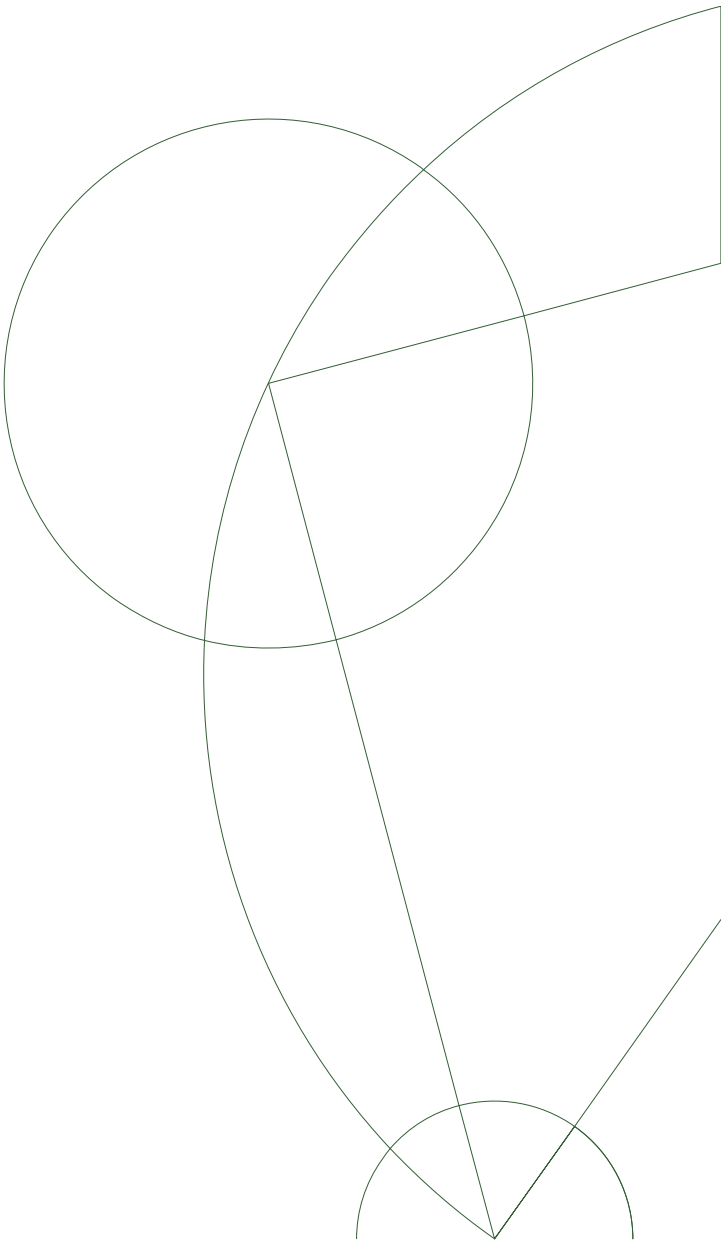
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

MSc thesis

Søren Lund Hess-Petersen - pws412

Using Model-Based testing to test MCP Instance-Specifications



Academic advisor: Michael Kirkedal Thomsen
<m.kirkedal@di.ku.dk>

Submitted: November 13, 2019

Using Model-Based testing to test MCP Instance-Specifications

Søren Lund Hess-Petersen - pws412
DIKU, Department of Computer Science,
University of Copenhagen, Denmark

November 13, 2019

MSc thesis

Author:	Søren Lund Hess-Petersen - pws412
Affiliation:	DIKU, Department of Computer Science, University of Copenhagen, Denmark
Title:	Using Model-Based testing to test MCP Instance-Specifications /
Academic advisor:	Michael Kirkedal Thomsen <m.kirkedal@di.ku.dk>
Submitted:	November 13, 2019

Abstract

Dansk Resumé

Preface

This Master's thesis is submitted in fulfilment of the master programme in Computer Science at the University of Copenhagen for Søren Lund Hess-Petersen.

