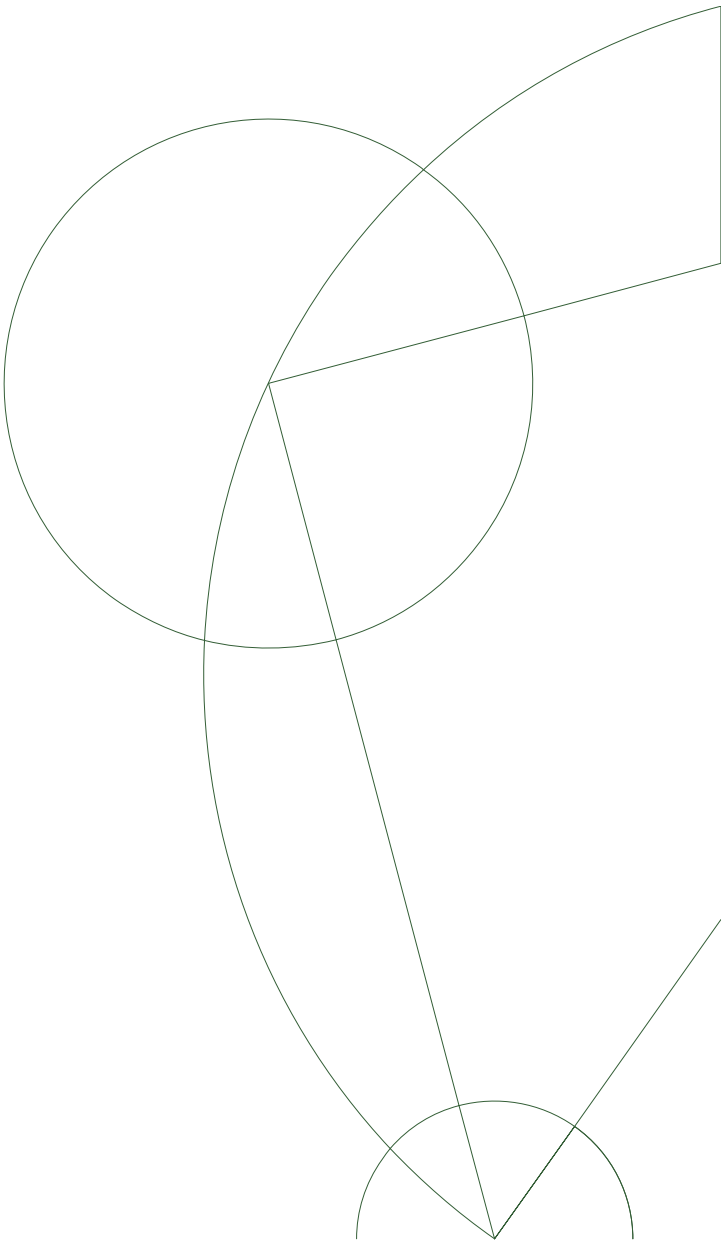
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

MSc thesis

Søren Lund Hess-Petersen | pws412

Using Model-Based Testing to test MCP Instance-Specifications

Rage Against the Finite State Machine
Boats on Parade



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

Submitted: February 28, 2020

Using Model-Based Testing to test MCP Instance-Specifications

Rage Against the Finite State Machine
Boats on Parade

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

February 28, 2020

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 / Rage Against the Finite State Machine Boats on Parade
Academic advisor:	Michael Kirkedal Thomsen <m.kirkedal@di.ku.dk>
Submitted:	February 28, 2020

Abstract

TODOWRITE

Dansk Resumé

TODOWRITE

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, need to be thoroughly tested, which often is not done adequately 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 in turn, can be used to verify that a certain piece of software adheres to the specification.

1.1 Description of the Project

In this project, model-based testing will be utilized, in order to verify that instances fulfills the requirements, set by specifications in the MCP. Model-based testing is a technique that allows for verification of overall *model behavior*, rather than *function correctness*. The idea behind this method is, as the term *model*-based testing suggests, to create models which describe the properties that given test-cases need to reflect. Once an accurate model is created further testing can be streamlined, as a simple test can affirm an entire aspect of the tested software. The goal of the project is to create both a manual and an automatic model-creating module for the MCP. The manual model creator will be run by the execution of commands, each creating different aspects and behaviors of the corresponding model, which, ultimately can be used for model-based testing. The automatic model-creator module, however, will draw information from the MCP specification xml-files, creating equivalent models, based on user-defined input. In the case of automatic model creation, the idea is that the originators of the maritime service will be

the ones to author the xml-file, outlining the specifications, and in extension hereof creating the maritime models.

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

This thesis will go into means of creating true and fair representations of models, describing maritime services. All tools, required to create models, both manually and automatically, will be analyzed, summarized and utilized, which will be described in the report.

1.4 Limitations

Methods of improvement upon both manual and automatic model-creation modules will be present, however not all will be utilized. This includes monadic parsers, and model verification through quickcheck, however in future work, these topics are relevant to be elaborated upon.

1.5 Structure

The remainder of the thesis will consist of 7 chapters, covering the following topics:

Chapter 1: Introduction
This chapter provides a description of the problem statement, the learning objectives and the project itself, as well as containing this list.

Chapter 2: Background
This chapter describes the techniques, theories, and components that are used or analyzed throughout the thesis.

Chapter 3: Analysis
This chapter examines potential solutions to the problem statement, and

how to implement them into the MCP. This chapter contains advantages and disadvantages to each presented solution, which are summed up at the end of the chapter.

Chapter 4: Work/Design
This chapter presents the work that went into implementing the desired functionality in the MCP.

Chapter 5: Results
This chapter includes the results that came from the implementation, described in the previous chapters.

Chapter 6: Discussion
This chapter discusses the results, described in Chapter 5.

Chapter 7: Conclusion
This chapter concludes the thesis and compiles a short summary, highlighting central takeaways.

Background

2

2.1 The Maritime Connectivity Platform

The Maritime Connectivity Platform (MCP), formally known as the Maritime Cloud, is a platform, developed by EfficienSea2 [3], which is led by the Danish Maritime Authority. 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. 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.

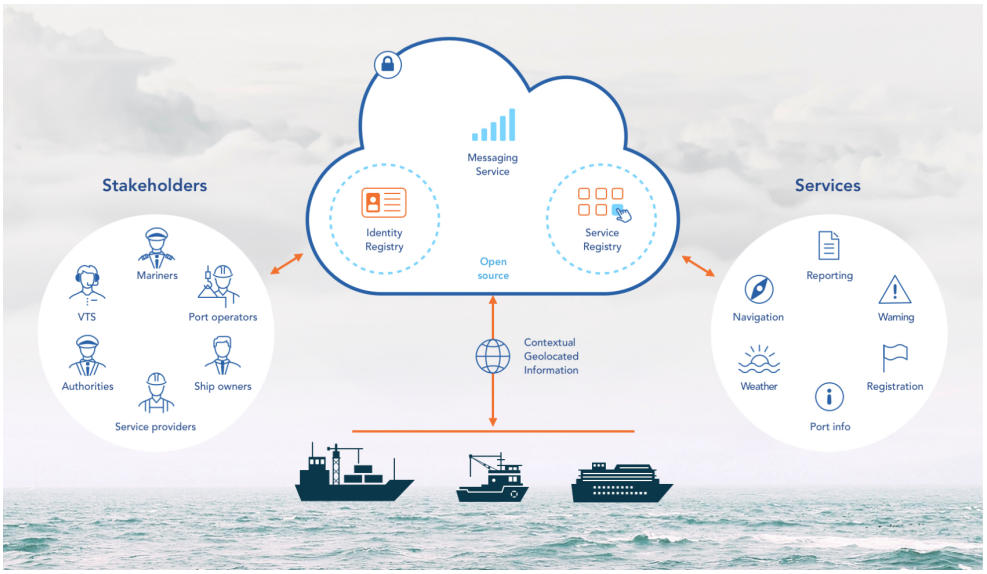


Figure 2.1: Diagram, describing the structure of MCP [3].

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 that 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

What the Identity Registry provides for the Maritime Stakeholders, the Service Registry provides for the Maritime Services. Here all Maritime Services are registered and stored. The Service Registry provides 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

The messaging service in the MCP, enables the exchange of information across the different platforms, thereby linking their systems to each other. The messaging service accounts for the available means of communication, as well as the geographical coordinates of the destination and origin of the information, being sent [3].

2.2 Parsing

Parsing is a widely known tool in computer science. The term expresses a function, taking a string of input, from which it yields a parse tree that it constructs when applying the specific parser logic upon the input string. The parser performs a lexical analysis, meaning that it converts sections of the string to tokens, that are ultimately easier to handle by an interpreter. When using Erlang to define parsers, a library family known as parser combinators is very popular, in that it allows the programmer to embed domain specific language directly into the parser. The concept of combinatory parsing is using one or more of these libraries of higher-order functions, known as parser combinators. Parser combinators take in parsers of any kind imaginable as input and returns a new set of parsers as an output.

2.2.1 Monadic Parsing

Before 1995, implementation of top-down parsing, using parser combinators ran in and used both exponential time and space complexity, when applied to ambiguous, context free grammars. At that time Frost and Szydlowski demonstrated that memoization can enhance parser combinators in order to optimize time complexity to polynomial [6]. One year later through the use of monads, Haskell-specific parser combinators were reconstructed by Frost which introduced systematic and correct threading of memoization throughout execution.

2.3 Interpretation

In solutions, utilizing parsers, an interpreter is often introduced to handle the result that the parser returns, and for this reason, interpreters share the same syntax that is used in their corresponding parsers. The job of an interpreter is to execute the parsed commands, and therefor their implementation can vary to an even higher extend than that of a parser. Often-times a parser-interpreter combination is used to identify, generate, as well as execute domain-specific languages. Since automatic utilization of domain-specific language is an integral part of the problem-statement, using a parser- and interpreter-combination is straightforward to the solution.

