TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science

Magnus Teekivi 192277IAPM

MODEL-BASED TESTING OF WEB APPLICATIONS

Master's Thesis

Supervisor
Evelin Halling
PhD

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Magnus Teekivi 192277IAPM

VEEBIRAKENDUSTE MUDELIPÕHINE TESTIMINE

Magistritöö

Juhendaja
Evelin Halling
PhD

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Magnus Teekivi

May 24th, 2021

Abstract

The goal of this thesis is to evaluate and apply model-based testing to the web application developed by Milrem Robotics which is used as a user interface for the company's THeMIS unmanned ground vehicle. Two model-based testing implementations are developed as part of this thesis, one that utilizes primarily UPPAAL and DTRON, and another one that utilizes primarily GraphWalker. Interaction with the web application is implemented using the browser automation tool Selenium. To improve test maintainability, the Page Object Model test design pattern is followed.

During this thesis, it was found that setting up a model-based testing implementation that uses primarily UPPAAL and DTRON is more complex and time consuming than the other one, but it can be used to test more complex systems. This solution lacks the possibility of obtaining the test trace. Setting up GraphWalker turned out to be easier, but that also lacks some desirable features like out-of-the-box easy test splitting or test execution replay. UPPAAL is closed source and free only for non-commercial academic use, whereas GraphWalker is an open source project licensed under the permissive MIT license. GraphWalker is less capable of modelling certain complex systems, but the thesis showed that the system can be modelled with it. All of this suggests that GraphWalker is the tool to move forward with.

The thesis also suggests future work both with regards to the test implementations and the tools itself. In the case of GraphWalker, it is possible to contribute code to help get new features and fixes implemented. As of this thesis, the author has started initial contribution work.

The thesis is in English and contains 58 pages of text, 9 chapters, 21 figures, 0 tables.

Annotatsioon

Käesoleva lõputöö eesmärgiks on teostada rakendusuuring, mille käigus realiseeritakse mudelipõhine testimine ettevõtte Milrem Robotics robotsõidukite kasutajaliidese peal. Antud kasutajaliides on oma olemuselt veebirakendus, mis on mõeldud jooksutamiseks puutetundliku ekraaniga tahvelarvuti peal. Käesolevas lõputöös toimub mudelipõhise testimise rakendamine kahe erineva lahenduse kaudu. Üks lahendustest põhineb peamiselt tööriistadel UPPAAL ja DTRON. Teine lahendus põhineb peamiselt tööriistal GraphWalker. Kasutajaliidesega liidestamine on lahendatud brauserite automatiseerimiseks mõeldud tööriistaga Selenium. Testide hallatavuse parandamiseks, eriti olukorras, kus kasutusel on paralleelselt mitu testimislahendust, on kasutusel disaini/testimismuster *Page Object Model*.

Lõputöö raames teostatud rakendusuuringu jooksul selgus, et UPPAAL'il ja DTRON'il põhineva testimislahenduse üles seadmine ja edaspidine kasutamine oli GraphWalker'iga võrreldes keerulisem ja aeganõudvam. Samas võimaldab UPPAAL modelleerida keerukamaid süsteeme. Lõputöö autori teada ei ole antud lahenduse puhul paraku võimalik tekitada testimise jooksutamisest testisammude logi. Selle tõttu võib olla raskendatud avastatud vigade taastekitamine. Seevastu oli GraphWalker'i üles seadmine ja kasutamine enamasti lihtsam, ent ka sellel puuduvad mõned soovitud võimalused, nagu näiteks tööriista poolt võimaldatud lihtne testide tükeldamine või varasema testikäivituse taasesitamine. UPPAAL on suletud lähtekoodiga, tasuta võib seda kasutada vaid mittekommertsiaalseteks eesmärkideks akadeemilises kasutuses. GraphWalker seevastu on avatud lähtekoodiga projekt, mis on välja antud lubava litsentsi MIT alusel. Ehkki GraphWalker on modelleerimise koha pealt vähem võimekas, näitas lõputöö käik, et antud süsteemi puhul on see rakendatav. Lõputöös ei jõutud keskenduda keerulisematele alamsüsteemidele, kuid autor leiab, et ka nende puhul saab antud tööriista rakendada, juhul kui eeltingimused on teada ja süsteem käitub deterministlikult. Eeltoodu põhjal leiab autor, et edasiseks kasutuseks on antud süsteemi testimiseks rohkem potentsiaali GraphWalker'il.

Lõputöö toob välja ka mõned potentsiaalsed kohad, millega võiks edaspidi tööd teha. Esiteks saaks tegeleda süsteemi keerulisemate osade testidega katmisega, samal ajal vähendades *mock*-imist; selle kõrval saaks testide katvust üldisemalt laiendada. Teiseks

toob töö välja ka need aspektid, mida võiks antud tööriistade endi puhul täiendada. Kuna GraphWalker on avatud lähtekoodiga ning omab piisavalt aktiivset arenduskommuuni selle ümber, siis on võimalik huvilistel anda oma panus antud tööriista edendamisele. Autor toob välja, et näiteks testikäivituste taasesitamine on üks potentsiaalne arendus, millega võiks GraphWalker'it täiendada. Selle konkreetse arenduse osas on autor käesoleva lõputöö kirjutamise ajal teinud esialgset tööd. Samuti toob autor välja mõned väiksemad paranduste ja täienduste võimalused.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 58 leheküljel, 9 peatükki, 21 joonist, 0 tabelit.

List of abbreviations and terms

adapter	a software component that fits an SUT into a test solution
API	Application Programming Interface
CI	Continuous Integration
CSS	Cascading StyleSheets
DTRON	Distributed TRON
ETSI	European Telecommunications Standards Institute
guard	a condition which determines if an edge can be entered
HTML	HyperText Markup Language
IDE	Integrated Development Environment
MBT	Model-Based Testing
mock	a software component replacement that reduces dependencies
POM	Page Object Model
SUT	System Under Test
TDL^{TP}	Test Scenario Definition Language
TRON	Testing Real-time systems ONline, a testing tool based on UPPAAL
UGV	Unmanned Ground Vehicle
UI	User Interface
URL	Uniform Resource Locator

Table of Contents

1 Introduction **11**

2 Technical Background **12**

2.1 Model-based testing . . . 12

2.2 Web application UI testing . . . 15

2.3 Page object model pattern . . . 15

3 Tools Used **17**

3.1 UPPAAL . . . 17

3.2 DTRON . . . 19

3.3 TDL^TP . . . 19

3.4 GraphWalker . . . 21

3.5 Selenium . . . 25

3.6 Java . . . 26

3.7 JUnit 5 . . . 27

4 The System Under Test **29**

5 The Testing Setup **33**

5.1 Page Object Model classes . . . 34

5.2 GraphWalker . . . 36

5.3 UPPAAL + DTRON . . . 42

6 The Process **46**

6.1 Initial setup . . . 46

6.2 Initial testing . . . 47

6.3 Testing with GraphWalker . . . 48

6.4 Testing with UPPAAL + DTRON . . . 50

6.5 Further process and improvements . . . 54

7 Evaluation **57**

7.1 Evaluation of GraphWalker . . . 57

7.2 UPPAAL + DTRON . . . 59

7.3 Evaluation of MBT in general . . . 60

7.4 Evaluation of POM design pattern . . . 62

7.5 The role of traditional testing . . . 63

7.6 Feedback from the company 64
7.7 Summary of evaluation 65

8 Future Work **66**

9 Summary **68**

Bibliography **69**

Appendix 1 - Non-exclusive licence for reproduction and publication of a graduation thesis **71**

Appendix 2 - HeaderBar.java **72**

Appendix 3 - DtronOperatorAdapter.java **73**

Appendix 4 - Initial scenario **76**

List of Figures

1 Overview of the Taxonomy [1] 14

2 A small example of an UPPAAL system 18
3 A small example of a model in GraphWalker Studio 22
4 GraphWalker Studio overview with its sidebar open 23
5 GraphWalker Studio sidebar: the "Model" section 23
6 GraphWalker Studio sidebar: the "Element" section for e_IncrementCounter 24
7 GraphWalker Studio sidebar: the "Execution" section 24

8 Overview of the SUT and related components and parties, adapted from [13] 29
9 The vehicle selection view 30
10 The initial configuration view 31
11 The main view with vehicle mode buttons shown 32
12 The main view top bar with lights and camera items 32

13 Overview of the testing setup 33
14 POM classes related to the "Change Date" view 36
15 The MainViewVehicleModes GraphWalker model 39
16 The SettingsViewModes GraphWalker model 40
17 The modes tab of the settings view in the SUT 40

18 Selenium dependency pom.xml entry 46
19 The initial scenario GraphWalker model 49
20 The initial scenario UPPAAL SUT model 52
21 The initial scenario UPPAAL SUT model with trap variables TS1 and TS2 55

1. Introduction

Software testing helps to improve and maintain software quality, therefore it plays an important role in software development. There are different approaches to software testing. Among traditional software testing methodologies are automated tests in the form of unit, integration and user interface (UI) tests; in general, these are manually created.

A more novel approach is to apply model-based testing (MBT). In the case of MBT, tests are generated based on one or more models. These models represent the behavior of the System Under Test (SUT), while being notably simpler than the system being modelled. This kind of testing can have several benefits. One important benefit is that model-based tests can often be easier to maintain. Should the SUT change, it suffices to update its model and/or model implementation accordingly to make the tests work again. Another benefit of MBT is that tests generated based on models can more easily achieve better test coverage while potentially including more exceptional situations. Achieving the same with more traditional testing techniques may require higher experience from test authors and can often require a lot of work. Models are generally more concise than test code and thus may be understood by more people than just programmers.

As part of this thesis, the author tests a web application using different, alternative MBT tools. One of them is a combination of primarily such tools as modelling software UPPAAL and distributed testing tool DTRON. The other tool that is used independently from the previously mentioned ones is an open source MBT software GraphWalker. In both of the cases, interaction with the SUT (which in this case is a web application) is implemented using Selenium, applying a test design pattern named Page Object Model.

The application to be tested is Milrem Robotics THeMIS user interface project. The user interface is based on web technologies (HTML, CSS and JavaScript/TypeScript), thus it can be tested with software intended for web application testing.

The goal of this thesis is not to achieve full test coverage, but to evaluate, analyze and compare the testing process with the mentioned two alternative tool combinations and with MBT in general. In the case of GraphWalker, thanks to it being open-source software, the author also puts (initial) effort into making the tool better for everyone.

2. Technical Background

The following sections are intended to give an introduction to the technical topics this thesis builds upon. The main technique applied in this thesis is model-based testing. The latter is used to perform web application testing. To ease the development and improve the maintainability of the web application tests, the Page Object Model design pattern is followed.

2.1 Model-based testing

Utting *et al.* [1] describe the process of model-based testing and provide a taxonomy based on it. The aim of testing in general is to show that the System Under Test (SUT) behaves differently than what is intended. If testing does not expose failures, then the goal is to at least gain some confidence that the system behaves correctly.

"Model-based testing (MBT) is a variant of testing that relies on explicit behaviour models that encode the intended behaviours of an SUT and/or the behaviour of its environment. Test cases are generated from one of these models or their combination, and then executed on the SUT." [1]

In the case of traditional testing, tests are created in a way that can often be unstructured, nonreproducible, undocumented, without detailed description of test design decisions, and rely on the skills of individual engineers. The fact that MBT utilizes the models of the intended SUT and possibly the environment can help reduce the mentioned problems.

"MBT encompasses the processes and techniques for the automatic derivation of abstract test cases from abstract models, the generation of concrete tests from abstract tests, and the manual or automated execution of the resulting concrete test cases." [1]

A test case can be described as a finite structure of input and expected output. The exact structure depends on the type of the system.

Utting *et al* describe the process of MBT as the following five steps:

1. Build a model of the SUT based on informal requirements or existing specification. This model is often referred to as being a *test model* as its abstraction level and focus are dependant on testing objectives. While in some cases it is possible to utilize a single model for testing and SUT design, it is desirable to have some independence between the test model and any development models to avoid errors in the development model propagating to the generated tests. It is preferable for the test models to be more abstract than the SUT otherwise validating them would be equally difficult to validating the SUT itself. At the same time, the models must be precise enough so that "meaningful" test cases can be generated from them.
2. Choose test selection criteria, that instruct the automatic generation to arrive at a test suite which complies with the test goals defined for the SUT. Test selection criteria might be based on certain functionality of the system, structure of the test model, data coverage heuristics, stochaistic characterizations including pure randomness or user profiles, properties of the environment, and well-defined set of faults.
3. Transform test selection criteria into formalized *test case specifications*. Provided a model and a test case specification, an automatic test case generator must be able to derive a test suite. In the case of state coverage of a finite state machine, the corresponding test case specifications might be about reaching specific states of the machine. In this sense, a test case specification is essentially a high-level description of a test case that is desired.
4. Given the model and test case specifications, a set of test cases is generated that satisfy all the test case specifications. If the set of test cases satisfying a test case specification with regards to a model is empty, then the test case specification is considered to be *unsatisfiable*. In the usual case of several or many tests cases satisfying a test case specification, the test case generator will choose a single one of them. Some test generators make it possible to minimize the test suite to cover a large number of test case specifications with a small number of generated test cases.
5. Run the test cases. This process can be manual (executed by a physical person) or may be automated by the use of a *test execution environment* which also records test verdicts. It is also possible to generate and run the tests interleavingly, which is known as *online* testing.

The model and the SUT are at different levels of abstraction. Therefore it is necessary to bridge those levels in order to run a test case. Utting *et al* give an example abstract test case for a bookshop web site which can be written as *checkPrice(WarAndPeace) = $19.50*, where *checkPrice* represents the name of the webservice used, *WarAndPeace* is the book that is the subject of the query, and $19.50 is the expected result value. Executing a test case begins by concretizing the test inputs with the help of a component called the *adapter* (in this case it may involve arriving at a detailed web service call). Once the concrete

output is received from the SUT, the adapter is used to abstract the output into a high-level result that can be compared with the expected result.

Utting *et al* proceed by proposing a taxonomy of model-based testing that is based on the previously described process. As can be seen from Figure 1, at the highest level the taxonomy is divided into Model Specification, Test Generation, and Test Execution. The Model Specification part of the taxonomy is divided further into Scope (either input-only or input-output model), Characteristics (deals with timing, (non)determinism and time dynamics), and Paradigm (represents various ways in which the model can be described).

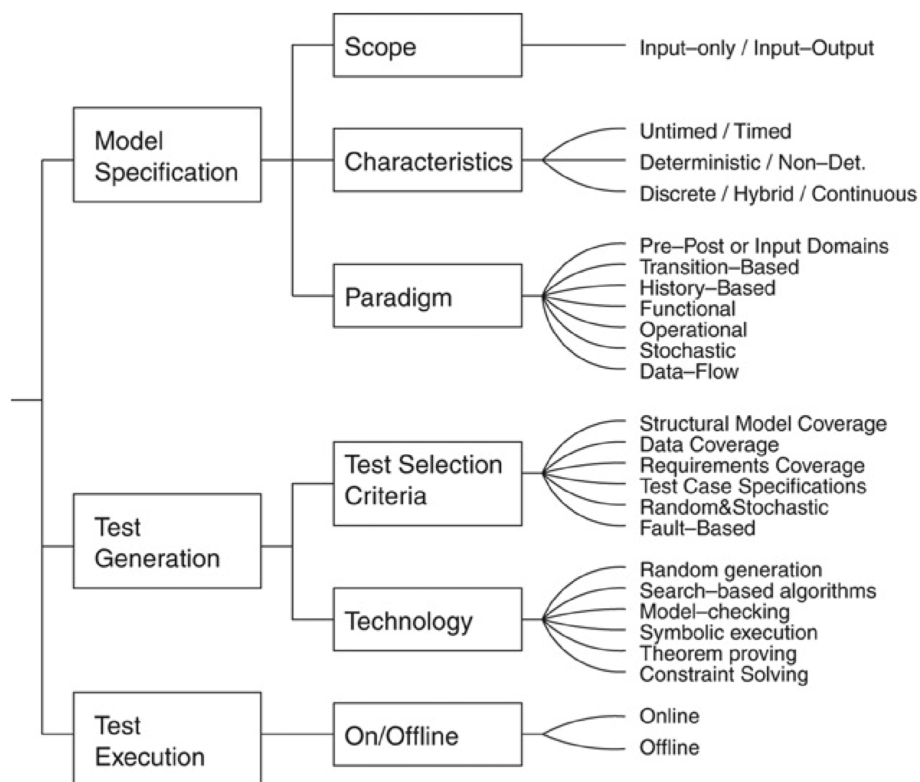


Figure 1. Overview of the Taxonomy [1]

The Test Generation part of the taxonomy is divided into Test Selection Criteria (includes various ways of specifying test coverage requirements and other methodologies of specifying test goals) and Technology (consists of different solutions for test generation).

Finally, the Test Execution part of the taxonomy deals with online test execution (by which tests are generated and executed interleavingly) and offline test execution (according to which tests are generated beforehand and can be run later, possibly multiple times).

Model-based testing determines the conformance between the test models and the System Under Test.

"Conformance testing is a kind of functional testing: an implementation of a protocol entity is solely tested with respect to its specification. Only the observable behaviour of the implementation is tested, i.e. the interactions of an implementation with its environment; no reference is made to the internal structure of the protocol implementation. In practical conformance testing the internal structure of the entity is usually not even accessible to the tester: the computer system in which the entity under test is located need not be accessible. The tester can only observe the entity by communicating with it." [2]

2.2 Web application UI testing

Automated web application UI testing is typically done by starting the System Under Test, extracting information from the UI and sending commands to it, usually by emulating key-presses and/or mouse actions. This kind of testing can be more time consuming, regarding both development and execution. In comparison, unit testing a backend functionality in code level can be often done by directly executing calls and checking return values, which may consume noticeably less effort. In contrast, web application UI testing on one hand requires that there is a way to obtain current state from what is displayed, especially if black box testing is performed. There's also the need for performing actions in the UI, and that may be in the form of emulating mouse movements, clicks, scrolling, various touch actions and/or keypresses etc. In addition, there is the issue of synchronization/waiting — the UI may take some time to start up or react to certain actions. In a single-thread backend test, these kind of timing issues might not be as prevalent.

2.3 Page object model pattern

Page Object Model[1] (POM) is a design pattern used in testing. According to this pattern the internal details of various *Page Objects* in the System Under Test that are to be tested should be encapsulated within their own classes in testing code. In the realm of browser based testing, this notably includes selectors for various elements in a Page Object. These classes can contain getters for various states, items, content on the page, and also methods for performing actions regarding the Page Object.

The main benefits of the Page Object Model design pattern are that it reduces code duplication and (partly thanks to reduced duplication) improves code maintainability. If something changes in the System Under Test, the relevant Page Object Model can updated to take the change into account. As long as the change isn't so major that the existing API of the Page Object Models should be changed (e.g. some functionality is removed or

[1] https://www.selenium.dev/documentation/en/guidelines_and_recommendations/page_object_models/, accessed April 12th, 2021

moved), the tests that use the POM classes can remain working unaltered. So, for example, if the placement of some button or the HTML structure of changes within the same page, changing just the relevant POM class is usually sufficient. The design pattern is still useful even when dealing with major breaking changes.