Contents

Introduction

1

The Maritime Connectivity Platform [1] (MCP) is, as the name suggests a platform that connects maritime services, functioning as a common infrastructure by offering safe and reliable information exchange between various maritime actors. A particular way of utilizing this is that a user of the MCP is able to develop his or her own software, and distribute it across the world. Such digital maritime components, however needs to be thoroughly tested, which often is not adequately done through simple unit testing. Thus, software components, distributed through the MCP need a thorough test-suite, for which the obvious solution is to implement model-based testing. The MCP has provided specifications, which describe rules that software needs to adhere to. These specifications will be used to create the model, that will verify the software. The specifications are provided in `xml`-format, and are, as of now, not used in any formal degree. Ideally, the provided specifications should be used to automatically generate models, which can be used to verify that a certain piece of software adheres to the specification.

1.1 Description of the Project

In this project, QuickCheck [2] will be utilized in order to verify that instances fulfils the requirements set by specifications in the MCP. QuickCheck is a library, which allows for model-based testing of software. Originally, QuickCheck was written in Haskell, but has since been extended to more than 50 programming languages. The idea behind this method is, as the term *model*-based testing suggests, to create models which describe the properties that the test-cases need to reflect. This means that in stead of conducting unit-tests, the model should reflect what needs to happen with an arbitrary input- and function-combination, and ideally catch every special case that either has not been accounted for in the model or in the analysed software. Once an accurate model is created further testing can be streamlined, as a simple test can affirm an entire aspect of the tested software, in stead of just *one* particular example. The goal of the project is to create an automatic test suite for the MCP, which performs better than a unit-based test suite. In the long run, such a test suite has the capacity to increase reliability, efficiency

and ultimately make way for better distributed, and more uniformly created software components in the Maritime Connectivity Platform.

1.2 Learning Objectives

- Utilizing QuickCheck in creating viable models, that describe the specifications, provided in MCP specifications.
- Interpreting MCP specifications in order to generate relevant tests.
- Parsing MCP specifications in order to generate relevant models.
- Applying property-based testing of functions and specifications in a maritime environment.

1.3 Scope

1.4 Limitations

1.5 Structure

- **Chapter 2: Background**
  This chapter describes the techniques, theories, and components that are used or analyzed throughout the thesis.
- **Chapter 3: Analysis**
- **Chapter 4: Work/Design**
- **Chapter 5: Results**
- **Chapter 6: Discussion**
- **Chapter 7: Conclusion**

Background

2

2.1 The Maritime Connectivity Platform

The Maritime Connectivity Platform (MCP), which is formally known as the Maritime Cloud is a platform, developed by EfficienSea2 [3], which is led by the Danish Maritime Consortium. MCP is a communication framework, that is to ensure efficient, reliable and secure communication, and exchange of information in the maritime sector. The goal of the platform is to connect maritime stakeholders with maritime information services.