2.4 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.

```
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
```

Figure 2.2: Example of Property-Based Testing [4]

.

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 discover 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 2.2, which describes a snippet of Haskell quicksort code. A property of quicksort is that a list, sorted once is equivalent to that list, sorted twice, and this is what the function `prop_idempotent` validates.

2.5 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 , is a way of implementing model-based testing and involves predefined models, upon which a number of tests are executed. Creating the tests manually ahead of execution allows the tester to implement an additional layer of complexity to the created models. Furthermore, implementing this strain of model-based testing allows for custom-tailored testing, suited to fit issues, unique to the corresponding model. 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. Depending on the developer, as well as the implementation techniques of models, a relatively high leaning curve may be introduced, leading to exclusions of parties, not qualified to take on this task. Figure 2.3 provides a high-level diagram, describing this principle, using arbitrary measurement units of time consumption, skill level, and model complexity along its y-axis, and amount of models along its x-axis.

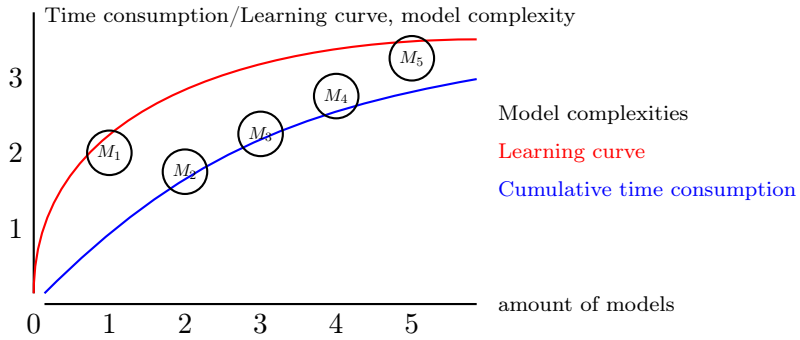


Figure 2.3: Time consumption and learning curve of serial model building and testing.

Figure 2.3 shows the time consumption increasing for every model built, while the learning curve initiates very steeply, however diverges, as the creator of the models learns the used techniques.

Sequential model building- and testing , is a way of implementing model-based testing, that 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. Input, containing all necessary information that would otherwise be entered manually is, however, still needed, and thus scalability is not completely eliminated.

Figure 2.4 provides a high-level diagram, describing this principle, using arbitrary measurement units of time consumption, skill level, and model complexity along its y-axis, and amount of models along its x-axis.
Like Figure 2.3, Figure 2.4 introduces a learning curve, as the user will need to become familiar with the input needing to be specified, in order for the sequential model creation to start, however, this is severely more gradual, while also featuring a lower end-point than the one seen in Figure 2.3. The cumulative time consumption is also lower, compared to serial model building, due to the fact that this process will run semi-automatically, only needing input from a user, before handling all building-related tasks.

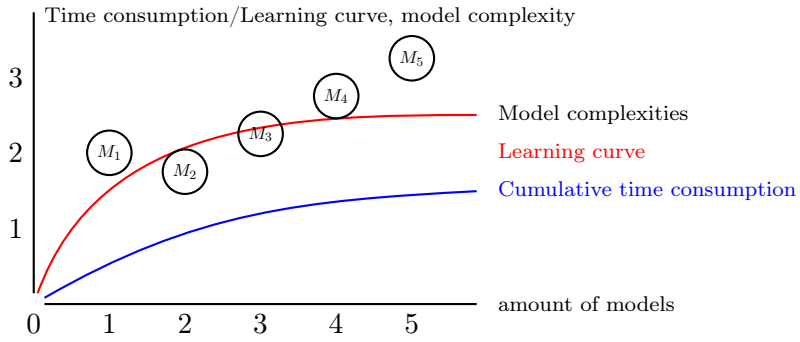


Figure 2.4: Time consumption and learning curve of sequential model building and testing.

2.5.1 Finite State Machines (FSM)

A finite state machine is, as the name suggests, a mathematical machine, consisting of a collection of states and actions. 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 2.5, which describes the flow of a turnstile, which allows for one pass through, after which it prompts for a coin before allowing another pass through.

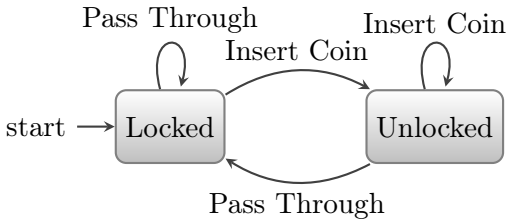


Figure 2.5: Example of a finite state machine, describing a turnstile. Furthermore, this figure describes Table 2.6.

In addition to describing the concept of states in a FSM, Figure 2.5 and Table 2.6 also displays a collection of actions. An action is an operation that is performed along with a corresponding state, which can potentially change that state. The described case, features the two actions 'Insert Coin' and 'Pass Through'. The action 'Insert Coin' alters the state of the turnstile, only if performed when the state of the turnstile is 'Locked', as it is not possible to unlock an already unlocked lock. Similarly, the action 'Pass Through' only changes the state if the turnstile is in its unlocked state before performing the action, as passing through a locked turnstile is not possible.

Table 2.6, seen below displays the relationships between the actions, states and output, produced when interacting with the turnstile, using its intended functionality. The table features the four columns, with 'State' describing the state before execution of the 'Action' column. 'Next State' describes the state after the action has taken effect, and the 'Output' column describes the flow of events, leading to the next state. Note that the Output column features event flows that do *not* lead to a change in state, as well as outcomes that do.

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 2.6: Example of a finite state machine, describing a turnstile. Furthermore, this table describes Figure 2.5.

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 and potential methods of generating said models.

3.1 Data

To provide a holistic description of a service instance, three documents are necessarily provided. Below are the descriptions of the documents with the service instance `E2 - NW-NM DMA` as an example.

E2 - NW-NM DMA Service Instance
The purpose of this service instance document and its xml-defined counterpart is to describe a DMA instance of the REST-based technical design of the MW- NM service specification, according to the guidelines given in the Service Description Guidelines.

E2 - NW-NM REST Service Technical Design
The purpose of this service technical design document and its xml-defined counterpart is to describe a REST-based technical design of the MW-NM service specification, according to the guidelines given in the Service Description Guidelines.

E2 - NW-NM Service Specification
The purpose of this service specification document and its xml-defined counterpart is to provide a holistic overview of the MW-NM service and its building blocks in a technology-independent way, according to the guidelines given in the Service Description Guidelines.

Of the three documents, only the last, `Service Specification` is to be used to describe specifications of the service, and as such only this will be studied closer.

3.2 Service Specification grammar

The service specification document is constructed using a general xml-format. The majority of the parser grammar of the document can be seen in Figure 3.1 as well as in Figure 3.2, while the latter is in a severely reduced and generalized form. A minority of the grammar has been cut from Figure 3.1 as it was deemed irrelevant, however the parser grammar in its entirety can be seen in appendices, A.2 and A.3.

As mentioned above, the service specification is to be used to verify the behavior of the maritime service from a technical stand point. At the moment, the xml-document provides a technical description of its corresponding maritime service, but the most commonly used expression method is free text. As this is a very non-technical design choice, a large portion of the obvious technical advantages provided by the xml-format, is lost.

As it is possible to utilize free text though the use of natural language processing [5], no modifications are strictly necessary, however, even with utilization of such or similar methods, implementation hereof would be unstable in terms of usability. This is due to the fact that free language formulation varies at an unforeseeable degree, and as such creating uniform models based on this would add a great layer of complexity.

For a more sustainable solution to the problem, see Section 4.1.

The xml-version of the service specification can be found in Appendix A.3.

3.3 Testing the MCP

The core principle in the MCP is to restructure and streamline 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.

```
ServiceSpecificationSchema ::= specifications

specifications ::= spec specifications
    | ϵ

spec ::= name
    | status
    | ID
    | version
    | description
    | keywords
    | isSpatialExclusive
    | authorInfos
    | requirements
    | serviceDataModel
    | serviceInterfaces

authorInfos ::= authorInfo authorInfos
    | ϵ

authorInfo ::= aSpec authorInfo
    | ϵ

aSpec ::= ID
    | name
    | description
    | contactInfo

requirements ::= requirement requirements
    | ϵ

requirement ::= rSpec requirement
    | ϵ

rSpec ::= ID
    | name
    | text

serviceDataModel ::= definitionAsXSD

serviceInterfaces ::= serviceInterface serviceInterfaces
    | ϵ

oSpec ::= name
    | description
    | returnValueType
    | parameterTypes
```


Figure 3.1: Snippets of full parser grammar of Service Specification Schema. (Found in appendices A.2, and A.3)

```
ServiceSpecificationSchema ::= specifications

specifications ::= spec specifications
    | ϵ

spec ::= specifications
    | spec
    | ϵ

spec ::= string
```

Figure 3.2: Reduced parser grammar of Service Specification Schema.

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.
   This trait will be used to visualize the structural components of maritime services. Just as point one and two, this functionality will be useful for creating a quick and clear projection of how the maritime service behaves.

4. Increase reliability and efficiency of maritime services.
   Through correct implementation of points one, two, and three, 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 2.5 there are two main types of model-based testing techniques: serial and sequential model building-and testing, and as these two techniques both provide different advantages utilizing either one will be a trade-off.