In one industrial case study [3] it was found that utilizing the Page Object Pattern helped reduce the time required for repairing a test suite by a factor of roughly three and the number of modified lines of code by a factor of roughly eight.

The term *Page* Object Model refers to, as its name implies, to whole pages in the UI. There is a related concept named Page *Component* Object Model, which is about specific components in the UI. This is especially useful if there are components in the UI that are present in different kinds of pages. There's also the possibility that Page Object Models can return instances of (different) Page (Component) Object Models. This can help reduce duplication and improve maintainability even further.

As mentioned previously, one of the benefits of model-based testing is that it can make test maintenance easier. The Page Object Model design pattern improves the maintainability aspect even further. Maintenance-wise, whereas the benefits of model-based testing can be easily seen with regards to larger changes to the SUT (e.g. altering the way states in the SUT can be reached), the Page Object Model design pattern deals with the internal, more implementation-specific side of the SUT.

3. Tools Used

This chapter gives an overview of the main tools used in this thesis. UPPAAL, DTRON and TDL^{TP} form one combination of tools used in this thesis to implement model-based testing. GraphWalker in conjunction with JUnit 5 forms the other combination. In both cases the web application UI testing part is implemented using Selenium. Java is the programming language used in this thesis.

3.1 UPPAAL

As said on its homepage[1]:
"Uppaal is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.)"

The name UPPAAL is derived from the two universities that are developing the tool: Uppsala University, Sweden (Department of Information Technology) and Aalborg University, Denmark (Department of Computer Science).

The implementation of Uppaal applies the theory of timed automata. A timed automaton is an extension of finite-state machine that adds clock variables. The values of theses clock variables evaluate to a real number and all of the clocks progress synchronously. There is also the addition of bounded discrete variables that make up the state. These variables are used in a similar way as in programming languages [4].

Definition of Timed Automaton[4].
"A timed automaton is a tuple (L, l_0, C, A, E, I), where L is a set of locations, $l_0 \in L$ is the initial location, C is the set of clocks, A is a set of actions, co-actions and the internal τ-action, $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset, and $I : L \to B(C)$ assigns invariants to locations."

"[...] $B(C)$ is the set of conjunctions over simple conditions of the form $x \bowtie c$ or $x-y \bowtie c$, where $x, y \in C, c \in N$ and $\bowtie \in \{<, \leq, =, \geq, >\}$." [4]

[1] https://uppaal.org/, accessed April 10th, 2021

Figure 2 gives a small example of a system in UPPAAL. Using the terminology of UPPAAL, there are two processes with two locations each — Process1 contains FirstLocation and SecondLocation; Process2 contains ThirdLocation and FourthLocation. The location pairs in both processes are connected by two directed edges. The FirstLocation and ThirdLocation are the initial locations of Process1 and Process2, respectively. Let's first look at the edge leading from FirstLocation to SecondLocation. The *rand : int[0,10]* part is called a *Select* and it makes it possible to select a random value and save it to a variable. In this case, when the edge is traversed, a random integer between 0 and 10 is selected and saved into a (local) variable named *rand*. The *canEnterSecondLocation* is the *Guard* of this edge — the edge can be traversed only if this guard evaluates to *true*. The *channel_a?* part represent this edge's *Sync* or synchronization with regards to channels. In this case, provided that its guard is satisfied, the edge will be traversed when a synchronization with *channel_a!* comes from the other process in the system via the traversal of the edge leading from ThirdLocation to FourthLocation. Finally, the x = *rand* part is the *Update* of this edge. It is typically used to update the values of one or more variables after an edge has been traversed; in this case the value of the variable *rand* is assigned to the variable x. All of the previously mentioned parts of an edge are optional. Looking at the edge leading from SecondLocation to FirstLocation, we see that it has a *Sync* with value *channel_b!*. Traversing this edge will synchronize the edge of Process2 leading from FourthLocation to ThirdLocation as the latter has a corresponding *Sync* of *channel_b?*.

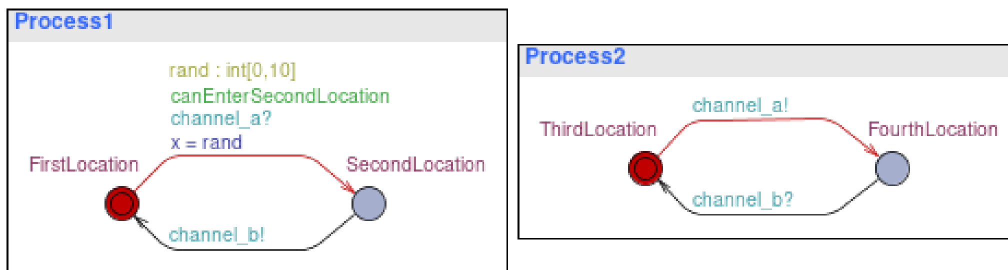


Figure 2. A small example of an UPPAAL system

UPPAAL uses a C-like programming language that can be used in global and local declarations to declare variables and channels, and define functions; the language can be used to declare processes in system declarations. In the case of the previous example, channels *channel_a* and *channel_b* were declared globally, whereas the variables x and *canEnterSecondLocation* were declared locally for Process1, and the processes Process1 and Process2 were declared as part of system declarations.

UPPAAL Yggdrasil [5] is a tool built into UPPAAL that can be used to generate offline test cases. In the UPPAAL model editor, it is possible to set test code that should be executed when a specific location is entered or exited or when a transition is taken. UPPAAL Yggdrasil can then generate symbolic test cases that correspond to UPPAAL traces. The

symbolic test cases are then concretized with the test code supplied in the editor. The test code is in plain text which makes it possible to use arbitrary programming languages for it. UPPAAL Yggdrasil is not utilized in this thesis.

3.2 DTRON

The UPPAAL toolbox provides an extension named TRON (Testing Real-time systems ONline) [6], which is originally meant to be used for conformance testing, but can also be utilized for model-based discrete control. To make TRON model-based control module work together with a controllable object, it is necessary to have an *adapter* on the latter side. TRON can handle a single *tester-testee* pair and doesn't naturally scale for more than one tester and testee. This in turn means that TRON isn't easily usable for distributed control applications. What negatively affects the experience of using TRON is that it is necessary to put a lot of work into implementing adapters between controllers and control objects. This in turn means that any kind of configuration changes require re-writing code on both sides of the affected adapters [7].

Distributed TRON (DTRON) [7] is a framework that extends the TRON tool with the ability to multicast messages between TRON instances that run in parallel. DTRON implements messaging in *publish* and *subscribe* fashion by utilizing the Spread[2] toolkit. The usage of *the dependency injection* design pattern makes the *controller*-controllable object pairs *loosely coupled* which in turn allows much better scaling.

The term multicast means sending a message to possibly multiple recipients. To make use of DTRON, the control models can be provided with designated transitions so that DTRON can intercept them and inform other control agents about it. These designations are defined by *synchronized transitions*. Thanks to the fact that multicast is used for message passing, agents can dynamically join and leave the environment without there being a need for re-configuration of the existing infrastructure. The only thing necessary is an agreement or protocol covering message definitions, their data and the timing of their traversal in the multicast [7].

3.3 TDL^{TP}

TDL^{TP} (Test Scenario Definition Language) [8] is a high-level test scenario specification language that can be used for specifying complex test scenarios that can find its use in model-based testing of mission critical systems.

[2] `http://spread.org/`, accessed May 19th, 2021

There have been previous attempts at implementing test purpose specification languages for MBT. Because of the lack of standardization, high-level tests created in some of them are strongly tool-dependent and are therefore usable only in testing processes involving specific tools. Other test languages have been created which are meant to be used in specific domains, their usability in other domains is therefore limited. There have also been created test scenario specification languages but one of the first of them had loose semantics which restricted its use as a consistent test description language. Concrete test scripting languages may have strict semantics but they are not well suited to use in describing test scenarios in a higher level — the syntax of those is reminiscent of imperative programming languages [8].

European Telecommunications Standards Institute (ETSI) has developed the Test Description Language (TDL) [9]. Although TDL provides one of the most advanced test purpose description languages it still has some limitations. One limitation of TDL concerns the fact that its timing semantics are limited [8].

Uppaal allows the usage of Timed Computation Tree Logic (TCTL). An important limitation of that is the fact that the TCTL syntax in Uppaal tool does not allow nesting operators, which makes the TCTL expressions flat with regards to temporal operators. These kind of flat expressions can't be used to represent more complex properties [8].

The Test Scenario Definition Language — TDL^{TP} [8] is an extra language layer "for test scenario specification that is expressive, free from the limitations of 'flat' TCTL, interpretable in Uppaal TA, and suited for test generation." TDL^{TP} generates a test model based on a SUT model and a TDL^{TP} coverage expression. For this, the SUT model's structural elements of interest, also called test coverage items (TCIs), have to be labelled with Boolean variables referred to as *traps*. A group of *traps* is called a *trapset*.

The syntax of TDL^{TP} expressions in Backus-Naus form is:

⟨*Expression*⟩ ::=
    '(' ⟨*Expression*⟩ ')'
   | *'A'* ⟨*TrapsetExpression*⟩
   | *'E'* ⟨*TrapsetExpression*⟩
   | ⟨*UnaryOp*⟩ ⟨*Expression*⟩
   | ⟨*Expression*⟩ ⟨*BinaryOp*⟩ ⟨*Expression*⟩
   | ⟨*Expression*⟩ ~> ⟨*Expression*⟩
   | ⟨*Expression*⟩ ~> '[' ⟨*RelOp*⟩⟨*NUM*⟩ ']' ⟨*Expression*⟩
   | '#' ⟨*Expression*⟩ ⟨*RelOp*⟩ ⟨*NUM*⟩

⟨*TrapsetExpression*⟩ ::=
    '(' ⟨*TrapsetExpression*⟩ ')'
    | '!' ⟨*ID*⟩
    | ⟨*ID*⟩ '\' ⟨*ID*⟩
    | ⟨*ID*⟩ ';' ⟨*ID*⟩

⟨*UnaryOp*⟩ ::= '*not*'
⟨*BinaryOp*⟩ ::= '&' | '*or*' | '=>' | '<=>'
⟨*RelOp*⟩ ::= '<' | '=' | '>' | '<=' | '>='
⟨*ID*⟩ ::= ('*TR*')⟨*NUM*⟩
⟨*NUM*⟩ ::= ('0'..'9')+

This thesis makes use of TDL^{TP} interpreter [10]. It uses a slightly altered syntax for TDL^{TP} expressions.

3.4 GraphWalker

The previously mentioned three tools form one combination of model-based testing tools that are used in this thesis. GraphWalker[3] is an open-source model-based testing tool that is an alternative to that combination. GraphWalker comes with the browser-based Studio capable of creating and editing models, and visually showing how the models would be traversed when executing tests. GraphWalker is written in Java and thus natively supports model implementations in this programming language.

GraphWalker is based on finite state machines which are visualized as graphs in the Studio, i.e. vertices that are connected by edges. Each vertex and edge can have a name and each non-empty name is associated with a Java method. Vertices correspond to states and this is where checks/assertions are performed. Edges correspond to actions.

To make the models easier to maintain and more understandable, it can be beneficial to split them up. In GraphWalker, this can be achieved by utilizing shared state (also called shared name). Each vertex can have a shared state of string type and GraphWalker can directly jump between vertices of different models if these vertices have the same shared state.

When executing a test in GraphWalker, there needs to be a combination of a path gener-

[3] `http://graphwalker.github.io/`, accessed April 10th, 2021

ator and a stop condition. A path generator tells GraphWalker how the models should be traversed, while a stop generator determines when the test execution should stop. GraphWalker comes with several path generators and stop conditions.

GraphWalker can interface with the system under test in different ways. There's a Maven plugin which can help create Java interfaces from models. Then there's the option to interface with the SUT over WebSocket or a Restful API. Test can be executed offline or online.

Figure 3 gives an example of a model in GraphWalker Studio. Following the naming convention of GraphWalker (prepend vertex names with *v_* and edge names with *e_*), there are three vertices — *v_InitialVertex*, *v_SecondVertex* and *v_SharedVertex*. The first of the three is, as its name implies, the initial vertex. *v_SharedVertex* has shared state, which means that it is possible to traverse from this vertex to vertices in other models with the same shared state. *v_SecondVertex* is a regular vertex, i.e. it is not initial nor does it have shared state. There are also three edges — *e_IncrementCounter* (leads from *v_InitialVertex* back to itself), *e_EnterSecondVertex* (leads from *v_InitialVertex* to *v_SecondVertex*) and *e_EnterSharedVertex* (leads from *v_SecondVertex* to *v_SharedVertex*).

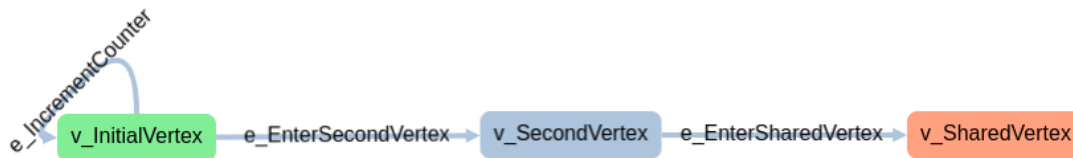


Figure 3. A small example of a model in GraphWalker Studio

GraphWalker shows less information in its model editor view (as compared to Figure 2 that showed an example UPPAAL model). Therefore, let's look at some more details of this example model that can be seen in other parts of the GraphWalker Studio user interface. The Studio can work with multiple models — a tab corresponds to each open model. By clicking the bottommost icon in the left-side vertical toolbar, a "Properties" sidebar is opened next to the toolbar (Figure 4 gives an overview of the Studio UI with the sidebar open). At the top of the sidebar, there is a section titled "Model" (Figure 5). Here, the name of the model can be specified and it is also possible to specify initial actions which is often used to initialize model variables — variables defined with the prefix *"global."* are global, other variables are local to the model. In the case of this model, the name is "Model" and as an initial action the model-local variable *counter* is set to be zero.

Below the "Model" section is the "Element" section which applies to concrete elements of the model (vertices and edges). Figure 6 shows the "Element" section for the edge *e_IncrementCounter*. The section allows to set the name for the element, set the shared

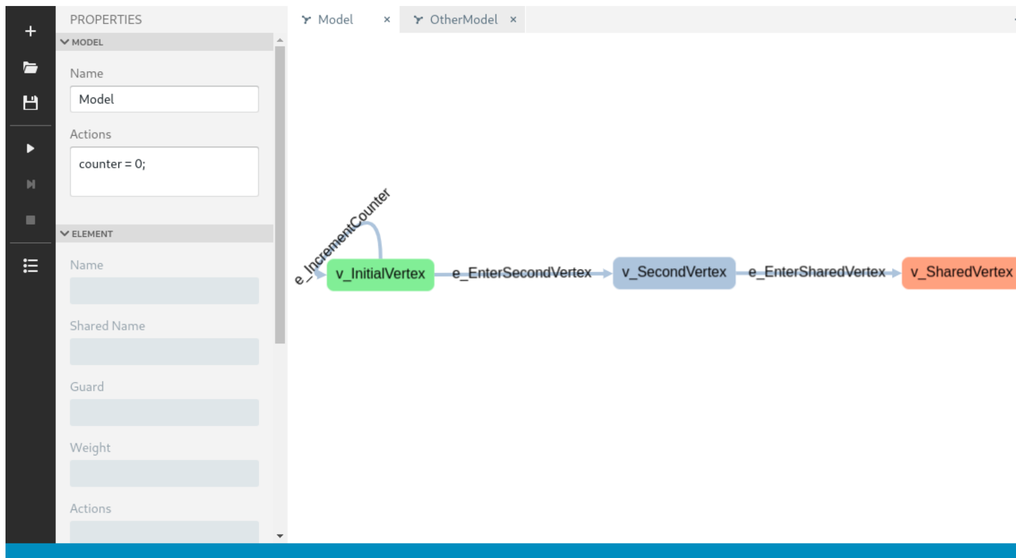


Figure 4. GraphWalker Studio overview with its sidebar open

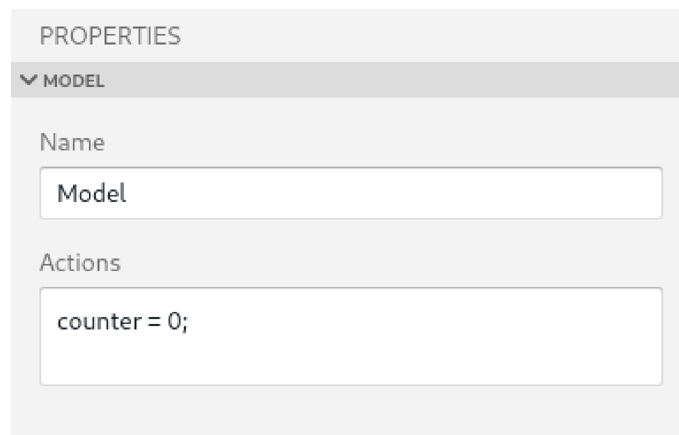


Figure 5. GraphWalker Studio sidebar: the "Model" section

name for vertices, set the guard and weight for edges, and specify actions, requirements, toggle the "Start element" (i.e. initial) attribute for both vertices and edges. In the case of *e_IncrementCounter*, the weight is 0.8 and action naturally is to increment the counter by one.

What is not indicated in the figure is that the edge *e_EnterSecondVertex* has *"counter > 5"* as its guard and 0.2 as its weight.

Finally, the sidebar contains the section titled "Execution" (Figure 7). The values specified here are used when the model is visually traversed in the GraphWalker Studio. The value of "Generator" specifies the path generator together with the stop condition. In this case, the path generator is "weighted_random" which chooses a random neighbouring edge as the next element of the path while honouring the edges' weight values, and the stop condition is "edge_coverage(100)" which stops after full edge coverage is achieved. When

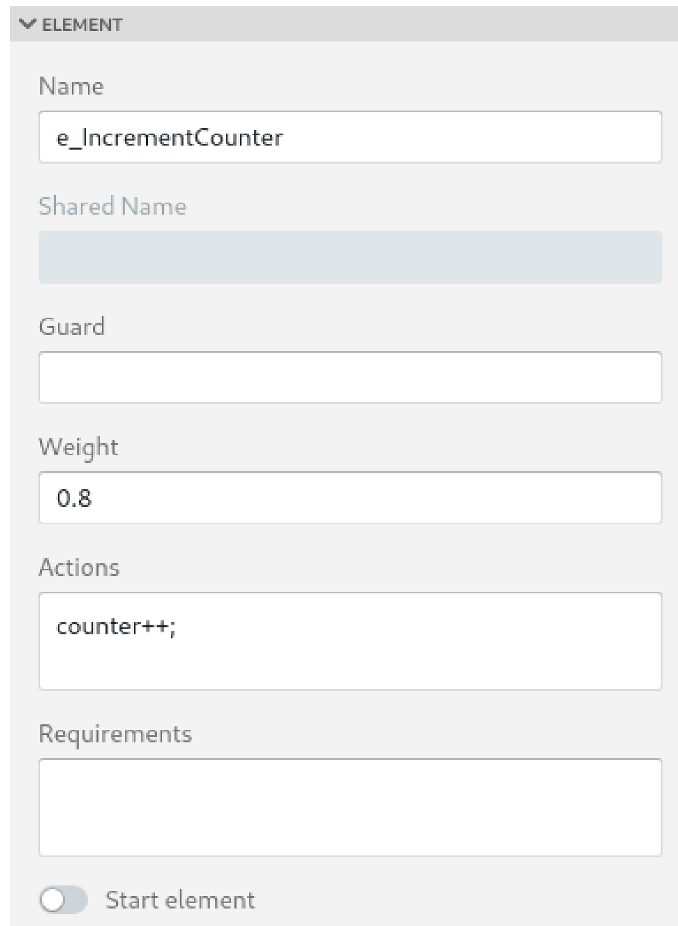


Figure 6. GraphWalker Studio sidebar: the "Element" section for e_IncrementCounter

visually traversing this example model in GraphWalker studio, it can be seen that the edge *e_IncrementCounter* is traversed at least six times before *e_EnterSecondVertex* can be considered, as the counter has to be more than five for the latter edge's guard to allow traversing it. Taking into account that we are using a path generator that honours edge weights, the incrementing edge can likely be traversed more than six times (as its weight is 0.8) before traversing the edge (with weight 0.2) leading to the second vertex. The value of "Delay" (in milliseconds) can be used to adjust the speed of traversal.

