Radboud University Nijmegen

Master Thesis
Computer Science

MODEL-BASED TESTING OF WEB APPLICATIONS

26 May 2015

Author:
J.R. Monsma, BSc
Student No. s4236777

Internal Supervisor:
Dr. ir. G.J. Tretmans

Second Supervisor:
Dr. G. Gousios

External Supervisor
Ing. B. Duijs, MSc

Radboud University
Comeniuslaan 4
6525 HP Nijmegen
The Netherlands
Faculty of Science
Huygens building
Heyendaalseweg 135
6525 AJ Nijmegen
The Netherlands

Abstract

Software shifts more and more to the online world. Testing web applications requires documents that specify what their correct behaviour is. Lack of documentation for smaller web application makes testing difficult. In this study, we try to find a solution to test web application based on a model-based testing approach. We use model-based testing for detection failures in web applications, the differences between the behaviours of the System under Test(SUT) and what is expected based on the specification. We have chosen for the model-based testing approach because it automates the testing process, adapts quicker to changes, it is time saving, and less error prone if the system is modelled correctly.

Using G∀ST as a model-based testing tool, we generate test cases based on the model of the web application. PhantomJS, a headless browser is used to access a web application. With an adapter, we established the communication with the model-based testing tool and the headless browser. The adapter communicate with sockets to the model-based testing tool and translate received input from the model-based testing tool to useable input for the SUT.

To create a generic model that is capable to test different web applications, we assume that every web page is the same, based on its content. Each page can hold links, events, images, and Styling- and JavaScript code. Based on this abstraction we created a small model that is applicable to test many web applications without any changes to the model. This model clicks on links and check for content errors. We extended this model to test forms, based on three options. Testing results show that the model is generic and capable of testing different web application. However, larger applications the amount of pages tested decreases based on the random generation of test cases by G∀ST, pages are tested multiple times instead of testing undiscovered pages.

Testing webshop applications, the model requires an extension to test features that occur in a webshop. Results show that the model does not suits itself to test different webshop applications. The checkout phase of a webshop differs and it is hard to model it to a generic solution. However, the functionalities of a webshop stays the same and there we have chosen to allow changes in the model. Testing results show we can add and buy products, but the same problem with hold with generating test cases.

Based on testing two web applications with forms and two webshop applications we can say that we can improve quality of web applications with model-based testing. However, the improvement is not enormous. There is improvement required on the test selection by G∀ST. To increase the quality even further more quality characteristics can be used to test on.

Acknowledgments

I would like to take this opportunity to thank the people that have helped me with this thesis. First, I would like to thank Jan Tretmans and Bard Duijs for their supervision and their clear view and thoughts about this topic.

Furthermore, I would like to thank Bard Duijs and Laurens Alers for giving me the opportunity to write this thesis at Bluenotion. It was great to spar with you guys to get fresh new insights. I also want to thank you for giving me the freedom to sail my own course and tackle certain problems I faced.

I would also like to thank the developers of Bluention for setting up a test environment and solving some internal problems we faced during testing of the use cases. I appreciated their opinions and insights on the topic and their advice on how to address a certain problem.

I would like to thank Pieter Koopman for his help with the Model-based testing tool G∀ST.

Lastly, I would like to thank my friends and family, especially for the refreshing coffee breaks and having someone to talk to, and my parents and girlfriend for the pep talks when I needed those.

Contents

ABSTRACT ... 1

ACKNOWLEDGMENTS ... 2

1 INTRODUCTION ... 5
- 1.1 Area of online web applications ... 5
- 1.2 Problem statement ... 6
- 1.3 Research questions ... 7
- 1.4 Research context ... 9
- 1.5 Research method ... 9
- 1.6 Thesis outline ... 9

2 RELATED WORK ... 10

3 BACKGROUND ... 12
- 3.1 Web applications ... 12
  - *3.1.1 Forms ... 13*
- 3.2 Software testing ... 16
- 3.3 Model-based testing ... 18
  - *3.3.1 Modelling languages ... 18*
  - *3.3.2 Labelled Transition System ... 18*
  - *3.3.3 Finite State Machine ... 19*
  - *3.3.4 Extended State Machine ... 20*
- 3.4 Model-based testing tools ... 22
  - *3.4.1 Overview of tools ... 22*
  - *3.4.2 Graphwalker ... 22*
  - *3.4.3 TorXakis ... 23*
  - *3.4.4 G∀ST ... 23*
- 3.5 Choice of modelling language and model-based testing tool ... 25

4 TEST ARCHITECTURE ... 27
- 4.1 Overview of the current situation ... 27
- 4.2 Overview of the test architecture ... 28
- 4.3 Content Management System ... 30
- 4.4 The System Under Test ... 30
  - *4.4.1 Routing Engine ... 30*
  - *4.4.2 Application models ... 32*
  - *4.4.3 Views ... 33*
  - *4.4.4 Controllers ... 33*
- 4.5 The adapter ... 34

5 BEHAVIOUR OF WEB APPLICATIONS ... 37
- 5.1 Common testing ground ... 37
- 5.2 Our testing ground ... 39

6 THE SMALL MODEL ... 42
- 6.1 Existing model ... 42
- 6.2 Which parts of a web application need to be tested? ... 43
- 6.3 The small model ... 44
- 6.4 The form extension ... 48
- 6.5 Results ... 51
  - *6.5.1 Use case: Lookinsharp ... 51*
  - *6.5.2 Use case: Bluenotion ... 53*

7 WEBSHOP MODEL **56**
7.1 WEBSHOP APPLICATIONS 56
7.1.1 Products *56*
7.1.2 Checkout phase *57*
7.2 MODEL 59
7.3 RESULTS 63
7.3.1 Matrascenter.nl *63*
7.3.2 Wordpress Webshop application *66*
8 REFLECTION **69**
9 CONCLUSION **72**
9.1 COMPATIBILITY OF THE TEST TOOL 72
9.2 COMPATIBILITY OF WEBSHOP 72
9.3 ANSWER RESEARCH QUESTIONS 73
9.4 KNOWN ISSUES 77
9.5 FUTURE WORK 77
10 REFERENCES **79**

1 Introduction

We will start this section with a small introduction about web applications, followed by section 1.2 where we will discuss the problems surrounding the topic of web application testing. In section 1.3, we define the research questions related to our problem. In section 1.4, we will state the research context. The research method is explained in section 1.5. At last, we mention our thesis outline in section 1.6.

1.1 Area of online web applications

More and more things are shifting to the Internet. Applications that are built for offline use are now available online[2, 3]. People have the need to be online[4]. Some web applications are publicly available while others are private and only accessible through login credentials[5]. Web applications are built with multiple programming languages and can make use of different design patterns. One way of building web applications is by using the .NET framework combined with the MVC design pattern. The MVC (Model-View-Controller) pattern is used to divide a complex system into three components, each component with their own responsibilities. The model contains data, the view the presentation, and the controller the logic of the system. The web applications we are currently interested in are based on the .NET framework supported with this MVC design pattern[6]. For many years, when applications, for offline purposes, were developed less effort was spent in testing the software.

Nowadays more effort is spent in testing and measuring quality of software. One of the reasons is that customers demand a higher quality of software. Testing is now integrating in development methodologies, like agile development[7]. With the latter, testing is a part of the development cycle. The development of software contains at least several cycles and this amount is increasing depending on the size of the software. In each cycle, testing is performed on the developed code. By testing during these development phases more faults can be detected, resulting in more sustainable and higher quality software. The gain in sustainability results from the fact that solving faults during development time or even preventing them on the drawing board is cheaper than fixing the same faults afterwards[8]. The software is of better quality, because we know the system is test and possible faults that could occur are prevented. However, this does not guarantee that the software is free from bugs. It is possible that faults in the software still exists but were not detected with testing.

While more effort is nowadays being spent on the testing of normal software applications, there is still a lack in testing effort during the development of web applications. Simple web applications, that only have the purpose of serving informational content, are often either insufficiently tested or not tested at all[9, 10]. Hence, these web applications still contain faults. A specification is often used during the testing of software. Otherwise, we do not know what we have to test. However, in most of the cases there is not any specification available, when developing web applications. Most of the time there is no documentation available. This makes knowing what to test hard.

A great example is the lack of proper documentation at the company Bluenotion for small web application. From experience, we know that the actual design of the web application is the only available documentation. Furthermore, the small web applications are badly tested and still contain faults after the application is going live.

1.2 Problem statement

As explained in section 1.1, more and more effort is being spent on the proper testing of software. One of the reasons is to deliver higher quality software. Insufficient testing of web applications still results in unwanted errors and consequently, the quality diminishes. Users will not use such an application and revenue will decrease. By adequately testing a web application, we can prevent those errors from happening and prevent the drop of quality. However, there are still difficulties with testing of web applications. Of course, manually testing web applications is possible but then a different problem arises. Manual testing is time consuming and there is often little to no time available to test after the development of the web application. A better and desirable option is automated testing or model-based testing. We want to propose a plan to solve these testing problems with web applications and create a foundation that we can use to model-based test web applications.

One of the problems with the automated testing of web applications is the lack of proper documentation. Simplistic web applications, that only have the purpose of serving informational content, are often implemented without (proper) documentation. This can lead to inconsistent behaviour and bugs. Without documentation, there is no explanation what the preferred behaviour of the web application is, which makes testing even harder. Even worse is that those simplistic applications are often not even tested. Still even those simplistic web applications can contain bugs. Therefore knowing what kind of aspects we need to test, and the way to it are important.

A second problem with the testing of web applications is that multiple programming languages are involved. With the involvement of multiple languages, more mistakes happen when combining those languages. To let a web application work properly, languages like HTML for the mark-up, CSS for the looks of the application, and JavaScript for functionality are commonly used. For dynamic web applications, where our focus in testing lies, an additional language is used. This is used for retrieving data from databases and displaying the retrieved data. Certain parts of the web application can be tested with unit tests, but this is time consuming due to writing specific tests for each individual application and unit tests do not deal very well with changes.

Today's web applications are not static anymore but dynamic. Static web applications are web pages containing information that is not changing anymore. Dynamic web applications can deal with changing information. By accessing a content management system (CMS), the owner of the web application can add, remove, or change the data. A good example is a blog. Once a week we write an article about a certain subject. We write text about the subject and save it in the CMS. When the data is saved, it is almost instantly visible in our web application. When we have a blog made in a static way, we have to open the source code and change the information we want, save it and upload the new source file. It is intensive, time consuming, and requires some knowledge about the programming languages mentioned above. In dynamic web applications it is, for owners, easier when changing information and require no programming skills to use them.

Because of dynamic web applications, owners are able to change, add, or remove information that is readable on their application instantly. The adjustments are stored in a database. When users visit the web application, through a web browser, the necessary data is retrieved from the database and is displayed in the browser. With the changes the owner makes, it could be possible that the application does not work properly anymore. For example, links to certain parts of the application are not working, due to typos or the page does not even exist. Testing with unit tests, which is used to test small parts of a system are therefore not enough because they do not cover and adapt well to the changing part in dynamic web applications.

With the introduction of the programming language JavaScript, certain functionality is added to the web application. This programming language runs on the client-side of the application, in this case within a web browser. With JavaScript, it is possible to submit forms or add a product to the shopping cart through an Asynchronous JavaScript and XML (AJAX) call to the server. This functionality is preferably tested on the client-side of the application by accessing it through a web browser. This is done by an event activated by the user through the browser. The other approach of testing AJAX calls is by manually creating the calls and providing the correct data that is required for this AJAX request.

Now that we know what kind of difficulties we can face during the testing of a web application, we want to find a way to solve these difficulties. Automated testing of software can be performed by testing it with a model. This approach is called model-based testing. We want to see if we can apply model-based testing techniques to test web applications. Especially in finding a generic approach to test web applications. This way the model can be applied to multiple web applications. In the following section, we define a set of research questions that can help us to see whether model-based testing can be applied to web applications.

1.3 Research questions

In section 1.2, we named several problems that we have to deal with when we want to test web applications automatically, preferably by a model-based testing approach. This brings us to our research question:

RQ 1 How can we improve the quality of web applications by applying model-based testing techniques?

It is important to know what part of a web application are required to be tested otherwise we are testing the wrong parts. Are we interested in the graphical user interface or only interested in the functionality of the web application?

RQ 1.1 What quality aspects are important with testing web applications?

We are interested in testing web applications with a model-based testing approach. Therefore, it is important to know if it is possible to test web applications with this approach. By finding a model, which is required to test with model-based testing, that is capable of simulating our web application we can automatically test the problems we face with testing web applications. We can run the test multiple times, even when owners change the content of their application, this way we make sure that each link that is added by the owner also works.

To model a representation of a web application we need to know how we model a web application without proper documentation. How do we translate input and output actions from a web application to a model-based testing tool? What information does the input and output actions contain, what kind of abstraction is used? To see how those techniques can be applied and achieved we defined the following sub questions:
RQ 1.2 How can we apply model-based testing techniques to web applications?

When using model-based testing techniques a model is required to test the System under Test (SUT). Therefore, we want to see what kind of models already exist in the literature and how those can be applied when testing web applications in a model-based way. Thus, we define the following sub question:
RQ 1.3 What are the advantages and disadvantages of existing modelling languages, found in literature, with model-based testing of (web) applications?

Certain models have disadvantages we do not like because then the model would be better of use. It could be possible that some existing models have different advantages that we would like to combine. In that case, we will need a new model. This leads to a new sub question:
RQ 1.4 How can we create a new model that is capable of testing web applications that deals with the disadvantages of existing models?

It is also important to know how the new model performs compared to manual testing and the coverage of the model. The performance is needed to help us determine if the quality of the web application increases or not. This leads us to the next sub question:
RQ 1.5 How does the new model perform?

We initially have the intention to test web applications developed with the .NET framework. To achieve a much broader audience we would like to see what the capabilities of the model are with regard to web applications developed in different languages. This brings the following sub question:
RQ 1.6 What are the possibilities with the new modelling framework to test different web applications?

To be able to test our model we need a selection of tools that makes testing possible. What tools do we need to translate test input to real actions in a web application and how do we translate actions from the web application back to test output. What tools do we need to generate automated test cases based on the type of model that is used?
RQ 1.7 What set of tools are required and available to test web applications?

We want to compare results from the model-based testing tool with results when testing web applications manually this to determine the improvement of quality. How many faults are found during automatic testing compared to manual testing the web application? We are also interested to see if certain types of faults are not found in this approach with model-based testing.
RQ 1.8 How does the tool perform in coverage with regard to other existing tools and manual testing?

With the answers of these sub questions, we hope we can answer our main research question in section 9.

1.4 Research context

This study is performed at Bluenotion, a web development company specialised in building websites, webshops and custom web applications. Applications are built with the .NET framework combined with the MVC methodology (Section 4.4). Bluenotion has interests in the ability to automate test those type of web applications, to increase the quality of their products. Currently testing is done manually with some error detection tools.

1.5 Research method

We start with a literature study about the subject, which is model-based testing of web applications. In our literature study, we found enough information about model-based testing and about testing web applications, but both topics combined left us with only a few relevant papers. We used these as a base to continue finding more relevant information. Based on the relevant literature we found out that it should be possible to test web applications with a model-based approach. With that in mind, we wanted to know if it is possible to find how general our solution could be, so it is possible to test multiple web applications with the same model. To find a solution we first need to know what we want to test.

We identified important quality aspects that we want to test in web applications. Most quality aspects are ought important to the company Bluenotion. To this list we added other quality aspects that other companies find relevant to test on web applications. We narrowed this huge list down to a selection based on the primary requirements of Bluenotion and aspects that are interrelated.

Based on our quality aspects we search for the best type of modelling language that could deal with our requirements. Our test tool is chosen based on the modelling language we selected. We then created a test environment to see how our model performs. We started with a small web application to see how our model works. We made some small changes according to our result and started to extend the model so we could test forms. Based on this result we extended our model so we could test a full webshop application, now named as webshop. We compared the findings from the test tool against manual testing of the webshop. We also tried to test our solution on a different webshop to see what changes were needed to our model. We also compared coverage with a different tool called Crawljax.

1.6 Thesis outline

In section 2, we state related work within the field of testing web applications. We mention background information in model-based testing and on web applications in section 3. The test architecture that we use is defined in section 4. In section 5, we state the differences in what the companies and we consider relevant to test on web applications. The new defined model (section 6.3) based on the improvements of existing models (section 6.1) and an extension with forms, including its results on two use cases are mentioned in section 6. In section 7 we extend our model even further to be able to test webshop applications. The results of this webshop model based on the new uses cases are mentioned in section 7.3. We start finishing with a reflection based on the choices we made during this research in section 8. We conclude this thesis in section 9, including known issues (section 9.4) and future work (section 9.5).

2 Related work

In our research effort for finding relevant literature about model-based testing of web applications, we only found a few papers that we find relevant. We do found enough literature about both topics seperately, model-based testing and web applications.

A document written for the software testing conference STANZ in 2010, is found to be useful and is about model-based testing of web applications[11]. In this document, the model-based testing tool TestOptimal is used to test web applications and the modelling language FSM to model the web application. Their approach resulted in a huge detailed model, making it unreadable but suitable to test many different quality characteristics.

A similar approach is done by using StateCharts as a modelling language[12]. Ogaard tries to model web applications as it appears to users. The level of detail is now based on the content of each web pages instead of the web page itself used in [11]. The content is based on the HTML structure of the web page, meaning that every HTML element that matters is represented as a “Blob” in a StateChart. HTML elements that matter are elements that have a functionality on the web page like links, images, form elements, and buttons.

Work done by Andrews takes a different approach where the web application is partitioned into clusters where, for each cluster a separate FSM is created and one application FSM is to cover all the clusters[13]. A similar approach with the same way of clustering is done by Kung only based on an object oriented model[14] and Karam uses the same techniques but uses Workflow Graph Models[15].

Ernits approach to test web applications is based on both client- and server-side testing[16]. Custom scripts are made to intercepts messages on other interfaces used besides the front-end to test the server side of the application. NModel is used as the modelling framework to define model programs. Labelled Transitions Systems (LTS) are used to support the semantics of these models programs used in NModel.

A different aspect within testing web applications is testing Ajax calls. Ajax, Asynchronous JavaScript and XML is a bundle of technologies used to simplify implementation of rich and dynamic web application. The main purpose of Ajax is asynchronous communication between the client (web browser) and server (web application). Marchetto and Tonella describe a state-based approach to test Ajax calls within web applications[17]. Mesbah and van Deursen proposed a method to test AJAX applications automatically, based on a crawler[18]. This crawler, named Crawljax, tries to infer a state-flow graph based on all user interface actions.