3.4 Example Model Structure

Throughout this report, I will use a map-requesting service as an example. In this example the maritime service will act as an intermediate, broking between a ship and a company, where the map is being held. The ship will have to submit authentication information to the maritime service in order to get a the requested map. The model, which has been built upon the example protocol has been illustrated in Figure 3.3. The lines numbered 1-6 illustrate interaction between the three entities `Ship`, `Service`, and `Company`, where *1 and *2 respectively, illustrate a submitting of authentication information to the service, and the service accepts the information and sends a response. *3 and *4 illustrates the service's request of the requested map, and its subsequent receiving hereof. *5 illustrates the forwarding of requested information to the ship. Figure 3.4 represents an FSM, illustrating the `ship` entity in Figure 3.3.

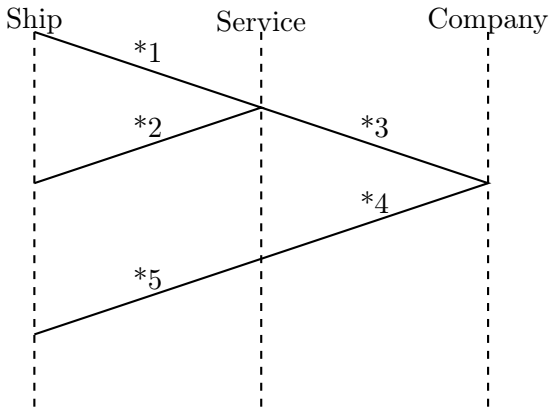


Figure 3.3: Early draft of a model example, where a ship requests information from a service.

The FSM starts in the `No map loaded` state, and has the three available actions, `trash map`, `request map`, and `await response`. Performing `trash map` or `await response` in the initial state will yield no results, as no map is available to be trashed and no response is on the way. If `request map` is performed, the state will be changed to the intermediate state `awaiting map`, where `await response` is the only action that will change the state, as no map can be trashed, and requesting another map will change the state to the same, but where a different map is being waited for. The third and final state `map loaded` will allow the actions `trash map`, which will change the state to `no map loaded` and `request map`, which will, respectively, trash the loaded map, and request a new one, changing the state to `awaiting map`.

Figure 3.5 represents an FSM, illustrating the `service` entity in Figure 3.3. The FSM has the two states, `idle` and `active`, the former being the initial state. Whenever the entity receives a valid response the state changes to the `active` state, and when the request has been properly handled, the state changes back to `idle`.

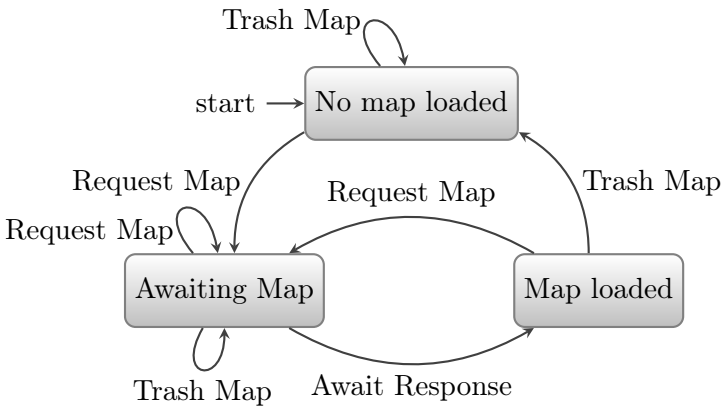


Figure 3.4: FSM, describing the `Ship` entity in Figure 3.3.

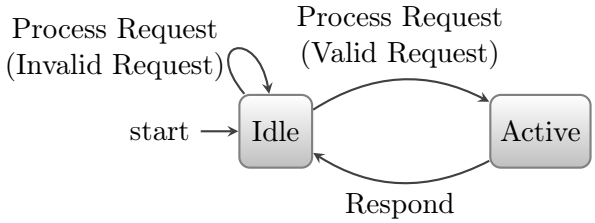


Figure 3.5: FSM, describing the `Service` entity in Figure 3.3.

3.5 Uses of model-based testing of the MCP

The main advantage that comes from model-based testing is the large variety in testing algorithms it produces. If a model is to be machine readable the techniques, listed below can be implemented fully automatic, and if not, they can still be utilized manually. The following techniques can be applied to true and fair maritime service-models in the MCP:

Finite state machines

As mentioned in Section 2.5.1, a FSM is made up of states, and the ability of a simulating program to change state through a collection of actions. Executable paths are determined, which can be used to create test cases, as well as illustrate the behavior of the system. Furthermore FSMs are closely related to the further techniques, that will be explained below.

Proving theorems

Automatic theorem proving is a technique originally produced for proving logical formulas, which is a concept that can be rewritten to suit model-based testing. In order to create test cases, system behavior is assigned to equivalence classes. Following this step, predicates can be formed from equivalence classes and logical consequences of the equivalence classes.

Symbolic execution

Much in the likes of an FSM, a symbolic execution of a program or system will generate executable paths, along with symbolic values, describing how they can be accessed. A symbolic execution engine will analyze possible variations of input in order to create a map of the software in question. Said map can subsequently be used to better understand, test, and develop the software in question.

Model checkers

Model checkers, also known as property checkers, are methods of determining if an FSM meets a specific set of requirements. The model checkers detect if requirements are met, either by determining examples or counterexamples. When a path, representing either an example or a counterexample is discovered it is logged as a *witness*, which can be mutated upon when generating test cases.

Markov chains

Markov chains are named after the Russian mathematician Andrew Markov, and are also known as usage models. A Markov chain, or usage model, is made up of two primary components; firstly an FSM, which represents every available action of a given system, and secondly an operational profile, which is a statistical indicator of what has been or will be used. The FSM is used by the operational profile in order to generate the statistical analysis, and the operational profile is used by the FSM to derive operational tests.

The scope of which features the above techniques will be able to test, will be deemed by the actual maritime service model, however, the expansive array of methods will cover maritime services extensively.

3.6 Use-Cases

In this section, I will present the utilization of the desired tool, that is implemented throughout Chapter 4 using techniques described in Chapter 3. This entails three examples, described in Figures 3.6 through 3.9, and visualized in Figure 3.10. Figures 3.6 and 3.7 describe intended use of the implementation, while Figure 3.9 describes unintended use of the implementation. Actor/system-interaction is described in Figure 3.10. The steps described in

Use Case	System verifies model, first iteration
Actors	Service Instance Designer (SD)
Basic Flow	1. SD will submit a service specification. 2. SD submits all information as per the old service specification syntax. 3. SD submits all information to generate maritime model. 4. Model concept is tested and accepted. 5. SD submits the service instance specification.

Figure 3.6: First example of correct use of the solution.

Use Case	System verifies model, second iteration
Actors	Service Instance Designer (SD)
Basic Flow	1. SD will submit a service specification. 2. SD submits all information as per the old service specification syntax. 3. SD submits all information to generate maritime model. 4. Model concept is tested and rejected. 5. SD repeats step 3. 6. Model concept is tested and verified. 7. SD submits the service instance specification.

Figure 3.7: Second example of correct use of the solution.

Figures 3.6 and 3.7 are similar up until 4, where one example accepts, while the other example rejects the model. As the Service Instance Designer produces a valid model in Figure 3.6, the fact that the system is used correctly is trivial, and while the initial model is rejected, based on the desired behavior in Figure 3.7, the final result is a valid model concept, and thus the software has been used as intended.

In Figure 3.8 the user wishes to surpass the model-testing system, and successfully avoids warnings by creating a dummy model and validates this against a set of loose tests. As the model-generating occurs before service instance development- and implementation, there is no way of knowing if the software that is developed is similar in any way to what is indicated by the maritime model.

Use Case	User creates dummy model
Actors	Service Instance Designer (SD)
Basic Flow	1. SD will submit a service specification. 2. SD submits all information as per the old service specification syntax. 3. SD submits information to generate simple dummy model. 4. Model concept is tested and verified. 5. SD submits the service instance specification.

Figure 3.8: Example of incorrect use of the solution through false positive.

In the case of Figure 3.9 a model is created, using the system, similar to above, however, the result of the checked input is ignored by the user. This should still be possible, as the limitations of the system, should not be projected onto the created software, but in some instances, it could also lead to invalid model-concepts being iterated upon further.

Use Case	User ignores model warnings
Actors	Service Instance Designer (SD)
Basic Flow	1. SD will submit a service specification. 2. SD submits all information as per the old service specification syntax. 3. SD submits all information to generate maritime model. 4. Model concept is tested and rejected. 5. SD submits the service instance specification.

Figure 3.9: Example of incorrect use of the solution through ignored warnings.

Figure 3.10 describes the use of the system, which can lead to the instances, seen in Figures 3.6, through 3.9. As denoted, the user has the ability to create from which feedback, and the model itself, can be continuously viewed. When the user is satisfied with the model it can be submitted. To allow for maximum flexibility in the system, as little constraints are placed on the user, when interacting with the model creator, which is what allows for incorrect use hereof, as seen in Figure 3.9. No precautions can be done, however, which will prevent incorrect use of the model, as seen in Figure 3.8.