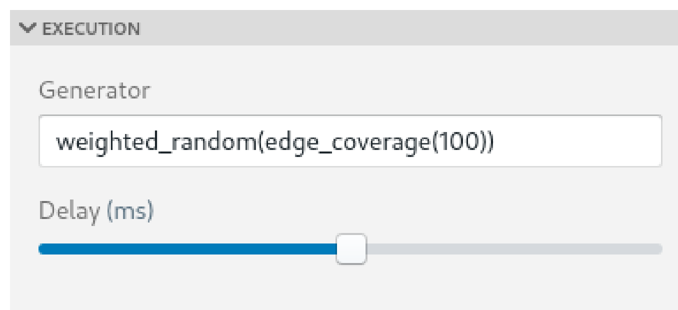


Figure 7. GraphWalker Studio sidebar: the "Execution" section

3.5 Selenium

Selenium[4] is a tool that enables the automation of browsers. It is primarily used for browser-based testing and this is how it is used in this thesis, too. Selenium has official bindings[5] for the following programming languages:

- Ruby
- Java
- Python
- C#
- JavaScript

Garcia *et al.* [11] present a survey of Selenium ecosystem. With regards to the background of Selenium, they point out that the initial version of this testing tool is nowadays known as Selenium Core. Selenium Core is a JavaScript library that operates on the so-called Selenese commands. These commands have three parts:

1. Command (the action to be performed in the browser)
2. Target (locates/identifies the HTML element on which the command will performed)
3. Value (necessary data for the command, such as text to be entered in a web form)

The authors of Selenium created a newer project called Selenium Remote Control (Selenium RC) which added a scripting layer to Selenium Core. Selenium RC is based on a client-server architecture — scripts written in various programming languages (such as Java, Python and several others) take the role of the clients which send Selenese commands to Selenium RC server (an intermediate HTTP proxy). The Selenium RC server launches web browsers as necessary and injects the SUT with the Selenium Core JavaScript library which performs browser automation based on Selenese commands.

There were notable limitations to this approach. Due to the fact that its implementation of browser automaton was based on JavaScript, some commands aren't possible, such as file upload/download and handling of dialog boxes. There is also considerable overhead to the performance of test execution. These are among the reasons why a new tool called Selenium WebDriver was developed. Whereas its use case is similar to Selenium RC — WebDriver, too, makes it possible to automate browsers using different programming languages — its architecture is completely different. WebDriver overcomes the previously mentioned limitations by relying on browsers' native automation support instead of inject-

[4]`https://www.selenium.dev/`, accessed April 10th, 2021
[5]`https://www.selenium.dev/downloads/`, accessed April 10th, 2021

ing a JavaScript-based library into the SUT. This native support for browser automation is implemented by a platform-dependent component called the driver. The driver for Google Chrome is called *chromedriver* and the Firefox counterpart is called *geckodriver*. The Selenium client and the driver communicate over HTTP in a form of JSON messages standardized by W3C. Selenium Core and RC have been deprecated in favor of WebDriver.

Other projects in the Selenium family include the Selenium Grid (makes it possible to drive web browsers running in parallel on remote hosts) and the Selenium IDE (a tool that makes it possible to record, edit and playback Selenium scripts).

An important concept used in Selenium are selectors. Selectors make it possible to find HTML elements of the SUT. Some of the selectors are based on different attributes, such as the element's *id* or *class*. There are also more powerful ones utilizing CSS selectors or XPath. Once an HTML element is located, its attributes can be queried and actions can be performed on it. It is also possible to find the element's (direct or indirect) children, again utilizing selectors.

One modern alternative to Selenium is Cypress[6], a JavaScript-based testing tool that runs in web browsers. In a technical analysis [12] that compares Selenium and Cypress, it was pointed out that the latter can handle dynamic web applications better (especially when considering its automatic waits and simplified asynchronous testing). The analysis included testing a real world website (AliExpress[7]) with both tools. It was found that Cypress tests ran somewhat faster and consisted of less code than equivalent Selenium tests. Regardless of the previously mentioned advantages of Cypress, this thesis uses Selenium because it is much easier to integrate it with the model-based testing tools introduced previously, considering that the tools don't have JavaScript support.

3.6 Java

Java is a statically and strongly typed programming language. This is the language that is used in this thesis notably for programming the page object models for Selenium, the adapter for DTRON-based testing, and model implementations for GraphWalker-based testing.

This language was chosen, because this is what DTRON and GraphWalker support natively. Being a strongly and statically typed language, Integrated Programming Environments (IDE) can be more helpful when dealing with Java as compared to some other, dynamically

[6] `https://www.cypress.io/`, accessed May 15th, 2021
[7] `https://www.aliexpress.com/`, accessed May 15th, 2021

typed programming languages.

With some extra effort, other programming languages could have been chosen. Alongside Java, DTRON also has language bindings[8] for C++. Under the hood, DTRON uses Spread[9] as its publish-subscribe messaging service. Besides Java, Spread also supports[10] the programming languages C/C++, Perl, Python and Ruby. This means that C++ could be used with DTRON natively, but C++ is a quite complex programming language, and the possible performance gains are not worth the extra development time in the thesis author's opinion. This leaves us with C, Perl, Python and Ruby which can be used with DTRON if some extra development work is applied. The C programming language does not fit well into a modern testing environment and besides that, it is not supported by Selenium. The other three are dynamic programming languages, which can be beneficial in some ways, but a statically typed programming language like Java can help avoid common errors better and is also better to use in an IDE.

GraphWalker natively supports Java. At least in theory, it would be possible to also use other languages that run on the Java Virtual Machine (JVM). These include Kotlin, Clojure, Groovy and Scala, among quite a few others. It is also possible to get native support for Python and C#/.NET by using an alternative GraphWalker-based MBT tool named AltWalker[11]. For this thesis neither of the described options is used, mostly to reduce the complexity of the technological stack.

3.7 JUnit 5

JUnit 5 is *"[t]he 5th major version of the programmer-friendly testing framework for Java and the JVM"*[12].

JUnit 5 is used in conjunction with GraphWalker. Firstly, it is used in model implementation classes — namely to implement assertions in methods corresponding to model vertices. JUnit 5 is also used when specifying different test cases, which can differ in the path generators and stop conditions used. The usage of this tool in this case makes it possible to run multiple test cases and see which of them pass and which fail.

JUnit 5 is not used in the case of UPPAAL + DTRON, instead conformance testing is

[8] `https://cs.ttu.ee/dtron/usage.html#Language_bindings`, accessed April 23rd, 2021

[9] `http://spread.org/`, accessed April 23rd, 2021

[10] `http://spread.org/SpreadPlatforms.html`, accessed April 23rd, 2021

[11] `https://altom.com/testing-tools/altwalker/`, accessed April 23rd, 2021

[12] `https://junit.org/junit5/`, accessed April 10th, 2021

utilized for those. In that case, there is a model that represents how the SUT should behave, which is then compared with the real SUT. Any difference between the model of the SUT and the real SUT halts the testing process with a report of failure.

The approach is different in the case of GraphWalker. Here, the model is more abstract and doesn't describe in detail how the SUT should behave. Instead, these details are provided in the model implementation. It is up to the model implementation to perform checks/asserts in the vertex methods to determine whether the SUT performs correctly. This thesis uses JUnit 5 for these asserts.

4. The System Under Test

The system to be tested as part of this thesis is the user interface web application project for THeMIS[1] Unmanned Ground Vehicle (UGV) developed by Milrem Robotics[2]. Among other features, it allows the user to see the UGV camera image, the vehicle statuses, and give commands to the vehicle. The current frontend of the user interface was initially developed as part of the thesis author's Bachelor's thesis [13].

Figure 8 gives an overview of the SUT components, related components and parties. Technically, the user interface has two main layers. It has a frontend based on web technologies (HTML, CSS and JavaScript/TypeScript) that runs in a Chromium-based browser and connects to a backend over a WebSocket. The frontend and backend run on the same device — a touchscreen-equipped tablet. The backend is what connects with the vehicle and also interfaces with a joystick. The user uses the joystick and interacts with the frontend using the tablet. The joystick is used to give direct movement commands to the connected UGV; for safety reasons, this is something that cannot be done via the tablet touchscreen/frontend.

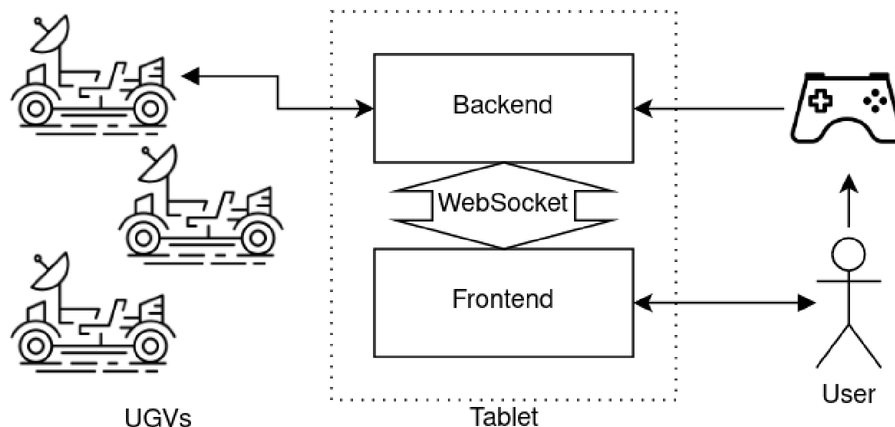


Figure 8. Overview of the SUT and related components and parties, adapted from [13]

For demonstration and testing purposes, there is a mock which replaces the WebSocket part of the user interface and by doing that, allows the UI frontend to be executed without running the backend. This means that one can open a local or intranet URL in a web

[1] `https://milremrobotics.com/defence/`, accessed April 10th, 2021
[2] `https://milremrobotics.com/`, accessed April 10th, 2021

browser and try out the UI. This mock was used throughout the testing process in this thesis. From one point of view, using the mock makes the testing process simpler and allows to put more effort into the testing solutions themselves. The mock is almost entirely free from persistent state so refreshing the browser is all that is necessary to start from a fresh state. The mock also makes it possible to simulate certain events like vehicle faults and connection loss. These events can be triggered via keyboard keys, which makes them also usable when demonstrating the UI. But on the other hand, using a mock means that bugs caused by the backend are not in the scope of testing. Fortunately, the tests themselves can stay mostly unchanged when moving towards more integrated tests.

The following gives an overview of the user interface's initial views through screenshots. Once the UI is started, the vehicle selection view is initially shown (Figure 9). If there are vehicles to connect to, these will be listed as buttons, otherwise an appropriate message is shown.

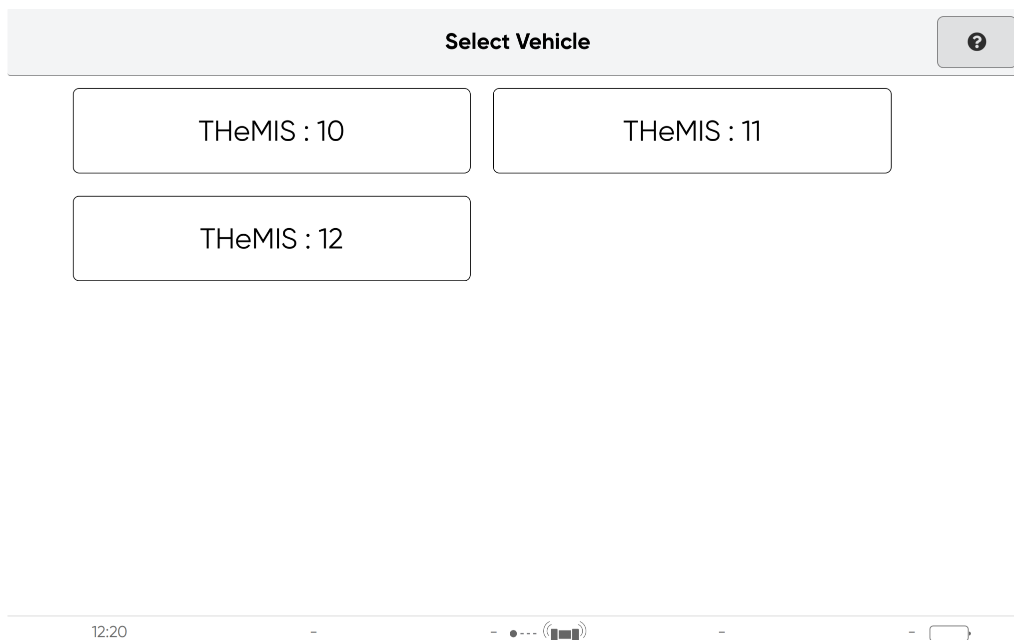


Figure 9. The vehicle selection view

Clicking on the button corresponding to a vehicle navigates the user to the initial configuration view (Figure 10). In this view, it is possible to configure some (mostly vehicle) modes prior to actually connecting to the vehicle. At this point, it is also possible to discard initial configuration and go back to vehicle selection by clicking on the left hand side button of the top bar (called the header bar).

If the user does not discard the initial configuration, but instead clicks on the header bar's right hand side button, then the configuration is submitted and the main view is opened (Figure 11). The content area of the main view shows the currently chosen camera streams.

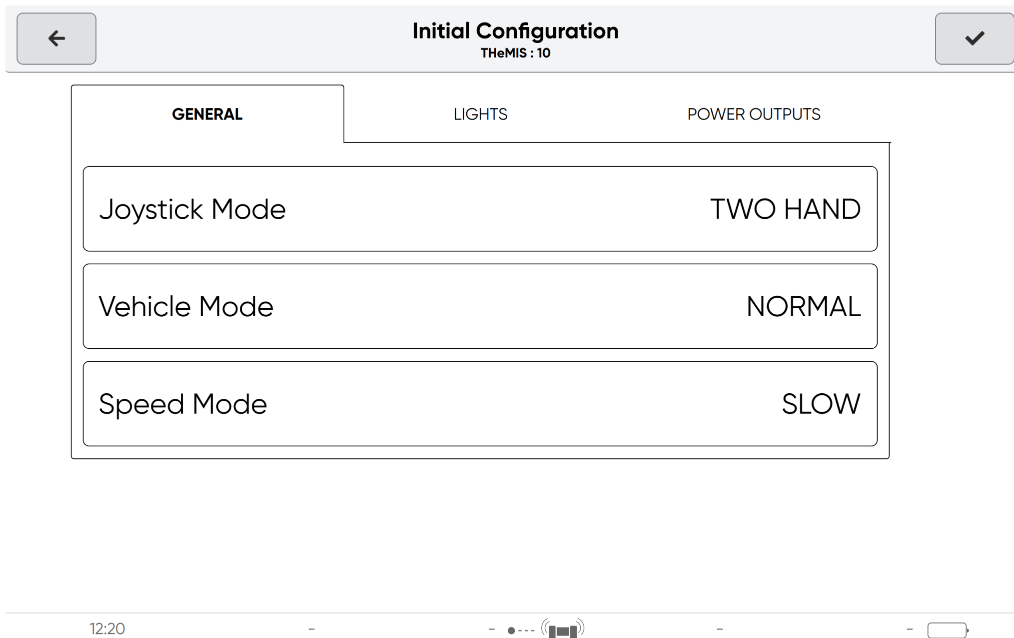


Figure 10. The initial configuration view

The view has a top bar that can be used to change vehicle modes — clicking on one of the four buttons reveals a dialog box with options; if the leftmost flip button is pressed, items for toggling vehicle lights and changing the active camera are revealed (Figure 12) — choosing these items open a second bar with appropriate options below the top bar. There's a button on the right side of the top bar which opens the settings view. The bottom bar of the main view shows various data from the vehicle.

There are two broad parts of the UI with regards to how it is used — manual driving (the so called Drive UI) and navigation. Initially the focus of this thesis is to test the Drive UI. The user interface can be built with or without the navigation feature. If the navigation feature has been enabled during building, there is an additional item in the main view top bar next to lights and camera options which allows the entry into several navigation features, such as the mission planner. This is a notably more complex part of the SUT, even when considering only the frontend. Due to this, it can also be more difficult to test, especially when the usage of mocking is reduced. Some initial work has been made to start testing this part of the SUT.

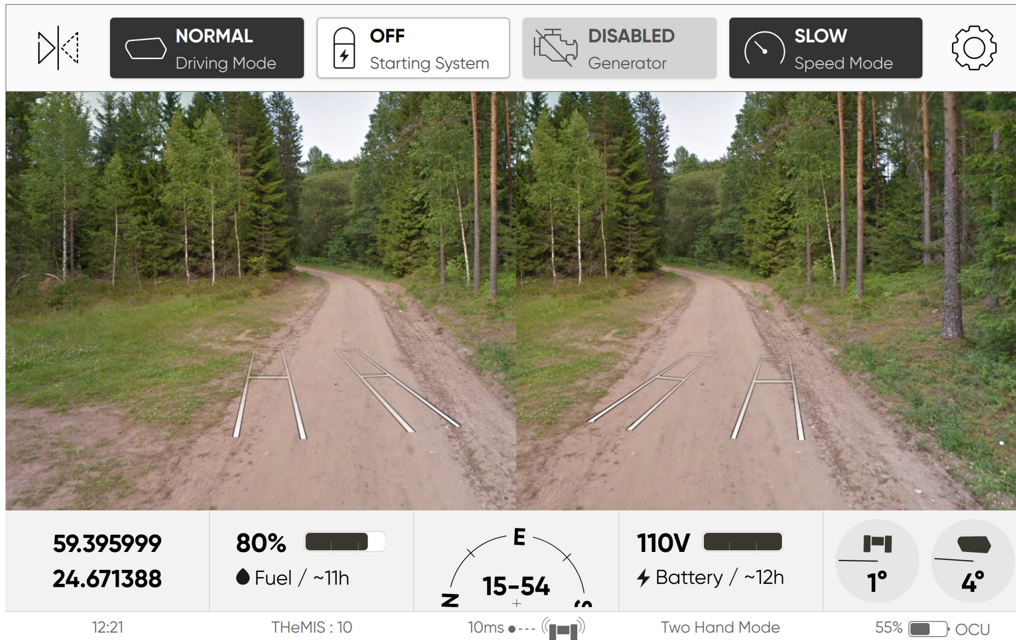


Figure 11. The main view with vehicle mode buttons shown

Figure 12. The main view top bar with lights and camera items

5. The Testing Setup

Figure 13 gives an overview of the testing setup used in this thesis.