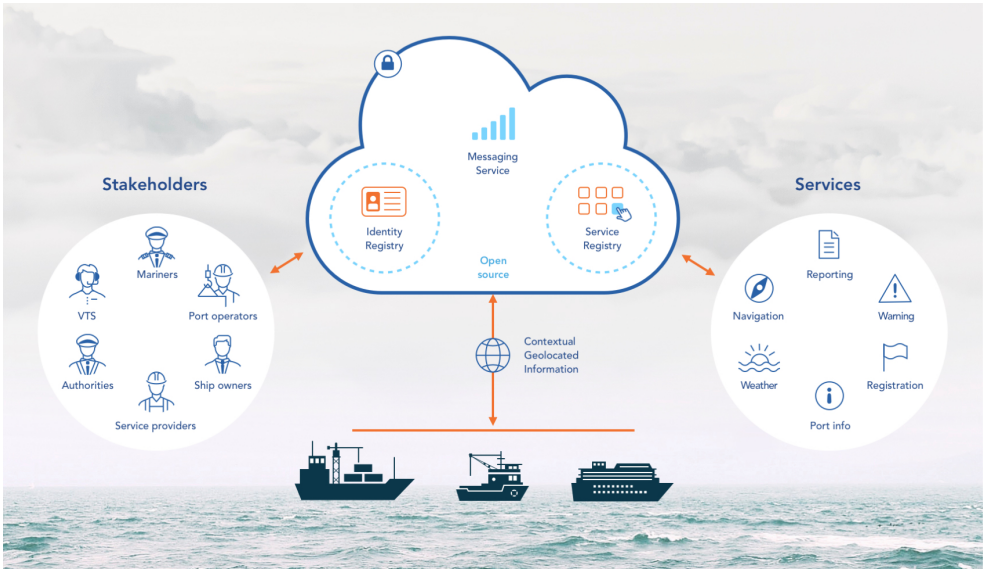


Figure 2.1: Diagram, describing the structure of MCP. [3]

A high-level diagram, describing the structure of the MCP can be seen in Figure 2.1. Here it is shown that maritime stakeholders are connected with the maritime services through the Identity Registry, the Service Registry and the Messaging Service.

2.1.1 Identity Registry

Here the relevant information regarding maritime stakeholders are stored. This information needs to be authorized and stored at a safe location in order for the security of the MCP to be sufficient. The Identity Registry on the MCP is equivalent to the Central Person Registry of a country. The Identity Registry is vital to the MCP in it ensures the solution's authenticity, integrity, and confidentiality. Maritime stakeholders are, through the Identity Registry provided with a single login to all Maritime Services. [3]

2.1.2 Service Registry

The Service Registry acts to Maritime Services as the Identity Registry acts to the Maritime Stakeholders. Here all Maritime Services are registered and stored. The Service Registry holds both commercial and non-commercial, as well as authorized and non-authorized services, either free of charge or for a fee. The Service Registry is comparable with the App Store or Google Play in that it distributes services of all kinds to registered users. [3]

2.1.3 Messaging Service

"An information broker that intelligently exchanges information between communication systems connected to the platform, taking into account the current geographical position and communication links available to the recipient." [3]

2.2 Monadic Parsing

Text parsing is a functionality, supported in virtually every programming language.

2.3 Property-Based Testing

Property based testing, also known as Automation Testing, is the principle of testing properties of code, rather than instances, as is done in manual unit testing. Unit tests are fast and easy to set up and execute, however, in most instances they only provide coverage to a certain degree. Depending on the complexity of the given program, most programmers can come up with tests, covering most of the given program, however many edge-cases are unintuitive and would require extensive testing to determine manually.

Property-based testing solves this problem by setting up test cases that take into account mathematical models, describing the desired behavior of the code. Once such a test case has been created, semi-random execution instances can be run to the point of literal exhaustion, and thus a relatively small program can test for thousands of occurrences at once.

An example of this can be seen in Listing 2.1, that describe a snippet of Haskell quicksort code. A property of quicksort is that a list, sorted once is equivalent to a list, sorted twice, which is what the function `prop_idempotent` validates.

```
import Test.QuickCheck
import Data.List

qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort lhs ++ [x] ++ qsort rhs
    where lhs = filter (< x) xs
          rhs = filter (>= x) xs

prop_idempotent xs = qsort (qsort xs) == qsort xs
```

Listing 2.1: Example of Property-Based Testing. [4]

2.4 Model-Based Testing

The principle of Model-based testing is to check behavior of software against predictions made by a model. Behavior can be described in a variety of manners, including data flow, control flow, dependencies, decision trees/cycles, and state transition machines. Model-based testing is good at describing the behavior of a system, when it reacts to a specific action, which is determined by another specific model. Using this technique, the behavior of a system can easily be determined and validated. Currently there are two main types of model-based testing-techniques:

- **Serial model building- and testing:**
  This way of implementing model-based testing involves predefined models, upon which a number of tests are executed. Creating the model ahead of execution allows the tester to implement an additional layer of complexity to the created models. Depending on the amount of models, this is, however, a significantly slower process than the alternative, in that each model requires a similar amount of time to construct. Figure 2.2 provides a high-level diagram, describing this principle, using arbitrary units of measurement along it's x- and y-axis.

