160
    =20190907-21:09:23.000479453\x0111=0\x0148=0\x0122=8\x0120020=1\x0138
    =0\x0140=1\x0159=0\x0129=7\x01453=1\x01448=0\x01447=D\x01452=1\x012376
    =22\x0121018=0\x0121800=0\x01552=1\x0154=1\x016399=1\x0110=238\x01
state.send_test_request(data=data(...))
Draw 3: FIXT.1.1\x019=84\x0135=1\x0134=2\x0149=00000101\x0156=Market\
    x01115=00000101\x0152=20190907-21:09:24.000484172\x01112=0\x0110=149\
    x01
state.teardown()
```

Listing 5.3: Heartbeat assert violation

a simple sequence of messages, that the OEG fails to correctly process every message sequence number, especially in cases where a malformed message is rejected.

Chapter 6

Conclusions and future work

The aim of this thesis project was to apply property-based testing (namely the model, data generators, post-conditions and a state update function) in the context of a real problem, i.e. a financial market platform. With this in mind, we started developing a model, based on the existent FIX specification of the OEG module that we wanted to test. In order to properly test the market platform, more precisely the message entry gateway, we had also to develop test data generators as well as a post-condition that we wanted to verify.

6.1 Discussion

We achieved this objective by developing a test framework, divided into three major sections, representing the different dimensions of the property based testing methodology. The first section, and also the most import one, refers to both the message fields generators and the modules for generating an entire message. The fields generators are all grouped up in one single python file, imported by each python module responsible for generating a specific type of message. These generators, and the modular way they were built, ease the creation of new message types, in the future.

The tool has also 10 modules: one for generating an Header section for each message, six for the administration messages and three for the generation of application messages. The second main part of the tool is made by the model definition. The tool uses a specific class that stores the necessary states, e.g. the message sequence numbers, to emulate what is happening on the trading chain side.

Finally, on the main python class, the tool defines the state transition functions, also known as rule based state machines in the Hypothesis framework, each one updating the state machine of the testing environment. An effort metric is present on the table 6.1, reflecting the programming effort of each module of the property based testing tool developed.

Table 6.1: Effort metric for each section

Code section	**Lines of Code (LOC)**
Fields generator	551
10 message modules	659
Main class	280

In the end, we managed to achieve our main goal which was applying property based testing in a critical infrastructure that is a financial market platform. The tests made on the target revealed that the program did not comply with the post-condition that we initially defined, based on the FIX specification. This failure would not be possible to detect if the test approach had been a fuzzing one. This type of methodologies can only scratch the surface of a program, by testing for failures in the parsing of data, e.g. memory corruption errors like buffer overflows. When facing failures derived from the logic of a program, the fuzzing methodology is not the best approach to take.

6.2 Future work

With this modular infrastructure, if we want to add more message types to the testing environment, we only have to create one new python module and one rule based function, for each new type of message added. If we wish to test other post-conditions, and depending on the type of messages each post-condition is focused on, most of the times we only need to add new assertions on the rule functions already created.

There are a couple of post-condition validations that can be added with minimum overhead. These are the checksum and sequence number fields validation. We can validate whether the checksum returned from OEG is valid, by calculating each checksum value of an OEG message and then check it against the value that appears on the message. The sequence number field assertion can be achieved simply by asserting that the sequence numbers, present on the OEG message replies, are at least higher than the sequence numbers that we have already received from OEG. This type of validation can also be applied to the Header field LastMsgSeqNumProcessed, returned by OEG, which enables another way of detecting the failure mentioned.

There are many post-conditions that can be further validated as well as many message structures that can also be added to the framework in order to increase the complexity of the tests. This is eased by the modularity of the developed program that demonstrates one of the biggest advantages of defining properties rather than defining individual test cases.

Bibliography

[Ammann und Offutt 2016] Ammann, Paul ; Offutt, Jeff: *Introduction to software testing*. Cambridge University Press, 2016

[Andjelkovic 2019] Andjelkovic, Stevan: *quickcheck-state-machine: Test monadic programs using state machine based models*. 2019. – URL http://hackage.haskell.org/package/quickcheck-state-machine. – Accessed: 2010-09-21

[Claessen und Hughes 2011] Claessen, Koen ; Hughes, John: QuickCheck: a lightweight tool for random testing of Haskell programs. In: *Acm sigplan notices* 46 (2011), Nr. 4, S. 53–64

[trading company 2019] company, European trading: *OEG CLIENT SPECIFICATIONS – FIX 5.0 INTERFACE*. 2019

[Howden 1980] Howden, William E.: Functional program testing. In: *IEEE Transactions on Software Engineering* (1980), Nr. 2, S. 162–169

[Hughes 2016] Hughes, John: Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane. In: *A List of Successes That Can Change the World*, 2016

[Lamoureux, Robert and Morstatt, Chris 2019] Lamoureux, Robert and Morstatt, Chris: *Financial Information eXchange*. 2019. – URL https://www.fixtrading.org/. – [Online; accessed 25-September-2019]

[MacIver 2016a] MacIver, David R.: *Hypothesis*. 2016. – URL https://hypothesis.works/. – Accessed: 2019-04-29

[MacIver 2016b] MacIver, David R.: *Welcome to Hypothesis!* 2016. – URL https://hypothesis.readthedocs.io/en/latest/. – Accessed: 2019-04-29

[Myers u. a. 2004] Myers, Glenford J. ; Badgett, Tom ; Thomas, Todd M. ; Sandler, Corey: *The art of software testing*. Bd. 2. Wiley Online Library, 2004

[Pan 1999] Pan, Jiantao: Software testing. In: *Dependable Embedded Systems* 5 (1999), S. 2006

[Sutton u. a. 2007] Sutton, Michael ; Greene, Adam ; Amini, Pedram: *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007

[Takanen u. a. 2018] Takanen, Ari ; Demott, Jared D. ; Miller, Charles ; Kettunen, Atte: *Fuzzing for software security testing and quality assurance.* Artech House, 2018