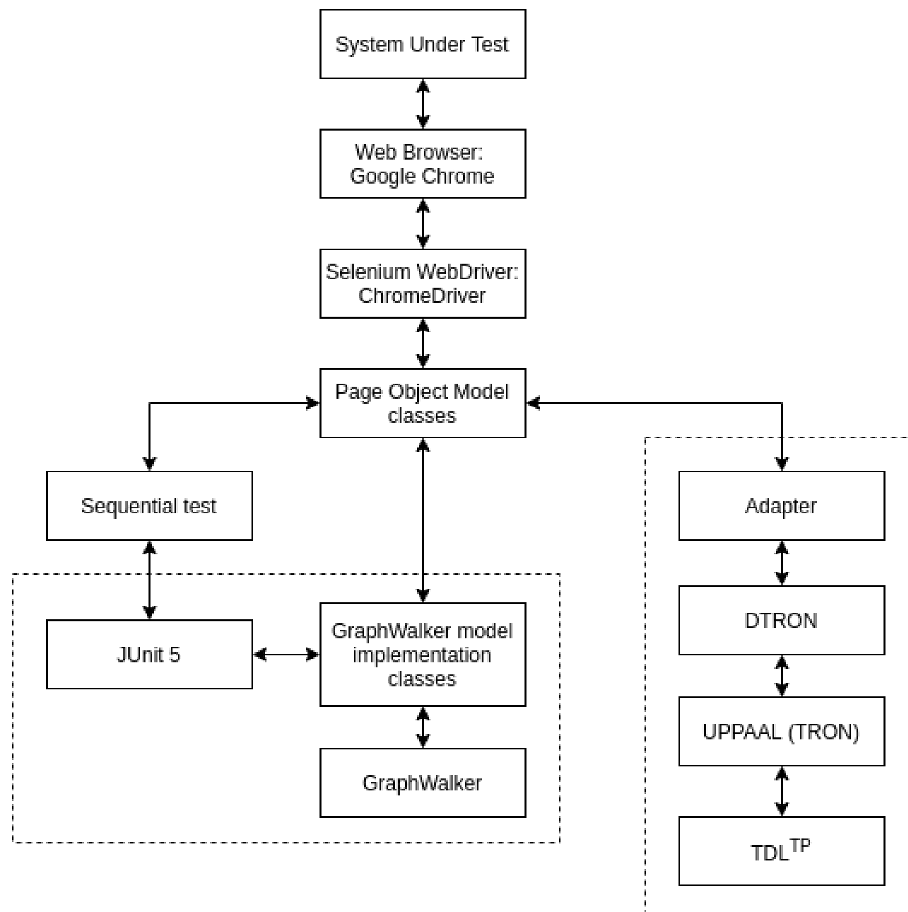


Figure 13. Overview of the testing setup

As the System Under Test is based on web technologies, it runs in a web browser, specifically Google Chrome / Chromium. Programmatically interacting with the browser running the SUT is achieved using Selenium WebDriver. The WebDriver variant corresponding to Google Chrome / Chromium is called ChromeDriver.

Interacting with the WebDriver is almost completely delegated to Page Object Model classes. These classes correspond to specific views and components of the SUT, allowing to obtain information and perform actions on them while encapsulating internal details such as locators. Instantiating the WebDriver, navigating it to the URL where the SUT is available, and quitting it once tests have finished running are steps that are performed

outside of POM classes — performing these is the responsibility of the higher-level test solutions.

There are three higher-level test solutions implemented in this thesis. Besides two MBT solutions, there is a more traditional test suite which can be considered as "Sequential test". As the term "sequential" implies, it tests the System Under Test in one long run instead of being split into smaller units. Therefore this test solution isn't suitable for actual testing because failure in some part of it halts the whole test and the test report only shows a pass/fail verdict for the whole test run. Implementing a full-blown traditional test suite is out of the scope of this thesis — the "Sequential test" exists solely to ease the development and demonstration of POM classes. This test uses the JUnit 5 library for test suite definition and assertions.

The other two test solutions apply the MBT approach with online test generation. One of them is based on GraphWalker. This model-based testing tool generates Java interfaces that correspond to test models, whereas each method corresponds to each named edge and vertex in the model. The actual test code that interacts with the SUT in the form of assertions and actions is placed in GraphWalker model implementation classes. Interacting with the SUT is done via POM classes. The assertions are implemented using JUnit 5.

Finally, there's the solution utilizing the combination of mainly UPPAAL and DTRON. Here, interacting with the SUT (through POM classes) is implemented in an adapter class which exchanges messages with DTRON. The adapter listens to messages that are about getting specific information from the SUT or performing a specific action on the SUT. It then uses the POM classes to perform the necessary work and sends information or confirmation of action back to DTRON. The latter is connected to UPPAAL (specifically its extension TRON) which executes the UPPAAL model that is passed to DTRON on the command line. The UPPAAL model specifies how the SUT should behave in response to input. In this solution, there are no explicit assertions. Instead, verifying that the SUT behaves as expected is done via conformance testing between the expected behavior (according to the test model) and the actual behavior of the SUT.

More detailed description of the main components involved in MBT follows.

5.1 Page Object Model classes

To ease the development and maintenance of tests, the Page Object Model (POM) design pattern is followed. The benefit of this approach is even more substantial due to there being the previously mentioned three higher-level test solutions. All three of them interact with

the SUT through the usage of the same set of POM classes.

There are POM classes corresponding to entire views of the SUT. What this kind of view POM classes have in common is that these typically have a method that checks whether the view is currently open in the SUT. These classes also encapsulate the specific nature of the view. For example, there is a POM class called *VehicleSelectionView* which has a method for getting the currently active vehicle name and another method for selecting a vehicle by its name.

Not all tested SUT views have a corresponding view POM class. Instead, there are some general components in the SUT UI for which Page Component Object Model classes were created (for simplicity, these can also be considered to be POM classes). For instance, most views in the SUT UI have a header bar at the top which contains the view title, a left-hand-side button to go back to the previous view (if there is a view to go back to), and optionally a right-hand-side button for miscellaneous actions (such as confirming). Therefore a POM class named *HeaderBar* was created which has getters for the title and subtitle, methods to click the "go back" button and to click the miscellaneous action right side button. The code for this rather simple POM class is provided in Appendix 2. The title from *HeaderBar* is frequently used to check whether a specific view is open, both in and out of view POM classes.

One essential part of the POM classes are various kinds of selectors for locating the relevant HTML elements. These selectors have a type of *org.openqa.selenium.By* and are stored as class fields at the beginning of the class body. This makes it easier to update the POM in case of changes in the SUT. Then there are methods which make use of the selectors to enable the querying of various information from the view or component and to perform actions on them.

Most POM classes take a *WebDriver* instance as one of their constructor arguments. In this thesis, there is one exception — the constructor of POM component class *Vertical-RangeInput* takes a *WebElement* instance instead. But that constructor is package private and is meant to be used by another POM class *VerticalRangeInputGroup* which is constructed using a *WebDriver* instance. One POM view class that in turn constructs three *VerticalRangeInputGroup* objects — one each for the inputting of year, month and day — is *ChangeDateView*. Figure 14 shows a screenshot of the "Change Date" view in the SUT that is annotated with regards to POM classes. To avoid overcrowding the figure, only one *VerticalRangeInputGroup* and one *VerticalRangeInput* instance is annotated. It can be seen that the *ChangeDateView* POM class also includes an instance of *HeaderBar*. This instance is used in the view POM class' *isOpen* method which uses the header bar title for

its check. The *StatusBar* is global and independent from the view.

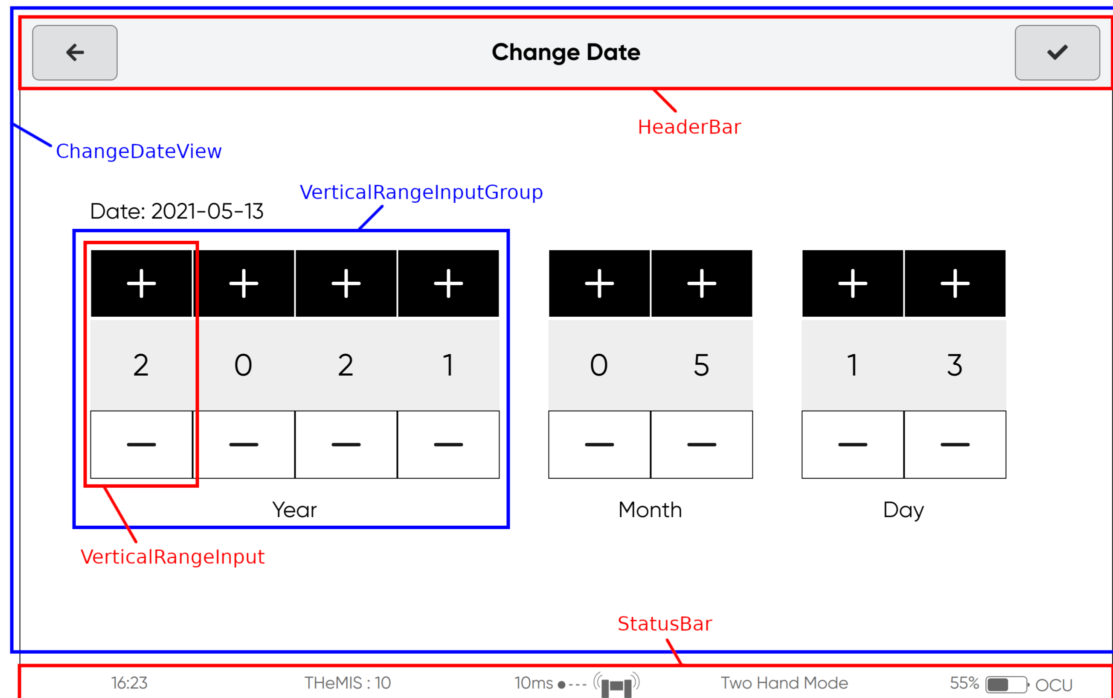


Figure 14. POM classes related to the "Change Date" view

Several POM classes corresponding to components take an additional argument identifying the concrete component instance in the SUT. For instance, there's a POM class named *SettingsButton* representing a settings button which has a option name and may optionally display a value. If we want to know the value of the speed mode setting, we can first construct a *SettingsButton* with a *WebDriver* instance and the string "Speed Mode", then we can use the method *getValue()* on the constructed instance.

The POM classes implemented in this thesis use lazy evaluation and are uncached. HTML elements are located each time methods are called on the classes. No locating is done upon construction. This results in two main implications. First, it is possible to construct the POM classes regardless of whether the appropriate HTML elements are present in the SUT at construction time. And secondly, calls to getters (and naturally to action methods) result in errors if the respective HTML elements aren't available. This means that when moving between views, it can be necessary to explicitly store the return value of getters into variables.

5.2 GraphWalker

GraphWalker Studio was used to model the System Under Test so that it could be tested in an MBT fashion using the GraphWalker testing tool. The SUT is modeled as multiple

models each corresponding to a tested view, or (if the view is bigger or more complex) parts of it. This kind of splitting was performed in order to ease the maintenance and understanding of the models. All of the models were put into a single GraphWalker-specific .json file. The other options would have been to either put each model or a group of them into separate files. The author of this thesis decided in favor of the single file option since it eases editing of multiple models together (each model is in placed into its own tab in the Studio UI) and this way it is possible to visually "execute" the models in the Studio to make sure that they are correctly traversable in combination with each other.

After a GraphWalker models .json file has been created, the next important step is to have GraphWalker generate Java interfaces that correspond to each model in the .json file. Using the graphwalker-maven-plugin this can be achieved by executing the command *mvn graphwalker:generate-sources* in the project root. This command goes through all the .json files in *src/main/resources/* directory and attempts to parse them as GraphWalker model files. For each model in a valid GraphWalker .json file a corresponding Java interface is created. There is also a second command *mvn graphwalker:generate-test-sources* which behaves in the same way but looks in the *src/test/resources/* directory. The models in this thesis are put in *src/main/resources/gwmodels/* so the former command is used. The generated Java interfaces have a name corresponding to the appropriate model name and the interfaces are annotated with *@Model(file = "path/to/model_file.json")*, an annotation from the *graphwalker-java* package. The interfaces contain *void* methods that take no arguments and are named after each unique non-empty vertex and edge name. The model can also contain vertices and edges that are unnamed. This kind of unnamed vertices can be used where there is no check/assertion to perform. Unnamed edges can similarly be used if there is no action to perform on the SUT — for example, this can be used to jump from a specific assertion/state vertex to a more general one that is the source of other transitions.

Actual testing is realized in classes that extend GraphWalker's *ExecutionContext* class and implement the previously mentioned model interfaces. Therefore, for each model to be tested, there is a corresponding model implementation class. These kind of classes make use of POM classes to interact with the SUT and have to be provided with a *WebDriver* instance. Due to the way model implementation classes are instantiated, it is more straightforward to have no explicit constructor for them (or at least have a no-argument constructor). It is necessary to have a way to share a single *WebDriver* instance between the model implementation classes, because all of them should interact with the same instance of the SUT. One way would have been to create an interface that allows setting a *WebDriver* instance and let model implementation classes implement this interface. That would have been a bit cumbersome. Instead, the author of this thesis decided to create a class called

WebDriverSingleton used by all of the model implementation classes which ensures that a single instance of *WebDriver* (more specifically its subclass *ChromeDriver*) is in use.

The model implementation classes obtain a *WebDriver* instance from *WebDriverSingleton* and use it when constructing the POM class instances, most of which are stored as fields of the class so that they can be reused. Most of the methods corresponding to vertices contain one or more assertions to ensure that the SUT is in a state that is compliant with the state implied by the vertex. Vertices that have shared state and which lead the execution out of the model typically have an empty/stub method in the corresponding model implementation class — the assertions for these vertices are performed in some other model implementation class method that corresponds to a vertex linked via shared state.

Some model implementation class methods perform waiting in addition to actions or assertions. For example, this is necessary for a button in the SUT main view called "Starting System". This button can be used to toggle the vehicle system on and off. As it takes some time for the system to turn itself on or off, the UI accounts for that — after toggling the system by choosing an appropriate option in a dialog box, the button is disabled for a few seconds until the system enters the desired state. For this reason the relevant model implementation class called *MainViewVehicleModesTest* waits until the starting system button is clickable after toggling it. This waiting is also done before clicking the button since testing is executed at a rapid pace and initially (for a short time) the button might not be clickable due to the way the SUT is implemented. Technically, the wait is implemented using Selenium's *WebDriverWait* so that the test execution can proceed as soon as the button is clickable.

There are different ways how coverage of different options in the SUT could be achieved. For instance, in the model *MainViewVehicleModes* (depicted in Figure 15), there are vertices and edges for each of the clickable vehicle mode buttons and each choosable option. This makes it possible to handle conditions for specific buttons being clickable at the model level by using guards, and also gives greater control over test coverage via path generators and stop conditions.

In contrast, clicking through the modes in *SettingsViewModes* (depicted in Figure 16) was implemented through the random selection of a mode and a corresponding option in the model implementation class *SettingsViewModesTest*. The model corresponds to the modes tab of the settings view (Figure 17), which has six modes with two or three options. Turning on "EMG Mode" opens a confirm dialog which is accounted for by the *v_EmgConfirmModal* vertex led to by an appropriately guarded edge

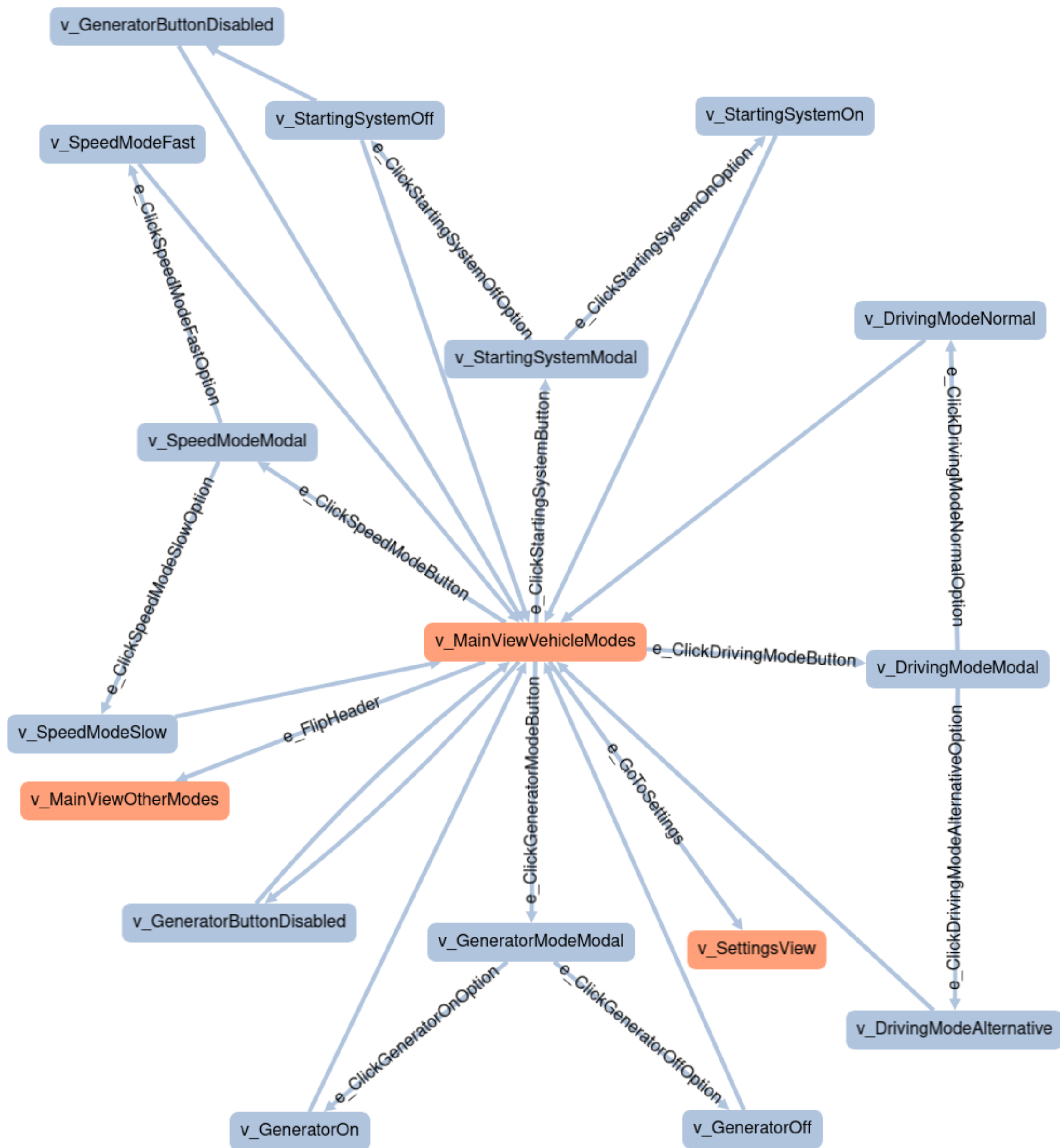


Figure 15. The MainViewVehicleModes GraphWalker model

e_ChooseRandomValue. Covering all of the modes through vertices and edges in a model would make the model quite bloated. On the other hand, due to the way *SettingsView-ModesTest* is implemented, full edge coverage is a bit more difficult to achieve. The "Test data and GraphWalker"[1] wiki page gives an approach utilizing a model variable and edge guards, which can help achieve full coverage of the model even in the case of model implementation-aided data selection. In this approach, there is an edge and vertex which test some feature with different values, with the model implementation class taking the next one from a list each time. The edge's guard initially allows traversing the edge; as soon as there is no more data to use, the guard no longer allows traversing this edge. There

[1] `https://github.com/GraphWalker/graphwalker-project/wiki/Test-data-and-GraphWalker`, accessed May 1st, 2021

is another edge with an opposite guard that makes it traversable as soon as there is no more test data. In this case, all of the elements can be covered only after all of the test data has been used, thus an 100% edge or vertex coverage stop condition works as intended.