Other literature we found is about testing sub parts of web applications. Testing the checkout phase of a e-commerce system is done in[19]. The checkout phase is about adding products to a shopping cart and the steps it takes to buy these products. Other interests are in testing HTML forms. Hajiabadi and Kahani proposed a method to derive input data to test forms[20]. Their approach tries to fill in and test forms automatically based on ontologies. A complete different approach by Bae is about Graphical User Interface(GUI) testing[21]. However, it is out of the scope of this research but is still relevant in testing web applications.

We consider the approach to test web application with TestOptimal as a good reference. We do take other references about model-based testing into account, only we mainly focus on their approach to test certain quality characteristics and not much on their modelling language. Web applications that make use of AJAX calls will not be tested as in the approach mentioned by Marchetto or Mesbah, but probably assumed as a synchronous call. To compare the testing results from our testing tool we use Crawljax as a different testing tool to see what differences there are. Furthermore, we take into account the detailed information about testing forms and a shopping cart.

3 Background

In this section, we discuss background information about our study. In section 3.1, we will provide information about web applications. We focus on the information that is needed to retrieve data from the web application by a user approach. In section 3.1.1, we give detailed information about forms within a web application. This level of detail is necessary to retrieve and set information to those form fields. We explain more about software testing in section 3.2. In section 3.3, we give details about modelling languages used with model-based testing followed by section 3.4 where we provide information about existing model-based testing tools. We end this section with section 3.5 where we discuss our choice of modelling language and model-based testing tool.

3.1 Web applications

The World Wide Web started with static web pages. We now call this era the Web 1.0. The Web 2.0 is the successor of Web 1.0. It is not an upgrade to the World Wide Web but merely a term to address the changes happened in this field. One of the founders of Web 2.0 is Tim O'Reilly and he describes in his paper the shift from Web 1.0 to Web 2.0[22]. Web applications in the Web 2.0 era are dynamic. Users can generate data, which is stored in databases. With scripting languages like PHP, C#, and Ruby, data is retrieved and surrounded with HTML code to display it properly in a web browser. HTML (HyperText Markup Language) is a language, which is used to create webpages. HTML uses tags to mark-up a webpage. An HTML document consists of a tree of elements and text leaves. Each node element is denoted with a start stag, such as "<html>", and a closing tag, such as "</html>". Elements have to be nested, such that they are not overlapping each other. In Example 1 the top line shows a wrong example of nested elements and the bottom line shows a correct example.

```
<p>This is <em>a <strong>wrong</em> mark up!</strong></p>
<p>This <em>is <strong>a correct</strong> mark up!</em></p>
```

Example 1 – Wrong and correct example of nested html elements.

When a web browser requests a web page, the HTML document is sent from the server that hosts the web application to the user's web browser. The web browser tries to parse the HTML document in a Document Object Model (DOM)[23]. To do that it first requests, by TCP, all the files that are mentioned in the HTML document. Those could be image-, JavaScript-, font- and style sheet files, in short media files. It could occur that one of those files does not exist this file will not be parsed into the DOM object. A DOM object is an in-memory representation of the HTML document, style sheet, and JavaScript files combined. The DOM object defines the HTML elements as objects, properties of all HTML elements, methods to access all HTML elements and events for all HTML elements.

A style sheet describes how a certain HTML element should look like. We can define for each HTML element for example its height, width, background-colour, and font size. It is programmed in the language Cascading Style Sheet (CSS)[24]. The properties for each element mentioned in the CSS files are set to the corresponding properties for each element in the DOM object it matches. It is possible to define properties for a single HTML element, a type of html element or HTML elements that matches the name of the class or id attribute that is set to that element.

JavaScript files mostly contain functions that evoke certain events. With JavaScript it is possible, to bind events to certain HTML elements. Those events are also stored in the DOM object that matches the html element. An event can be evoked by a keystroke, mouse action (click or movement), or time event.

In the DOM object, we have the possibility to search for html elements that match certain requirements. Using the methods "querySelector" and "querySelectorAll" we can find the elements that matches our description. If multiple elements matches while using the "querySelector" method, only the first result is returned. If we use the "querySelectorAll" method all elements that matches are listed. It is also possible to combine attributes and element names to only search for specific elements. By searching with the following command, we only search for links that have a class name menu and contain the attribute 'href':

$$document.querySelector('a.menu[href]')$$

The HTML elements we are interested in can be found by their attributes or element name. Links in a web page are represented by the "<a>" tag. Each link tag holds an attribute named "href" that describes what page is opened when clicked on. Additionally we can also identify html elements by the attribute "name", "action", "id", and "class". For the name, action, and id-attribute we know that their value must be unique. For the class attribute, we know that multiple can occur.

A useful and interesting property of an element in the DOM object is the property "textContent". This property holds all the text that is used within its html element including its children elements. If we take the correct html code from Example 1, we can ask the "textContent" from the paragraph element <p> and this will result in the text: "This is a correct mark up!". If we ask the textContent from the element "<strong>" results in the text: "a correct". This property could be useful for testing values displayed on a screen for example product prices and product amounts in a webshop application.

We could also use the events that are bound to a certain html element in the DOM object. For example, a click event is by default added to the <a> tag. Through JavaScript, we can fire that event so it looks like a mouse clicked on that element. The same hold for changing checkboxes in forms and submitting forms, which will be explained in Section 3.1.1. For more attributes and detailed information on the DOM object and the HTML document we refer to the HTML Specification[25].

3.1.1 Forms

With web applications, it is possible to submit forms. Forms are used in many ways for many purposes. For example, forms are used to add products into a shopping cart or leaving a comment in a blog. We mention forms specific, because it contributes to functional testing of web applications. In some occasions, forms are even necessary to continue testing and discovering more pages that are only accessible by submitting a specific form.

The HTML tag <form> has certain attributes. The name attribute is used to identify a form and must not be an empty string and should have a unique name amongst the other forms that exist. The action attribute specifies a URL where the form should get the data from or sends the data to. The way a browser knows whether it should send or get data is defined by the "method" attribute. The method attribute exists of two values GET or POST. The POST method submits data to a specific data source mentioned in the action attribute. The GET method requests data from a specific data source mentioned in the action attribute.

Besides its attributes, the form tag also consist of a set containing one or more HTML elements. There are four kinds of elements that are usable to a form and are listed in Table 1. It is possible to nest those elements within a form with other html elements like the <div> tag for design purposes.

HTML element	HTML representation
Input	<input type="text" name="lastname" value="Mouse">
Textarea	<textarea name="message" rows="10" cols="30"> The cat was playing in the garden. </textarea>
Select	<select name="cars"> <option value="volvo">Volvo</option> <option value="saab">Saab</option> <option value="fiat">Fiat</option> <option value="audi">Audi</option> </select>
Button	<button name="click" type="button" onclick="alert('Hello World!')">Click Me!</button>

Table 1 – Elements used in a form, example by W3Schools [26].

Each element used in a form has some attributes that are required. The name attribute is required and should have a unique name to identify the field. The value attribute is required for the input and select elements but is allowed to be empty. When a user enters some text in the field, the entered text is set to the value attribute of that specific field. The type attribute is used for the input element to determine what purpose this field has. The html element "<input>"does not have a closing tag like other html elements but is self-closing. A few other html elements hold the same behaviour like the <img> tag. An input element can have the following types:

- Hidden
- Text
- Password
- Checkbox
- Radio
- File
- Button
- Submit
- Reset

With the introduction of HTML 5 more types have been introduced[27]. Among those are the types like “email”, “tel” (telephone) and “datepicker”. We do not cover the HTML 5 field types because those elements are not commonly used and only modern browsers have support for those elements.

The hidden input type is used to add values to the form that users do not have to see and use. It is possible to add multiple hidden fields but each field should have a unique name. Often this input type is used to add a security aspect to the form to protect it from a cross-site request forgery attack (CSRF)[28]. When a page with a form is requested, a random value is generated and set to this hidden field. When the form is submitted, the server checks the hidden value. If the value matches the generated value, the form is accepted as a genuine submission. Hidden input fields are not visible on the screen but if we search in the html code, we can find the following structure:

$$< input\ type = "hidden"\ name = "_csrfkey"\ value = " - 8625438235" >$$

The input type text is an input type that is a single line field and can hold text as a value. Commonly it is used for users to fill in their names and addresses. It can hold any text a user writes, but when a form is submitted, validation of the field value can be done on the back-end side of the application. This validation is based on the validation properties specified in a model used in the web application see section 4.4.2 for more on these application models. If the value does not match against a certain set of rules, the form with the data is send back and the web page displays errors concerning the fields that are filled in wrongly.

The password type can hold any text a user types in. It looks almost the same as the text type but only the text that is written is displayed in dots. This is done to hide the real password that is entered.

The radio type is used to allow the user to choose between a set of answers. It represents a choice between a set of values. An example of how a radio button is used is by a question: “what kind of food do you like to most?”. The user can choose between three options, apple, banana, and pineapple. Only the value of the radio button that is checked is sent to the server. All the options where the user can choose between have the same name attribute but he value attribute differs. This way we know based on the value which option is selected.

The checkbox type is used to allow the user to choose. It represents a choice between zero or more elements. Each checkbox that belongs to the set of choices should have the same name attribute and should have different values in their value attribute. If none of the options is checked, an empty value is submitted on submission. If one or more options are checked those are all sent in a list.

A special type is the file type. This type allows you to choose a file from your computer and upload this to the server. It requires the form element to have its method argument set to POST and the additional “enctype” attribute set to the value “multipart/form-data”. In the back-end of the web application, we should deal with this multipart form to receive both the form data and the chosen file. Due to implementation issues, we cannot test this kind of forms that contain ‘file’ inputs. In API we use to address a headless browser would not attach the selected file to this input element. This is know bug to PhantomJS.

The last input type is the button type. The input type is used for buttons in a form. Often, when it is clicked, it activates a JavaScript function. The "value" attribute in this type of input element is not used to send stored information to the server but the content of the value attribute is displayed as text in the button. Often those buttons are used to help users automate things. For example, they can be used to check/uncheck all the option for a specific question. There are two more types of buttons to choose from besides the normal button type. Those are the reset and submit type. Their names already specifies their purpose within a form. The reset type resets the whole form to their default values, which is often an empty value. The submit type is a button that submits the form to the specified url in the action attribute of the form.

3.2 Software testing

Now that we have seen what web applications and forms are, we focus on the testing part of software. Testing is a growing aspect in the development of software. Nowadays more effort is spent in testing software. By testing software, we minimize the amount of defects that can occur in the software. Defects occur in software because humans make mistakes. Based on the testing results we can conclude if the reliability and quality of the software is increased. However, this is not a guarantee that the software is error free. With testing, not every possibility is tested and therefore it is still possible that errors can occur. The reason why not every possibility is tested is that the amount of possibilities to test are with huge software immense and not reachable to test within the time that is given.

Before we test software, we need to know what testing means, otherwise we do not know what we are doing. Within the field of testing varying definitions of testing exists. We find the following definitions:

"Testing is a technical operation that consists of the determination of one or more characteristics of a given product, process, or service according to a specified procedure."

Definition 1 – Testing definition by ISO[29].

"Testing is the process of executing a program with the intent of finding errors."

Definition 2 – Testing definition by Myers[30].

"Software testing is a technical process performed by executing/experimenting with a product, in a controlled environment, following a specified procedure, with the intent of measuring one or more characteristics/quality of the software product by demonstrating the deviation of the actual status of the product from the required status/specification."

Definition 3 – Testing definition by Tretmans[31].

We prefer Definition 3 over the others because testing is not only about finding errors and determining what characteristics belong to the software. It is also about comparing and showing deviations to those characteristics. We use Definition 3 to all references to the word testing. In our definition of testing, we find the word quality. Quality defined by the ISO 9126-1 standard is:

The totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs

Definition 4 – Quality definition by the ISO 9126-1 standard.

According to Definition 4 quality is about a set of characteristics that is important to the product. Software quality can be categorised into six characteristics each with their own sub characteristics mentioned in the ISO 9126-1 standard:

- Functionality – suitability, security, accuracy
- Reliability – maturity, fault tolerance, recoverability
- Usability – operability, understandability, learnability, attractiveness
- Efficiency – time behaviour, resource utilisation
- Maintainability – analysability, changeability, stability, testability
- Portability – adaptability, installability, replaceability

So testing software is about measuring the quality of the software. Testing software can be performed in different ways. Each sort of test is based on one or more software characteristics. There exists many different ways to test software. In Table 2, we give a short overview of what sorts of testing there exist but there are many more than listed. The testing approaches we mention in Table 2 are applied on many different levels of testing. Some are testing a small part of a system (Unit-testing) where other are testing a complete system (System testing). For a more detailed explanation of each sort of testing can be found in[30].

Sorts of testing			
Functional	Acceptance	Duration	Performance
Interoperability	Unit	Black-box	White-box
Regression	Reliability	Usability	Portability
Security	Compliance	Recovery	Integration
Factory	Robustness	Stress	Conformance
Developer	Third-Party	Production	Module
System	Alpha	Beta	Specification

Table 2 – Sorts of testing, used in the course Testing Techniques at Radboud University.

There are three ways to perform tests. One of those ways is manual testing. Manual testing is labour intensive but very flexible. However, it is not the preferred way to test software. The opposite of manual testing is automated testing. With automated testing, we create testing scripts and execute those automatically. In this approach, less effort is spent in performing the tests, but the creation of those scripts still requires enough time. A drawback with testing is regression testing, executing test scripts while the code is changed. Often the scripts will not work anymore. A different approach to test software is model-based testing. We will explain model-based testing in section 3.3.

3.3 Model-based testing

Model-based testing is a relatively new technology to test software. A model that describes the desired behaviour of the System Under Test (SUT) is the key point in model-based testing. The desired behaviour is often specified in the specification document of the SUT. Model-based testing goes beyond automated testing because it algorithmically generates specified amounts of test cases based on the model of the desired behaviour. A model-based testing tool (Section 3.4) generates the test cases based on the model.

The SUT is tested through a black-box approach. Black-box testing means that we only observe the output of the system considering a certain input, without knowing the code behind it. The observed output from the SUT is compared to the expected output of the system given by the model. The opposite of black-box testing is white-box testing where the internal structure of the system, i.e. the code, is the foundation of testing[32]. To be able to test the SUT with the generated test cases an adapter is often required. An adapter translates the generate input from the model-based testing tool to useable input for the SUT and the observable output form the SUT to the readable output for the tool.

3.3.1 Modelling languages

Testing (web) application with a model-based testing approach requires us to use a model. A model represents the correct behaviour of our system. The specification of the system, a documentation about what the system is capable of doing, specifies what the system does when certain elements are used and what reaction the system should give on those used elements. For example, the specification tell us what should happen if a user clicks on a link in a browser. The system should respond with the requested file or should provide an error. All in all the specification tells us what is required to trigger the event and what to expect as output of that event. With these known input and output, we can make a model of the specification.

With a model, we can generate algorithmically test cases to verify if the SUT is behaving accordingly. A model can be described in different ways each with their own distinction. Mind that each model-based testing tool uses a different model to generate test cases. In the following sub sections, we highlight the most common modelling languages used in model-based testing. Besides the modelling languages mentions in the next sub sections others exists like PetriNets[33], StateCharts[34] and Abstract State Machines[35].

3.3.2 Labelled Transition System

Labelled Transition Systems (LTS) consist of a set of states and a set of transitions between those states. There is one state that is the initial state called s_0. Each transition is labelled by an action. States represent the state of the system and labels represent the observable actions of the system. Labels are taken from a global set L. Formally, an LTS is a quadruple

$$(S, s_0, L, \rightarrow)$$

where S is a set of non-empty states and L the set of input labels. We also have a special label τ that is not an element of all the labels $\tau \notin L$ and stands for an internal action. The transition relation $\rightarrow$ is a subset of the product from the set of states, labels and output states [32]:

$$\rightarrow \subseteq S \times (L \cup \{\tau\}) \times S, with\ \tau \notin L$$

We write $s \xrightarrow{u} t$ if there is a transition labelled u from state s to state t, i.e. $(s, u, t) \epsilon \rightarrow$[36]. States that cannot do an internal action are called stable, whereas states that cannot do an output or internal action are called quiescence δ. One of the well-known theories within the scope of LTS is the input/output conformance relation, also known as **ioco** [37]. A visual representation of an LTS is shown in Figure 1.

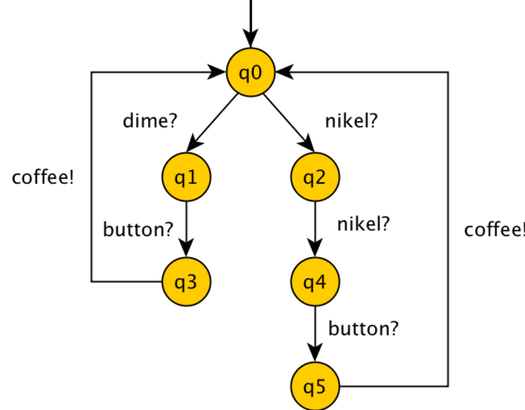


Figure 1 – A visual representation of a labelled transition system.

Equivalence between two LTSs can be reached by determining if their traces are the same. Trace equivalence is defined as follow:

$$I =_{tr} S \quad holds\ iff\ Traces(I) = Traces(S)$$

Meaning that the two LTSs I and S are trace equivalent if the traces of the two LTSs are the same. Trace equivalence is used in testing purposes. Traces of an LTS are defined as:

$$traces(p) \equiv \{\sigma \in L^* \mid p \overset{\sigma}{\Rightarrow}\}$$

A trace is a sequence of observable actions $p \overset{\sigma}{\Longrightarrow} q$ where $\sigma \in L \cup \{\tau\}$. The trace σ can contain more than one observable action. All the possible traces of process q in Figure 1 are:

$traces(q) = \{\epsilon, dime, dime \cdot button, dime \cdot button \cdot coffee, nikel, nikel \cdot nikel, nikel \cdot nikel \cdot button, nikel \cdot nikel \cdot button \cdot coffee\}$

Some states the system can be $\boldsymbol{after}$ a trace are defined as[32]:

$$p\ \boldsymbol{after}\ \sigma \equiv \{p' \mid p \overset{\sigma}{\Longrightarrow} p'\}$$

An LTS p is deterministic if $\forall \sigma \in L^*, p\ \boldsymbol{after}\ \sigma$ has at most one element. The LTS shown in Figure 1 is deterministic.