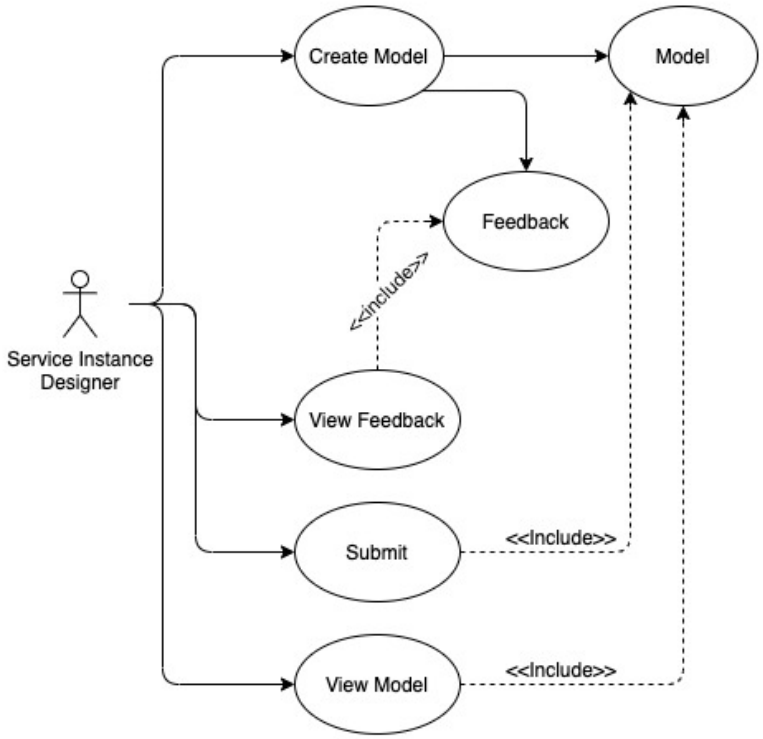


Figure 3.10: Use case diagram describing Figures 3.6, 3.7, 3.8, and 3.9.

3.7 Summary: Advantages and Disadvantages

Issues will inevitably present themselves with a project such as this, however the scope and nature of these issues will vary with each approach possible. As different approaches offer tailored advantages to circumvent correlating unique concerns, another set of unique concerns will necessarily present themselves. In this section I will sum up the approaches presented throughout Chapter 3, along with their corresponding arising advantages and disadvantages.

3.7.1 Manual model-creation

Advantages

- Manual model creation will have a shorter initial implementation time, as an arbitrary single model is faster to build than its auto-model-generator counterpart.
- Built maritime service models can be fully customized at once in order to describe all aspects of its maritime service counterparts.
- Having customized each maritime service model to its corresponding maritime service counterpart allows for an equally customized test suite. Following this, the techniques, described in Section 3.5, will allow for exhaustive tests via manual model-creating.

Disadvantages

- The time consumption of manually creating each model will expand near-linearly with each maritime service, that is submitted, and subsequently needs a maritime service model.
- Having each maritime service model being created individually and manually will inevitably introduce much diversity both in the sense of the available standard actions and implementation style of the model.
- The complexity as well as way of implementing testing techniques explained in Section 3.5 varies to an extent, even without consideration of non-uniform models. Elaborating on the case that each maritime service model differs both in purpose, implementation style and exhaustiveness, creating test suites similar to those described in Section 3.5 for each new maritime service, will become an extensive project.

3.7.2 Automatic model-creation

Advantages

- The first and foremost advantage of automatic model generation is the scalability it provides. This comes as the implementation of *one* maritime service model generator module, ideally, will be able to automatically assemble each maritime model without further interference, as opposed to manual model generating.
- As an automatic model generator abides by a set collection of rules and metrics, and would create models following the instructions of an xml specification-file, generated models will never deviate from the uniformity of the generator.
- The uniformity in the generated models will further show advantage, as this will streamline test suite generation either because of the fact that similar models promote similar manual test suite generation, or because that it allows for near-total automation.

Disadvantages

- The obvious disadvantage that comes with automatic model generation, is the lack of customization in models. As the generator should handle all models that can be described through the xml specification files, it needs to be generalized, which in turn means less specific models, which lowers the ability to custom-fit each model to its maritime specification counterpart.
- Another inevitable downside of having automatically generated - however less specific - models is the fact that test suites can become less specific, and in extension hereof less exhaustive of the maritime service.

To sum up, both methods will be advantageous, as they will each allow for extensive testing, with the former being more custom-tailored and so a lot more time consuming in the long run, and the latter being more generic, but much ultimately less time consuming in the long run.

Work/Design

4

In this chapter I will describe the work I have done in order to implement model-based testing into the MCP. The workload that is contained in the implementation and description herein includes a model-building module, which is able to simulate given examples of maritime protocols. Furthermore, I will update the structure of the xml specification files, based on which I will introduce a parser/interpreter implementation, thus allowing me to automatically generate maritime models, equivalent to the one that I have created manually.

4.1 Updated Data

As discussed in Section 3.1, in order to create uniform models upon which to execute tests, the structure of the xml-specification files needs to be altered in a manner that adds uniformity to the xml-specification files. This rules out the use of natural language for reliable model-generation.

One way to implement these changes is to make the user able to assign variables, based on a fixed collection of available options. I will now present three fields, which are designed to be used in order to maximize uniformity in model generating, as well as give maximum flexibility in future model design.

Entity

An entity is the most basic component of the models, without which, they would be devoid of any data. These describe the physical real-world objects, which the model is to simulate, and these are also the subjects of all transactions, which occur throughout execution of a model. At the moment support for three entities exist, however, implementation of support for additional entities is possible. Multiple instances of each entity type is allowed in a model.

Ship

If comparing the MCP to the App Store or Google Play, a ship would undertake the role of a user. The user can be the one utilizing the maritime service or symbolize other users, however, no distinctions are naturally implemented between those two types.

Service

The Service entity represents a maritime service, which is being modeled. If multiple instances of a service entity occurs, this will correspond to cross app interaction, using the same comparison as above.

Company

Company entities serve as nodes, which hold data or similar goods, which can be desired by ship entities for a variety of reasons. If multiple company entities exist at one time, they will not necessarily need a service entity to connect them.

Relation

A relation symbolizes a bond between two entities. In the case of Figure 3.3, relations can be seen between `Ship` and `Service`, as well as `Service` and `Company`.

A ship entity can share a relation with service entities, as they would have no way of communicating with company or other ship entities without any prior software. In the case that a ship communicates with another ship, it will need an intermediate service entity to handle communication.

A service entity needs to be able to share a relation with all other entities in order to act as an intermedium between ship entities and company entities, while also being able to interact with other service entities.

A company entity can share a relation with a service entity as well as other company entities. Two company entities do not necessarily need service entities in order to be able to share information similarly to the real world, where companies use different communication channels when communicating with companies and citizens.

Dependency

A dependency is, as the name suggests, a dependency from one entity to another, and so the first argument of a dependency must be a relation that describes which two entities are in question, as well as which entity depends on the other. The second argument that a dependency must receive is a constraint function. This function's purpose is to describe what one entities require from the other entity in order to proceed with a given execution. In the case of Figure 3.3, `Service` has the two con-

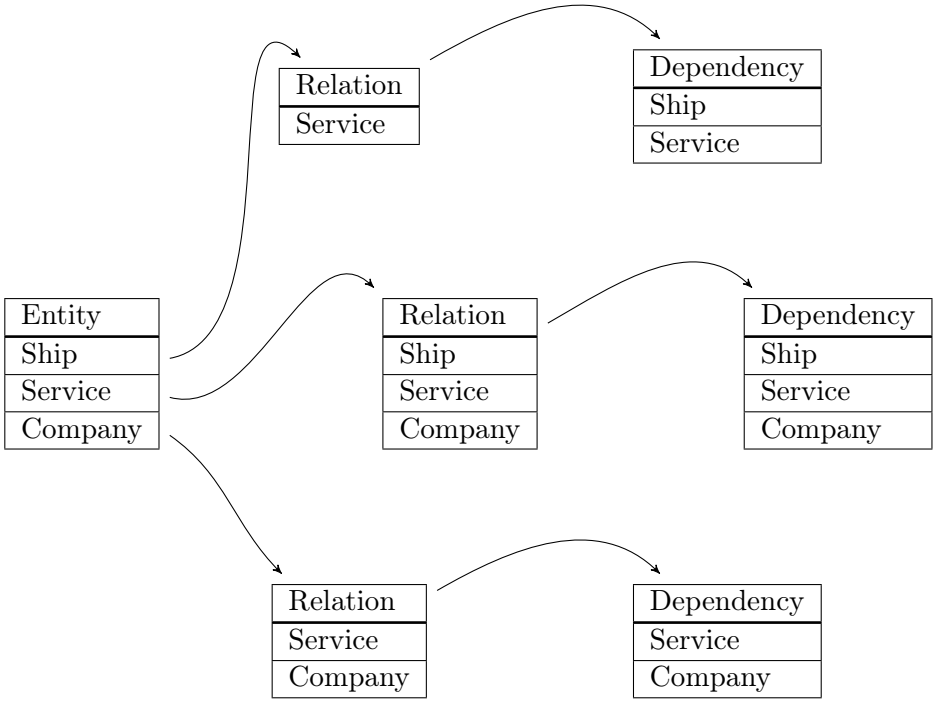


Figure 4.1: Available relations and dependencies for ships, services, and companies

straint functions `user/1`, and `psswd/1`, declaring that a correct user and password must be provided in order to make a request.

If two entities co-depend on each other, a relation must be made from the first to the second, as well as from the second to the first. As only the constraint function limits what the dependency entails, an entity *can* depend on itself.

Information

All entities are able to store information, which can be requested, transfered, and deleted. This is meant to represent various data, be it maps, keys, messages or other.

Figure 4.1 describes which options are excluded at selections of entities, relations, and dependencies. As can be seen, Figure 4.1 coincides with 4.3, and therefor the description given in this section describe both figures.

4.2 Updated Parser Grammars

The fields described in Section 4.1 furthermore need to be translated into parsable xml, and an example of this is given in Listing 4.3. This parser grammar serves as an updated version of the parser grammar described in Listing 3.1, however shaved for information[1] that is irrelevant to the model-generating process. While this information can of course be included in the xml-specification file, it will be deemed irrelevant after parsing, and thus not included further in the model-generating process. The parser grammar presented in Listing 4.3 is designed to form a clear and unambiguous collection of entities, along with a similarly unambiguous description of relations, exchange-pattern, and security precautions. The parser grammar abides by the rules presented in Section 4.1 and Figure 4.1.