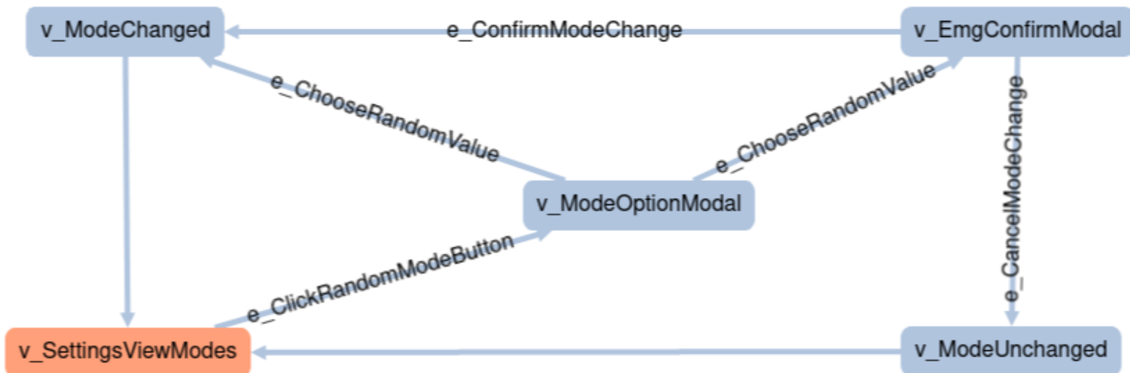


Figure 16. The SettingsViewModes GraphWalker model

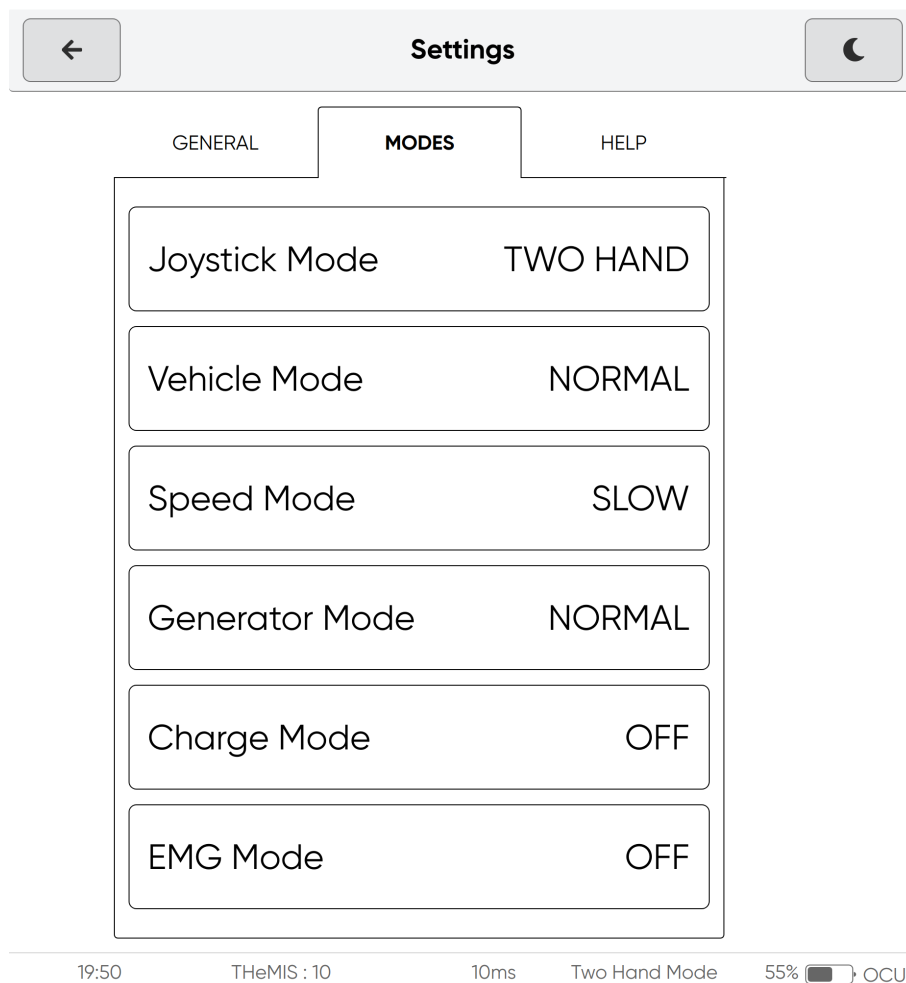


Figure 17. The modes tab of the settings view in the SUT

Model-based testing typically involves more than just a model. It also involves some kind of a test selection criteria or a more formal test case specification. In the context of GraphWalker, path generators and stop conditions can be considered to be playing this role.

A single model could be used to test the system in different ways. Besides the previously mentioned edge and vertex coverage stop conditions, straightforward approaches supported by GraphWalker out-of-the-box include traversing the test model randomly until a specific time duration has passed and traversing the shortest path to a certain edge or vertex. It is also possible to combine stop conditions with logical *and* and *or*; path generators can be combined in a daisy-chain fashion. In GraphWalker, path generators and stop conditions are applied to each model separately.

Given a larger model of the SUT (technically consisting of several GraphWalker models), it can be desirable to test specific parts of it in isolation. In the context of this thesis, some example parts are initial vehicle configuration view, the vehicle mode buttons in the main view and the modes tab in the settings view. To achieve this, it is necessary to instruct GraphWalker to first enter the respective part via the shortest path and to then stay within its bounds. Making GraphWalker enter the part is possible by programmatically applying relevant shortest path generator *a_star* together with stop condition *reached_vertex* with an appropriate vertex to each preceding model to the one that contains the part. This requires some manual work to determine which vertices lead closer to the necessary model.

Ensuring that test execution remains in the intended part of the SUT model needed some extra development work. Essentially, it is necessary to avoid edges which lead out of the model part to be tested. To achieve this, a custom path generator can be created. One initial idea was to create a path generator which avoids traversing edges that lead to vertices with shared state. This seemed to make sense as the intention was to initially test a single model without leaving it and vertices with shared state are through which execution can leave a model. Unfortunately, this solution didn't work out because the entry point vertex in several models had shared state and was traversed during within-part testing. These entry point vertices in practice didn't allow exiting a model but checking that could have required notable effort. The next attempt was to create a path generator that includes or excludes specific elements (edges and vertices). Seeing that test execution could traverse quite a few different elements and that there are fewer edges that should not be taken, it was decided that a blacklist of elements would make more sense than a whitelist. Thus, *ElementExcludingRandomPath* was created. It works like the built in *RandomPath* path generator with the addition of it excluding named elements passed to its constructor. However, due to technical limitations, it does not explicitly extend the built in class, instead it is based on code from *RandomPath*.

Putting the pieces together, testing a specific part of the SUT involves setting *a_star(reached_vertex(. . .))* with the appropriate vertex as the path generator and stop condition for the models leading up to the model of the part, and using *ElementExcludingRandomPath*

on the destination model with edge names that make execution exit the model. Like *RandomPath*, *ElementExcludingRandomPath* also takes a stop condition. But when using stop conditions that enforce element coverage, it has to be taken into account that a high coverage requirement might not be satisfiable due to excluded elements. Time duration stop condition naturally does not have this caveat.

Using the mentioned pieces, test methods are created which test different aspects of the SUT. Some of them apply to parts of the SUT model and there is also a test that applies to the entirety of it for a specific time duration. These methods are annotated with JUnit 5's @*Test*. This means that it is possible to start the test suite and get a pass/fail feedback after each executed test. Some care has to be taken to ensure that test failure indeed results in a fail status. There is a way to let GraphWalker itself throw an error in case of test failure, but in that case there won't be a test summary output. Therefore, to get the test summary even in case of failure and still have a correct test verdict, it is necessary to manually throw an exception if errors occurred during testing.

There are some limitations to what can be tested with GraphWalker. With regards to the SUT of this thesis, one of the limitations can be seen when considering faults that can be reported by the UI. One way to test faults is in a controlled environment where it is known when a fault is going to happen or be produced. This can be modelled and tested with GraphWalker. But in a realistic environment, faults from the connected vehicle can occur at random times and these can interrupt the testing process. Because there is a single active vertex or edge at a time in the case of GraphWalker, covering this and other similar situations is not feasible.

5.3 UPPAAL + DTRON

The second model-based testing solution of this thesis makes use of primarily UPPAAL and DTRON. UPPAAL is used to model the expected behavior of the SUT. In the initial solution, it was also used to model the tester. Later on, initial work was performed towards adopting TDL^{TP} in order to ease the creation of the tester part and to allow specifying test cases. DTRON is used to connect the actual SUT into the testing process via a custom adapter and it also initiates the test execution.

Communication between UPPAAL models and the adapter is achieved through the use of channels and associated variables. This thesis utilizes channels for three different purposes — initializing the SUT, getting values from the SUT and performing actions on it. At the UPPAAL side, there are two channels for all three purposes, one for sending a request (its name has a *i_* prefix, which stands for *input* to SUT) and another for receiving a value or

confirmation (its name has a *o_* prefix, which stands for *output* of SUT). The variables associated with a specific channel are prefixed with the channel's name.

Specifically this thesis uses the following channels together with their associated variables:

i_init
: Request the SUT to be initialized. Technically it means that the browser is refreshed by navigating it to the URL where the SUT is available. No associated variables.

o_init
: Received once the SUT has been initialized in response to *i_init*, i.e. the browser has finished loading the web app after the refresh. No associated variables.

i_getval
: Request a value from the SUT. The value to obtain is determined by the variable *i_getval_key* and in certain cases also the variable *i_getval_arg* (arg stands for argument).

o_getval
: Received in response to *i_getval* together with the variable *o_getval_value* that contains the requested value.

i_act
: Request an action to be performed on the SUT. The action is specified with the help of the variable *i_act_key* and in some cases also the variable *i_act_arg* (arg stands for argument).

o_act
: Received in response to *i_act* after the requested action has been performed. This channel has no variables associated with it. The channel is used so that the model can be synchronized with the pace of actions in the SUT.

Although UPPAAL supports integers, strings and other types, DTRON supports sending and receiving only integer variables. Therefore it is necessary to use integer constants and to have a two-way mapping between these and strings. For this, a Java enum called *UiConstant* was created on the adapter side. The items of *UiConstant* are associated with an integer named *id* and a string *label*, the latter of which is derived from the item name if it is not explicitly provided. The label is explicitly provided for those items which correspond to strings/texts in the SUT UI. Other items that correspond to e.g. *getval* or *act* keys don't need this kind of explicit label so it is not provided. The *UiConstant* enum

contains static methods that can be used to obtain a *UiConstant* item based on an id, label, or boolean. For the latter, there are the special items *FALSE* and *TRUE*, which correspond to the *id*s 0 and 1, respectively.

It is necessary to have integer constants on the UPPAAL side that correspond to the same *UiConstant* items and their *id* values. To reduce manual work, an utility called *UiConstantExporter* was created. This is a short and simple class that iterates over all the items of *UiConstant* and outputs a *const int* definition that assigns to each constant derived from the enum item names their respective *id* value. The output of this utility can be directly copy-pasted into UPPAAL declarations.

The component that fits the actual SUT into this testing solution is called the adapter. A shortened version of its code is presented in Appendix 3. The adapter is implemented as a Java class that starts by connecting to DTRON via its Java bindings, creating an instance of *ChromeDriver* and navigating the latter to the URL where the SUT is available. It then sets up DTRON listeners for the channels *init*, *getval* and *act* (DTRON automatically prepends the channel names with *i_* or *o_* as appropriate). Upon receiving a message, the listeners for *getval* and *act* convert the respective associated *key* and possibly *arg* from integers to the appropriate *UiConstant* items using their *id* values. The listeners then use the *key* and possibly *arg* (both of type *UiConstant*) to determine which value should be obtained from the SUT or which action should be performed on it.

The requested interaction with the SUT is performed using the Page Object Model classes. In the case of a *getval* request concerning a textual value, the result from the used POM class is converted to the respective *UiConstant* item. Boolean results are represented using the special *UiConstant* items *FALSE* and *TRUE*. Once the value has been obtained, the *getval* response is sent which contains the *id* of the obtained *UiConstant* as the channel-associated variable *value*. In the case of an *act* request the action is performed using the appropriate POM class, and after that an empty *act* response is sent as the confirmation that the action has been fulfilled.

In this testing solution, there are no explicit assertions in the code. Instead, what is being tested is the conformance between the actual SUT and the model of it. The tester part of the model requests for values via *getval* and actions via *act*. Both the model of the SUT and, via the adapter, the actual SUT receive these requests. The model of the SUT describes the expected behavior of the actual SUT (i.e. with which messages and which variable values the SUT should respond to requests). If the actual SUT behaves differently from what is expected based on the model of the SUT, the test execution is halted with an error message in DTRON console output that shows which values were different.

Executing the UPPAAL + DTRON testing solution begins with running the adapter. With that running, the next step is to start DTRON on the command line. The mandatory arguments for DTRON are the model file in XTA format, TRON time unit in microseconds and the TRON timeout (how long to execute the test) in the latter time unit[2]. For some reason, it is not possible to directly use model files produced by UPPAAL version 4.1.24 with DTRON version 4.18. Attempting to do so results in the following error:

```
Exception in thread "main" java.io.IOException: ParseError at [row,col
    ]:[1,3]
Message: The markup declarations contained or pointed to by the
    document type declaration must be well-formed.
```

This error can be fixed by opening the model .xml file in a text editor and removing its second line:

```
<!DOCTYPE nta PUBLIC '-//Uppaal Team//DTD Flat System 1.1//EN'
'http://www.it.uu.se/research/group/darts/uppaal/flat-1\_2.dtd'>
```

After the described problem has been dealt with, it is possible to use the model with DTRON. For example, the following command executes the exampleModel.xml model with a time unit of $1000\ \mu s = 1\ ms$ for the duration $5000 \cdot 1\ ms = 5000\ ms = 5\ s$ in eager delay mode:

```
java -jar dtron-4.18.jar -f exampleModel.xml -u 1000 -o 5000 -P eager
```

The eager mode is recommended for beginners to normally use[3]. In this mode, transitions in model are taken as soon as possible.

Once started, DTRON will perform online test generation until the specified amount of time elapses or an error occurs. In either case, the adapter remains running. The UPPAAL models of this thesis begin by requesting the SUT to be initialized, so it is not necessary to restart the adapter before starting a new test run.

UPPAAL works with a network of timed automata, whereas multiple processes are executed in parallel. This makes it possible and easier to handle more complex parts of the SUT. An example use case from the SUT used in this thesis, again, is the faults system — with UPPAAL it is possible to model the fact that vehicle faults can happen and be shown in the SUT practically at any moment.

[2]https://cs.ttu.ee/dtron/usage.html, accessed May 4th, 2021
[3]https://cs.ttu.ee/dtron/index.html, accessed May 4th, 2021

6. The Process

This chapter begins with an overview of how the thesis project was set up. The chapter then proceeds by describing the process of testing the SUT with the main focus on the usage of GraphWalker and UPPAAL + DTRON.

6.1 Initial setup

The initial setup of the project for this thesis began with the creation of a Java project that uses Maven[1] as its build automation tool. Maven was a natural choice because this is what GraphWalker and DTRON support out-of-the-box. The project's Maven dependencies and related configurations are stored in the *pom.xml* file. IntelliJ IDEA[2] was used as the IDE (Integrated Development Environment) in this thesis.

The following steps were about downloading programs, setting up dependencies and the environment. Selenium was chosen as the tool for interacting with the web technology-based SUT. The main step of setting Selenium up is adding it to the list of Maven dependencies in the *pom.xml* file (its entry is shown in Figure 18).