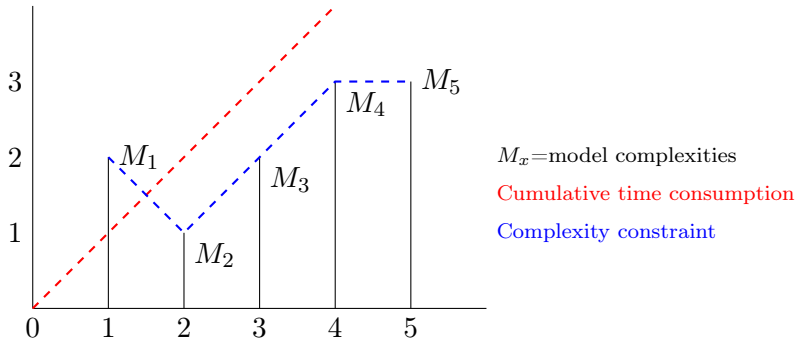


Figure 2.2: Time consumption and complexity restraint of serial model building and testing.

Figure 2.2 shows the time consumption increasing for every model built, while the model complexity is directly matching the requirement, set by each specific model.

- **Sequential model building- and testing:**
  This way of implementing model-based testing involves on-the-fly model creation- and testing. Here a program is designed to take in arguments, describing different model behaviors, upon the basis of which the program tests the models immediately after they are generated. Compared with the implementation described above, this technique scales to a much better degree, as the program will only need to be written once in order to create a virtually infinite amount of models. The created models are, however, constrained to the lowest common complexity level, shared across all of the generated models. Figure 2.3 provides a high-level diagram, describing this principle, using arbitrary units of measurement along it's x- and y-axis.

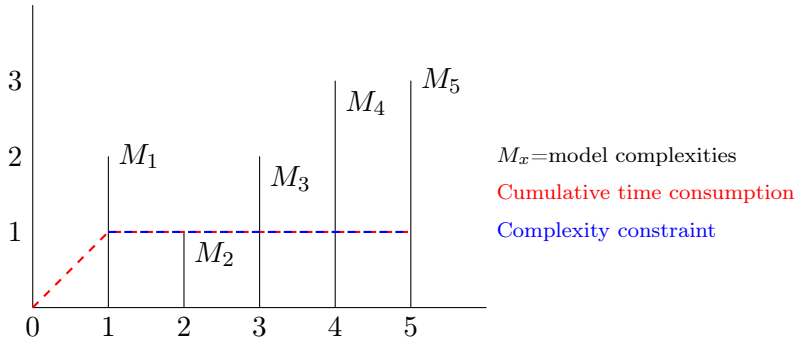


Figure 2.3: Time consumption and complexity restraint of sequential model building and testing.

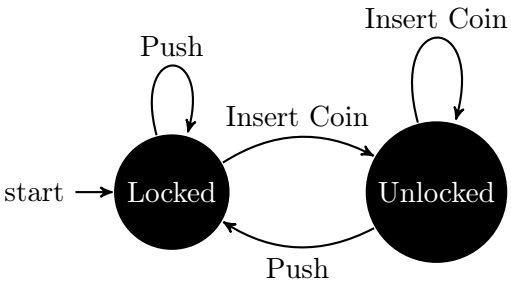


Figure 2.4: Example of a finite state machine, describing a turnstile. Furthermore, this figure describes Table 2.1.