4.2.1 Updated General Grammar

The updated xml-specification files will be limiting the flexibility of the models in some aspects, as not all maritime services would follow the structure given in Figure 4.3, however, the example given is the earliest version of the domain-specific language. Following complex structures of various maritime services, the domain-specific language of the specification files can scale in complexity with the requirements set up by the maritime services.

Listing 4.2 describes a reduced form of Listing 4.3. It can be seen that Listing 4.2 includes all information stored in Listing 3.2, which indicates that, after implementation of the model-generating process, there is no direct, urgent need to improve or change the parser, given the generalized format of the xml-specification files.

4.2.2 Updated Service Specification Grammar

Figure 4.3 displays the updated parser grammar, and subsequently shows the newly added information, which the service specification file will need to include. The added information fields are designed to represent the fields, displayed in Section 4.1.

[1] aSpec, authorInfo, authorInfos, contactInfo, dataExchangePattern, definitionAsXSD, description, isSpatialExclusive, keywords, oSpec, operation, operations, parameterType, parameterTypes, ptSpec, rSpec, requirement, requirements, returnValueType, serviceDataModel, serviceInterface, serviceInterfaces, siSpec, spec, specifications, status, text, typeReference, version

```
ServiceSpecificationSchema ::= specifications

specifications ::= spec specifications
    | ϵ

spec ::= specifications
    | spec
    | ϵ

spec ::= string
```

Figure 4.2: Reduced parser grammar of the updated Service Specification Schema

4.3 Technical Implementation

The project is implemented in Erlang, and contained in the files `aux.erl`, `interpreter.erl`, `mmods.erl`, (maritime models) `parser.erl`, `tests.erl`, and `main.erl`, along with the `xml`-folder containing models and simulations in xml-format.

4.3.1 The mmods module

The core of the project is contained in the `mmods.erl` file, which contains the functionality that creates the finite state machine models. This module creates processes which include the following fields:

`Type`
: This field is described by a single atom, which is only able to take the forms of either `ship`, `service`, or `company`. This is used to check the legality of relations, when adding these.

`Relations`
: The `Relations` field is described by a tuple list, containing a set of relations, as well as its corresponding dependencies, if any exist. Dependencies are represented by a list of functions, each evaluating to `true` or `false`.

`Self`
: The `Self` field is made up of the process ID, corresponding to the entity.

`Info`
: The `Info` field consists of all data representation, which can take any form.

```
ServiceSpecificationSchema ::= specifications

specifications ::= entities specifications
    | ϵ

entities ::= ent entities
    | ϵ

ent ::= type
    | name
    | relations
    | dependencies
    | information
    | requests

type ::= atom

name ::= String

relations ::= relation Relations
    | ϵ

relation ::= ent

dependencies ::= dependency dependencies
    | ϵ

dependency ::= relation
    | constraint

constraint ::= function

information ::= info information
    | ϵ

info ::= variant

requests ::= request requests
    | ϵ

request ::= relation
    | info
    | answers

answers ::= answer answers
    | ϵ

answer ::= variant
```


Figure 4.3: Updated parser grammar of Service Specification Schema

The functions `start/1`, `add_relation/2`, `add_dependency/3`, `add_info/2`, `remove_info/2`, `transfer_info/3`, and `request_info/4`, all add their corresponding self-descriptive functionality through the use of the `gen_statem` (generic state machine) module. The getter-functions `get_state/1`, `get_type` ↪ `/1`, `get_relations/1`, and `get_info/1`, perform the simple task of retrieving relevant informations for the provided process ID.

Having implemented this functionality, creating a model is a matter of simply using the functions as building blocks to create the desired maritime models. For examples of how to do this, see files `examples.erl` and `tests.erl`.

4.3.2 The parser and interpreter modules

The parser module

The parser module utilizes the `xmerl` library of functions to standardly perform well formed parsing of xml files. Additionally, the `xml_parser` ↪ `/1`-function uses the accumulator function `accFun`, which accumulates all field values, that are *not* purely whitespace and/or newline. Along with the hook function `hookFun` that binds the parser output to a tuple, consisting of an atom, describing the parsed node, along with its contents. Furthermore, `hookFun` dictates that pure text contents are just that - not a list, consisting of one string, which it would be otherwise. The parser module is created, following both the xmerl reference manual [7], an example posted online [8], and the parser grammar, described in Figures 4.2 and 4.3.

The interpreter module

The interpreter module exports the single function `interp/1`. This function takes in an Erlang term of the form, described above [2] and recursively loops the content value in order to exhaust the information, stored in the term. The `interp/1` function builds maritime models by adding entities, info, relations, dependencies, and lastly performing information requests, all in said order. Each adder function dynamically searches its corresponding keyword, rather than retrieving by index in the xml term, which allows for variation in the structure of the xml, along with interchangeability in the order of the adder functions. Two constraints do exist in the order of which components are added to the model, the first being that entity addition must always come first, as no data can be bound to nonexistent entities. The other constraint is that dependencies must be added *after* relations, as a dependency can only be placed on top of an existing relation. Finally, the interpreter shows a snapshot of all entities' states, as is after addition of all data.

[2] A tuple

4.3.3 Executing Instructions

In an Erlang-shell, all files can be compiled with `c(file)`, where after manual model generation can be run from the `mmods.erl` file [3]. For automatic model generation, the `main.erl` file gathers the functionality of the parser- and interpreter-modules, which in turn draws upon the mmods-module. From here, the user has the possibilities of parsing an `xml`-file, interpreting an Erlang term though the uses of `main:fparse/1`, `main:interpret/1` or using the combined function `main:fparse_and_interpret/1`, which fully converts an `xml`-file to a working maritime model.

For efficient, exception-free use of the model generators, its functions should be used following this precedence:

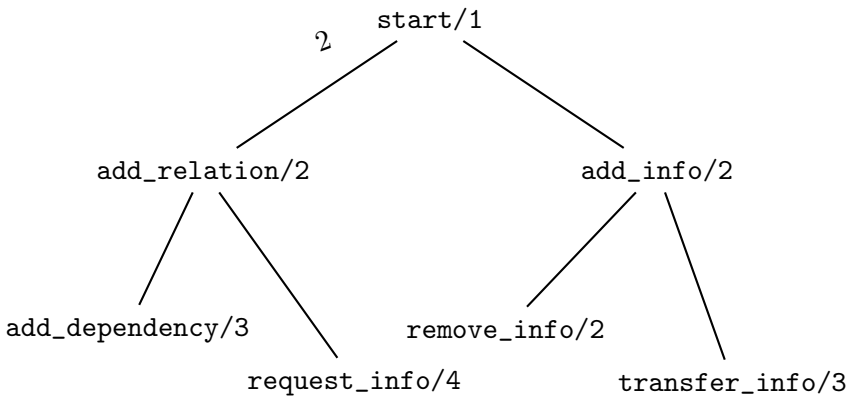


Figure 4.4: Function precedence, when using `mmods.erl`.

Figure 4.4 displays the order that needs to be followed when using the mmods module. The reason for this requirement is the fact that a number of the API functions rely on information, given in earlier functions. A relation cannot exist without at least two entities being added to the model beforehand, just as a dependency needs a relation in order to exist. Likewise, information cannot be added to a nonexistent entity, and that same information cannot be removed or transfered without it having been added first.

[3] See `tests.erl` and `examples.erl` for examples

An information request, as per the API function `request_info/4`, carries the four arguments, `from(entity)`, `to(entity)`, `requested(information)` and `answers(list)`. Despite the fact that it would seem like this action would require at least one dependency to be satisfied by `answers`, and that the information needs to exist, neither is the case. As dependency is a (possibly empty) list of constraints, none of these *need* to be added for a request to be made. Likewise, a request can be made for nonexistent information, or information not stored in the `to`-argument entity, in which case no information will be returned to the requester. This in turn, means that no information needs to be added to a model before a request can be made.

Using a model post-creation is both possible though the uses of manual and automatic model-creating, by interacting directly with the variables in the Erlang shell, in which the model has been created.

If a model is to be generated directly in an Erlang-shell, the same rules apply, as if it was run from a file, or compiled from an xml-file.

Results

5

In this chapter I will provide a layout of the experiments, which I have set up, in order to verify and present the functionality of the implementation as a whole. This includes testing of all modules contained in the implementation, as well as the final demonstration, that a model generated manually, can be replicated automatically.

5.1 Testing

Testing of the implementation has been divided into two parts, one of which is detailed in Section 5.1.1, the other in Section 5.1.2.

5.1.1 Testing the Manual Model Creator

The manual model creator module contained in the file `mmods.erl`, exports 11 functions, of which 7 are model creation functions[4], 4 are data-fetching functions[5] and a single one is a simulation function[6].

All testing of the mmods module has been contained in the `tests.erl` file, which includes multiple function-specific tests for each API function, exported from the mmods module. After compilation, all tests can be run in an Erlang shell, either as the normal zero-input functions they are, or as the `main_funcs/0` function, which encapsulates all created function-specific test cases. All tests are created to return the atom `true` if they succeed, or the corresponding error message, if they do not. All API-function-specific tests returned true, indicating their correctness. The getter functions were only tested with a single case, `get_all/0`, due to their implementation being extremely simple.