3.3.3 Finite State Machine

Finite State Machine (FSM) has a finite set of states. Formally a deterministic FSM is defined as a sextuple $(S, I, O, s_0, \delta, \lambda)$ where [38],

- ⇒ S is a finite non-empty set of states;
- ⇒ $s_0 \in S$ is the initial state;
- ⇒ I is a finite non-empty set of inputs;
- ⇒ O is a finite non-empty set of outputs and includes $\emptyset$, the null output;
- ⇒ δ is the state transition function, $\delta: S \times I \rightarrow S$
- ⇒ λ is the output function, $\lambda: S \times I \rightarrow O$

The main difference between an LTS and an FSM is that an FSM has alternating input and output labels whereas an LTS can have input and output occurring in a random order. Furthermore, an LTS is allowed to have infinite set of states and an infinite set of labels, whereas an FSM must have a finite set of states and a finite set of input alphabet. A Finite State Machine responds on a given input i given in a state s. This produces a state transition $\delta(s, i)$ and the output $\lambda(s, i)$. A visual representation of an FSM is given in Figure 2, where the input is shown before the / and afterwards the given output. The empty output is shown as ∅.

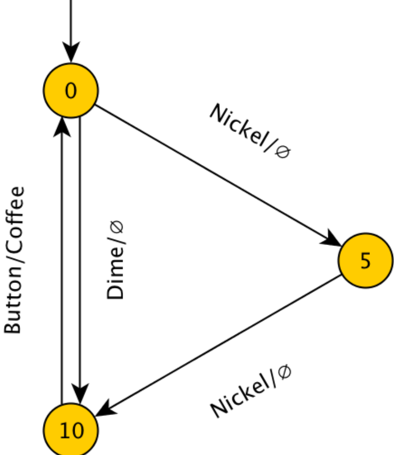


Figure 2 – A visual representation of a finite state machine.

It is possible for an FSM to be non-deterministic. This requires some changes to the formalisation of a deterministic FSM. Instead of the two functions, the state transition and the output function we only need one behaviour function[39]. This behaviour function is defined as $h: S \times I \rightarrow P(S \times O)$. In the behaviour function $P(S \times O)$ is the set of all subsets of the set $S \times O$, because it is possible to have multiple outputs and new states with the current state and input action.

3.3.4 Extended State Machine

Extended State machines (ESM) have similarities to FSMs but are extended with variables. The modelling language ESM is used to model a system in the model-based testing tool G∀ST, see section 3.4.4. During the transition from the current state to another, new values can be assigned to these variables. Using these variables, we can make an abstraction to our model by diminishing the use of states. With these variables we can build predicates that can be used as a guard for certain transitions. Guards affect the behaviour of the state machine by enabling transitions only when the condition holds. Strictly, an ESM is formalized as

$$(S, s_0, I, O, D, \delta_r)$$

where S is a set of states and s_0 represents the initial state and $s_0 \epsilon S$. The symbol I represents the Input alphabet and O the Output alphabet[1]. So far, an ESM looks formally the same as an FSM. However, it is allowed for the set of states, input, and output to be infinite. As mentioned we want to make use of extra variables. To be able to use variables we have to parameterize the state to store these variables, otherwise we cannot use the variables to build predicates.

Using variables we introduce domain D, which is denoted as the space of all possible values of these variables. Because of the additional variables, we have to change the transition relation (δ_r). Elements in the δ_r are a tuple of (s, i, p, a, o, t) and can be read as "in a state s with input i under the condition that p holds, we update the variables by the actions a, given output o an go to state t". A transition can be visually represented by a labelled arrow from one state to the other by:

$$s \xrightarrow{i,p\ /a,o} t.$$

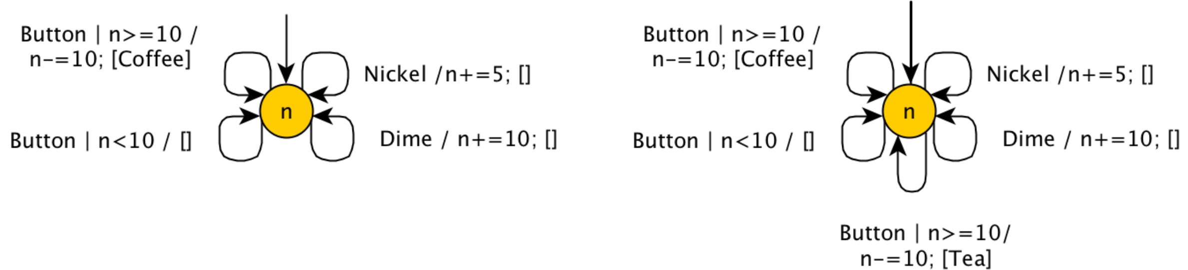
We allow omitting trivial components. This applies to the predicate $p = True$, the identity action $a = id$, and an empty output. Our transition relation is now formalized as a set of these tuples:

$$\delta_r \subseteq S \times I \times P(D) \times [O] \times A(D) \times S$$

In the transition relation, the output is denoted as $[O]$ meaning that we have a sequence of elements of type O. In the transition relation $P(D)$ denotes all the first order predicates over the domain $I \times D$, which includes the input alphabet as parameter and $A(D)$ denotes all possible actions to update those variables. It is not obligated to use predicates and update actions in every transition.

For an ESM to be deterministic we need the output and new state to be uniquely determined by the current state and input. However, we have to take into account the use of predicates as well. It is possible that we have more than one transition with input i leaving state s. Let $\delta_{r,j} = (s, i, p_j, a_j, o_j, s_j)$ with $j \epsilon \underline{n}$ denote these j transitions. The variable values X_k for which predicate p_k is true need to be mutely disjoint, hence $X_j \cap X_k = \emptyset$; $\forall j \neq k$ and $j, k \epsilon \underline{n}$.

It is not required to have a deterministic model, because this modelling language also supports non-determinism. For a model to be non-deterministic there are $s \in S$ and $i \in I$ with more than one tuple in the δ_r. We have to update the transition function because otherwise the transitions and outputs are unrelated to each other: $S \times I \to P(S \times O)$



Example 2 - Deterministic coffee machine (left) and non-deterministic coffee machine (right).

An example to show the difference between non-determinism and determinism is shown in Example 2. In Example 2, predicates are shown after the | mark. Update actions are shown directly after the / mark, before the output list, and closed with a semicolon. In Example 2 the left coffee machine is deterministic and the right coffee machine non-deterministic. The difference is the additional transition $Button/[Tea]$ in the right coffee machine. This makes the right coffee machine producing either coffee or tea. It is unknown what to expect when with the input Button it is either Tea or Coffee. Because when n is 10 and input Button is used the output will be $\{(n = 0, [Coffee]), (n = 0, [Tea])\}$.

3.4 Model-based testing tools

In this section, we discuss several model-based testing (MBT) tools. There are different kinds of model-based testing tools each depending on the kind of model that is used[32]. Not only is the type of model important but also the quality aspects that are tested, the level of formality, accessibility, and observability of the system being tested relates to the choice of tool. The tools we discuss use the models explained in section 3.3.

3.4.1 Overview of tools

During our search of model-based testing tools, we found a collection of tools (Table 3). We made a selection based on the following conditions to reduce our collection:

- ⇒ Still in development
- ⇒ Supports on-the-fly generation (Section 3.4.4)
- ⇒ Provides useful documentation
- ⇒ Open source or commercial

The MBT tools mentioned in Table 3 contain several free tools like Spec Explorer[40] and TestComposer[41]. Spec Explorer, developed by Microsoft is written for testing reactive, object-oriented systems. Spec Explorer has the possibility to test offline and on-the-fly. Offline testing means that tests are generated and executed against the SUT, whereas "on-the-fly" testing, test generation and –execution are interleaved. A great feature of the tool is that it supports non-deterministic behaviours, e.g. messages received in different orders than published[35]. Tools like JTorX[42], TorXakis, and SpecExplorer can deal with the **ioco** relation mentioned in section 3.3.2.

AETG	**Graphwalker**	RT-Tester	TestGen (INT)
Agatha	Gotcha	SaMsTaG	**TestOptimal**
Agedis	JTorX	SeppMed MBTsuite	TGV
All4Tec MaTeLo	**NModel**	Smartesting CertifyIt	Tigris
Autolink	OSMO	**Spec Explorer**	TorX
Axini Test Manager	ParTeG	Statemate	**TorXakis**
Conformiq Qtronic	Phact/The Kit	STG	T-Vec
Cooper	QuickCheck	TestComposer	Tveda
G∀ST	Reactis	TestGen (Stirling)	Uppaal-Cover
			Uppaal-Tron

Table 3 – Overview of MBT tools.

3.4.2 Graphwalker

Graphwalker is a model-based testing tool built in JAVA. Graphwalker has support for generating tests online and offline[43]. Graphwalker supports both FSM and EFSM as type of models. Graphwalker has four options to walk through a model:

- ⇒ Random, also known as the Drunkard's walk
- ⇒ Quick random, tries the shortest path but in a fast fashion way
- ⇒ A* algorithm[44]
- ⇒ Shortest all paths, the cost of every edge is set to 1.

The quick random walk is defined as follows[43]:

- ⇒ Choose an edge not yet visited by random.
- ⇒ Select the shortest path to that edge using Dijkstra's algorithm[45]
- ⇒ Walk that path, and mark all the edges that are being executed as visited.
- ⇒ When reaching the selected edge in step 1, start all over, repeating steps 1-> 4.
- ⇒ The algorithm works well on very large models, and generates reasonably short sequences. The downside is when used in conjunction with EFSM. The algorithm can choose a path that is blocked by a guard.

Based on one of the four options, Graphwalker walks through the model to generate tests. The purpose of Graphwalker is explained by the name itself. The tool walks through a graph, based on a certain algorithm. The tool is still in development and recently version 2 has been released. Based on our own findings the tool is very basic and requires time to implement. On each transition, if wanted also on each state, we have to determine if the SUT is still behaving according to the model. Therefore creating a test environment that needs little adjustments to test a different web application requires many changes.

It also turned out that the documentation was not accurate and still lacking information. After some contact with the owners, we found out that more functions can be used but were not documented. We have chosen not to use this tool with testing web applications based the amount of implementation work, the brief experience, and troubles we faced in a shout amount of time.

3.4.3 TorXakis

TorXakis is a model-based testing that makes uses of LTS models. TorXakis is developed by ESI-TNO and is still in development. The tool is based on the model-based testing tool TorX[42], extending it with symbolic test generation capabilities. Test generation and execution are performed on-the-fly. TorXakis is written in Haskell[46], a functional programming language and uses its own logical syntax to create models. TorXakis is used as the model-based-testing tool in a recent project to test Electronic Passport[47].

3.4.4 G∀ST

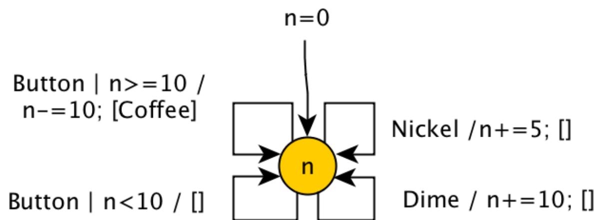
Generic Automatic Software Test-system shortly G∀ST[48], is a model-based testing tool and uses models modelled in the languages FSM (Section 3.3.3) and ESM (Section 3.3.4) to generate test cases. G∀ST is written in the function programming language named CLEAN[49]. There are two ways to use G∀ST, which is property based or state based. We are interested in state based testing. G∀ST has the possibility to test software that is written in different programming languages by communicating over sockets. It is often not possible to communicate directly between G∀ST and the SUT, therefore an adapter is developed. The adapter is needed to translate the generated test output to readable input data for the SUT. It also translates the output, given by SUT as a reaction on the input, to readable input data for G∀ST. G∀ST has the possibility to generate test output “on-the-fly”, meaning that test generation, -execution and –analysis are alternatingly performed, only as far as needed, and that no explicit test case is generated[50].

With state based testing we test based on conformance. A SUT is conform the specification modelled in G∀ST if the observed transitions are part of the specification or the specification does not specify anything for this combination of state and input: $\delta_r(s,i) = \emptyset$. Conformance in G∀ST is formalized as:

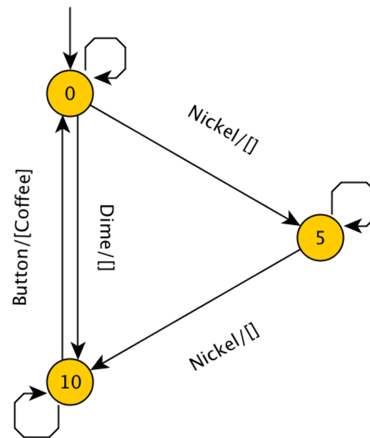
$$\text{SUT } conf \text{ SPEC} \equiv \ \forall_\sigma \in traces_{spec}\ (s_0). \forall_i \in init\big(s_0\ after_{spec}\ \sigma\big)\forall_o \in [O].$$
$$(t_0 after_{sut}\sigma) \overset{i/o}{\longrightarrow} \Rightarrow (s_0 after_{spec}\sigma) \overset{i/o}{\longrightarrow} [51]$$

The observed output o should be allowed by each input i in the $init$ after each trace σ. A trace σ is a sequence of inputs and associated outputs for a given state s. If a model is non-deterministic multiple outputs are allowed, As said it is possible that the specification does not specify anything for a certain combination of state s and input i, this empty trace refers to its own state: $s \overset{\epsilon}{\Rightarrow} s$. It is possible to combine a trace $s \overset{\sigma}{\Rightarrow} t$ and a transition $t \overset{i/o}{\longrightarrow} u$ to the trace $s \overset{\sigma;\, i/o}{\Longrightarrow} t$. The inputs that are allowed in the current state s are $init(s) \equiv \ \{i|\exists o. s \ \overset{i/o}{\longrightarrow}\}$. The states that are available after applying the trace σ are: $s\ after\ \sigma \ \equiv \ \left\{t \middle| s \overset{\sigma}{\Rightarrow} t\right\}$. There is a possibility to have infinite many traces and unbounded long trace, only if the state machine contains a loop. Having a transition from state s that refers to itself $s \overset{i/o}{\longrightarrow}$ s is already enough to cause a loop[51].

In the definition s_0 is the initial state of the specification and t_0 is the initial state of the SUT. The actual states of the SUT are never used because we only compare inputs and outputs in this conformance definition. We use black-box testing and there can only compare inputs and outputs, we do not know the internal state of the SUT. However, input and output of the SUT and the model (specification) should be the same according a specification.



Example 3 – c1, coffee machine 1[1].



Example 4 – c2, coffee machine 2[1].

In Example 3 and Example 4 we have two ESM models of a coffee machine. We use $c1$ asa specification and $c2$ as a SUT, we can do this because both examples are modelled as an ESM. The unlabelled transitions in the examples are applicable to any input that are not explicitly specified and produces the empty output. On the first looks, both models should behave the same. According the to the trace [{Dime,[ ]}, {Dime,[ ]}] both the specification and SUT are behaving the same. The same hold for the trace [{Nickel,[ ]}, {Nickel,[ ]}, {Button,[Coffee]}], which produces in both models the same output [Coffee]. However, if we look further we can find an input sequence where $c2$ is not a correct implementation of the specification.

For the trace [{Dime,[ ]}, {Dime,[ ]}, {Button,[Coffee]}, Button] the specification $c1$ produces Coffee for the second time, whereas the SUT $c2$ produces an empty output. We can say that the SUT c2 is not conform to the specification $c1$.

G∀ST uses a generic algorithm to generate input data. This generated list of input data is infinitely long and lazy evaluation is introduced to generate only a fraction of this list that is actually needed[50].

G∀ST uses pseudo random number to make slightly different permutations in the generated list, so that not the same input is constantly chosen. G∀ST checks whether this generate input is possible to use in the current state of the specification. If not, the input is rejected and continues to test with the following input action in the list. We reduce the generated list even further so G∀ST only generates input actions that do make sense in the current state of the model, so we have less rejected input to generate. We do not want to generate random links that do not make sense at all.

From the generated input, in a certain state, G∀ST observes the output from the SUT. If the observed output matches the expected output from the model, the trace is accepted and the state changes accordingly. Each input sequence is started from the initial state. If the end of a sequence is reached, a reset function is required to restore to the initial state. If non-conformance is detected, i.e. observed outputs do not match the expected outputs, the test is aborted and a counter example is given.

3.5 Choice of modelling language and model-based testing tool

In the previous sections, we discussed several modelling languages and model-based testing tools. For our test architecture we need a modelling language to model a web application. We have chosen for an ESM as a modelling language. The reason why we have chosen an ESM is due to its capabilities. It is possible to model non-deterministic and deterministic systems and an ESM can make use of variables to define predicates that can be used in guards. It possible to use variables with an LTS only some changes has to be made with the LTS we defined in section 3.3.2. However, it is not possible to use variables with an FSM.

We expect from testing a web application that we always receive an output after each input, because if a request is made to the web application a response should be send back. In an FSM and ESM, the input and output are a pair in every transition, whereas in an LTS we have to specify the input and output as a separate transition.

Another difference with the modelling languages is the amount of states, input, and output. An FSM should have finite set of states, input, and output whereas in an ESM an infinite set is allowed. An LTS should have a countable finite set of labels.

If we want to use variables to store information in the state, we can leave out an FSM as a modelling language. This leaves us with the choice between an LTS and an ESM. There main reason why we prefer an ESM above an LTS is because of the model-based testing tools each modelling languages supports. An LTS is mainly used as an modelling language in the model-based testing tool (J)Torx and its successor Torxakis. Torxakis is not yet mature enough to use in this thesis research. The tool contains some unwanted behaviour and uses his own language to describe an LTS model.

Using an ESM as a modelling language leaves us with the choice of multiple model-based testing tools. We described in section 3.4.2 the model-based testing tool Graphwalker. This tool can use an ESM as a modelling language. However, the tool itself is not useful enough based on the implementation work that needs to be done and its approach it uses to test the model.

The tool TestOptimal is also suitable to use an ESM as a modelling language[52]. We did not discussed this tool in section 3.4, only mentioned it as a possible model-based testing tool. We did not do any detail research about this tool, because we focussed more on Graphwalker and G∀ST. This tool is a commercial tool and is not our preferred model-based testing tool to use. We prefer open source model-based testing tools.

One of the tools that is free to use is the model-based testing tool G∀ST. We are familiar with this tool because we used this testing tool in a project exercise in the course Testing Techniques given at the Radboud University of Nijmegen. Based on this familiarity we have chosen to use this model-based testing tool. We know the language, CLEAN where we have to write our model. Comparing this G∀ST with other testing tools we focussed on, this is most mature tool and the best option with an ESM as modelling language.