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>3.141.59</version>
</dependency>
```

Figure 18. Selenium dependency pom.xml entry

The SUT used in this thesis is developed for usage with Google Chrome / Chromium. Therefore it is also necessary to have that browser installed and to have its corresponding WebDriver called ChromeDriver available on the system.

For writing test cases and assertions in the sequential test and GraphWalker test solutions, the JUnit 5 library is used. Including the library into the project merely requires adding the appropriate entry to *pom.xml* dependency list and updating Maven dependencies in the IDE.

[1] `https://maven.apache.org/`, accessed May 6th, 2021
[2] `https://www.jetbrains.com/idea/`, accessed May 6th, 2021

Moving on to the first MBT solution used in this thesis, setting up GraphWalker consists of two main activities — downloading GraphWalker Studio[3] which can be used to create and edit models, and adding several dependencies to *pom.xml* so that generating model interfaces and actually running tests is possible. GraphWalker is split into several Maven packages. The ones that are used in this thesis are *graphwalker-core*, *graphwalker-java* and *graphwalker-maven-plugin*

Setting up the second MBT solution that is primarily based on UPPAAL and DTRON requires more work. First, it is necessary to download UPPAAL[4], UPPAAL TRON[5], DTRON[6] and TDL^{TP} [7]. The obtained files should be extracted as appropriate. For DTRON to work, it is necessary to set the value of the environment variable *TRON_HOME* to the path of the directory which contains the UPPAAL TRON archive unpacked contents. It is also necessary to add an entry for DTRON to the project dependencies in *pom.xml*.

Finally it is necessary to have access to the SUT. It can be achieved primarily in two ways. If applicable, one way would be to build the SUT, store it locally and access it via a *file://* URL. Another option would be to run the SUT in a local or remote server and access it via an *http://* URL.

6.2 Initial testing

After the necessary tools and libraries have been set up, it is possible to start testing. The first thing to do is to figure out what should be the first scenario to test. It should not be a too complex scenario. The scenario could be documented by taking screenshots of the SUT UI and annotating these with rectangles that show where to click and how to verify that the UI is in the correct state.

In this thesis, the initial test scenario (called scenario 0) was about choosing a vehicle, confirming its configuration and changing speed mode in the main view. Appendix 4 describes the initial scenario in greater detail.

Having created an initial scenario, the next step is to create a basic traditional test following it. The goal of this test is to just go through the verification and action steps of the scenario. At this stage, the test need not use POM classes nor apply any kind of MBT. There may be

[3] `https://graphwalker.github.io/#download`, accessed May 6th, 2021

[4] `https://uppaal.org/downloads/`, accessed May 6th, 2021

[5] `https://people.cs.aau.dk/~marius/tron/download.html`, accessed May 6th, 2021

[6] `https://cs.ttu.ee/dtron/download.html`, accessed May 6th, 2021

[7] `https://github.com/tanelprikk/ee.taltech.cs.mbt.tdl.interpreter`, accessed May 6th, 2021

a single test method that just goes through all the steps. One can use the help of Chrome Devtools[8] to come up with the necessary selectors. Work on the test until it manages to test the scenario.

Once the initial form of the test has been created and runs successfully, it is time to apply the POM design pattern. There can be POM classes for both components and views. Looking at the scenario and the SUT, one should consider which way of creating POM classes makes more sense, taking into account views' uniqueness and component reusability. The process of adopting the POM design pattern could be done in a step-by-step fashion — take a part of the initial basic test, create a POM class for it, move code into it, generalizing it as appropriate, and use the POM class in the test; repeat these steps until the whole test has been refactored to interact with the SUT through the POM classes. One part that should not be moved into the POM classes are assertions — these should stay in the test class. However, if desired, one kind of verification may be done in the POM classes, namely checking upon construction whether the appropriate component or view is available/active and if not, throwing an exception[9]. This kind of verification was not used in this thesis.

The POM classes that were created for the initial test scenario of this thesis are component POM classes *HeaderBar*, *OptionModal*, *VehicleModeButton* and the view POM class *VehicleSelectionView*. At this point, it was not necessary to create view POM classes for the vehicle configuration view or the main view — the component POM classes were enough in this case.

6.3 Testing with GraphWalker

The goal now is to test the initial scenario with GraphWalker using the same POM classes created previously. The first step towards the goal is to create a model in GraphWalker Studio, the latter of which was downloaded during the initial setup. After an empty model has been created in GraphWalker Studio, vertices and edges should be created that correspond to assertions and actions, respectively. These vertices and edges should be named descriptively. The naming convention of GraphWalker states that vertex names should start with *v_* and edge names should start with *e_*. As the initial test scenario was not too complex, it was sufficient to use a single model in this case.

If the model contains cycles, GraphWalker can traverse it for longer periods (depending on the path generator and stop condition used). Therefore it can make sense to add some

[8] `https://developer.chrome.com/docs/devtools/`, accessed May 24th, 2021
[9] `https://www.selenium.dev/documentation/en/guidelines_and_recommendations/page_object_models/`, Accessed May 18th, 2021

cycles to the model. For example in this thesis, cycles were formed in two places of the initial scenario model. In the SUT, it is possible to go from initial configuration view back to vehicle selection. Therefore an appropriate edge was added from the configuration view vertex to the initial vertex. And secondly, the initial test toggled speed mode from slow to fast; in the GraphWalker test it was made so that the speed mode is also toggled from fast to slow, forming another cycle. Figure 19 depicts the initial scenario GraphWalker model.

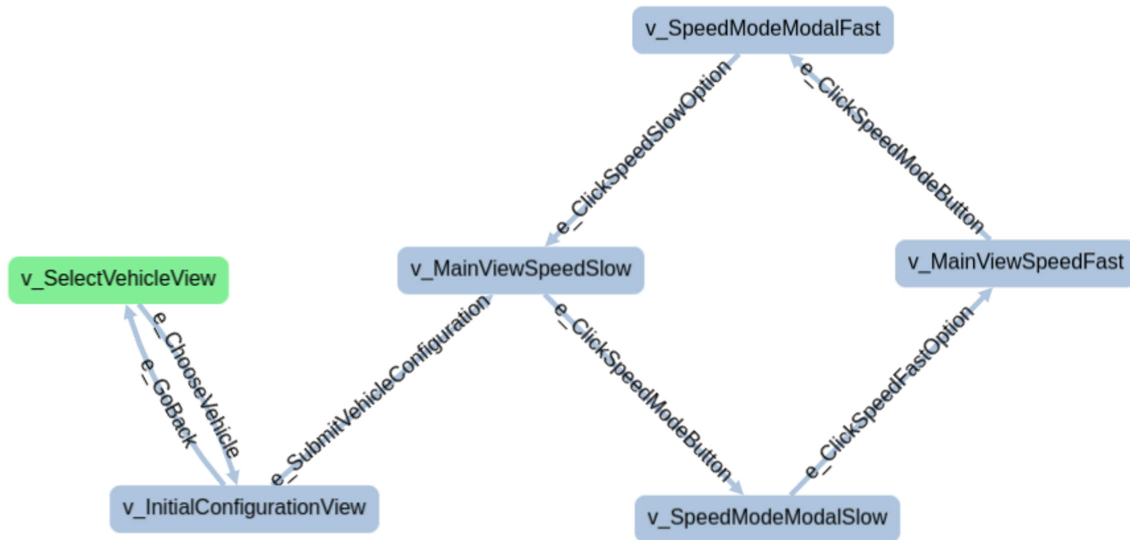


Figure 19. The initial scenario GraphWalker model

Once the model has been created, it should be saved and stored in a subdirectory in either the project's *src/main/resources* or *src/test/resources*. Putting it into a subdirectory ensures that the generated Java interfaces are usable within Java packages (otherwise the interfaces are in the default package and cannot be accessed from anywhere else). The Java interface corresponding to the model should now be generated by executing one of these commands provided by the GraphWalker maven plugin: *mvn graphwalker:generate-sources* (if the model is located under *src/main*) or *mvn graphwalker:generate-test-sources* (if the model is located under *src/test*). Executing the appropriate command will create one Java interface per each model in each of the GraphWalker model .json file. The model interfaces will be put in *target/graphwalker/* under the equally-named subdirectory as the source model. In the current case there will be a single Java interface. The interface contains *void* no-parameter methods that are named after each unique non-empty vertex or edge name.

Now it is possible to create the model implementation, i.e. a class that implements the previously generated Java interface. The implementation performs assertions in methods that correspond to vertices (whose name by convention starts with *v_*) to verify that the SUT is in a state that complies with the vertex in question. Actions on the SUT are taken in methods that correspond to edges (whose name by convention starts with *e_*).

The model implementation class can use the same POM classes that were created and used for the initial test. The class should first set up a ChromeDriver instance (which is a subclass of WebDriver), store it in a field of the class, and navigate it to the URL where the SUT is available. More commonly used POM classes could be constructed in the same setup method and be also stored in a field of the class. This setup method can be annotated with JUnit 5's *@BeforeEach* annotation. Further interaction with the SUT will be performed by the POM classes that commonly take a WebDriver instance as a constructor argument. The methods corresponding vertices will use the POM classes' getter methods to obtain values from the SUT and use JUnit 5 assertions to compare them against expected values. Analogously, the methods corresponding to edges will use the POM classes' action methods to interact with the SUT.

To actually execute[10] the GraphWalker test, a test method has to be created. In this case where there is only a single model, the method can be added to the model implementation class itself. For it to be easily executable the method can be annotated with JUnit 5's *@Test* annotation. The execution method should first construct an executor. In the case of a single model, this can be done using GraphWalker's *TestBuilder* class. A part of setting up the *TestBuilder* instance is specifying the path generator and stop condition. Having constructed the executor, the next step is to execute it and obtain a result. If the result had some errors, then these can be output. It is also possible to output some information and statistics of the test run. To ensure that this kind of information is available for also failed test runs, the executor's *execute* method should be called with its *ignoreErrors* parameter being valued as *false*. Because the errors are being ignored, an exception must be explicitly thrown so that the test run is correctly marked as failing if there are some assertion or other errors.

After the test method has been created, it can be run. One possibility is to use the test execution capability of IntelliJ IDEA for that. Running the test method will perform online model-based testing. Naturally, if running the test exposes some issues with the model implementation, these should be resolved and the test should be run again, repeating the process until the issues have been resolved.

6.4 Testing with UPPAAL + DTRON

Work towards implementing this testing solution can start with modelling the SUT and tester in the UPPAAL application. For simplicity, it is a good idea to begin with the same simple initial scenario used for initial traditional and initial GraphWalker testing described

[10] `https://github.com/GraphWalker/graphwalker-project/wiki/Test-execution`, accessed May 18th, 2021

above.

Some of the decisions that should be made early on, are about which UPPAAL channels and associated variables to use. The initial UPPAAL + DTRON testing solution used two of the three kinds of channels mentioned in the previous chapter — one pair for getting values from the SUT (*i_getval* and *o_getval*) and another pair for performing actions on the SUT (*i_act* and *o_act*). In the case of SUTs with more kinds of different actions, it can make sense to use additional channels.

To be able to use the chosen channels for obtaining values of different kinds and performing different actions on the SUT, channel-associated variables can be used. These kinds of variables are associated with a specific input or output channel and have the channel named prefixed to them. This thesis uses *key* and *arg* variables for both the *i_getval* and *i_act* channels. It would have also been possible to manage with a single variable for each. However, using two variables potentially reduces the number of total values necessary, simplifies the adapter and may also ease the understanding of the model. For certain other complex SUTs, it can make sense to use more than two variables for a channel.

Because DTRON supports sending and receiving only integer values, it makes sense to define integer constants for each value that is used with the channel-associated variables, with the possible exception of values that are directly integers (e.g. counts). The usage of these constants helps to greatly improve the understanding of the models and the maintenance of them. The values of these constants might be universally unique or be unique with the context of its usage (e.g. the number 3 might mean one thing for *i_getval_key* but another thing for *i_act_key*). In this thesis, at first the constants were unique within the context of usage, but it was eventually decided in favor of globally unique constants. For starters, it is enough to manually define the constants as *const int* variables in the UPPAAL's global "Declarations". It might make sense to define these constants in parallel with modelling the system.

Let's continue by modelling the tester. The function of the tester is to send requests to the SUT — requests for action but also for getting values. The latter is necessary even though the tester model will not actually use the values obtained. The values obtained from the actual SUT will be validated against the SUT model that is yet to be created.

It would be logical for the tester model to begin by obtaining some value or a few of them that can be used to verify that the SUT is in the correct initial state. For the SUT used in this thesis, the value to obtain could be the header bar title. The complete obtaining of that value can be done with three edges and locations — the

first edge updates a channel-associated variable so that the correct value is obtained: *i_getval_key=GETVAL_HEADERBAR_TITLE*, the second edge has a Sync of *i_getval!* to send the request, and the third edge has a Sync of *i_getval?* to synchronize with the retrieval of the value. Actions are performed with a similar sequence of edges, with the appropriate variables, and *i_act* and *o_act* synchronizations. The *o_act* channel has no associated variables — its use is synchronizing with the pace of actions in the SUT. Overall, the entire tester model follows the pattern of obtaining one or more values, performing an action, and then repeating this process.

The next step is to model the SUT. While it is possible to model the SUT in a way that resembles the tester model (at first, the initial SUT model created by the thesis author turned out this way), it is better to model the SUT in a less sequential fashion. This makes the model accept input in a more similar way to the real SUT — a user might not perform actions exactly in the order expected. The UPPAAL SUT model for the initial scenario is depicted in Figure 20.

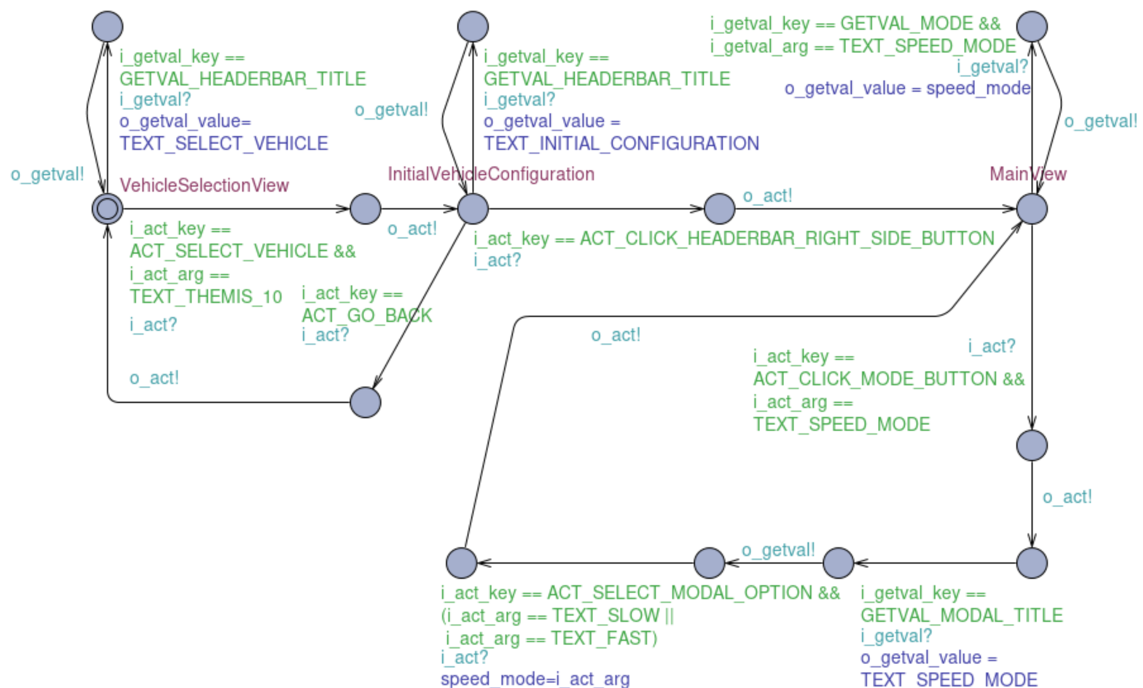


Figure 20. The initial scenario UPPAAL SUT model

The usage of channels in the SUT model is the inversion of the usage in the tester model. For example, whereas in the tester model *i_getval!* and *o_getval?* were used, the SUT model utilizes *i_getval?* and *o_getval!* — i.e. the sequence is *i_getval!* (tester), *i_getval?* (SUT), *o_getval!* (SUT), *o_getval?* (tester), by which the tester model sends a request which is received by the SUT model, then the SUT model sends a response which is received by the tester model.

The main locations of the SUT model correspond to views of the SUT. There are also locations that correspond to modals / dialog boxes being open. In each view (and also modal), there are specific values and actions that can be requested. Ensuring that only valid requests are allowed is through guards on the edges that synchronize with *i_getval?* or *i_arg?*. The implementation of the SUT model of this thesis' initial scenario is somewhat restrictive — for example, it is not possible to request the current speed mode from the respective main view button label while the modal for speed mode is open, even though in reality it is possible. This kind of limitation allows for a simpler SUT model. Overcoming this limitation might be done by using a separate model for modals, or by simply adding more edges (although the latter would worsen the readability of the model considerably).

In the tester model it was necessary to have three edges per a request. In the SUT model, two edges suffice for a response. This is because the first "listening" edge could be supplied with a guard, a synchronization and an update. It is possible to place the update on this edge since it has to be set when the response is sent. An equivalent combining isn't possible in the tester model because an update that sets the necessary variable values has to occur before the request. All of this makes the SUT model more readable than the tester model. Once both of the models have been created, the UPPAAL .xml file should be saved and its <*!DOCTYPE*> should be removed.

What's left to implement is the adapter that utilizes DTRON and fits the real SUT into the testing process. The adapter listens to messages from the same channels (except the channel's *i_* or *o_* prefix is omitted in code and instead is handled by DTRON). The adapter can be a Java class which has a standard Java main method. The adapter should first initialize and connect to DTRON. Then it should create an instance of the WebDriver to be used (in this thesis it is the ChromeDriver).

The adapter should be equipped with the same constants that were defined in the UPPAAL file. Some constants map to strings of the SUT, so it is necessary to have a two-way mapping between each integer constant and its string counterpart. One way to achieve this would be to use two *HashMap*s — one that maps from integer to string and another one that maps vice versa. Instead, in this thesis a Java enum was created that stores an item for each string in the SUT. Each item has an integer ID and the string the item corresponds to. The enum also contains methods that can be used to obtain an enum item from its id or its string. At this point, this enum contained only items that correspond to string values in the SUT; it did not contain other *key* or *arg* values.

The main part of the adapter consists of listeners for each of the channels. These listeners use the POM classes to interact with SUT and fulfill requests of values and action. The

integer values from the channel-associated variables that map to strings are translated into the latter. Similarly the string values from the POM classes are translated into the integer constants before sending them to DTRON.

Once the adapter has been created, it can be tried out with the model. For this, the adapter should be started first. Then it is time to execute DTRON on the command line, providing the model filename alongside with some other options as arguments, as described in the previous chapter.

6.5 Further process and improvements

After having created the POM classes and set up basic testing for the initial test scenario, it is possible to proceed further and also improve different aspects of the testing setups.

One logical way to proceed is to create a new test scenario that extends the initial one. For example, for this thesis the next step was to add the toggling of other main view modes besides the speed mode. Having specified the scenario, review of the POM classes followed together with changes as necessary. There is the option of first implementing the new test scenario in the sequential test, because jumping straight into implementing it in MBT solutions might be a bit more difficult. Then it is necessary to update the MBT solutions to cover the new test scenario.

Moving forward, as larger and larger test scenarios are tested, there are several things that can be improved in the MBT solutions. As the models get bigger, one apparent improvement is to split the models up. In GraphWalker, this can be done using vertices' shared state. Together with splitting, usage of *TestBuilder* needs to be replaced with *TestExecutor* and it would make sense to use a separate class for test execution. Splitting the UPPAAL tester and SUT models can be achieved with the help of internal channels (the thesis author naming; means simply channels that are not used over DTRON).

It can also be desirable to test a part of the SUT and not the entirety of it. With GraphWalker, one way to achieve this is through the usage of different path generators and stop conditions for each of the models, creating custom path generators as necessary. In the UPPAAL + DTRON side of things, the similar (or more flexible) outcome could be achieved with the usage of TDL^{TP}.

With GraphWalker, several test methods were created (utilizing JUnit 5's @*Test*) that test different parts of the SUT and/or in a different way. This test methods could be started as a collection. The result is that each test will have a pass/fail outcome attached to it.

Working with mutliple GraphWalker .json files that contain at least some models with the same name can have some confusing consequences. In this case, running the appropriate GraphWalker Maven plugin command will overwrite the model interfaces with the same name. It might not always be the latest model that remains on top. Therefore it is advisable to prevent this situation by adding a suffix (such as ".skip") to the older model filenames, or delete them, if desired.

For it to be possible to specify test scenarios, the usage of TDL^{TP} can be introduced to the UPPAAL + DTRON testing solution. For this, appropriate edges should be annotated with trap variables and those variables should be declared in the global "Declarations". Those edges should be annotated that correspond to branching or decision points of interest. This kind of annotation means that the respective trap variables are assigned to be *true* as part of the appropriate edges' Update part. Figure 21 depicts a trap-annotated initial scenario UPPAAL SUT model. Two of its edges leading out of InitialConfiguration location that are synchronized with *i_act?* are annotated with traps — the edge leading back to vehicle selection is annotated with TS1 and the edge leading towards main view is annotated with TS2.

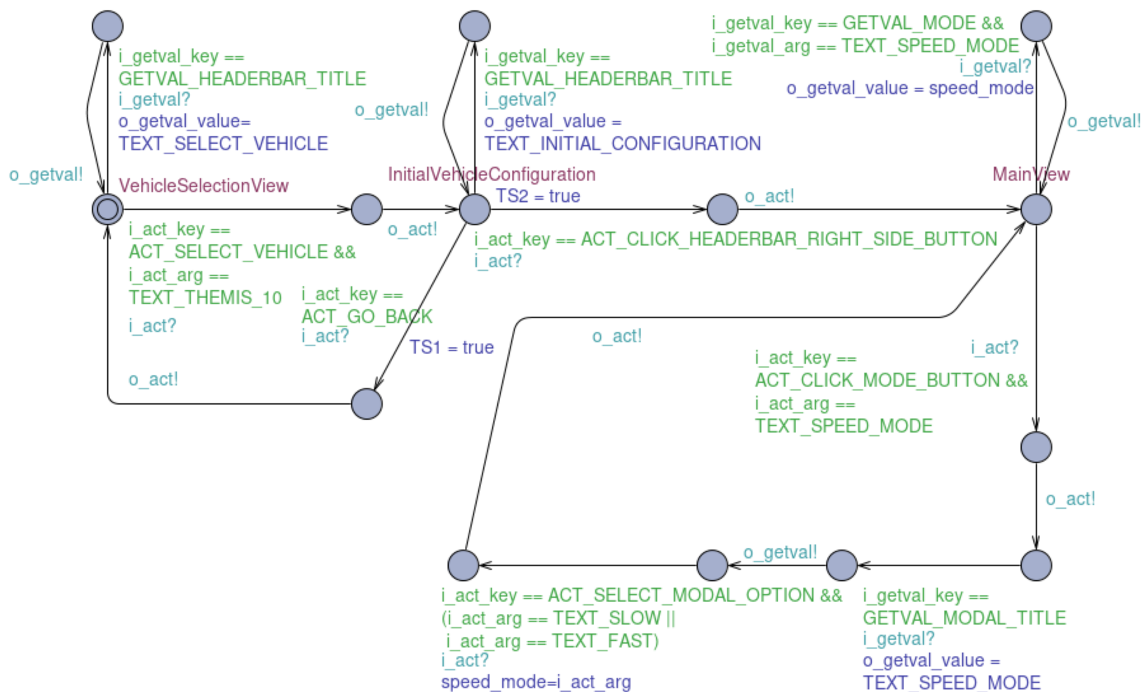


Figure 21. The initial scenario UPPAAL SUT model with trap variables TS1 and TS2

After the edges have been annotated with trap variables, the latter can be used in TDL^{TP} expressions. For example, the expression *-E(TS2) & E(TS1)* would mean that edge leading towards main view is avoided and instead the test will cycle between vehicle selection view and initial configuration. In this fairly simple model, there aren't many useful places to use trap variables. With a model with more decision points that involve actions, traps and

TDL^{TP} can be utilized to specify a smaller part of SUT to be tested in more meaningful ways.

The previous discussion involved the SUT model. It is still necessary to have some kind of a tester model. For this, Reactive Planning Tester (RPT) [14] can be used. Making actual use of RPT is out of the scope of this thesis.

One improvement that was done in the DTRON adapter is that all of the constants were unified into a single enum. Alongside that, it was possible to make the adapter output more informative by displaying the item name and (if available) the label each time an integer constant occurs. One more related improvement is that a simple utility class was created which outputs a *const int* definition with all the constants derived from the enum items.

7. Evaluation

This chapter provides an evaluation of the tools, methodologies used and work done in this thesis. It starts with the MBT tools used in this thesis (GraphWalker and UPPAAL + DTRON), then transitions over to MBT in general and Page Object Model pattern. A discussion on the role of traditional testing follows. Subsequently, feedback from the company is provided. The chapter concludes with a summary of the evaluation.

7.1 Evaluation of GraphWalker

The first obvious thing to mention is that GraphWalker is an open source project released under the MIT license. This means that GraphWalker is free to use for both non-commercial and commercial purposes and that it can be forked and modified as desirable. To get help, there is a discussion group[1] in Google Groups which can be used to ask questions concerning usage and problems, among other topics. Specific issues and bugs can be reported in the project's GitHub issues page[2]. On the other hand, there is no dedicated support team available. The members of the community help in their free time and will.

The fact that in GraphWalker models are complemented with model implementations in code, can help make models more concise and easier to understand. To ensure that the models are readable, it is important (as in programming generally) to choose descriptive names for models, their edges and vertices. For bigger models, it is beneficial to split them up, which is achievable through the use of vertex shared states. From another point of view, the model editor GraphWalker Studio does not show many attributes of vertices and edges on the graph view. This may make understanding some complex models that make heavy use of e.g. guards and actions more cumbersome because to see them, it is necessary to click on the elements. It can be taken as a compromise, since if all of the attributes were shown on the graph area, the overview of the model would get crowded.

Using GraphWalker Studio is mostly straightforward. The user needs to learn how to create vertices and how to connect them with edges. The user should also know the meaning and usage of different vertex and edge properties. However, during the work on this thesis, the

[1] `https://groups.google.com/g/graphwalker`, accessed May 8th, 2021

[2] `https://github.com/GraphWalker/graphwalker-project/issues`, accessed May 8th, 2021

author author stumbled upon an annoying bug. The bug is that under some circumstances each click on an edge or vertex of the model resets the view's pan and zoom. This made using the editor quite cumbersome. Eventually, the author found a way to overcome the issue — pan and zoom the model as desired, click another tab to open a different model and then open the previous tab. Doing so has proven to set the chosen pan and zoom as the ones that are reset to.

It would be useful to be able to reorder the tabs in GraphWalker Studio to keep the order of models logical. This is not currently supported in the tool. To reorder the models the author of this thesis edited the .json file created by the Studio after prettifying it. What's also missing from the Studio is the ability of undoing or redoing changes.