The remaining part of the `tests.erl`-file includes the scenario-oriented tests. As the description entails, these tests describe the outcome of seven

[4] `start/1`, `add_relation/2`, `add_dependency/3`, `add_info/2`, `remove_info/2`, `transfer_info/3`

[5] `get_state/1`, `get_type/1`, `get_relations/1`, `get_info/1`

[6] `request_info/4`

scenarios; one for every scenario-function test. The tests, contained in this section of the file are:

multiple_coms/0
: This test case shows that a single service is able to determine which company to draw information from in an information request. The service `Serv` is connected to the three companies `Com1`, `Com2`, and `Com3`, which hold the atoms `this`, `is`, and `correct` respectively. For each request, as `Ship` requests the same service entity `Serv` for the three atoms, `Serv` must select which company to fetch the data from.

multiple_serv/1
: This test case shows that multiple service entities can link a ship entity and a company entity. The ship entity `Ship` and the company entity `Comp` are linked thrice by the service entities `Ser1`, `Ser2`, and `Ser3`, and the three atoms `this`, `is`, and `correct` are requested. The three atoms are requested one at a time, through each relation, and no matter the order, the right combination is fetched. This function is executed six times; one for each permutation of the order in which to request to the three service entities.

service_choose_company/0
: This test case shows that when a service entity is linked to several company entities, each containing different data, it is able to determine which company to fetch data from. Here the ship `Ship` is mutually linked to `Serv`, which is then mutually linked to companies `Com[1,..,5]`. As `Ship` requests the atom `c`, `Serv` will have to determine that `Com1` holds the relevant information, and thus this is transfered to `Ship`.

bad_connection_from/0
: This test case shows what happens to a request when the service entity is linked to a company entity, without the connection being mutual. In this case, as `Shp1` and `Shp2` request `Serv`, `Serv` passes the requests on to `Comp`. `Serv` is able to make the requests, however, as no channels of communication are set up from `Comp` to `Serv`, no information is able to be returned, and thus no ships receive information.

bad_connection_to/0
: This test case works similar to `bad_connection_from/0`, however, in this case, the connection from `Serv` to `Comp` is missing. Thus, `Serv` cannot find the correct entity to pass the request on to, and thus no ships receive information.

ok_connection/0
: This test case works as the last two test cases should ideally have worked.

That is, two ship entities request a service entity, which fetches data from a company entity, and thus, data is returned to the ship entities.

first_come_first_served/0
: This test case shows that information is removed from a service entity, after it is sent. In this case, the two ship entities, `Shp1` and `Shp2` are both mutually connected to the service entity `Serv`. The service entity holds the atom `a_steak`, and when both ship entities request said atom, it will be distributed to whomever requested first.

The scenario-oriented test cases are all included in the wrapper function `main_scenarios/0`, which, when run, yields that all cases pass. The scenarios wrapper function is included in the top-level wrapper function, and thus, to execute *all* tests, function-oriented or scenario-oriented, simply execute

```
$ tests:main().
```

5.1.2 Testing the Automatic Model Creator

The automatic model creation algorithm makes use of the mmods module, and as this has been tested for correctness in Section 5.1.1, it is fair to assume that this module will also function correctly in the context of the automatic model creator.

The two modules that distinguish the automatic model creator from its manual counterpart is the parser and the interpreter. As the parser is a standardized xml-to-Erlang, there is no need to check for edge-cases, correlating directly to the current implementation, as there are none. Therefore the parser is tested with the function `parse_tests/0`, which includes all test cases. The cases tested for are correct handling of strings, integers, lists, multiple sub-tags in one tag, a tag occurring in another tag with the same name, and handling of spaces. The test is run with `tests:parse_tests().` in an Erlang shell, which will reveal that all tests pass.

One test that does not pass, is `test_interp_anonymous_function/0`. This is due to the fact that handling of anonymous functions is not implemented at the time of writing. Support hereof would need to be handled in the `aux:token/1`-function, as this is the mechanism that translates functions in the Erlang-term, resulting from `fparse/1`.

5.2 Experiments

The primary goal of the project is to construct maritime models, describing maritime services, in the shape of finite state machines, both manually and automatically, followed by an assessment of the advantages and disadvantages of each type. As summarized in Section 3.7, manual model-building provides more control to the author, while automatic model building allows for more

streamlining in the model-building process. Thus, as each method provides unique advantages to the process, a feasible metric of judging the products of each method is their respective attributes. That is, which FSM is able to describe scenarios to the highest degree. In order to determine this, I will compare the results of each model building method, starting with the example, given in Figure 3.3.

5.2.1 Model Equivalence

Figures 5.1 and 5.2 display the results of manual and automatic model creation respectively. By manually inspecting them, it can be seen that the only variance between the two is the process IDs, and despite of this, the relationships between the entities, are evidently equal in the two models. Also, from manual inspection, it can be seen that information, types and dependencies, distributed between entities are also equal to that of their manual/automatic counterparts.

```
[{company,[{<0.79.0>,[]}],<0.80.0>,["map"]},
 {service,[{<0.78.0>,[]},{<0.80.0>,[]}],<0.79.0>,[]},
 {ship,[{<0.79.0>,[fun aux:user/1,fun aux:psswd/1]}],
       <0.78.0>,
       ["map"]}]
```

Figure 5.1: Maritime model, resulting of executing the commands, contained within the function `examples:protocol`

```
[{company,[{<0.84.0>,[]}],<0.83.0>,["map"]},
 {service,[{<0.85.0>,[]},{<0.83.0>,[]}],<0.84.0>,[]},
 {ship,[{<0.84.0>,[fun aux:user/1,fun aux:psswd/1]}],
       <0.85.0>,
       ["map"]}]
```

Figure 5.2: Maritime model, resulting of interpreting the file `protocol.xml` with `main:parse_and_interpret('xml/protocol.xml')`

A visual representation of the points described is displayed in Figure 5.3, where 5.3a represents the manually created model, and 5.3b represents the automatically created model. In both subfigures, nodes symbolize entities, containing three fields. The first line in each entity node represents its type, the second its unique ID, with the third representing the information held by the entity. All edges in the figures are oriented, symbolizing relations from one entity to another. Functions displayed beside a relation, represents a dependency, associated with that relation. If not evident in Figures 5.1 and 5.2, Figure 5.3 clearly show how the collection of entities' states are uniform in relation to their counterparts'.

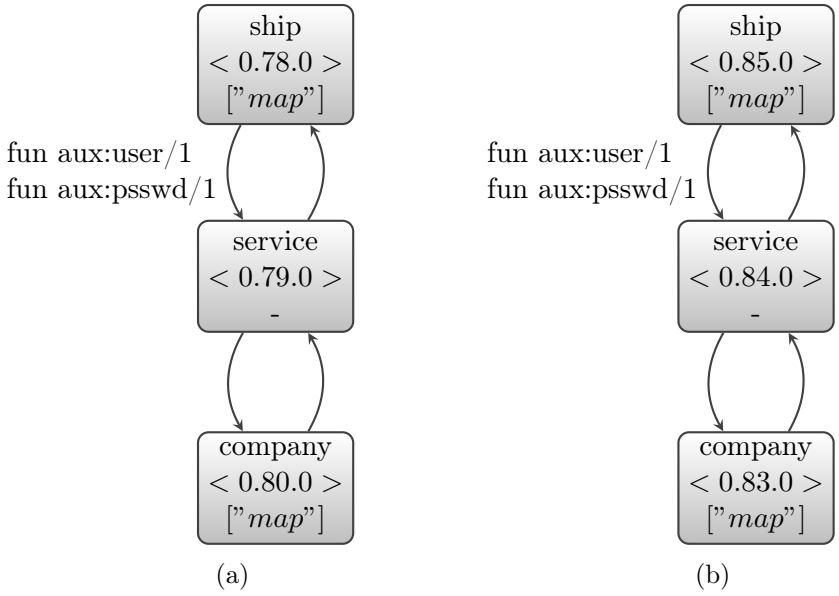


Figure 5.3: Visual representation of Figures 5.1, and 5.2.

Having concluded that the models[7], described in Figures 5.1, and 5.2, are equal, in order for their FSMs to be equivalent, all available *actions* correlating to the states must also be equal. To determine if this is the case, I will look to the functionality of the manual, as well as the automatic model creator.

The functionality of the automatic implementation is built to directly utilize the manual model building module, and thus their functionalities are near identical. They are only limited by the functionality of the parser and interpreter's ability to feed instructions to `mmods.erl`, such as the example given in Section 5.1.2.

[7]Collections of all states in the FSM

5.2.1.1 Further Examples of Model Equivalence

In order to verify that the example, described above in Section 5.2.1 is not a unique occurrence, the function `equiv_models/2` has been created. This function compares two maritime models, according to the metrics used, when comparing models in Figure 5.3; Types, constraint functions, and information have to exactly match, while relations have to match mapped values. The function `equiv_models_tests/0` utilizes the model comparer function, in order to test that the protocol example holds true with other examples as well. The examples given to `equiv_models_tests/0`, are all models created manually in the file `examples.erl`, as well as their automatic counterparts, created in the 'xml' folder. All test cases passed, further proving the implementation's ability to create equivalent maritime models.

Discussion

6

Following the successful development of the solution, described in the preceding chapters, problem statements arise dealing with integration into the MCP, use, future use and the balancing of the compromise that is choosing which model generator to utilize. The following chapter will deal with these topics. Section 6.1 will describe potential approaches to initial integration, while Section 6.2 will subsequent future use, both practical and technical, following the inevitable need for further development. Section 6.3, will contain an assessment of which model generating functionality is most likely to be widely used, going forward. Eventually, Section 6.4 will provide a recap of the results obtained in Chapter 5, in relation to what is described in sections 6.1 through 6.4.

6.1 Launch and Integration into the MCP

For the solution described throughout this thesis to be integrated, various changes need to be made with regards to the infrastructure of the Maritime Connectivity Platform. The first, and most obvious, change that should be implemented is the technical implementation of the new functionality. Next comes the need for educating the user-group, as these will have no preceding knowhow as to make the system work in accordance with their needs. This will be necessary whether use of manual or automatic model generating is utilized.