4 Test architecture

In this section, we discuss the architecture of the test environment we use. We will enlighten all aspects of the architecture, stating the purpose of each item in the architecture and their communication between each item. In section 4.1, we will give an overview of the flow how current web applications in our setting are used. In section 4.2, we will give an overview of the test architecture we are going to use. In the following sections, we will discuss the parts of the architecture separately, where in section 4.3 we will discuss the Content Management System, followed by section 4.4 where the System Under Test will be discussed. In the final section 4.5, we will discuss the adapter that is used in the test architecture for communication purposes.

4.1 Overview of the current situation

It is important to have a test architecture. It describes what each component does and its communication to other components it is connected to. To setup a test architecture it is important to know what components there are if the system is used in real-life. In Figure 3, we give an overview of systems that are currently involved when using a web application.

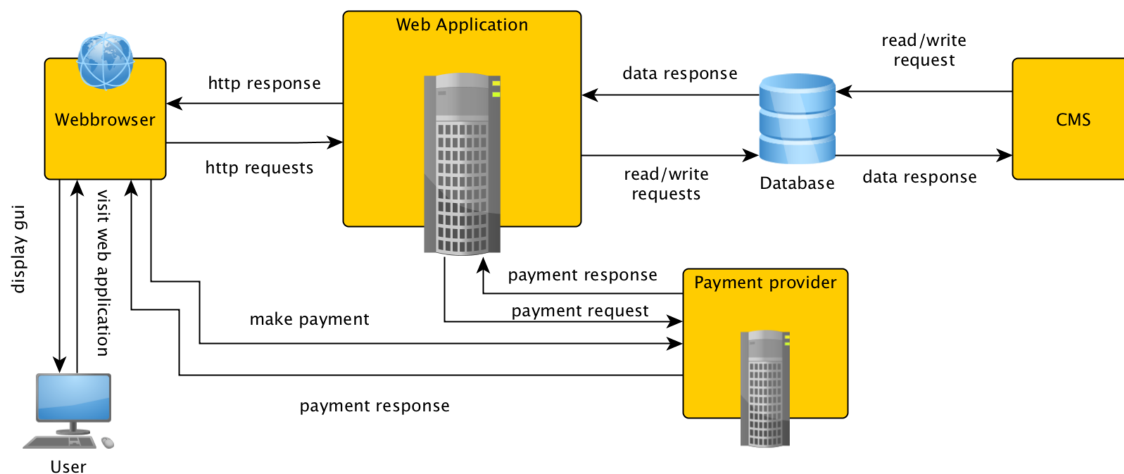


Figure 3 – Overview of systems involved.

The normal flow to use the application is that a user visits the web application by a browser on his/her computer. The browser tries to make a connection with the web application by sending a request. The web application receives this request and based on the URL it tries to retrieve data from the database. How the web application retrieves the correct data based on its URL will be explained in section 4.4. Based on this data, the web application creates a complete HTML document and send this document back to the web browser it received the request form.

Creating a new web page or adding images to a web page is done through the Content Management System (CMS). The CMS communicates with the same database the web application uses. Changes made by the owner to the application are almost instantly online. The CMS will not be part of the test architecture but is needed for owners to maintain and update their web application. For this reason, we will explain the CMS shortly in section 4.3. The web application requests data from the database, but it can also write data to the database. The latter is not often used, but in situations where the application is used as a blog, it is common. Users can post comments on a blog item. This comment is written to the database.

In Figure 3, we also mention the payment provider, which is needed to pay for the products a user wants to buy in this webshop. If the user clicks on the buy button, he is redirected to the payment provider. This payment provider does not belong to the web application and is an external company that delivers this service to the webshop. Users can easily pay with this provider and this payment provider almost instantly transfers the money to the account of the owner of the webshop. As already said users are redirected from the webshop to the payment provider, they are leaving the webshop application. When the payment is successful or aborted, users are directed back to a page of the webshop. The webshop first requests to the payment provider the status of the transaction of the user we bought the items. This status request allows to webshop to know if the payment is completed successfully or not and can corresponds with a message accordingly.

4.2 Overview of the test architecture

Now that we have seen how currently users are using the web(shop) application, we need to make a test environment to test a web application according to a user's approach. In Figure 4, an overview of a test architecture we are using is given, each block is explained in the following paragraphs.

The Web Application in Figure 4 is the SUT. The SUT responds on the HTTP requests received from a browser. The received request will first be handled by the routing engine of the web application (Section 4.4.1). All the data that belongs to a page, that is requested, will be retrieved from the database and is surrounded with HTML code in the corresponding view (Section 4.4.3).

We need a web browser to open the SUT and a tool that can receive commands and can execute these on a web browser. The latter is required otherwise, we cannot extract information we want to test a web application.

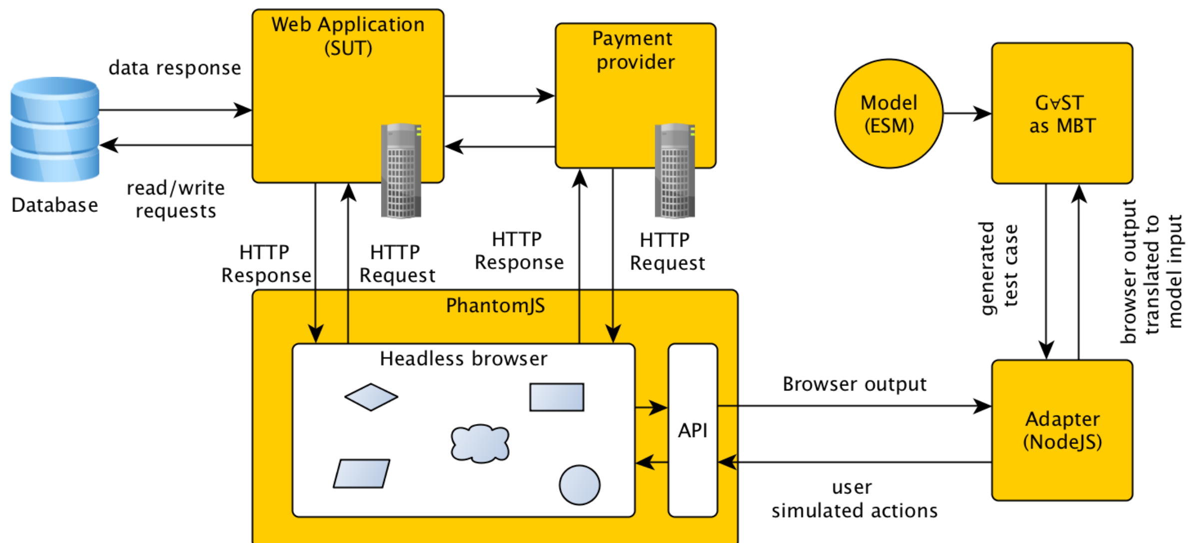


Figure 4 – Overview of the test architecture we use.

We found a couple of tools where, for us, Selenium[53] and PhantomJS[54] came out best. The tool Selenium has support for multiple browsers. Selenium is a tool that automated web browsers. It can open a browser on a computer and access all its functionality. The functionality of a browser can be addressed by the API of this tool. An advantage of this tool is that we can actually see what happens while testing a web application, because the browser is visible on a screen. Selenium has support to test simultaneously on multiple browsers. Due to previous experience and difficulties we faced with Selenium, we have chosen not to use this tool. In the course Testing Techniques given at the Radboud University of Nijmegen, we tried to use this tool. We faced many difficulties trying to run Selenium and run tests on a browser. This reason kept us from starting with Selenium before trying the other tool we found, which is PhantomJS.

PhantomJS is a headless browser, meaning that they removed the Graphical User Interface (GUI), but all functionality of the browser remains. PhantomJS uses the Webkit engine that is also used in the familiar web browser Safari. PhantomJS provides a JavaScript API to address this headless browser. By using the API, we can open web pages and retrieve information on the page. One drawback could be that we do not have a real-time visual image of what is happening. The headless browser within PhantomJS, allows making a rendering of the web application at the moment it is requested. This gives us the capability of capturing an image of the web application on moments it is desired. One of those moments could be when the web application returns an error. This way we have a visual representation of the moment the error is occurring.

PhantomJS communicates with a web application the same way as done through a normal web browser. It requests the web page and receives an HTML file of the page back. This page includes several links to other files that are needed to render the web page. Those files are Style sheet-, JavaScript-, Font-, Image files and more. Each file is requested likewise the main file.

To connect PhantomJS with a model-based testing tool we require an adapter, there is no way we could directly communicate with both tools because each tool has its own language. The preferred way to communicate is with an adapter using sockets. PhantomJS does not have the ability to communicate with sockets to an MBT like G∀ST or to an adapter. Because of the lack of socket communication, we searched for a different tool that has support for sockets and could easily use the API from PhantomJS. We found a tool, named NodeJS[55], also built on a JavaScript language but is able to communicate with sockets and could use the API from PhantomJS.

The purpose of the adapter is to serve as a server, this way the adapter keeps running, and multiple clients can connect to this server. It possible to add observers as a client to see what is happening, without interfering the MBT tool. We created a client script to see if we were able communicate by sockets and test if we could open a web application and click on a link. The adapter is used for translating generated test cases by the model-based testing tool to useable input action to the headless browser and vice versa. More about the adapter is explained in section 4.5. The MBT tool used in this case is G∀ST and makes use of an ESM as a model.

4.3 Content Management System

Today's web applications, think about informational applications but also webshops, are commonly provided with a Content Management Systems (CMS). It allows owners to easily add, change, or remove content on the application. The CMS we use is called "MyNotion" and is developed by the company Bluenotion. MyNotion is designed to collaborate with the web applications we test. The CMS delivers functionality to manage websites and webshops.

When owners of the application create a new page, they can choose between predefined page types. Each predefined page type looks different. During the development of the web application, for each predefined page type a view is created. Each page type has a predefined number of text fields and album images that belong to a view defined in the web application, which is explained in section 4.4.3. By doing this, owners are restricted to the set of fields and albums that are allowed but guarantees that owners can almost do nothing wrong to dysfunction the web application.

Of course, there are still possibilities to dysfunction the web application. However, this will probably be due to bad programming. Examples like empty text fields or image albums that are not captured and causes application errors.

4.4 The System Under Test

In this section, we will discuss the relevant parts of the SUT. We will discuss these parts because they are relevant in understanding what happens at the back-end of the applications we test. We suggest reading this section only if there is no knowledge in how web applications works, especially web applications built with the MVC principle. It also explains where errors that can be found during testing are located in the code.

4.4.1 Routing Engine

When a web request to the application is made, for example visiting the following link: http://www.bluenotion.nl, the request first gets to the routing engine of the application. The routing part is necessary to deliver the right content back to the browser. The routing engine tries to parse to URL and redirects the request to the right controller so the right content is served. The controller determines the appropriate action method to handle the request. It is possible to capture certain URL's and specify where they redirect too. The Routing engine reads those special cases from top to bottom and tries to find the first one it matches with [56]. If no match is found the default routing setting is used.

http://www.bluenotion.nl/blog/index/12

Example 5 - An example of a URL applicable in the application.

URL's are built in different parts. The first part of a URL is the domain name. We use Example 5 as an illustration. The domain name in Example 5 is www.bluenotion.nl. The second part of the URL is "blog". This second part always corresponds with the name of a controller. The controller will be explained in section 4.4.4. The third part in the example is "index" and refers to the name of the action method in the controller. The last part of the URL is the parameter given to the action in the controller. In this case, the parameter is an integer with a value of 12.

Nevertheless, it is possible to deviate from the default routing settings. We can take a blog as an example. Each blog item has an id and a title for the application. It is preferable to have nice user- and search engine friendly URLs like:

http://www.bluenotion.nl/blog/first-item-of-the-website-001

Example 6 - URL for blogs

We still want the URL to redirect to the controller "blog", with action "index" and an id as integer. With the URL mentioned in Example 6, the controller will be reached with the default settings but the action 'index' will not be reached. In this situation, we should create a new routing case that matches the URL we want. With Regular Expressions, it is possible to create a routing case. Example 7 shows a routing case that should match the URL mentioned in Example 6 and matches new blog items with different ids and titles.

```
routes.MapRoute(
            "Blog", // Route name
            "blog/{title}-{id}", // URL with parameters
            new { controller = "blog", action = "index",
                  id = UrlParameter.Optional }, // Parameter defaults
            new { controller = @"[^\.]*", action = @"[^\.]*" }
        );
```

Example 7 – Routing case for a blog item.

In Figure 5 a flow of the routing engine is given. As mentioned after the url is parsed the routing engine tries to find a matching route as shown in Example 7. If no matching route is found a HTTP 404 error is returned.

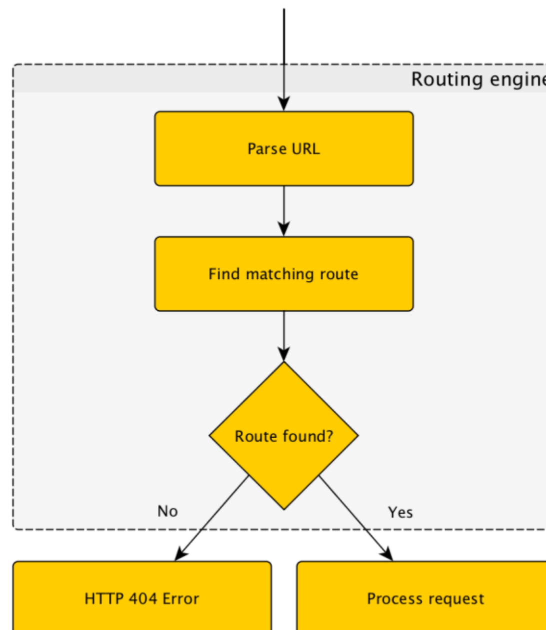


Figure 5 – Flow of a routing engine.

4.4.2 Application models

Models also occur in a .NET applications. They differ from the term used within the field of model-based testing. We use the term application models for models used in the .NET Web application. Application models basically describes the business logic, data access login, and validation logic [57].

Normally application models should contain data access logic but this is commonly separated into a service layer. In this way, we keep the models and controllers clean of excessive database calls. All these calls are managed in one service layer to keep a better overview. All the applications we host have the same database calls and are managed in this layer. An additional advantage is that we only have to update the service layer to fix a bug instead of writing correct code into each application.

Bluenotion also separated the data access login into a service layer. For each type of application Bluenotion hosts, website or webshop it has a default set of application models. Some of the application models used in a website are also applicable to a webshop because a webshop is an extension of a website. A good example of an application model used in both applications is the Page model. This page model holds data about the page, concerning textual information that is displayed on page, images and more. The application models describe what data they should contain and are retrieved through the service layer, previously discussed. It is possible to apply validation on these sort of models. Each data attribute that is mentioned can have certain validation or restriction rules. Rules like the type of variable, for example a string, integer or boolean can be applied. Other things are also possible like the length or range of the value, if it is required, or are allowed to be empty. All these kind of validation types can be added by annotating the attribute with the correct annotation.

Application models are not always used to retrieve data from the database and display it in the application but also to represent forms in an application like a login or contact form. In this way, it restricts the user what he is allowed to fill in. This is based on the restrictions mentioned in the model. These type of models are called ViewModels because they behave like an application model but are only used in Views [58]. The validation after submitting the form should always be performed server-side, but with these restrictions mentioned in the model, it is also possible to validate this on the client-side.

Client validation is feasible because the HTML generated from the model is annotated with those restrictions. With some JavaScript code or a library, it is possible to validate the form before it is submitted to the server. This reduces the amount of submission to the server. Validation on the server-side is done automatically only we have to catch if the validation is wrong. If a submitted form is wrong, the data of the form is returned and an error message for each wrong input field is shown accordingly. The user will see what he submitted and an error message for each field summarized or displayed next to its belonging field.

4.4.3 Views

In this section, we discuss how views and its data are constructed. Views represent what should be displayed on a browser. We discuss with some details how views are constructed, because views represent parts of the HMTL document that is sent to the browser.

Views are used to display data from an application model to users on a screen. It has to know the structure of the model to display the data correctly. Many views can display the same model in many different ways. The structure of a view is a combination of CSS, HTML, razor code, and other web rendering code. Razor is a programming language based on C# and is used in the views to display the data in the application models and other defined variables[59]. It contains normal programming statements like for-loop, if-else and while. Razor uses the '@' symbol for all its statements. Razor code in a View can be used in three different ways:

- Single statement block @{ var myMessage = "Hello";}
- Multi-statement block @{…var myMessage = "hello"; var name = "John";…}
- Inline statement @{<p> Hi @name, this is my message: @myMessage</p>}

Without using the '@' symbol, the code written above will be shown as plain text. It is possible to use partial views within views as well. This is relevant when we have the same piece of code used in multiple views. It is better to separate this part because it is less error prone and better for maintainability. A different aspect of .NET MVC is the ability to define sections. It allows us to add HTML, CSS, or JavaScript to a section defined in a parent view. We can define a section named 'JavaScript' in the layout view. Each view uses the layout views and holds most of the HTML code that is similar in each view. It is now possible to uses the defined section 'JavaScript' in other views. In the section, we can enter html code, text and include files. In this way, we can include only CSS and JavaScript files that are specific to this view. By using sections to load different files it reduces the loading time of the application page, because not all the JavaScript and CSS files have to be loaded at once.

4.4.4 Controllers

Controllers are the centre point of the MVC principle [60]. It connects the application models and the views together. Controllers have a similar representation like a class defined in an Object Oriented Programming language, like Java. The class name is the controller name. Each controller has a constructor and multiple methods. In the controller, we can define and assign variables that are useful in every method defined in the controller, this way they are only assigned once.

As explained in section 4.4.1 the routing maps a certain URL to methods defined in different controllers. A method is like a function and could have several parameters as input. Each method has its own return type, except for private functions. There are different return types but the most used are ActionResult and JsonResult. ActionResult always returns a view that is assigned to the method whereas JsonResult returns a JSON object. JSON[61] is often used in Ajax[62] calls made in JavaScript.

Bluenotion made a service layer between the controller and the database as explained in section 4.4.2. The designated services are initialised in the constructor of the controller [63]. The services are therefore available in each method defined in the controller.

If a defined method has to possibility the receive a submitted form, the method should have a ViewModel with the proper data as in input parameter. The submitted form is parsed into the ViewModel. With this ViewModel we can check if a form contains no errors or empty fields. With a simple if statement it is possible to check if the model is valid or not. The application automatically keeps track of the state of the model. Therefore, the attribute "ModelState.IsValid" tells if the submitted form corresponds with all the requirements that are obligated within the ViewModel.