There is a limited way of verifying models in the Studio. This verification is simply through specifying a path generator and a stop condition and observing whether the model is traversed as expected. However, this might not work for all kinds of models. For example, there might be models which make use of model attributes for guards in some of the edges of the model, but those guards get evaluated to *true* when those model attributes are changed via the model implementation. Since this kind of traversal of the model via the Studio does not execute model implementations, some edges and vertices might not be traversable. In some other cases, the Studio could suggest that more of the model is traversable but in reality, the model implementation might restrict the traversability. GraphWalker Studio lacks more sophisticated ways to verify models.

GraphWalker comes with several path generators and stop conditions. Probably some of the more popular path generators are those that perform random walks of the models. The random path generators are usable in conjunction with a variety of stop conditions ensuring vertex or edge coverage, reaching a specific vertex or edge, or executing the test for a specified time duration.

One specific use case which can be achieved with the use of path generators and stop conditions is covering smaller parts of the SUT when given a larger model of it. GraphWalker does not come with built-in utilities for specifically this purpose. As path generators and stop conditions apply to each model individually, specifying a part of the SUT to test can be somewhat cumbersome. One way to achieve it would be apply shortest path generators to each vertex of the preceding models that eventually lead to the model which should be tested. The built-in path generators might not be enough to specify a smaller part of a model to test or to ensure that execution does not leave the intended portion of the model. What would make it possible is the usage of model attributes that are used in the guards of the edges and restrict which edges can be taken. Another solution (which was also used in

this thesis) is to create a custom path generator that e.g. can restrict which edges to take. Either way, these solutions can be fragile; their code can become verbose and be prone to duplication. In the opinion of the thesis author, covering smaller parts of the SUT/model does not come naturally with GraphWalker.

There are limitations to what kinds of SUTs can be tested with GraphWalker. For one, GraphWalker does not support clocks or timed automaton. And another limitation is that in GraphWalker, one model is executed at a time, making it difficult to model more complex SUTs with parallel processes.

After a test has been executed, it would be beneficial to see which vertices and edges were traversed in which order. This is especially important in case of failures. There might be cases where an error occurs only if some certain sequence of actions are taken. For this, it would be beneficial if the trace of each test execution could be stored into a file. This would make it possible to run a whole test suite and later examine how each one of the executions went. The described functionality is not currently available in GraphWalker. While there is a class called *ReplayMachine*, it is currently usable only if a live instance of a GraphWalker test machine is available.

7.2 UPPAAL + DTRON

UPPAAL is closed source and has a restrictive license according to which it is free for non-commercial use in academia only. The pricing for the commercial use license does not seem to be public. There does not seem to be a dedicated support team for UPPAAL. Similarly to GraphWalker, there is a discussion group[3] in Google Groups for UPPAAL. For bug reporting, there is a GitHub project[4] solely for that. DTRON also is closed source. The TDL^{TP} interpreter is open source.

The UPPAAL editor shows models in a more verbose way — most of the location and edge attributes are shown in the editor when displaying the model. Therefore it is usually not necessary to click on elements to see what attributes do locations and edges have. This makes the model displaying more crowded, and it is necessary to accommodate the attribute labels next to the elements.

UPPAAL as a tool is more traditional as it is a GUI-based desktop application. During its usage in this thesis, the tool was mostly stable. Sometimes errors occurred but these weren't fatal and it was possible to get around them. These issues could be caused by the

[3] `https://groups.google.com/g/uppaal`, accessed May 8th, 2021
[4] `https://github.com/UPPAALModelChecker/UPPAAL-Meta`, accessed May 18th, 2021

fact that the thesis author used a development snapshot of UPPAAL. The UPPAAL editor features undo and redo functionality.

UPPAAL features a C-like programming language that can be used for declarations. It has some limitations, such as it is not possible to access an array with an index derived via calculation.

With UPPAAL it is possible to formally verify the model. This can be performed through checking whether some conditions are guaranteed to hold. Testing the SUT with a verified model can give greater assurance that if the SUT conforms to the model then the SUT works as intended. This kind of formal verification was not done as part of this thesis.

For the SUT (more specifically the parts of the SUT) tested in thesis, the extra complexity of UPPAAL + DTRON solution might not make sense. This extra complexity could be necessary if testing more complex SUTs and be possibly useful for more complex parts of the SUT used in this thesis.

TDL^{TP} can be used to specify test scenarios. It deals with the SUT models. Additional work must be performed for the complementary tester model. Getting TDL^{TP} actually integrated into the UPPAAL + DTRON testing solution wasn't achieved as part of this thesis.

Both DTRON and the adapter output contain each message sent over channels together with the associated variables. If there is a conformance error between the model of the SUT and the actual SUT, then these details are also output. The console output is what can be used to debug issues. Unfortunately, to the best of the thesis author's knowledge, it is not possible to produce a replayable trace of test steps with DTRON. It might be possible with UPPAAL TRON that DTRON uses, but the necessary functionality appears to be not exposed via DTRON.

DTRON only supports sending and receiving integer values. Therefore it is necessary to have some kind of mapping between integer and string values. This can be an inconvenience. Due to this, it is not possible to have string difference indication — something that is possible with for example JUnit 5.

7.3 Evaluation of MBT in general

The part of the SUT that was tested in this thesis is not very complex. What reduces the complexity further is the fact that the mocked SUT was used. The lower complexity

allowed to concentrate more on the MBT testing solutions as compared to interaction with the SUT. When applied on this kind of a SUT, model-based testing did not expose errors that would not have been found by traditional testing or even attentive manual testing.

The goal of this thesis was to evaluate MBT on a real SUT through the use of two different tool sets. Covering all of the SUT was not the goal. This means that measuring code coverage of the MBT solutions would make little sense. Nevertheless, in the tests that were created, it could be observed that higher coverage is easier to achieve than with traditional testing. If MBT is applied to the more complex parts of the SUT and possibly with less mocking, the benefits of MBT will likely be clear.

This thesis tested the user interface part of a web application. In the author's opinion, MBT applies quite naturally to testing user interfaces. In this kind of a SUT there is one specific view active at each moment. And there are concrete actions that can be taken with regards to the current state. The "user" aspect of "user interface" means that the models describe what the users can do in the UI. Therefore, modelling a SUT UI can often be straightforward. Applying MBT to a UI can be in some ways easier than applying properly implemented traditional tests.

Though there wasn't much test maintenance to do during this thesis as the SUT was not updated in the process, observation still suggests that with MBT the maintenance process can be easier. With MBT, there can be less to update to fix a test, whereas with traditional testing multiple similar tests might need changing.

Through the experience from the two tool combinations used in this thesis, the authors opinion is that model-based testing is not currently fully applicable out-of-the-box. For SUTs that can be tested with the tool, GraphWalker comes quite close. However, its current version lacks a built-in functionality for producing proper test reports or replaying an older test run. GraphWalker does not make it easy to test arbitrary parts of the SUT. Some more complex SUTs might need a more capable testing tool such as the combination of UPPAAL + DTRON. Setting this combination up (together with TDL^{TP}) proved to be more cumbersome and time consuming. This combination, too, does not have a built-in way of replaying test runs.

The complexity of the test tool setup directly translates to the level of expertise required for creating and maintaining MBT tests. If the goal is to test (possibly a part of) a not very complex SUT in its entirety and basic test reporting is enough, then setting up and maintaining MBT testing with GraphWalker is not very difficult. After some general concepts have been learned, the further process shouldn't be too difficult. On the other

hand, setting up UPPAAL + DTRON + TDL^{TP} can require noticeably more expertise. Therefore the author's opinion is that the latter combination should be opted for if the additional power and features are necessary to test the SUT.

One additional benefit of model-based testing is that it can be used for certain kinds of non-functional testing. A straightforward way to use MBT for non-functional testing is to run a test for an extended time period. This test can use the same model that is used for functional testing. For example one may run a MBT test for several hours to find out if the system is reliable and stable. Using a random model walk for this can help ensure that different kinds of unusual situations may be tested during the long run.

Model-based testing, especially when applied to testing a web application, can be time consuming execution-wise. In this thesis, a kind of black-box testing called functional testing was effectively applied. This means that there is no interaction with the internals of the SUT during the testing process. What this means is that the test has to click through the UI to reach the part that is being tested, whereas in more of a white-box testing, the web application may be programmatically initialized to start straight at the view (or some other part) that is to be tested.

7.4 Evaluation of POM design pattern

The Page Object Model design pattern plays an important role of this thesis, since it was used to organize interacting with the SUT. Testing user interfaces of web applications can often be more difficult than testing some other kinds of SUTs such as backends. With user interfaces, it is useful to have a kind of abstraction layer between the SUT and the tests, and the POM pattern can be applied to implement it.

One apparent benefit of the POM design pattern is that it improves readability of the test code. The created POM classes encapsulate the internal details concerning how relevant HTML elements can be located, what processing has to be applied on obtained data and how actions can be performed. Thanks to this, the tests that use the POM classes can operate at a higher abstraction level.

Another important benefit of the POM design pattern is that it improves test maintenance. The main contributing factors for this are the encapsulation of internal details mentioned above, and also that the usage of POM classes reduces code duplication. If an internal detail of a UI component changes, it may often be sufficient to simply update the respective POM class in one place.

Implementing and testing POM classes themselves was a time consuming part of this thesis, especially when considering more complex views and components of the SUT. There were cases when a test failed because of an issue not in the SUT but in the implementation of a POM class. Sometimes a POM class for a component worked in one view but not in a different view, even though the component in the SUT was the same in both views. This meant that generalization issues of the POM classes were fixed as the testing progressed.

This thesis took the approach of not changing the SUT during testing. Due to this, there was some inconsistency in how HTML elements could be located; some selectors ended up quite complex and possibly fragile. Thanks to the benefits of the POM design pattern that are mentioned above, it is not difficult to adopt the usage of more consistent and better selectors once the necessary changes have been made in the SUT itself.

7.5 The role of traditional testing

The same POM classes can also be used for traditional (non-MBT) testing. Therefore, improving a POM class can benefit both MBT and non-MBT tests. In fact, to some extent the POM classes were used in this way during this thesis. It was easier and quicker to test a new POM class with a "sequential" traditional test. What is yet to be implemented are tests that apply to different parts in isolation, instead of simply going through steps sequentially and stopping the whole test in case of failure.

Model-based testing will not replace other kinds of testing; instead it should complement them. Considering the testing pyramid proposed by Mike Cohn [15], unit tests that test individual parts of the code are the foundation and should be most numerous, followed by tests for services in separation of the UI, and finally there are user interface tests, with the least amount of them. One of the reasons for this distribution of tests is the fragility of them. For example, if there is some change in the user interface, several UI tests may fail. Model-based testing is not mentioned in this testing pyramid. The opinion of the author of this thesis is that both regular and model-based testing sit at the same level of the pyramid.

While in theory, it would be possible to specify a single component to be tested with MBT, traditional testing fits better for this purpose. To make traditional testing of UI components better, it would make sense to put extra effort into starting the UI in the correct view instead of letting the test click through the UI to arrive at the desired view. This might require changing of the SUT and move the methodology closer to white-box testing. In contrast, MBT is useful for testing more components — or even the entire SUT — at a time.

7.6 Feedback from the company

The Quality Assurance (QA) lead of Milrem Robotics brought up several topics regarding the company's viewpoint of this thesis. They found that the thesis described how software development and automated testing could and should be handled at the frontend side; and that the thesis provides insights into how to improve the implementation of automation and speed of software testing.

They noted that if the SUT is not developed with automation in mind, eventual integration with automated frameworks can be too expensive or result in "hacky" easy to break automated tests. One example of this is the necessity of using XPath selectors that tend to be fragile versus adding IDs to elements of interest that can be used more reliably.

The QA lead pointed out that MBT can be the cheapest way to maintain and test frontend solutions, when there are multiple versions of them. This in turn brings up the topics of regression effort, time cost in case of back-ports and custom frontend solutions. Using manual labor for testing in this context can be really expensive.

Milrem Robotics currently maintains multiple frontend versions. The main reason for this is that updating UGV and UI versions is a complex process after they leave the factory. This in turn means that for every bug which wasn't identified on time, a fix must be deployed which requires backporting and testing with high coverage.

In the opinion of the QA lead, MBT can have great potential in case of the previously mentioned circumstances and can be applicable for all testing which is done based on automation. However this approach is not often considered as it requires some technical background, which may turn out to be a problem if there aren't people to hire. In it's current state, the MBT approach will not replace traditional automation because it is missing a few important elements which are required for everyday work. For example the approach is lacking in its reporting capabilities and integration with nowadays common system management tools (e.g. Jira, Jenkins). Nevertheless, MBT can have a certain place in release decision making, so the QA lead suggests that it should be used, if possible. However, if the mentioned shortcomings are dealt with, MBT could become the main automation approach.

In the QA lead's opinion, the POM classes and Selenium approach of this thesis is described as frontend automated testing should be. Therefore these can be reused and maintained without issues by other engineers. As one benefit of Selenium, the QA lead outlines that it is a raw platform which allows customization as demanded by corporation, therefore it is

flexible and covers all future needs.

7.7 Summary of evaluation

Choosing between the UPPAAL + DTRON and GraphWalker MBT solutions, the latter appears to be more suitable for this SUT, at least when considering the part of the SUT that has been tested so far. In the case of more complex parts of the SUT, it may still be possible to use GraphWalker, provided that the SUT behaves in a deterministic way and the pre-conditions are known.

Although MBT did not show clear advantages (when considering found errors) for the part of the SUT that was tested in this thesis, the hypothesis is that the advantages will be apparent with less mocking and more complex parts. Even if eventually the decision is to not move forward with MBT, the created POM classes can still be useful.

8. Future Work

This chapter presents some ideas for future work regarding the thesis.

One possible direction to move forward is towards covering some more complex parts of the SUT. A related direction is towards less mocking. Both of these, especially in combination, will show more of the usefulness of MBT.

Alongside with covering more complex parts of the SUT, it makes sense to cover more of it in general. The ultimate goal would be to cover all of the SUT, but it needs considering whether working towards this goal will pay off.

For there to be real benefits of testing, it needs to be integrated into the development process. For starters, running the tests may be initiated manually at certain times or after some features or fixes have been merged. In the longer perspective it would be beneficial to execute the tests as part of a CI (Continuous Integration) solution. Alongside with that, it may make sense to let the MBT solution run in a non-functional test fashion for example nightly and/or on the weekends, storing test reports and restarting in case of failure.

An important improvement — both for UPPAAL + DTRON and GraphWalker — would be better test reporting and the ability to replay test runs. Because UPPAAL and DTRON are both closed source, there might be little that can be done by outside users to improve this aspect of these tools. Thanks to GraphWalker being open source and having a welcoming community around it, it is possible to propose and discuss features and put effort towards implementing them. This was something the thesis author did for the ability to perform test execution steps playback in GraphWalker. The author discussed the feature in a Google Groups thread[1] and created a Pull Request[2] to GraphWalker that is intended to implement the first step towards this feature. As of writing this thesis, the feature is still mostly in initial planning phase.

One thing that what would make the GraphWalker testing solution better is a better way to

[1] `https://groups.google.com/g/graphwalker/c/qR8ZmBfltfg/m/qYp1Q1voBgAJ`, accessed May 9th, 2021

[2] `https://github.com/GraphWalker/graphwalker-project/pull/263`, accessed May 9th, 2021

specify test cases. The solution this thesis uses is pretty cumbersome and has resulted in excessive code. If the models are updated, the previously defined test cases may break in different ways. One possible solution would be to create a path generator that can operate with multiple models together and utilize some kind of annotations for vertices and edges that can be used when specifying the test case. Another potentially combinable solution would be to explore adding guards to model edges which can guide the model traversal.

There are also some smaller improvements that can be applied to GraphWalker itself. One of them is to fix the GraphWalker Studio editor pan and zoom reset which occurs under some conditions when clicking on model elements. Adding undo and redo feature to the editor could also prove to be useful. Then there is a more nice-to-have feature — being able to reorder GraphWalker studio tabs.

9. Summary

As part of this thesis, model-based testing (MBT) was applied to a System Under Test (SUT). The SUT used in this thesis was the web application user interface project developed by Milrem Robotics for use with the company's THeMIS vehicles.

The thesis used two different MBT solutions — one which primarily used GraphWalker and another that used the combination of UPPAAL + DTRON. Two different solutions were used with the aim of comparing different approaches. The goal of this thesis was not to fully cover the SUT with MBT. Interaction with the SUT UI web application was implemented using browser automation software Selenium. To improve test code readability and maintainability, the Page Object Model design pattern was adopted.

It was observed that UPPAAL + DTRON was in some ways the more powerful solution, but it was also more difficult to set up and use. UPPAAL makes it possible to model more complex SUTs, but for the SUT used in this thesis, this additional power was not strictly necessary. It was concluded that GraphWalker appeared to be a better fit for this SUT. The process of setting it up and continually using it was simpler. This conclusion was also influenced by the matter of licenses — UPPAAL is free for non-commercial academic use only, so if the company wants to keep using the MBT solution based on this tool, they would have to start paying a currently unknown amount of money. In contrast, GraphWalker uses the permissive open source license MIT, which imposes no limitations of commercial use.

There are several opportunities regarding future work. For one, it is possible to start moving towards higher test coverage, including the coverage of the more complex parts of the SUT together with less mocking. It can also be beneficial to include MBT testing into the development process at the company. Regarding GraphWalker, it was observed that test splitting could be improved. There are also some improvements, fixes and features that could be implemented into the tools itself; in the case of the open source tool GraphWalker, it is possible for anyone interested to put development effort into getting these implemented.

Bibliography

[1] Mark Utting, Alexander Pretschner, and Bruno Legeard. "A taxonomy of model-based testing approaches". In: *Software Testing, Verification and Reliability* 22.5 (2012), pp. 297–312. DOI: https://doi.org/10.1002/stvr.456. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.456. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.456.

[2] G.J. Tretmans. "A Formal Approach to Conformance Testing". English. PhD thesis. UT, 1992. ISBN: 90-9005643-2.

[3] M. Leotta et al. "Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. 2013, pp. 108–113. DOI: 10.1109/ICSTW.2013.19.

[4] Gerd Behrmann, Alexandre David, and Kim G. Larsen. "A Tutorial on Uppaal". In: *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures*. Ed. by Marco Bernardo and Flavio Corradini. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 200–236. ISBN: 978-3-540-30080-9. DOI: 10.1007/978-3-540-30080-9_7. URL: https://doi.org/10.1007/978-3-540-30080-9_7.

[5] Jin Hyun Kim et al. "Formal analysis and testing of real-time automotive systems using UPPAAL tools". In: *International Workshop on Formal Methods for Industrial Critical Systems*. Springer. 2015, pp. 47–61.

[6] Anders Hessel et al. "Testing Real-Time Systems Using UPPAAL". In: *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*. Ed. by Robert M. Hierons, Jonathan P. Bowen, and Mark Harman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 77–117. ISBN: 978-3-540-78917-8. DOI: 10.1007/978-3-540-78917-8_3. URL: https://doi.org/10.1007/978-3-540-78917-8_3.

[7] Aivo Anier and Jüri Vain. "Model based continual planning and control for assistive robots." In: *HealthInf 2012* Proceedings of the International Conference on Health Informatics (2012), pp. 382–385. DOI: 10.5220/0003783503820385.

[8] Evelin Halling et al. “TEST SCENARIO SPECIFICATION LANGUAGE FOR MODEL-BASED TESTING”. In: 18 (Dec. 2019), pp. 408–421.

[9] Philip Makedonski et al. “Test descriptions with ETSI TDL”. In: *Software Quality Journal* 27 (June 2019). DOI: 10.1007/s11219-018-9423-9.

[10] Tanel Prikk. “Implementation of an Interpreter for the Test Purpose Specification Language TDL(TP)”. Bachelor’s Thesis. Tallinn, Estonia: Tallinn University of Technology, June 2019.

[11] Boni García et al. “A Survey of the Selenium Ecosystem”. In: *Electronics* 9.7 (2020). ISSN: 2079-9292. DOI: 10.3390/electronics9071067. URL: https://www.mdpi.com/2079-9292/9/7/1067.

[12] Fatini Mobaraya and Shahid Ali. “Technical Analysis of Selenium and Cypress as Functional Automation Framework for Modern Web Application Testing”. In: *9th International Conference on Computer Science, Engineering and Applications (ICCSEA 2019). Presented at the 9th International Conference on Computer Science, Engineering and Applications, Aircc publishing Corporation*. 2019, pp. 27–46.

[13] Magnus Teekivi. “Development of Extensible Vue.js-based User Interface for Robotic Vehicles”. in Estonian. Bachelor’s Thesis. Tallinn, Estonia: Tallinn University of Technology, June 2019.

[14] Jüri Vain, Aivo Anier, and Evelin Halling. “Provably Correct Test Development for Timed Systems.” In: *DB&IS*. 2014, pp. 289–302.

[15] Mike Cohn. *Succeeding with agile: software development using Scrum*. Pearson Education, 2010.

Appendix 1 - Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I, Magnus Teekivi

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Model-Based Testing of Web Applications", supervised by Evelin Halling
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 - HeaderBar.java

In the following, the code for the POM class HeaderBar is provided.