A soft launch strategy should be used when inaugurating the functionality, as described in a blog-post on LiveChatInc [9]. Launching the solution gradually will let the user-base get acquainted with its use, while simultaneously allowing for necessary feedback towards the functionality. This will, in turn, decrease the risk of the project succumbing to any of the launch failures, as described in an article on Harvard Business Review [10], all of which are often connected with a hard product launch.

6.2 Future Use

Post-launch of the model-building component is a continuous development process. User-submitted feedback should be regularly implemented in order to ensure optimal user friendliness, and intuitiveness. Furthermore, as the functionality described in Chapter 4 is only the initial functionality, and ideally, as described in Section 6.2.1 below, future development is deeply rooted in the project.

6.2.1 Expanding Automatic Model Functionality

Embedded in the nature of the project is the need for future expansion of functionality, following increased demands from maritime service providers. The modularity that the solution has been implemented with will aid in this, as it allows for less complicated employment hereof. As described in Section 4.3, the complete solution has been divided into three fields, and thus, these are the areas where further implementation is needed, if support for additional model functionality is desired.

- `mmods.erl` The primary functionality will need to be implemented here. This entails the main API call to the finite state machine, along with its following desired result and side effects. Altering the implementation of this file will subsequently alter the manual as well as the automatic model generator.
- `parser.erl` This file will *not* need any adjustments in order to handle new functionality.
- `interpreter.erl` This file will need to be altered in order to handle additional information, picked up by the parser. In the case that support for model functionality has been implemented in `mmods.erl`, but not `interpreter.erl`, said functionality will not be included in the resulting model, and no error or exception will be raised. This is true, even if the additional design information is included in the xml-specification file.

6.2.2 Expanding Manual Model Functionality

As explained in Section 4.3.2, the manual component provides all core functionality of the automatic component. The fact that this module is underdeveloped in comparison to its automatic counterpart, in no way rules out its future use, as having only the core functionality encapsulated in a module, providing an all-inclusive API allows for virtually limitless further development. The automatic component is *one* example out of a virtually endless pool of additional, untapped functionality, based on the functionality, provided in the manual component. Expanding upon this, will be confined to future work, as potential solutions can scope to an extent similar to this project.

6.3 Manual versus Automatic Model Generation

A fundamental obstacle throughout the project is the learning curves, described in Figures 2.3 and 2.4, along with their corresponding sections. This obstacle is present in both scenarios, however, the learning curve is significantly steeper, using sequential[8] model based testing/model generating. A soft launch and extensive user-guide, as described in Section 6.1, will reduce the effects of the learning curve for both development techniques, however, not to the extend that this is completely negated.

Automatic model generation is, however, both the most user-friendly method and the one most suited for a visual builder interface. A visual builder interface could be in a style, similar to the Eclipse Visual Editor, Vex [11] for XML. Such an XML-builder should be implemented to always show the user which options are available to add to the xml and subsequent model, which would in turn ease work flow, placed on all other aspects of the solution.

6.4 Results Related to Use

Taking the findings of Chapter 5 into consideration, when comparing the usefulness of the two methods, one model-generating technique clearly stands out as being the most useful. Recognizing that automatic model-generation will provide more immediate user-friendliness, more scalability, and according to Section 5.2.1, functionality equivalent to that of manual model-generation.

A central perk of the solution is to, through model-based testing, visualize and predict intended behavior of maritime services. Being able to do this, will not only be advantageous in the ability to create software, more in the likes of what is desired; it will also speed up the process of doing so. Most development processes are implemented, using the much praised Scrum [12] framework. Developing software, using the Scrum framework entails a circular approach to the way the process plays out. Typically, such a process starts with the basic idea of the application. After this step, the loop starts with planning and analysis of intended functionality and behavior, followed by implementation, functionality and behavioral testing, and lastly, revision. After revision, the process initiates once again, starting with planning and analysis.

[8] Manual

Figure 6.1 visualizes a standard scrum framework, which can be followed, when developing a maritime service. The scrum method in question, although agile, can be a lengthy process, partly because of the 'testing' sequences. Furthermore 'Implementation' and 'Revision' may, in the worst cases, be performed to little benefit, if implemented on the basis of a faulty design.

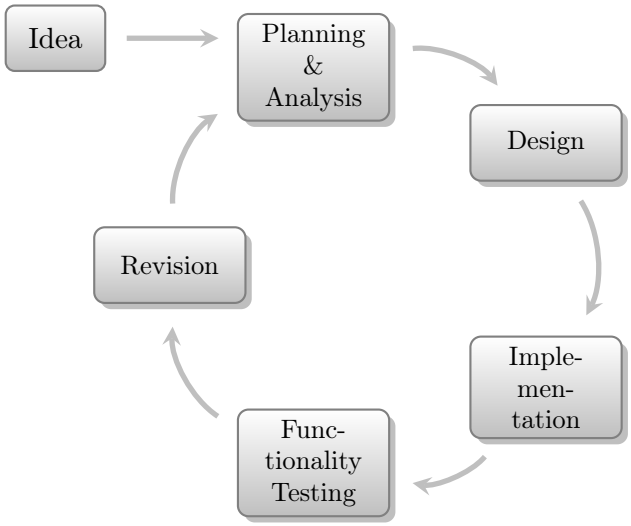


Figure 6.1: Scrum development cycle without automated model-based testing.

The automatic model-generator, described in Section 4.3, can aid in the issue, defined above. The tool will allow users to visualize the an intended version of their maritime service instance, before starting implementation, which will enable them to skip the steps 'Implementation', 'Revision', and 'Planning & Analysis' in a given scrum cycle. In an ideal use case scenario, such as the one, described in Figure 3.7, as crucial information is given before advancement, a user can, thus, skip large portions of cycles, thereby saving time by not follow dead design ends.

This principle is described by Figure 6.2, where 'Behavioral Testing' is added to a sub-cycle, only shared with 'Design', thereby creating a much smaller, and considerably faster, sprint.

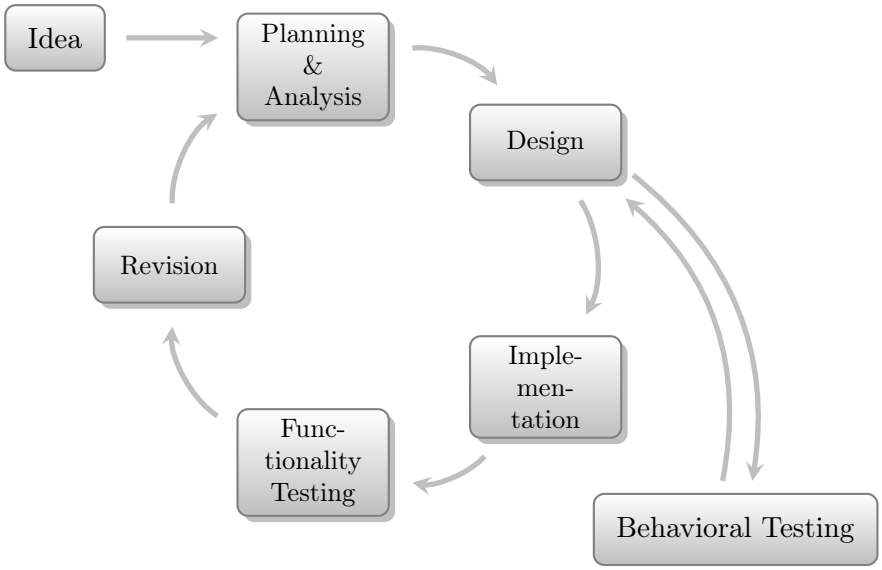


Figure 6.2: Scrum development cycle with automated model-based testing.

Manual model generation, in it's basic state, as described in Section 6.2.2, allows for limitless new functionality, and, as proven by the implementation of the automatic component, said new functionality does not necessarily need to be manual.

In conclusion, both components add relevant functionality to the MCP, and while automatic model generating provides the more present, needed, utility, the manual component provides a broad foundation for development of new ideas.

Conclusion

7

In this chapter, I will conclude the results of my analysis and work. This includes a breakdown of what has been achieved though the model-generating implementation, as well as suggestions to what can be achieved, by expanding upon the project in future work, seen in Section 7.2. Section 7.3 contains a summary, describing key takeaways from the thesis in bullet-point form.

7.1 Conclusions

In Chapter 2, an assessment was provided of methods, which could be utilized to implement a solution to the problem that was also described in Chapter 1. Specifically, the methods that were utilized in the solution, were the combination of a parser and an interpreter, which would feed commands to a finite state machine, thereby creating the maritime models. This process of implementing this was described in Chapter 4, and the process of determining this design is described in Chapter 3.

In Chapter 5 I verified the validity of the operations, performed in the implementation through extensive testing. That is, the testing, performed in Section 5.1, concluded that the model creators, described throughout Chapter 4 could create working models, which adhered to the set of rules and descriptions, described in Chapter 3. Furthermore, in Section 5.2 I have concluded that models generated through the manual method, have the ability to be equivalent to ones generated automatically, and vice verse.

Chapter 6 contains an assessment of the derived results, and relates this to the intended use of the application. This includes launch strategy, integration into the MCP and future use, both from a user-perspective, as well as from a technical standpoint. Strategies for future development are laid out for both the automatic and the manual model generator, and it is concluded that, while the automatic component may hold more immediate relevance towards integration in the MCP, both bring valuable functionality to the table.

7.2 Future Work

The project described throughout the thesis has been concluded, featuring satisfying results, however, this does not mean that no additional functionality is able to improve upon the implemented solution. Sections 7.2.1 through 7.2.5 displays a list of areas that, through continued work, could be shifted from compromises to all-embracing solutions.