Not all the controllers in the application exist as a physical controller. The application is dynamic and owners are allowed to create their own pages. To provide this dynamic behaviour a default controller is introduced. The default controller is used when other defined controllers are not applicable. The default controller checks if a page in the database exists that matches the URL from the routing. If not a 404 error will be displayed. If a page exists, the default controller uses the view that is bound to the page. This way the owner can create multiple pages.

4.5 The adapter

To communicate between an SUT and an MBT tool an adapter is often required. An adapter translates the generated test cases, from the MBT tool, to useable input for the SUT. It also translates the observable output from the SUT to input for the MBT tool. By using an adapter to make this translate, it should also be easier to replace the MBT tool with a different testing tool. Often communication is done through a socket connection whereas the MBT tool serves as a client and the adapter as a server. This way we only have to start the adapter once and the MBT tool when we want to start testing. It is also possible in this way to allow multiple connections to the adapter, by sending broadcast messages from the adapter. This way we can add an observer to see how the SUT is responding.

To build an adapter it is important to know what kind of messages or information will be send on the socket. To know this we have to know how to interpret a webpage. We know how we can retrieve information about a web page by accessing its DOM object. This way we can search for certain elements we need. We know that links to other pages have an "<a>" html tag. That tag always contains a "href" attribute that tells us where we go if we click on that link. We can search for all those href attributes to know what links the page contains. For testing forms we already know from section 3.1.1 that a form can be identified through the attributes "action", "class" and "id" and the elements within a form can be identified through their name attribute.

As we already mentioned, we communicate information to and from the adapter. We model a web page in such way that we extract only the information we are interested in and send the relevant information to the MBT tool. Therefore, the adapter should have some intelligence. Meaning that the adapter is responsible for extracting the necessary information. The headless browser provides a JavaScript API and therefore it is convenient to develop the adapter in JavaScript as well because we can directly address the PhantomJS API. Fortunately JavaScript provides support for socket communication and therefore we can communicate with our MBT-tool G∀ST.

For G∀ST it is only important to know if the web page is reachable or not and the relevant information, such as the set of internal links, current URL, visible forms, errors and visible products. The messages that G∀ST receives from the adapter are listed in Table 4. The top four rows in the table are only used in the small model and its extension defined in section 6. The complete table is applicable to the webshop model defined in section 7.

In the column "Message to G∀ST" in Table 4 the formatted strings are listed. Those formatted strings are needed to allow G∀ST to parse those strings to the correct output data type. The meaning of "#" within brackets is the length of the string next to it. This way G∀ST knows what the length of the string/description is. Each output typed will be send separately, but G∀ST requires that it should receive every output type. If no internal links, errors, forms or products exist the output message is therefore still send for the specific type, only with an empty list, for example "Errors [ ]".

Output	Message to GAST	Description
Id	"Id [#]string"	Url of the page that is currently open.
[Url]	Urls [Url [#]string, Url [#]string2]	List of internal links on the current page.
[Error]	Errors [Error [#]description, Error [#]description2]	List of errors that occurred during loading of the current page.
[Form]	Forms [[#]formId, [#]formId2]	List of forms that exists on the current page, found by their identifying attribute as its name.
[Products]	Products [(_Tuple3 id, amount, price), (_Tuple3 id2, amount2, price2)]	List of products that exists on the current page, represented as its product-id, amount, and price.
Amount	Amount #	Amount of products in the shopping cart.
Transport	Transport #	Current transport method that is selected.
Logged	Account Boolean	If the user is logged in or not.
Cart	Cart [(_Tuple3 id, amount, price), (_Tuple3 id2, amount2, price2)]	Products that are added in the shopping cart.
Payment	Payment	Current payment method that is selected.

Table 4 – Messages send from the adapter to G∀ST.

For the adapter it is important to receive useable information, because it has to know what to do. In Table 5, we list the messages that are sent from G∀ST to the adapter. The top five rows are only applicable to the small model and its extension defined in section 6, whereas the complete table is applicable to the webshop model defined in section 7. Each label holds one or more parameters. Each parameter is used in the message send to the adapter. The message that is sent is formatted in a JSON object but sent like a string. The adapter parses the received message into a JSON object. We prefer using a JSON object above multiple strings for the reason that it is readable and easier to use in the adapter. Because the web applications we test have fixed patterns, we can make a distinction in testing a category-, product-, information-, or form page. Each page has certain distinctive and overlapping elements to test.

Label	Message	Description
Visit website	{**type**:"start", **website**: website}	Open web application
Open url	{**type**:"click", **url**: url }	Click on requested link
SubmitEmpty formId	{**type**:"submit", **id**:2, **form**: formId	Submit form with empty values
SubmitFalse formId	{**type**:"submit", **id**:0, **form**: formId**}**	Submit form with false values
SubmitTrue formId	{**type**:"submit", **id**:1, **form**: formId }	Submit form with true values
Addproduct (productId, value)	{**type**:"addproduct", **id**: productId, **price**: value}	Add product to shopping cart
ChangeTransport	{**type**:"changeTransport", **id**: transportId}	Changes current transport method
ChangeAmount	{**type**:"changeAmount", **id**: productId, **price**: value}	Changes product amount by +1
ChangePayment	{**type**:"changePayment", **id**: paymentId}	Change current payment method
Delete	{**type**:"deleteProduct"}	Deletes first product in shopping cart
GotoStep2	{**type**:"gotoStep2"}	Goes register/login part of the checkout
GotoStep3	{**type**:"gotoStep3"}	Goes to payment options page
GotoStep4	{**type**:"gotoStep4"}	Goes to overview page of checkout
GotoPayment	{**type**:"gotoPayment"}	Goes to the desired payment provider
GotoStep4b	{**type**:"gotoStep4b"}	Goes to payment placed page
GotoStepDone	{**type**:"gotoStepDone"}	Goes to payment successful page

Table 5 – Messages send from G∀ST to the adapter.

5 Behaviour of web applications

In this section, we will discuss the behaviour of web applications that companies find important to test. In section 5.1, we discuss what is common to test with web applications. Section 5.2 discusses the testing ground we find important, partially based on the common testing ground.

5.1 Common testing ground

Web applications are available in many forms. Each application has its own purpose and behaves in a different way. However, at the lowest level of interaction almost each application behaves the same. This behaviour is based on the request and response principle. A browser requests certain information and the server responds back with an answer. It is one-directional communication. Almost all applications are built on this one-directional communication, but there are exceptions. There are applications that are bi-directional. Those web applications make use of websockets to establish a bi-directional communication. A nice example of how a bi-directional communication is implemented is the web application Shootr[1]. Bi-directional web applications takes a different approach to test and is therefore beyond our scope of this research.

So back to our request and respond applications. As we explained in section 4.5 when a webpage is requested an HTML object is received from the server. This HTML object can contain several links to other files. Those files are also requested and probably received. After receiving all files, the DOM object is initialized and ready to use. With having this DOM Object we can start testing.

There still remains a question which is what is important when it comes to testing web applications? In section 3.2, we mentioned the definition of testing and the quality characteristics that are applicable. With testing web applications, other quality characteristics are more important to measure quality [19, 64, 65]:

- Functionality
- Usability
- Human Interface
- Compatibility
- Reliability
- Performance
- Accessibility
- Security

In the next few paragraphs, we will discuss each characteristic shortly. Of course, other software characteristics still hold like maintainability. We still want a product that is maintainable and allows changes.

The quality characteristic functionality used with web applications is about mechanism used on the web application. Mechanisms like search options, fill in forms, (menu) navigation, level of scrolling (horizontal and vertical). It also contains domain specific mechanisms like product functionalities in an e-commerce system.

[1] http://shootr.signalr.net/

The usability characteristic is about understandability of the web application. This could be helping information with filling in forms, phone numbers and email addresses are correct annotated. This characteristic is hard to automate, because this is mainly about the interpretation of a user about the content of a web page. Of course, it is possible to see if certain parts are in place and function but it is almost not possible to test if it helps with understanding the contents of the page only assumptions can be made.

The characteristics interface and compatibility strongly related to each other. If we test compatibility of a web application, we mainly focus on the interface of the web application. A web application is a product that you want to show to the world. Therefore, it has to look good and no strange things should happen. Because browsers have implemented different rules for interpreting and rendering a web page, a web page could work fine in FireFox but does not look or function properly in Internet Explorer (IE) or even within older versions of the browser itself.These rules are defined in the CSS files that are included in the HTML document. The interface of the web application tells where every html element is located on a page and what design properties is contains, which are defined in a CSS file. Common web browsers with their recent version are FireFox(v38), Chrome(v42), Internet Explorer(v11), and Safari(v8). There is a huge difference in rendering a web page with IE7 or IE9 and higher. Other browsers do not differ that much with some older versions of the browser.

A different quality aspect companies also consider is mobility but relates to compatibility. Many mobile devices are used nowadays, each device with different screen sizes and other specifications. Companies want that their web application can be used on almost every mobile device everywhere on the world. However guaranteeing that the system is working on every device there is, is a struggle. Some may say this is a total different quality characteristic but we assume this mobility characteristic as a part of the compatibility characteristic.

Reliability with web application can have different options. The first option is about the uptime of the web application. Here the choice of server and its capacity is important. Preferably we want a server that can guarantee a uptime of 99,9% or higher. In addition, unexpected behaviour of the web application is also part of this characteristic.

Performance characteristic is about the speed of the web application. Here it is important to know how quick pages are loaded. Of course, it depends on the speed of the internet connection. Nevertheless reducing file sizes of media content and the HTML document could increase the loading speed of the page.

Accessibility is about how accessible is the web application. Is there any support for users that are blind or visually impaired? Certain html elements have attribute that help with telling blind people what this element is about. For example, a link element (<a>) should have a “title” attribute to tell what this link is about. With this attribute blind people know what this link is about. The same rule holds for the “alt” attribute in an image element

Security is lately also an important quality aspect to test. Criminals want to abuse web applications to gain profit of it. One way is by installing malware through advertisement that is shown on the web application. A different way is stealing credit card information by using SQL-injection. There are some standard security aspects that probably should hold for each web application, like preventing SQL-injections. However not every web application requires a secured connection by a SSL certificate.

5.2 Our testing ground

With our goal of testing web applications with a model-based testing tool, we diminish our set of quality characteristics that are stated in the previous section. Mainly the reason why we diminish the set of quality characteristics it is very complicated to implement every characteristic there is in one total package. Further to cover every characteristic that exist is too big to cover in this thesis.

We are not interested in the looks and feels of the web application. Therefore, it leaves out the interface characteristic and testing it on one browser is therefore sufficient. Our focus lies with testing the functionality of the web application. This means testing links, forms, media content, indirectly cookies, and mechanisms in forms. Indirectly we still have some overlap with interface testing. We want to test web applications through a user perspective. This means where a user clicks on we also click on, but without the interface and visual representation of the web application. We can achieve the user's actions the same way by accessing the DOM object and trigger that action. This way we do not have to rely on the user interface. However, if we cannot click on an element we also know that there could be something wrong with the interface. This way we still test our interface in some way. We also cover the interface characteristic based on testing if we have no errors in the CSS files that are included. If CSS files are missing, the interface can look different. Based on the user approach to test web application we heavily make use of the DOM object to achieve our tests.

We do not deal with compatibility. The web applications we test, built by Bluenotion, are by default not supported on older browsers and therefore not tested. If a web application is built for supporting older browsers, than these test cases should be build specific for that web application. Modern browsers are more compatible with each other and less changes are needed to behave identically.

Usability is partially covered by testing the content of the page, meaning that all files should be there and correctly loaded. Because is we miss media files we can conclude that the web page is probably not useable or readable. It is not interesting enough to build in spelling checkers to test if the written text is correctly spelled.

Performance is covered but in a different way. G∀ST, the testing tool we use, requires a timeout value. G∀ST stops testing after it did not receive any message within this timeout value. This can be translated to the response time of the website. If after a couple of seconds the web application still not responded with an answer this timeout could happen, especially if we test the web application on a local machine and the internet connection has no influence.

Accessibility is partially tested. We test if certain html attributes are added to html element it belongs therefore it should be readable enough to blind people. We test for existence of attributes like the “title” attribute in the <a> tag and the “alt” attribute in the <img> tag. We know that there are tools available that check a web application based on rules to make the web application readable for blind and visually impaired users[66]. Therefore, we do not test any further aspects of this characteristic.

Finally, we slightly cover the security characteristic as well. We test the possibility to login with fake credentials and have access to other person’s information. Furthermore, we also test if restricted pages are accessible without credentials.

While using a web application, especially a webshop, cookies can be set in a web browser by a web server and used. The purpose of cookies is mainly to identify users and server customized content. Webpages do not have any memory. A user going page to page is each time treated by the web application as a new visitor. With the use of cookies, this same user can be identified. There are basically two types of cookies that can be used: session cookies and permanent cookies.

Session cookies are temporary cookies and have a short lifespan. This means if we close the web browser that we use to visit the web application the cookies that are set are deleted. If you logout from an account the session cookies are also deleted. Some session cookies are also deleted after a certain time is elapsed. ASP.Net applications that use session cookies often have a lifespan of twenty minutes. Session cookies allow users to be recognized within a web application. A familiar example to explain session cookies is the shopping cart in any webshop application. A cookie is set to keep track of the items you added to your shopping cart. With each page visit, the server can identify users based on the id that is set in this cookie and therefor knows what each users has added to their shopping cart. The server keeps track of a local shopping cart for each users in its own memory.

Persistent cookies have a more permanent lifespan. They are stored in a folder that is assigned by the browser. Persistent cookies remain there until they are deleted manually or when they expire. The expiration date depends on the time that is set by the web application. Basically the owner of the application sets this. Nowadays this date is often set to never expire. In March 2015, the cookie law in the Netherlands is changed. The law is introduced to protect the privacy of the visitor concerning the use of cookies. With cookies, it is possible to track movements of individuals and create a profile that is used to address this user with personalized advertisements. Those cookies are named “tracking cookies” and belong to the persistent cookies. A web application is obligated to ask permission of the visitor to place these tracking cookies. It is allowed without consent to place analytical and functional cookies.

Functional cookies are cookies that are necessary to a web application to function properly, like keeping track of the items in a shopping cart mentioned above. Analytical cookies are cookies that are used for the purpose of statistics of the web application and give insight to the functioning of the web application. A good example is Google Analytics. With the Dutch cookie law, we have to make some changes to the Google Analytics code that sets the analytic cookie to guarantee the privacy of the visitors. Functional cookies belong to the session cookies. Analytical cookies belong to both types of cookies.

During testing of the web application mentioned in this thesis, cookies are enabled. This means cookies can be set and used. We do not test cookies in our test environment but we allow the web application to set cookies. Most of the cookies that are used are used for third parties, like Google Analytics and have not use for us to test with the web application. We do use the session cookies. Those session cookies are used to deal with the shopping cart items and information about the login status of a visiting user.
Each time we test a web application those cookies are set again with values generated by the web application. If we end a test those cookies are also deleted and have no further use. With our user approach, we can search for cookies that are set while testing the web application. However, we do not have any influence on those cookies, because they are set by the web application and are encoded. Because of this encoding we do not know what information is set within a cookie.

6 The small model
In this section, we will introduce a small model to see what we can test. We discuss in section 6.1 an existing way of modelling a web application. In section 6.2, we briefly recap what we are interested in to test. The small model is introduced in section 6.3 and followed by an extension of this model in section 6.4. In section 6.5, we will discuss the results of this extended model based on two use cases.

6.1 Existing model
It is possible to model a web application in a relatively simple way. This kind of model already exists and is used in some testing environments[11]. One of the weaknesses of this model is its state explosion. If the application that we test is huge in size, meaning it contains many features, pages, and elements that we want to test, the model will explode from the number of states and transitions it contains.

We can apply this approach to model the web applications we are interested in and use this model the generate test cases with an MBT tool. To achieve this kind of model we map pages in a web application one-to-one. This means that each web page represents at least one state in the model and each action that is possible on a page relates to a transition from its state to the same or another state. Actions in this case are links that refer to different pages within the application or actions that occur on the same page. To model a web application this way, it is possible to test many quality characteristics like the ones mentioned in section 5. Because each page is translated into a state, it is easier to write detailed tests scenarios for each page. Still with a huge web application and this way of modelling the model will be unreadable due to the state explosion and it requires lots of time to implement this.

To visualize this problem we take the webshop Amazon as an example. This company sells millions of products. Each product has its own page within the application that describes at least the product information, a number of related products, and a button to buy it. The minimal number of states based on this assumption will be:

$$\#\text{S} \geq \#\text{Products}$$

The web application does not only show product pages but it also contains other pages concerning the user's account and buying information. Each page contains links to other pages within the application. Amazon contains multiple menu's that reoccur in each page. Assuming that each link is a transition from the current state to another state. The number of transitions on a page will be:

$$\#\, Transitions_{page} \geq \#Links_{menu} + \#\, Links_{footer}$$

Leaving out the links that occur in middle segment of the web page, because that differs on each page. At least 50 links are showed in the menu and footer on each page. Amazon approximately sells 253 million products[2]. A simple math calculation shows that there are at least 253 million states and over 12.5 billion transitions. Giving a clear overview on this model, applied on Amazon, is hard due to the explosion of states and transitions.

[2] Number of products offered at Amazon
http://export-x.com/2014/08/14/many-products-amazon-sell-2/

There is another downside besides the state explosion when using this kind of model. The ease of reusing this type of model in different or even similar applications is hard, because we will have to work out each state and transition all over again for each application. Therefore, we want to create a model that can be used in multiple applications with the least amount of adjustments to this model.

6.2 Which parts of a web application need to be tested?

To find a model that is more abstract and better suitable than the approach mentioned in section 6.1, we first need to know what is relevant in testing web applications. As mentioned in section 5.2, we are not interested in the design of the web application, meaning that we do not test the user interface and compatibility characteristics. Important is that the application does not return any errors. Meaning that the requested page does exist and the page itself does not contain any errors. If a page does not exist a 404 error will be returned. As already said a page could contain errors itself even if the page exists. The different kinds of error types are listed in Table 6.

Error type	Explanation
Media	Image and Video files cannot be found.
Styling	Cascading Style Sheets (CSS), LESS, or SASS files cannot be found
JavaScript	JavaScript files cannot be found
JavaScript Execution	The runtime execution of the include code gives an error
Font	Font files cannot be found

Table 6 – Error types occurring on a page.

A JavaScript error is often followed by a JavaScript Execution error. The main file, which implements a certain JavaScript function, cannot be found. In one of the following JavaScript files that are loaded, this specific function is used but the application will return an error that the used function is not defined. Users do not see those errors unless they open a developers window in their browser. What users do notice is that certain functionality of the web application is not working, when you click on a button and nothing happens. In the worst case, when certain errors are occurring the web application can look different but could also get stuck due to an infinite loop.