Figure 2.3 shows that the time consumption halts to a stop when one model is built with the drawback that the model complexity constraint is set to the lowest common model.

2.4.1 Finite State Machines (FSM)

A finite state machine is, as the name suggests, a mathematical machine, consisting of a collection of states. An arbitrary FSM has a start state along with a collection of states that are accessed through various status- and/or input-combinations. An example of this can be seen in Figure 2.4 as well as Table 2.1. These describe the control flow of a turnstile, which allows for one pass through, whereafter it prompts for a coin before allowing another pass through.

State	Action	Next State	Output
Locked	Insert Coin	Unlocked	Unlocks turnstile, allowing passage.
Locked	Push	Locked	Blocks passage.
Unlocked	Insert Coin	Unlocked	Returns coin.
Unlocked	Push	Locked	Allows passage, locks.

Table 2.1: Example of a finite state machine, describing a turnstile. Furthermore, this table describes Figure 2.4.

2.4.2 State Charts

2.4.3 Unified Modeling Language (UML)

Analysis

3

In this chapter, I will present and analyze the data made available by the MCP in the three documents *E2 - NW-NM DMA Service Instance*, *E2 - NW-NM REST Service Technical Design*, and *E2 - NW-NM Service Specification*, which represent all data, made available to me. Furthermore, I will explain the necessity of testing the MCP, as well as present a description of a favorable model structure.

3.1 Data

3.2 Testing the MCP

The core component in the MCP is restructuring and streamlining maritime software sharing in a manner that can be done the world over, and the very nature of this statement dictates that the platform must be highly scalable. This adds the necessity of running quality-checks on all of the maritime services that are uploaded to the platform. To accommodate this issue, model-based testing immediately seems like the obvious solution, as this technique covers most of the required desired functionality. If implemented satisfyingly, a model-based testing suite for the MCP would be able to

1. Present or verify behavior of maritime services.
   There are many obvious common behavioral traits of maritime services, such as adding ships and maritime stakeholders, as well as removing them, however other behavioral patterns will often vary to the point of lowered manageability. A model, describing the maritime service in question will provide a clear and undeniable description of the service's behavior.

2. Visualize functionality of maritime services.
   Just as well as behavioral traits, functionality will differ greatly from one maritime service to another, and therefore it is very useful for a model to visualize said functionality, as well as verifying that it works as intended.

3. Present the structures of maritime services.

4. Increase reliability and efficiency of maritime services.
   Through correct implementation of points 1-3 it is possible to elevate the reliability and efficiency of the maritime services found on the MCP. This is due to the same reasoning that all testing is conducted on the basis of: the need for safe, consistent, and correct code.

As stated in Section 2.4 there are two main types of model-based testing techniques: serial and sequential model building-and testing.

As mentioned in Section 2.4, Model-based testing is a great tool for verifying the behavior programs. It can be implemented in a variety of ways, including automated, semi-automated, and manual.

3.3 Model Structure

3.4 Model-Based testing of MCP

3.5 Issues

Work/Design

4

4.1 Parser Grammars

4.1.1 ServiceSpecificationScema

4.1.2 ServiceDesignSchema

4.1.3 ServiceInstanceSchema

4.1.4 General Grammar

4.2 Implementation in the MCP

4.3 Technical Implementation

4.3.1 Executing Instructions

4.4 Testing

4.5 Experiments

Results

5

Discussion

6

Conclusion

7

Bibliography

[1] Maritime Connectivity Platform, https://maritimeconnectivity.net/

[2] QuickCheck, http://hackage.haskell.org/package/QuickCheck

[3] EfficienSea2's website
https://efficiensea2.org/solution/maritime-connectivity-platform/

[4] Real World Haskell, Bryan O'Sullivan, Don Stewart, and John Goerzen, Chapter 11
http://book.realworldhaskell.org/read/
testing-and-quality-assurance.html

Appendix A