```
public class HeaderBar {
  private final By titleBy = By.cssSelector(".headerbar h1");
  private final By subTitleBy =
      By.cssSelector(".headerbar .sub-title");
  private final By backButtonBy =
      By.cssSelector(".headerbar .go-back-button");
  private final By rightSideButtonBy =
      By.cssSelector(".headerbar .side:last-child button");
  private final WebDriver driver;

  public HeaderBar(WebDriver driver) { this.driver = driver; }

  public String getTitle() {
    return driver.findElement(titleBy).getText();
  }

  public String getSubTitle() {
    return driver.findElement(subTitleBy).getText();
  }

  public void goBack() {
    driver.findElement(backButtonBy).click();
  }

  public void clickRightSideButton() {
    driver.findElement(rightSideButtonBy).click();
  }

  public boolean isRightSideButtonClickable() {
    return driver.findElement(rightSideButtonBy).isEnabled();
  }
}
```

Appendix 3 - DtronOperatorAdapter.java

In the following, a shortened version of *DtronOperatorAdapter.java* is provided. It contains the class that fits the actual SUT into the UPPAAL + DTRON testing solution. The name part "operator" refers to the SUT's project name.

```
public class DtronOperatorAdapter {
  private static final String SUT_URL = "<URL>";

  public static void main(String[] args) throws SpreadException,
                                                IOException {
    Dtron dtron = new Dtron();
    dtron.connect();

    WebDriver driver = new ChromeDriver();
    driver.get(SUT_URL);

    HeaderBar headerBar = new HeaderBar(driver);

    IDtronChannel initChannel = new DtronChannel("init");
    dtron.addDtronListener(new DtronListener(initChannel) {
      @Override
      public void messageReceived(
          IDtronChannelValued iDtronChannelValued) {
        driver.get(SUT_URL);
        try {
          getDtron().send(initChannel.constructValued(
              Collections.emptyMap()));
        } catch (SpreadException e) { /* ... */ }
      }
    });

    IDtronChannel getvalChannel = new DtronChannel("getval");
    dtron.addDtronListener(new DtronListener(getvalChannel) {
      @Override
      public void messageReceived(IDtronChannelValued valued) {
        Map<String, Integer> values = valued.getVariables();
        UiConstant key = UiConstant.getById(values.get("key"));
        UiConstant arg = UiConstant.getById(values.getOrDefault(
            "arg", 0));
```

```
    UiConstant valueConstant;

    switch (key) {
      case GETVAL_HEADERBAR_TITLE: {
        String titleLabel = headerBar.getTitle();
        valueConstant = UiConstant.getByLabel(titleLabel);
        break;
      }
      // ...
      case GETVAL_MODE: {
        String modeLabel = arg.toString();
        VehicleModeButton modeButton = new VehicleModeButton(
            driver, modeLabel);
        String modeButtonValueLabel = modeButton.getValue();
        valueConstant = UiConstant.getByLabel(
            modeButtonValueLabel);
        break;
      }
      // ...
      default: {
        throw new IllegalArgumentException(
            "Invalid key for getval: " + key);
      }
    }
    Map<String, Integer> outMap = new HashMap<>();
    outMap.put("value", valueConstant.getId());
    IDtronChannelValued outValued =
        getvalChannel.constructValued(outMap);

    try {
      getDtron().send(outValued);
    } catch (SpreadException e) { /* ... */ }
  }
});

IDtronChannel actChannel = new DtronChannel("act");
dtron.addDtronListener(new DtronListener(actChannel) {
  @Override
  public void messageReceived(IDtronChannelValued valued) {
    Map<String, Integer> values = valued.getVariables();
    UiConstant key = UiConstant.getById(values.get("key"));
    UiConstant arg = UiConstant.getById(
        values.getOrDefault("arg", 0));
    switch (key) {
      case ACT_SELECT_VEHICLE: {
        String vehicleName = arg.toString();
        VehicleSelectionView vehicleSelectionView =
```

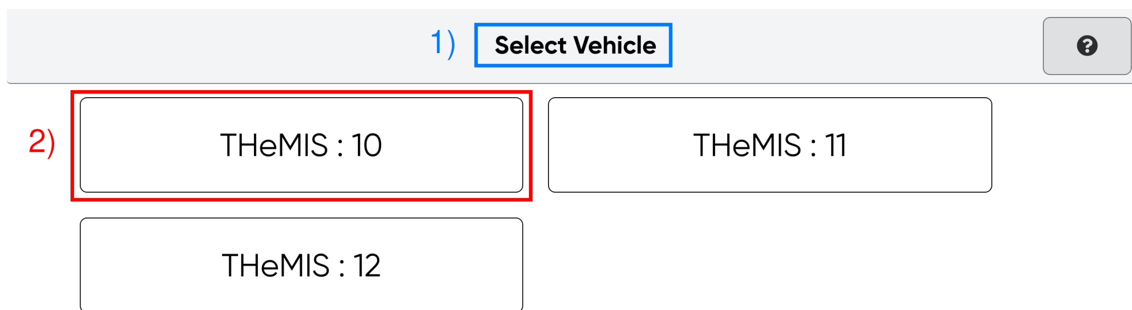
```
                new VehicleSelectionView(driver);
            vehicleSelectionView.selectVehicle(vehicleName);
            break;
          }
          case ACT_GO_BACK: {
            headerBar.goBack();
            break;
          }
          // ...
          default: {
            throw new IllegalArgumentException(
                "Invalid key for act: " + key.toVerboseString());
          }
        }
        try {
          System.out.println(
              "Sending empty response to act channel");
          getDtron().send(actChannel.constructValued(
              Collections.emptyMap()));
          Thread.sleep(50);
        } catch (SpreadException | InterruptedException e) {
            /* ... */ }
      }
    });

    System.in.read();
    driver.quit();
  }
}
```

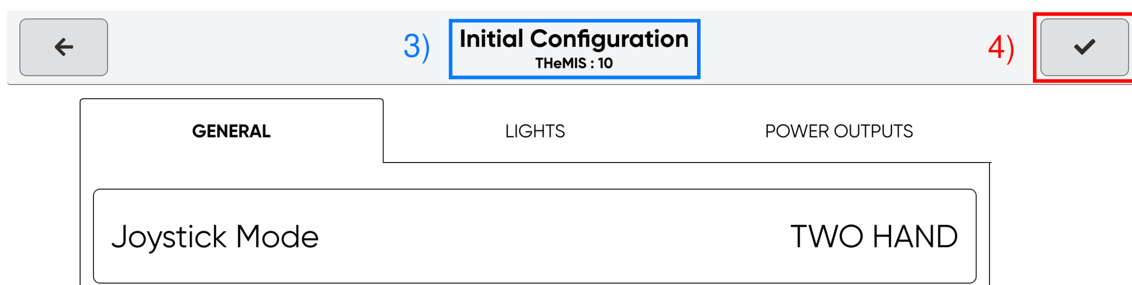
Appendix 4 - Initial scenario

In the following, the initial scenario created as part of this thesis is described in a step-by-step fashion with annotated (and cropped) screenshots of the System Under Test. Each step describes an assertion or an action.

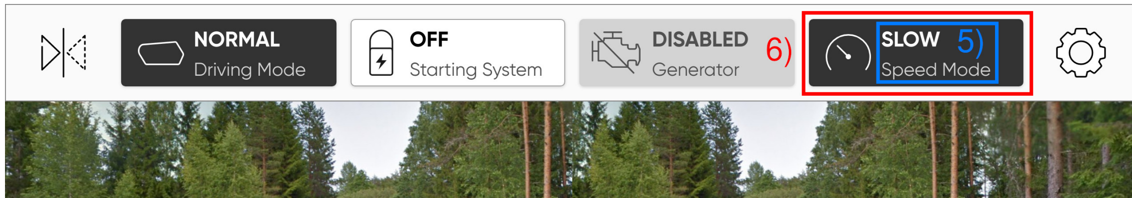
As part of the initial scenario, a vehicle is chosen, its configuration is submitted, and in the main view, speed mode is changed from slow to fast.



1) Assertion: Header bar title is "Select Vehicle"
2) Action: Choose vehicle named "THeMIS : 10"

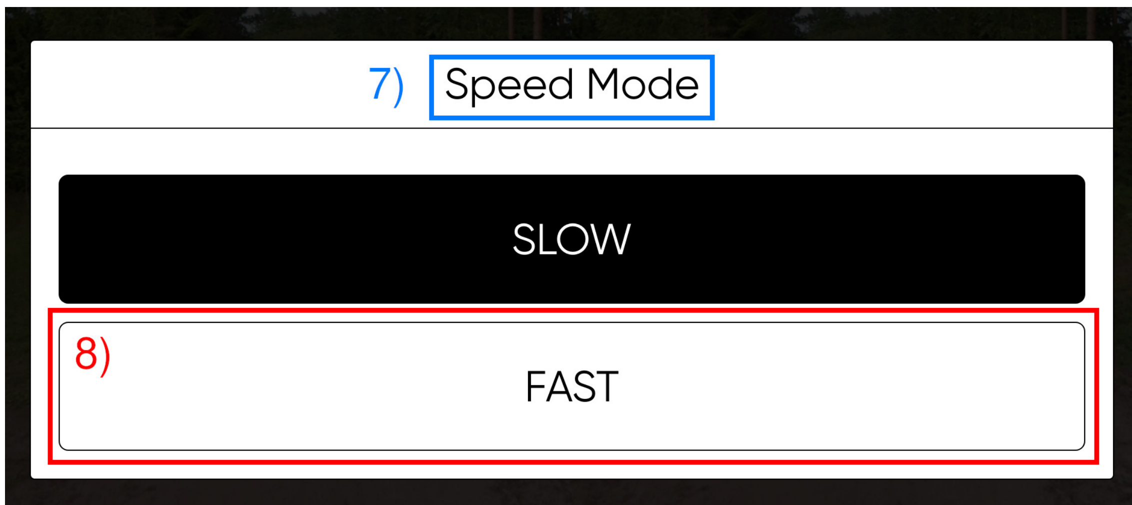


3) Assertion: Header bar title is "Initial Configuration" and subtitle is "THeMIS : 10"
4) Action: Click on header bar right side button



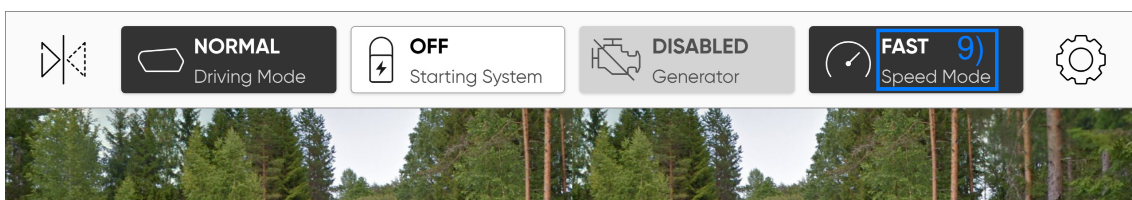
5) Assertion: Value of vehicle mode button with label "Speed Mode" is "SLOW"

6) Action: Click on vehicle mode button with label "Speed Mode"



7) Assertion: Option modal with title "Speed Mode" is visible

8) Action: Select the option "FAST"



9) Assertion: Value of vehicle mode button with label "Speed Mode" is "FAST"