We are not only interested in finding errors on a page but also on (successfully) submitting forms on a webpage. Forms are used not only for sending information to the owner of the application e.g. a contact form, but also for storing information on the web application e.g. a registration form. Finally, forms are also used for giving a user access to certain parts of the application e.g. a login form. We are interested in how forms react to different values used to fill in these fields. Each field allows different values to pass. We can think of the validation of an email address, zip code, or phone number. Testing forms are an important key to achieve full test coverage of the web application under test. Login forms, when authorized, give access to hidden sections of the application that could otherwise not be tested. Thus, it is required to successfully submit forms unless we want to test applications partially. Now that we know where we are interested in we can start making a definition of our small model.

6.3 The small model

If we want to test our web application, we need a suitable model. As already mentioned in section 3.3.1 many kinds of modelling languages exists. Labelled transitions systems (LTS), Finite state machine (FSM), and Extended State Machine (ESM) are commonly used languages. Each modelling language is used in a different kind of testing tool each with its own purpose as explained in section 3.4. Based on the choice we made in section3.5, we have chosen for an ESM to model our system and G∀ST as a model-based testing tool to generate test. ESMs are distinguished from Finite State Machines by the addition of variables.

The first step in making a model that should be able to test a full web application is to see if it is possible to test a web application at all. We define a small test model to see if our test architecture is capable of testing several pages in a web application. The pages contain information and links to other pages of the web application. We filter out the links that refer to external web applications and links that do not belong to the pages in this test model. The reason why we filter external links is that we are not interested in testing other web applications, like Google. We filter other internal links because we restrict the model to only test the pages that are modelled.

We take the corporate website of Bluenotion[3] as an example. The website application contains more than four pages but the following pages will be used in the small model:

http://www.bluenotion.nl/
http://www.bluenotion.nl/klanten
http://www.bluenotion.nl/portfolio
http://www.bluenotion.nl/portfolio/lookinsharp

The small test model will start with the homepage as starting page, because this is most used web page and default way to open a web application. We have to keep in mind that we should also accept different pages as a starting page as well. During the introduction of search engines, the way of accessing a web page is namely changed. Now we can enter the web application on the page that probably contains the most valuable information the person is looking for.

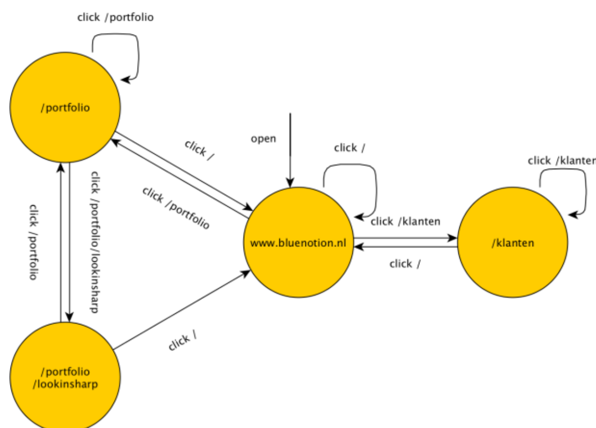


Figure 6 – Flow between the pages used in the small model.

[3] http://bluenotion.nl

In Figure 6, the relation between each page is shown. As we can see with four states, we already have ten transitions and this is only a part of the full application. Therefore, we do not want to model our application in the way explained in section 6.1. However, for testing the test architecture and its functionalities, mentioned in section 4, this small test model suffices.

As shown in Figure 6, the page "/portfolio/lookinsharp" is only accessible by visiting the "/portfolio" page whereas the homepage "www.bluenotion.nl" is accessible by all the pages. This tells us that each page in the application could refer to the homepage, meaning that it is probably an item occurring in a menu. Accordingly, we assume that each page holds at least one link referring to a different page of the application:

$$\#Page_{links} \geq 1$$

There could be a possibility that we end up in a livelock. This could happen if two pages are referring to each other and one of the pages has an incoming link from a different page as shown in Figure 7. In Figure 7, the livelock situation is caused by the cycle between state 2 and 3. There is no possibility to get out of this cycle, because there are no other links referring to other pages. Currently we allow this to happen, because we are going to run multiple tests each generating its next input action on the fly. Because of running multiple tests each generating other input sequences, we assume that we will cover and test almost every page. However, if we assume that every page has a menu containing a link to the homepage it is less likely to have livelock because a homepage should have more than one link to other pages.

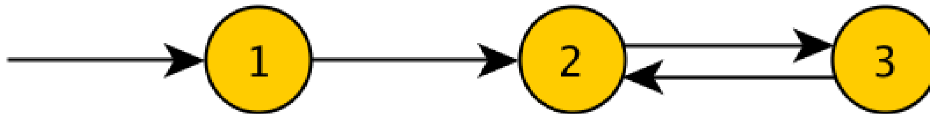


Figure 7 – Livelock situation.

With all this information, we created a model shown in Figure 8, which is both abstract and it is possible to extend it. We made this abstraction to the model based on the assumption that every web page is the same. Interpreting a web page at a low level, this assumption is true, because every web page consists of html code, JavaScript files, and media content. However, the content is not the same on every page but this is not relevant because we only test what we find on this page. If page does not contain any media files, we skip the tests that are used to test media files. The same hold if no JavaScript files are found. Based on this model we can extend the tests that we perform on every page. Therefore, it is possible to include more quality characteristics than we focus on. For example, it is possible to include a grammar checker to check the text on every page.

The model contains an initial state "INIT" and a "RUN" state. Therefore $S := \{INIT, RUN\}$ and $s_0 := INIT$.
The state "INIT" only has one input "Visit website" where the variable website is used for the website that should be opened. The "RUN" state also contains only one Input: "Open url". Our Input alphabet is now:

$$I := \{Visit\ website, Open\ url\}$$

The Output alphabet consists of a sequence of output labels always in the same order. The output labels are explained in Table 7.

$$O := \{Id, [Url_1, Url_2 \dots Url_n], [Error_1, Error_2 \dots Error_m]\}$$

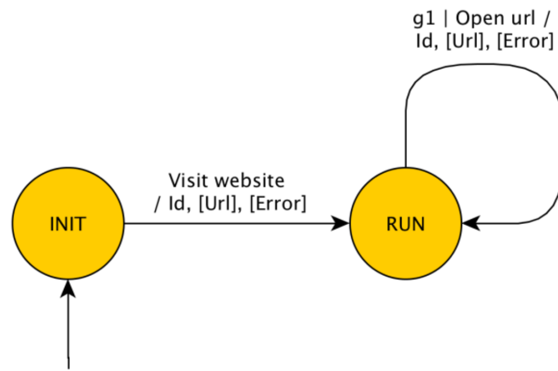


Figure 8 – Small model.

In our model, we make use of predicates. Those predicates are used to guard certain transitions. The one in particular we would like to know is, if the newly chosen url to click on exists in the current page. We do not want to open pages that do exist but are not reachable from the current page. We define the variable U for the list of URLs that exist on the current page whereas L is the set of all internal links that exists in the web application. The variable U should be a subset of the set of all URLs that exist in the web application.

$$U \subseteq L$$

The variable "url" in the input "Open" refers to the URL that the test tool should click on. There is one condition to check and that is if the url to be clicked also exists on the current page. The choice of using an ESM will be convenient now. We parameterize the state we are using. This way we can store variables in the state and use them to define predicates. We define two variables in the state. This first variable should hold the identifier of the webpage where the state currently is. This identifier is updated after every transition based on the output received from the adapter. The identifier is the value from the output label "Id", mentioned in Table 7. The second variable we define holds the urls we found on the current web page. Those urls are received from the adapter and can be found as the output label [Url]. We use the stored urls to allow G∀ST to generate the next url used in the input action Open.

Output label	Meaning
Id	Url of the opened link.
[Url]	List with Urls that occur on the opened page
[Error]	List with errors that occur on the opened page, to continue with testing this list should be empty.

Table 7 – Output labels and their meaning.

One other predicate we use is to check whether or not the output contains errors. To check this we introduce a variable E defined by

$$E := \{O_{errors} == \emptyset\}$$

If the error list contains elements, we stop the test and we show these error messages. The defined variables are each updated on every transition. We do not mention the check on the errors in the model, because this check applies for every transitions. This brings us to the actions that can update our variables, but not before we defined our Domain D. In section 3.3.4, we stated that D is a space of the domain of all state variables, therefore we need a domain that holds the variables we need, the id and urls on the webpage.

$$D := L \times P(L)$$

We have stated that L is the set of all existing internal links. The “id” variable holds a value in the set L and the urls of a page is a subset of L therefore the possible set of values is a subset of $L \times L$. Now that we defined our domain, only the actions that update our variables are all that is left. We only need two actions, one the initialize the variables (INIT) and one action to update all the variables (UPDATE). In Table 8, we summarized the action and their usage.

Actions	Usage
INIT	Initialize all variables
UPDATE	Update all variables on each page visit

Table 8 – Actions and their usage.

In the “init” state of the model, we initialize all the variables and after each transition, we update all the variables based on the output of the adapter.

The model shown in Figure 8 suits itself to be implemented in a model-based testing tool named G∀ST. The model is suitable for running tests exhaustive, meaning that we can run the tests until we have seen all possibilities. Of course, after a certain amount of test cases we hope that each page of the web application is visited at least once. The model allows pages to be visited more than once. This makes sure we have the ability to visit every page that is linked to the web application. Only with one condition, which is that, each page holds a links that refers to the homepage of the application. With having this ability in our model we can reach pages that are only reachable through one specific page. An example is shown in Figure 6 where the page “portfolio/lookinsharp” is only accessible by the page “portfolio”.

We have successfully implemented the model mentioned in Figure 8 in the model-based testing tool G∀ST and used the adapter described in section 4.5 for the communication between G∀ST and the web application. We used the cooperate website of Bluenotion as a use case to test our model. We found several JavaScript errors and page errors but we will discuss these findings in section 6.5.2 where we test the same use case only with a small extension to this model.

6.4 The form extension

Now that we created a base for testing web pages in a web application, we can extend our model such that it is capable of testing a full webshop application. The first step in reaching this goal is extending the model with the ability to test forms. Forms exist on a webshop application in the shape of a registration form, login form, newsletter form and contact form. A registration form creates an account on the webshop. This is required to purchase products. A login form is needed for allowing revisiting users to buy other products. This also grants access to different pages that are not visible when the user is not logged in, i.e. order overview, order details and account information.

There are many ways to test forms. In our user approach to test web applications, we are restricted in our choice to test forms. This restriction is because we cannot verify what data is sent to the server. We can only wait for the response back from the web application. We have chosen for an approach in which we are able to test all fields, each with their own restrictions. We are testing each form three times each time with other values to each field.

The first test shows if fields that are required, are behaving accordingly. Required fields have the property of not being empty on submission of the form. This means that when an empty form is submitted it should result in error messages for each field that is required. We know if this field is required or not based on its annotation in the html code.

The second test checks the ability of how fields responds to incorrect values. Certain fields are restricted by accepting only certain values. For example a form could contain an email field. Obviously, the value of this field should hold a valid email address, but we can all make mistakes and enter a wrong value in that specific field. Therefore when clicking on the submit button, the form should give an error saying that the field should hold a valid email address. In this test, we deliberately enter wrong values for each field according their type and kind of value it should hold.

The final test checks the submission of a correctly filled in form. Beforehand we know for each field what kind of values it should accept based on the documentation of the web application. We choose a subset of these values and enter for each field with their correct value. If the submission results in an error, we know that something is wrong. If the submission was correct, we expect that the form is removed from the page and a success message is shown. We have chosen that forms should be removed after a successful submission, because it is not common to submit the form again after a successful submission. However, after seeing the success message and we reload the page, the form should be visible, and allowed to be filled in again.

There is one thing we should address and that is the possibility to fill in a form in many ways. There are a million ways to define an email address, a phone number, and all the other typed fields that exist. There are even more possibilities if we test a combination of values for each field. However, we have chosen to test only these three types of testing a form and not to generate million possibilities to test a form. Because on a form submission with one faulty field, should already return the form and provide an error message. Why no test every field the same time. This is why we have chosen to test form three different times instead of testing millions of combinations.

To include testing of forms in the model, we have to extend the model discussed in section 6.3. As we already said, we are testing forms in three different ways namely empty-, false- and correct values. After each submission, we test if the outcome corresponds to the model. Therefore, we represent each submission of a form as a state and we test all three cases of a form submission after each other.

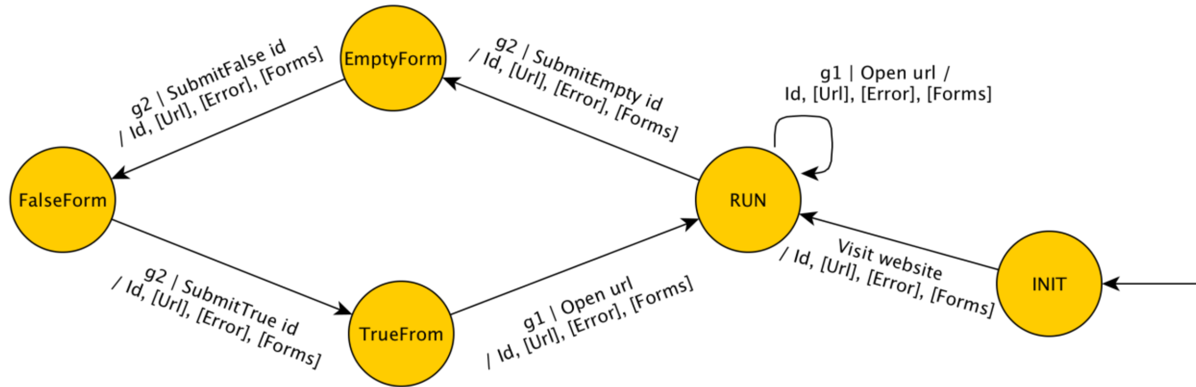


Figure 9 – Form extended model.

The extended model is shown in Figure 9. The model deals with submitting forms and visiting other pages. In the "RUN" state it is now possible to submit an empty form. G∀ST is implemented in such a way that if the page contains a form it can choose to test the form but can also continue to test other pages that are accessible through this page. This way we allow G∀ST to choose the path randomly based on the on-the-fly testing method.

With testing the forms, we keep track of the available forms on the page each time we test a form. This way, we make sure that the form we currently are testing still exists. It could happen, during bad programming, that submitting an empty form is accepted and therefore a success message appears. As already said we keep track of the forms available on each page and therefore should notice if this submission was accepted. We added a guard to check if the form we test still exist. The guards used in this model are listed in Table 9. Due to this form extension, our Input and Output alphabet is changed. Our Input alphabet is now:

$$I := \{Visit\ website, Open\ url, SubmitEmpty\ id, SubmitFalse\ id, SubmitTrue\ id\}$$

Moreover, the Output alphabet, which is explained in Table 10 is now:

$$O := \{Id, [Url_1, Url_2 \ldots Url_n], [Error_1, Error_2 \ldots Error_m], [Form_1, Form_2 \ldots Form_m]\}$$

The update actions defined in section 6.3 stay the same, however the amount of variables that are updated by these actions are increased. We additionally store the forms that we find on a webpage in the parameterized state. This way G∀ST can determine if it should open a link or test a form if one exists. If one or more forms exits on the page, there are two input actions available where G∀ST can choose between.

#	Guards	Meaning
G1	isMember(url, urls)	Check if the ur lot be clicked exists on the page
G2	isMember(form, forms)	Check if the tested form is present on the page.

Table 9 – Guards used in the form extended model.

Output label	Meaning
Id	Url of the opened link.
[Url]	List with Urls that occur on the opened page
[Error]	List with errors that occur on the opened page, to continue with testing this list should be empty.
[Form]	List of forms that occur on the opened page, each form has a unique identifier.

Table 10 – Output labels for the form extended model.

6.5 Results

We use the form extension model to test two use cases, both developed by Bluenotion. In the first use case, we take the web application from the company LookinSharp as SUT. The second use case is the web application from the company Bluenotion. Both applications have forms and enough pages to test. The LookinSharp use case holds 38 active pages in their CMS system, whereas Bluenotion application is a bit bigger containing 144 active pages.

Bluenotion has a tool running that registers errors triggered by visitors on a web application. It provides details about the error and on what page the error occurred. Those errors are detected while the application is live. This way of detecting is not idyllically, because we want to prevent those faults instead of detecting them afterwards. Besides the tool does only detect page faults i.e. page not found or an internal server error. Detecting missing media files and JavaScript faults are not in the scope of this tool.

In our findings we compare the results of our testing tool against those detected by the tool currently running at Bluenotion and the tool Crawljax[67]. Crawljax is a tool for automated crawling of web applications combined with testing modern web applications[68]. Crawljax is not a model-based testing tool but tests web applications based on an event-driven crawling engine. With the default settings of the tool, Crawljax tests pages on broken links, images, (JavaScript) events, and forms. It is possible to configure Crawljax to exclude certain elements for testing. Furthermore, it is possible to add plugins to extend the functionality of the testing tool. We mainly use Crawljax to test the coverage of our tool. We will use the default web interface with the default settings of the tool and using fixed input values for submitting forms. The approach we use to test forms is not comparable to Crawljax's approach.

6.5.1 Use case: Lookinsharp

Testing the web application from Lookinsharp (www.lookinsharp.nl) resulted in not finding any page errors with both tools. In addition, we did not found any errors registered by the tool Bluenotion currently has running. However, we did find one event error occurring on a page. This error is about a missing function in a JavaScript plugin. This plugin requires an additional plugin to cycle images but this JavaScript plugin is not added to the HTML code. In Table 11, an overview is given of the test results gathered from both tools. Testing the web application manually resulted in finding the same errors as both tools did. The time it took to test the web application manually was about 30 minutes.

	Crawljax	Test tool
Visited URLs	29	29
Seen URLs	29	29
Reachable pages	29	29
CMS pages	38	38
Not reachable	9	9
Number of edges	92	188
Time	9 minutes (exhausted)	59 minutes (3000 test cases)
Page errors	0	0
Event errors	1	1

Table 11 – Testing results from Crawljax and our testing tool for lookinsharp.nl

Both tools find identical results as shown in Table 11. The differences between the tools are the number of edges being tested and the amount of time it takes to test. Both tools tested 29 pages and found 29 pages. We mention this found property because it could happen that our model-based testing tool do find links but based on its pseudo random choices it did not test certain links. In Figure 10, an overview is given of the states found by Crawljax.