7.2.1 Model Functionality

As mentioned in Sections 6.2.1 and 6.2.2, expandability has been a top priority throughout the implementation process, as this leads the way for the entire basis of future use, however, mentioned in the same section, is how support hereof is entirely possible, due to the style in which the solution has been developed.

7.2.2 Improved XML-Parsing

In Section 2.2.1, monadic parsing was described as a means to parse text, using more efficient procedures, compared to the technique's predecessor, described in Section 2.2. The implementation of the parser module associated with this thesis features a *non*-monadic parser, which could potentially introduce scalability issues on the side of automatic model generation. Scalability is, in this case, only affected concerning the size of the xml specification-files, which will rarely reach sizes where this will pose anything but minor inconveniences, yet, introducing the power of the monad is the best solution in the long run.

7.2.3 Ignored Parsing

Another parser-related improvement is support for field-selection. At the time of writing, the parser module has no way of distinguishing relevant fields in the xml specification files from the irrelevant. In this project, this has been bypassed, by writing specifications in the correct form. The interpretation module will fail unless the argument it receives is on the form `{entities,Content}`. Unexpected values, contained in the `Content` variable, will be ignored, as the interpreter looks up values by identifier, and not by index, however, if the outermost tag, returned by the parser is on the wrong form automatic model generation will return an error.

7.2.4 Expanded Constraint Support

Section 5.1.2 describes missing support for anonymous functions in the automatic model generator, while Section 5.1.1, as well as `tests.erl` display this functionality as working in the manual model generator. It is concluded that the auxiliary function `token/1`, displayed in appendix A.1, is what comes up short, and so the shortcoming can be rectified with improved tokenization, in this function.

7.2.5 Improved Model-Testing

Section 2.4 describes a method of software testing that outperforms regular unit-based testing. Albeit usually utilized in development testing and function-performance testing, if applied to this field, property-based testing can contribute powerful additional functionalities to the solution as a whole.

7.2.5.1 Implementation

Implementation of property-based testing would ideally be included in a separate module. Included in the property-based testing module, should be a selection of functions, designed to take in a model, generated by the three modules described in Section 4.3, that would subsequently test aspects of the input models. The functions should be implemented with QuiviQ's quickcheck functionality for Erlang, and with the ideology that each test should characterize *one* functionality of the model[9].

7.2.5.2 Use

In a corporate setting, a user should be given the option to test for model properties at the time of submission. Given the nature of property based testing, and the ideology described above, users will be able to select which behavioral patterns to check for, and promptly receive the knowledge of whether or not this is supported in their maritime service model.

7.2.5.3 Benefit

Applying property-based testing to the maritime models, as described above, will give the users information about the behavior of their design. Striving for a property-based test module, as opposed to unit-based, will provide extensive coverage, increasing reliability towards the information given to the user. In turn, educating the authors of the maritime services, at an early stage, will heighten the likelihood that the functionality that is proposed is, in fact, that which the author has intended.

[9] To the possible extend

7.3 Summary

In conclusion, I believe that the solution presented in the thesis would pose a useful tool, if integrated into the Maritime Connectivity Platform. This would contribute with the functionality of model-generation, which opens the door to a wide array of added functionality.

The functionality and features of the maritime model generator and its result are:

- The ability to generate Finite State Machines, manually and automatically, the latter with the ability to be inserted directly in the structure of the already-integrated and used xml-specification files.
- Both methods of generating models introduce the same functionality, also featuring similar support of additional functionality.
- Support for further model-based testing through the automatic model generator.
- Support for further functionality through the manual model generator.
- An updated service specification structure, that allows for a more direct implementation of the automatic model generator into the infrastructure of the Maritime Connectivity Platform.

Bibliography

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

[2] QuviQ, http://www.quviq.com/

[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

[5] Natural Language Processing, Elizabeth D. Liddy, Syracuse University, liddy@syr.edu http://surface.syr.edu/cgi/viewcontent.cgi?article=1019&context=cnlp

[6] Monadic Memoization towards Correctness-Preserving Reduction of Search Richard Frost School of Computer Science, University of Windsor Ontario, Canada richard@uwindsor.ca http://richard.myweb.cs.uwindsor.ca/PUBLICATIONS/AI_03.pdf

[7] xmerl reference manual, http://erlang.org/documentation/doc-5.4/pdf/xmerl-1.0.pdf

[8] Starting to play with xmerl, Arif Ishaq, https://arifishaq.wordpress.com/2014/11/25/starting-to-play-with-xmerl/

[9] Soft launch vs hard launch, https://partners.livechatinc.com/blog/soft-launch/

[10] Why Most Product Launches Fail, Harvard Business Review, https://hbr.org/2011/04/why-most-product-launches-fail

[11] Vex, Eclipse, https://www.eclipse.org/vex/

[12] What is Scrum?, scrum.org, https://www.scrum.org/resources/what-is-scrum

Appendix

A

A.1 token

```
token(String) ->
  case String of
    % atoms
    "ship" -> ship;
    "service" -> service;
    "company" -> company;
    % functions
    "fun psswd" -> fun aux:psswd/1;
    "fun user" -> fun aux:user/1;
    "fun trivial" -> fun aux:trivial/1;
    _ -> error
  end.
```

Figure A.1: The `token/1`-function in the `aux`-file.

A.2 ServiceSpecificationSchema

```
ServiceSpecificationSchema ::= specifications

specifications ::= spec specifications
     | ϵ

spec ::= name
     | status
     | id
     | version
     | description
     | keywords
     | isSpatialExclusive
     | authorInfos
     | requirements
     | serviceDataModel
     | serviceInterfaces

authorInfos ::= authorInfo authorInfos
     | ϵ

authorInfo ::= aSpec authorInfo
     | ϵ

aSpec ::= id
     | name
     | description
     | contactInfo

requirements ::= requirement requirements
     | ϵ
```

Figure A.2: Full parser grammar of Service Specification Schema. (1 of 2)

A.3 E2 - NW-NM Service Specification

```
<?xml version="1.0" encoding="UTF-8"?>
<ServiceSpecificationSchema:serviceSpecification
    xmlns:ServiceSpecificationSchema="http://efficiensea2.org/
        ↪ maritime-cloud/service-registry/v1/
        ↪ ServiceSpecificationSchema.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://efficiensea2.org/maritime-cloud/
        ↪ service-registry/v1/ServiceSpecificationSchema.xsd
        ↪ ServiceSpecificationSchema.xsd "
    xmlns:xs="http://www.w3.org/2001/XMLSchema" >
  <name>NW-NM TP Maritime Cloud Service</name>
  <status>provisional</status>
  <id>urn:mrn:mcl:service:specification:dma:nw-nm</id>
  <version>0.3</version>
  <description>The NW-NM service specification defines a
      ↪ combined NW-NM TP model along with the actual service
      ↪ API used for accessing NW-NM data, as registered in the
      ↪  Maritime Cloud service catalogue.</description>

  <keywords>pNW, NM, Navigational Warnings, Notices to Mariners,
      ↪  MSI, Maritime Cloud Service</keywords>
  <isSpatialExclusive>false</isSpatialExclusive>

  <authorInfos>
    <authorInfo>
      <id>urn:mrn:mcl:user:dma:mcb</id>
      <name>Mads Bentzen Billesoe</name>
      <description>Responsible for the NW-NM service</
          ↪ description>
      <contactInfo>mcb@dma.dk</contactInfo>
    </authorInfo>
  </authorInfos>

  <requirements>
    <requirement>
      <id>urn:mrn:mcl:requirement:nw-nm:1</id>
      <name>Combined NW-NM model</name>
      <text>The data model should encapsulate a combined NW-NM
          ↪ model.</text>
    </requirement>
    <requirement>
```

```
    <id>urn:mrn:mcl:requirement:nw-nm:2</id>
    <name>Return all published NW-NM messages.</name>
    <text>The NW-NM service should make it possible to
        ↪ retrieve all published NW-NM messages from the
        ↪ given service provider.</text>
  </requirement>
</requirements>

<serviceDataModel>
  <definitionAsXSD>
    <!-- Too extensive - included in the "NW-NM Service
        ↪ Specification" document, Appendix B -->
  </definitionAsXSD>
</serviceDataModel>

<serviceInterfaces>
  <serviceInterface>
    <name>MessageService</name>
    <description>Works according to the request response
        ↪ pattern.</description>
    <dataExchangePattern>REQUEST_RESPONSE</dataExchangePattern
        ↪ >

    <operations>
      <operation>
        <name>get</name>
        <description>Retrieves an published NW-NM messages.</
            ↪ description>
        <returnValueType>
          <typeReference>Message[]</typeReference>
        </returnValueType>
        <parameterTypes>
          <parameterType>
            <typeReference>String</typeReference>
          </parameterType>
        </parameterTypes>
      </operation>
    </operations>

  </serviceInterface>
</serviceInterfaces>

</ServiceSpecificationSchema:serviceSpecification>
```

```
requirement ::= rSpec requirement
     | ϵ

rSpec ::= id
     | name
     | text

serviceDataModel ::= definitionAsXSD

definitionAsXSD ::= ϵ

serviceInterfaces ::= serviceInterface serviceInterfaces
     | ϵ

serviceInterface ::= siSpec serviceInterface
     | ϵ

siSpec ::= name
     | description
     | dataExchangePattern
     | operations

operations ::= operation operations
     | ϵ

operation ::= oSpec operation
     | ϵ

oSpec ::= name
     | description
     | returnValueType
     | parameterTypes

returnValueType ::= typeReference


parameterTypes ::= parameterType parameterTypes
     | ϵ

parameterType ::= ptSpec parameterType
     | ϵ

ptSpec ::= typeReference
```

Figure A.3: Full parser grammar of Service Specification Schema. (2 of 2)

TODO(s) remaining in the document:

TODOALL: 2
TODOWRITE: 2