In Table 11, we mention that there are 38 active pages within the web application. The active pages are counted in the CMS and tell us how many pages there are active and should be reachable within the web application. From the testing results, nine pages are active but cannot be addressed through the URLs that exist on the web application. This way the owner of the web application knows that not all information is accessible. Probably this page holds information that could be relevant to visitors. The owner probably forget he did not refer to this page in other pages he created or did not add a link to the menu of the application.

The time it takes to test the web application is much better with Crawljax than our testing tool. Crawljax tested all pages within 9 minutes, whereas our tool required almost an hour to test 3000 test cases and did not test every transition that exists. This difference can be explained by the approach taken to find and test web pages. Crawljax uses a shortest path algorithm to discover all the pages that exist, it achieves this by using the properties in a web browser like 'reload' and 'goback' in history[69]. The approach we take in our MBT tool is based on pseudo random choices. The pseudo random approach discovers more edges because it does not keep track of the already visited pages and the random choice to visit a certain page. However, it is time consuming, whereas Crawljax's approach is quicker, but more errors can possibly be discovered by repeatedly testing the same page as this turned out in the webshop use case we will mention in section 7.3.1.

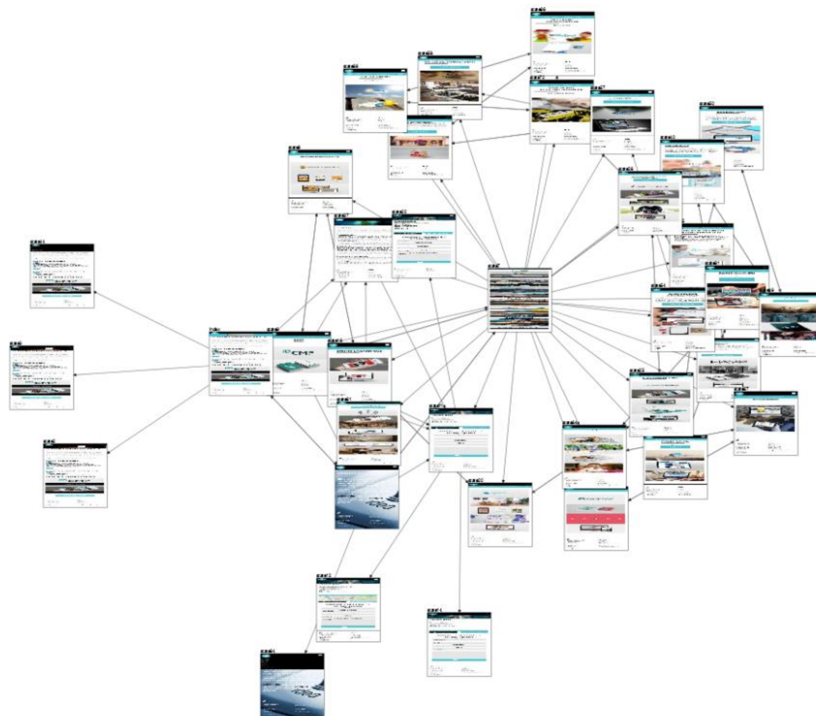


Figure 10 – State overview lookinsharp.nl, by Crawljax

6.5.2 Use case: Bluenotion

We also used our tool to test the web application from Bluenotion itself (www.bluenotion.nl). Running our model-based testing tool resulted in finding several errors. Results are shown in Table 12. The tool Bluenotion currently has running did not detect the event errors we found, however the page errors were detected but only afterwards.

	Crawljax	Test tool
Visited URLs	117	77
Seen URLs	117	118
Reachable pages	119	119
CMS pages	144	144
Not reachable	25	25
Number of edges	289	523
Time	29 minutes (exhausted)	1hour 35 minutes (5000 test cases)
Page errors	5	2
Event errors	3	5

Table 12 – Testing results from Crawljax and our testing tool for Bluenotion.nl

We compared the results gathered from our testing tool against the results found by Crawljax. From analysis of the Bluenotion application, we know that there are 119 pages reachable in the web application. In the CMS, there are 144 active pages. This means that several pages are active but not reachable through the application itself. As explained in section 6.5.1 this information is useful for the owner of the application. Testing the web application manually resulted in finding five pages errors and finding four event errors in all the pages that exists within the application. It took more than an hour to test the web application manually. The reason why we did not find the event error about the Google Maps plugin will be explained at the end of this sub section. Comparing the manual results with the results listed in Table 12 we can say that testing manually performs better than with a testing tool. We find more errors within less time. However testing a web application manually multiple times, when changes are made is not recommendable.

Based on the results we find a clear difference in the amount of pages that are visited with both the testing tools. Crawljax visited 117 pages whereas 119 are reachable. In Figure 11, the findings of Crawljax are visually shown. Page errors that are found are marked with a thick border. The difference in URLs visited by Crawljax and the URLs that are reachable is due to the interpretation of states with Crawljax, therefore it is possible that in the visual representation of Crawljax more states are shown than there are pages found. Within the use case, there are two links each having a different URL, which refer to an identical web page. This behaviour occurs twice in the use case. Crawljax interprets this as the same state because the DOM objects are the same, whereas the links are different. This way Crawljax interprets as if there are 117 pages. In our interpretation, we assume both links are individual pages and therefore we say that there are 119 reachable pages.

Our testing tool visited 77 pages whereas 118 links are seen. Due to the amount of test cases and the random choices G∀ST makes we only tested 77 pages. Our tool does see other links but due to the randomness, those were not tested (yet). G∀ST generates test cases pseudo randomly. It often chooses a page that is already tested in a previous test case. The choice G∀ST makes depends on a list of links that exists on the page. Often the next page to be tested is the first link that exists in this list.

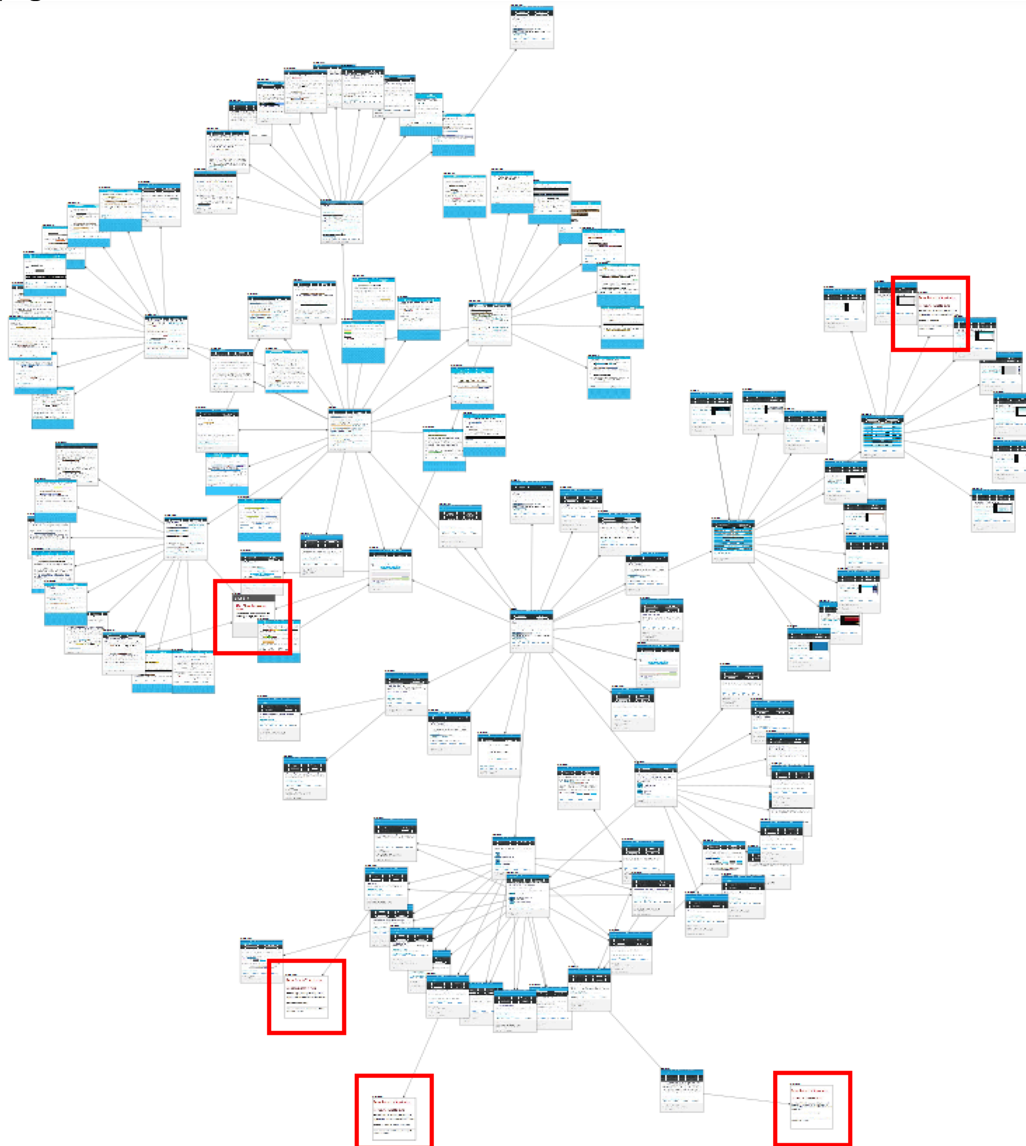


Figure 11 – State overview bluenotion.nl, by Crawljax

The list does not often change and the first few and last few links in this list are almost the same each time a page is tested. Those links are the navigation menu and footer links of the page. Because of this randomness, we test more edges, one page to another page, but we test less pages than Crawljax. Besides that, it will also take more time to test a whole application. It took an hour and a half to test 5000 test cases. We first started with 3000 test cases, however the results with 3000 test cases were not useful. Not even half of the existing pages were tested. We therefore increased this number to 5000 we better results were found.

We do however find more event errors than Crawljax. The two event errors we find in both tools are about validation of a form. The first event error is about a missing JavaScript file, whereas the second event error occurs because it is missing a JavaScript file. The next error this if found by both tools is an image file that is missing.

One of the errors we additionally found is the logging of certain data in the console window of the web browser. We do not want this behaviour and therefore we mark it as an error. Probably Crawljax is not configured to mark console messages as an error and therefor did not find this error.

The final error we found is a JavaScript error about the Google Maps plugin located at the contact page. The error we received was saying "Unable to delete property" and is caused when the Google Maps plugin is initialized. We did not see this error occurring in any browser even testing it manually. Besides the Google Maps plugin is functioning correctly. After some digging, we found why this error occurred. Recently Google updated their Google Maps API to version 3, the version that is used on the Bluenotion application is version 2. We can argue if this error is a real error because the web application and plugin are functioning properly. Our interest lies in unwanted errors and therefore we mark this as an error.

We also applied mutations to the web application. The reason why we did this is to see if deliberately made errors are found. We deliberately removed images, JavaScript files and even made changes to values in a form. All these changes are found with our testing tool.

Comparing the results from both use cases, we can say that our model-based testing tool is functioning properly, we do find nice test results even found an error that we did not know it existed. However, we did not find every error that exist in the web application. Finding all errors in the use cases is feasible but it will take much more time to eventually find those, due to the randomness of G∀ST.

7 Webshop Model

In this section, we discuss the modelling of the webshop application. This model is based on the form extension model and should be capable to test at least webshop applications build by Bluenotion. In section 7.1, we will discuss the additions that are made to a web application to make it a webshop application. The new model based on a webshop application is introduced in section 7.2 and the results based on two use cases are discussed in section 7.3.

7.1 Webshop applications

Now that we have a working model, introduced in section 6.4, to test a web application we want to extend this further to model a whole webshop application. A webshop application is more than just a web application. A webshop application allows a user to buy one or more products. This can be achieved by checking out the shopping cart of the user and following the steps that are required to buy the products successfully.

Besides buying products, there is also the ability to create an account during the checkout phase. This allows the user to come back and buy more products. It is possible to login based on the credentials that are entered when registering the account during the first time products are bought. When logged in, it is possible to check the order history of the account. It shows what the user has bought on each order and the total amount spent on each order. Furthermore, it also allows the user to change his contact and shipping addresses. While logged in and buying products, all the information that is required to fill in are already filled in based on the information the user provided while registering the account. This makes it easier for the user to continue the checkout phase because he only has to check if the information is still correct. This information is only available when the person has a registered account and is logged in. Changing contact information and showing the order history of the user is only available if the user is logged in. Therefore it is required to test if those pages can be opened or not and if they show the correct information.

In certain situations, it is possible to reach the pages that are only reachable by the login action, even if the pages are hidden. One way of doing this, is just by copying the link into a different browser and see what happens. The web application should not show the user's credentials of his account or from any other user, because the user is not logged in. Instead, it should show the user a message that he is required to login. We want to test these kind of security aspects.

7.1.1 Products

Besides the checkout phase, a webshop application also contains product pages. The product page looks the same for each product only the product information differs. A product page differs from other pages because it contains actions that allows the user to put the product in his shopping cart. Therefore, we allow the model to choose, if the page is a product page, to add the product to the shopping cart or continue testing other pages.

For adding a product to the shopping cart, it is important to know that the product is added. Somewhere on each page, usually on top right of the page, a count is shown of how many products there are in the shopping cart. This means that when a product is added this number should be increased by one. Because we know what product we add to the shopping cart, we can test if the product price that is shown on the page corresponds to what is entered in the database. If the prices do not match, we know something is wrong. A good explanation could be that the prices are shown without VAT instead of included with VAT. We keep track within our model what products are added to the shopping cart including their price. This is important because we need this information to test the checkout phase.

7.1.2 Checkout phase

The checkout phase of a webshop application consist of five steps. In the first step an overview of the added products in the shopping cart is given. Within this page, it is possible to change the amount for each product and to remove a product from the shopping cart. Furthermore, the user can give a comment about the order and he can choose the desired shipping method. Options for the shipping methods are pick-up or delivery with several shipping couriers, like DHL or PostNL.

In the first step, we test if the products in the overview matches the products that we keep track of in our model. In Table 13, we give an example how this overview looks like.

Product	Amount	Price per piece	Price
X	3	€49,95	€149,85
Y	1	€99,95	€99,95
		Subtotal	€249,80
		Shipping costs	€0,00
		Total	€249,80

Table 13 – Example of a product overview in the checkout phase

Each product in the overview has an amount displayed of how many of its kind are added to the shopping cart. Next to the amount the price per piece is shown. The last column in the overview shows the user the price he has to pay for the amount he orders of that specific product. In Table 13 we pay for product "X" €49,95 per piece and we order three of its kind, so the price will be $49{,}95 * 3 = €149{,}85$.

The subtotal mentioned in the overview, is the sum of all products times their amount. Depending on the chosen shipping method there could be shipping costs charged. Beforehand it is known what the costs are of each shipping method. The chosen shipping method could affect the total price mentioned in the overview.

As already said, the user can perform different actions in the first step. The amount can be changed, but also a product can be deleted. If the amount of a product is changed this directly changes the total price of the product, the subtotal price, and the total price of the order. If a product is deleted, not concerning the amount of the product, it affects the overview as well. The number of rows is reduced by one, the subtotal, and the total price of the order is changed.

The next step in the checkout phase is about the contact information. Here the user has the ability to login with an existing account or choose the register one. If the user chooses to register an account the user has to fill in a form to where and whom the company should ship the products to. Among the contact information like first and last name, phone number and email address the shipping address is required as well. It is even possible, if wanted, to mention an aberrant delivery address. This way the invoice will be sent to the normal address but the order will be shipped to the aberrant address. If the user chooses to login with an existing account the contact information and shipping address are already filled in, based on the information entered in the previous purchase.

The third step is about the choice of payment. The default option "payment in advance" is selected. Here the user has to pay in advance before the order will be shipped. Other options that are available depend on what the owner of the application prefers. The following options are possible:

- Cash on delivery
- Credit Card
- PayPal (Worldwide)
- Bitcoin
- iDeal (Netherlands), Mister Cash (Belgium), SOFORT (Germany)

The last payment option mentions several payment providers that make it easier to transfer money to the company. Each of the payment providers are similar to each other but only differ in their country coverage. The only thing the buyer has to do is authorize the payment by his bank account. The company receives the payment within seconds and can ship the items much faster than waiting for a payment done in advance.

The fourth step contains an overview of the previous steps. It shows the user what he is buying, where the products are going to be shipped to and what kind of payment method is chosen. By clicking on the "buy" button, the user accepts the terms and conditions and the products that he is buying. Depending on the chosen payment option, the user is redirected to the payment page. If the user has chosen for a payment provider he is redirected to the payment page of this provider. By doing this the user leaves the web application and visit the page of the payment provider. After the payment is completed, the user is redirected back to the final step in the webshop application. If the user has chosen for payment in advance he is redirected to the final step but a message is shown explaining where to transfer the money to. If the user has chosen for the option "cash on delivery" the user is also redirected to the final step.

If the user has chosen for a payment provider, it is important for the web application to know that the same user returns after paying. This is where cookies play an important role. While the user is visiting the webshop application, a cookie is stored on the computer of the user. This cookie contains an identification number, which the web application uses to identify users. When the user is redirected back to the webshop application this cookie is still alive. Based on the cookie the webshop application can determine if it is the same user. If it is not the same user, meaning that the identification number in the cookie has changed, the user will be logged out.

However, the status of the payment is still updated, because the payment provider also sends a status message to the webshop application notifying it about the new status of the payment of a certain order id. The webshop knows what order id belongs to a certain user.

The final step is nothing more than a web page that says: “Thank you for purchasing”. We mentioned all the steps and their possibilities that we can use in a webshop application. With this information, we can extend the model mentioned in Section 6.4, to be capable of testing a full webshop application.

7.2 Model

Based on the previous section where we introduced the parts that are important to test when testing a webshop application we need to extend our model to cover these parts. Those parts will be enlightened with reference to the new model in Figure 12. The five steps we mentioned in section 7.1.2 each represents a state in the model. Step 1 contains several actions, each of these actions can be found back in the model as a transition.

The deleteFirstProduct transition contains a guard that this transition can only be done if the shopping cart amount is higher than zero. Because it is not possible to delete a product in the shopping cart if the shopping cart does not contain any. After the completion of this action, the new cart amount should subtracted with the amount the deleted product had. We also test if the subtotal and total price are of the shopping cart is changed accordingly.

The changeTransport transition will change the shipping method to the id that is provided. If a certain shipping method adds cost to the total price of the shopping cart this will be tested. Of course, this transition contains a guard. Because we can only change the shipping method if there are more than 1 options available.

The transition changeAmount changes the amount of the product in the shopping cart that matched the productid that is provided. It adds 1 amount to the product in the shopping cart. With this transition, we can test if the shopping cart is correctly updated with new amount. We also test if the total price of the product, subtotal and total price are updated accordingly. Otherwise we have a mismatch in the total price we have to pay if these prices are not updated.

The last transition is the transition gotoStep2. This transition should bring the application to step2 of the checkout phase. In this state, the model has three options.

- Test login form, if not logged in
- Test register form, if not logged in
- GotoStep3, if logged in

We extend testing forms with an additional step this is the submitWrong step. This step is only used when we test the login form. We want to make sure nobody can login with wrong credentials. We had to extend the amount of input actions of the model, because there are two ways to login and we have the ability to register an account. Because these are special forms, we created new input actions and not using the default approach we modelled to test forms. Those input actions refer to the same command to the adapter but the name of the form differs.

There are two ways to login, one through the account page and through the checkout phase. Each way holds a different form name and redirects to a different page therefore we created different states but both forms have the same field names.

To test the login forms we require that there already exists one account to be able to login with. The username and password to login with both login forms is stored in the adapter. If we test the register form, we change the email address every time a register form is tested successfully with true values. We change the email address otherwise we cannot register multiple accounts. If we test the register form with true values, we are automatically forwarded to step3 of the checkout phase.

Once we are in step3 of the checkout phase it is possible to choose the payment option. The changePayment transition will change the payment method to the id that is provided. Often there is an option to pay by bank or by a certain payment provider mentioned in section 7.1.2. If go to step4 and we have chosen for the payment by bank, the purchase is immediately completed but we still have to transfer the money manually. However, in a test scenario we are not going to transfer money manually, therefore the order will remain open in the overview of the order history of the user's account.

After the selection of the chosen payment provider, we are forwarded to the overview page in step 4. We can only test if the data we have chosen matches the ones described in the overview. This means testing the chosen products, shipping method and the payment provider. If everything matches, we can continue to the payment page.

If we have chosen for a payment provider in step3, we are redirected to the gateway of the payment provider to pay the money automatically. Because we test the webshop application in a test environment, we are redirected to the test payment of the chosen provider. No real information is required to pay. If we have chosen for payment by bank or cash on delivery, we are redirected to step5 we are success message should be shown. In step5, we can continue to test new pages. This completes the new shop model.

Based on the shop extension the Input alphabet is now:

$$\begin{gathered} I := \{Visit\ website, Open\ url, \\ SubmitEmpty\ id, SubmitFalse\ id, SubmitTrue\ id, SubmitWrong\ id, \\ changeAmount\ (id\ , price), changeTransport\ id, changePayment\ id, deleteFirstProduct, \\ GotoStep2, GotoStep3, GotoStep4, GotoPayment, GotoStep4b, GotoStepDone, \\ AccountEmpty, AccountFalse, AccountWrong, AccountTrue, \\ LoginEmpty, LoginFalse, LoginWrong, LoingTrue, \\ RegistreEmpty, RegisterFalse, RegisterTrue\ \} \end{gathered}$$

The Output alphabet is also extended and is explained in Table 14;

$$\begin{gathered} O := \{Id, [Url_1, Url_2 \dots Url_n], [Error_1, Error_2 \dots Error_m], [Form_1, Form_2 \dots Form_m], \\ [Product_1, Product_2 \dots Product_m], CartAmount, CurrentTransport, LoggedIn, \\ [CartProduct_1, CartProduct_2 \dots CartProduct_m], CurrentPayment\} \end{gathered}$$

In the model shown in Figure 12 contains the variable "output" as an output message. This variable is an abbreviation of the output values mentioned in Table 14. We used this abbreviation because otherwise the model is not readable if all output values are shown in every transition. We also defined guards that are required to make sure certain transitions are only allowed if the guard holds. The guards used in the model are listed in Table 15.

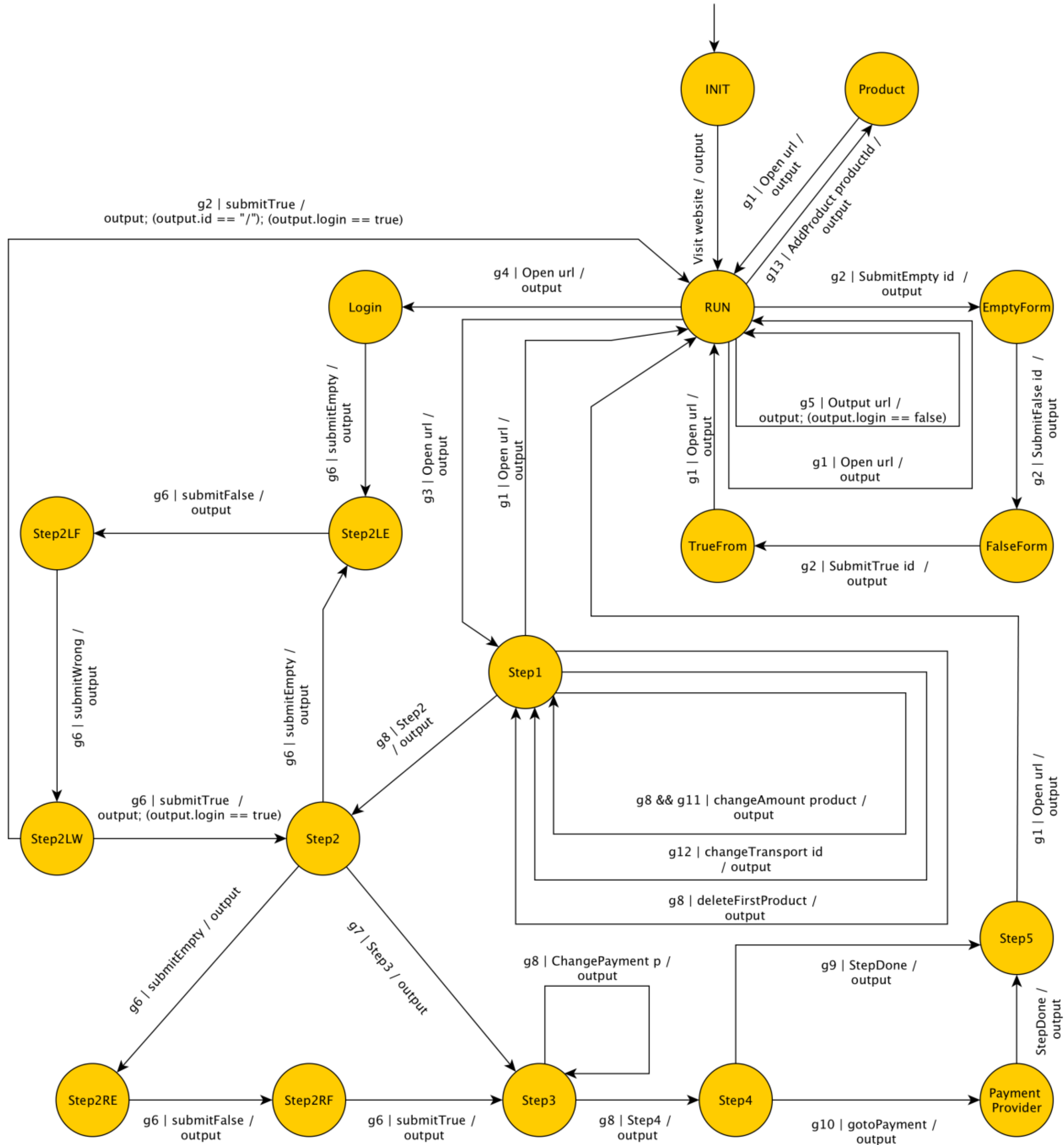


Figure 12 – The shop model

Output label	Meaning
Id	Url of the opened link.
[Url]	List with Urls that occur on the opened page.
[Error]	List with errors that occur on the opened page, to continue with testing this list should be empty.
[Form]	List of forms that occur on the opened page, each form has a unique identifier.
[Product]	List of products with their id and piece price that can be added to the shopping cart
CartAmount	Amount of items in the shopping cart.
CurrentTransport	Current transport id.
LoggedIn	Boolean value if the user is logged in or not.
Cart [Product]	List of products added to the shopping cart, each with their id, amount and piece price.
CurrentPayment	The current payment option that is selected

Table 14 – Explanation of the output labels in the shop model

#	Guards	Meaning
G1	isMember(url, urls)	Check if the ur lot be clicked exists on the page
G2	isMember(form, forms)	Check if the tested form is present on the page.
G3	url == "/cart/step1"	Url that is going to be opened equals Step 1 of checkout phase.
G4	url == "/Account/Login"	Url that is going to be opened equals login page.
G5	url == "/Account/Logout"	Url that is going to be opened equals logout page.
G6	Login == False	User is not logged in.
G7	Login == True	User is logged in.
G8	CartAmount > 0	Amount of shopping cart items is more than 0.
G9	CurrentPayment == 1	Payment method is a local option.
G10	CurrentPayment == 2	Payment method is a payment provider.
G11	isMember(product, cart)	Make sure the product we want to change the amount of exists in the shopping cart.
G12	Transport > 1	The number of transport options should be more than one. Otherwise, we do not have to change the shipping method.
G13	products > 0 && isMember(product, products)	Make sure we products exists on a page and we can add the product.

Table 15 – Guards used in the shop model.

7.3 Results

To see how the model-based testing tool performs we set up two webshops to be tested. The first web application is Matrascenter.nl and is a webshop application build by Bluenotion. The second webshop application is a temporary webshop that is setup for testing our tool. It is a complete different shop and created in PHP and build on the Wordpress engine. We have chosen for a different web application to see how generic our tool is and what modifications are necessary to our tool to be able to test a different webshop application. Our findings in testing both web applications resulted in some errors we did not oversee when testing manually.

7.3.1 Matrascenter.nl

While testing Matrascenter.nl, we found a couple of errors. The first error we found is about submitting an empty search form and resulted in an internal sever error. The error occurred because the application could not deal with empty search input. A second error we found is a missing style sheet file on the contact page of the application. The third is a JavaScript error about enlarging a photo. This error occurred on pages where it is possible to enlarge a photo. Besides the JavaScript error on those pages a missing image file also found. The fourth error we found is a page error. A certain page was not found and did not even exist. The last and probably the most important error we found is about the shopping cart. The error makes continuing the checkout phase impossible. The error occurred when buying twice after each other on the same account while logged in and shipping method is changed the first time to a non-default value. The second time the page of the shopping cart is opened the current shipping method is not selected. When continuing to the buying process an internal server error will occur based on a missing value.

We often faced timeouts or internal server errors while testing the webshop application. The timeouts happen because G∀ST did not receive any output back within several seconds after the input was sent. With a normal broadband internet connection, we assume that within ten seconds, the web application should be loaded and an output is sent back. However, the upload speed of the internet connection where the test server is located, which is a basic ADSL connection, is with 5 Mbit/s very low. This upload speed caused the timeouts we were having. A real web application is hosted in a datacentre where the internet connection is 1Gbits/s or higher. We tested if we faced the same timeouts with the web application only hosted in a datacentre. We increased the timeout value of G∀ST so G∀ST is able to receive usable output from the webshop application.

We also faced errors that are caused by the capacity of the test server. This will also cause an internal server error but reason is due to a timeout that occurred on a database call. The processing power and its memory are not realistic compared to a server located in a datacentre. The test server consist 2GB of memory and shares its processing power with other virtual machines that are running. One could say that the performance of the web application is poor because it generates timeouts and that causes internal server errors. We can find some truth in this opinion but we partially tested this web application on a live server, without the checkout phase. This resulted in no timeouts and no internal server errors.

We also compared the findings of our tool against those from Crawljax. We must state that Crawljax could not find everything and cannot test a full webshop application, due to the reason that with the default settings Crawljax does not deal with external pages like the payment provider page. An overview of states found by Crawljax is shown in Figure 13.

	Crawljax	Test tool
Visited URLs	382 (350)*	140
Seen URLs	382 (350)*	393 (333)*
Reachable pages	359	359
CMS pages	416	416
Not reachable	58	58
Number of edges	580	1357
Time	3h 51 minutes (exhausted)	1h 25 minutes (2900)
Page errors	5	6
Event errors	1	1

Table 16 – Test results from Crawljax and our testing tool for matrascenter.nl

Results from both Crawljax and our testing tool are shown in Table 16. Remarkable is that both tools found more links than that should be reachable. We found out that in the overview of the shopping cart that the product links hold a different link than how each product is originally found. These links should contain their category name in the url, instead of the text "product". We consider this as an error because this should not occur according the original implementation. In a category page, we find product links containing the name of the category, i.e. "matrassen":
http://www.matrascenter.nl/**matrassen**/polyethermatras-sg-25-soft-14-cm-80x200-5403

In the shopping cart overview we do not find this category name back in the url, instead the name "product" is found:
http://www.matrascenter.nl/**product**/polyethermatras-sg-25-soft-14-cm-80x200-5403

This is why we found 382 urls with Crawljax and 393 urls with the testing tool. The products that are added to the shopping cart are making the difference in this number. Removing those incorrect product links in the shopping cart overview, because they should be named properly, we are left with 350 urls found by Crawljax and 333 urls found by the testing tool. From this result, we can conclude that Crawljax added 32 products to the shopping cart in the time it tested the application and our testing tool added 60 products to the shopping cart.

From the random choices G∀ST makes, our testing tool only tested 140 unique pages. In those 140 pages, the tool successfully purchased a number of products several times. We had to change a few settings in Crawljax to test more pages and possible add products to a shopping cart. We allowed Crawljax to visit each page more often instead of only once. Because if Crawljax while visiting the shopping cart overview page and chooses to click on a link instead of checking out the shopping cart we are not able to test this page again. We also had to increase the maximal number of states that could be found, and we had to increase the maximal search time several times to find more pages. Because with enabling the option visiting the same page multiple time it take longer to find more pages. We ended up with a minimal search time of four hours to find useful results. Unfortunately, we were not able to purchase products successfully. One reason is that Crawljax does not deal with external links, like the payment provider page. The other reason is that it could not fill in a form that is required for contact information.

Because Crawljax does not test forms the way our implementation does, it cannot find the error the testing tool found about the shipping method. The other errors we found with the testing tool were also found by Crawljax.

Manual testing this web application took several hours. Mainly because of the size of the application. We faced the difficulty to keep in mind what pages we already tested. We overlooked one page that did not exist and the error we found in the shopping cart. Manual testing web applications of this size, we would not recommend it to anyone. It was a struggle to find out if we test every page and is time consuming.

Figure 13 – State overview from matrascenter.nl, by Crawljax

7.3.2 Wordpress Webshop application

We also tested a complete different webshop application. This web application is written in PHP instead of .NET. During testing, we found out that our current generic model shown in Figure 12 is not completely useable in this use case. The way we modelled the checkout phase, with the five steps, is not applicable to this webshop. The webshop uses less steps to buy products. Therefore, our generic model does not hold anymore. However, all the functionality we modelled, changing the shipping method, changing the product amount, deleting a product, and changing the payment option are still applicable to this webshop. We therefore have chosen to change to model according the steps this webshop requires. The new model is shown in Figure 15 and can be found at the end of this subsection. We removed the states Step3 and Step 4. We removed the possibility to login within the checkout phase and changed the changePayement transition from Step3 to Step2.

We also found out during testing of the web application that our implementation on testing forms did not work properly. The forms we test in .NET web applications are annotated. These annotations we use to test if an input field shows the correct error message. Unfortunately, the form fields are not annotated by the PHP implementation we use. We changed the interpretation of how we test form fields to deal with elements without annotations. The results from both Crawljax and our testing are shown in Table 17.

	Crawljax	Test tool
Visited URLs	18	32 (21*)
Seen URLs		32
Reachable pages	22	22
CMS pages	22	22
Not reachable	0	0
Number of edges	46	251
Time	9 minutes (exhausted)	3h 30 minutes (5000 test cases)
Page errors	0	0
Event errors	0	0

Table 17 – Test results from Crawljax and our testing tool for the Wordpress webshop application.

Both tools did not find any error while testing the webshop application. One remarkable thing is that our testing tool found and tests more pages than there really exist. The explanation for this strange behaviour is that the web application generates a different code each time a user wants to login, this code is used for security reasons but visible in the URL of the page. With our tool, it is possible to visit a page more often and the tool visited the login page frequently. After filtering out those duplicate links, which referred to the same page, the tool tested 21 unique URLs.

Based on the gathered result we found that both tools did not test the search form. We know from the testing result from G∀ST that the model-based testing tool did find this search form, but probably based on the random choice G∀ST makes it did not test the search form

We have an assumption why Crawljax did not test this search form, but we do not know this for sure. The search form does not contain a submit button and the submission is started when the users presses the enter key. Probably Crawljax does not know that the enter key starts this submission and therefore did not test the search form. Other forms like the login form, which contain a submit button are tested with Crawljax.

Figure 14 gives an overview of states found by Crawljax. Remarkable is that Crawljax did not find all existing pages. When we analysed the results of Crawljax we find out that Crawljax did not complete a purchase. We found out in the results that it stopped in a state where the web application should be redirected to the payment provider. This is possible because Crawljax only accepts internal url and does not redirect to other web applications. In theory, Crawljax should be able to complete the whole checkout phase by choosing the payment option “payment by bank”. Running Crawljax multiple times we did not completed the checkout phase in any of the test runs.

We made some changes to this web application to see if Crawljax is capable of completing the checkout phase if we only allow payments by bank. We removed the external payment provider from the possible payment options and we started testing with Crawljax again. This time we indeed completed the checkout phase of the webshop application. However we did not see the payment success page occurring. Somehow, Crawljax does not continue this transition and due this reason Crawljax will not find every page that exists.

Comparing the findings and testing results from both testing tools, we can say that our tool is performing better in testing the checkout phase of a webshop application than Crawljax does. Unfortunately, the generic modelling approach does not hold for the webshop model we introduced in section 7.2. Different webshop applications requires changes to this model. Nevertheless, we are still pleased with the results we found.

Because we did not found any errors with testing the application, we added some mutations to the web application. We deliberately added errors to the web application to see how the test tool responds to those errors. We added a JavaScript file that calls none existing functions and made changes to the registration form. The tool discovered both errors. It detected the none existing function calls and errors when submitting wrong data within a form.

From the previous form extended model we knew that the performance of G∀ST will decrease if the amount of pages to test increases. This same behaviour is shown with testing the webshop model. With the wordpress use case, we tested almost every webpages, but this application only contains a few products. Testing the Matrascenter.nl use case the amount of test paged decreased. With this use case we also found a unwanted behaviour in the checkout phase of the application. We did not found this error with Crawljax or with testing the webshop application manually. We believe that if we make improvement to G∀ST to find more web pages and decrease the time it takes to test this tool could be useful in testing web applications.

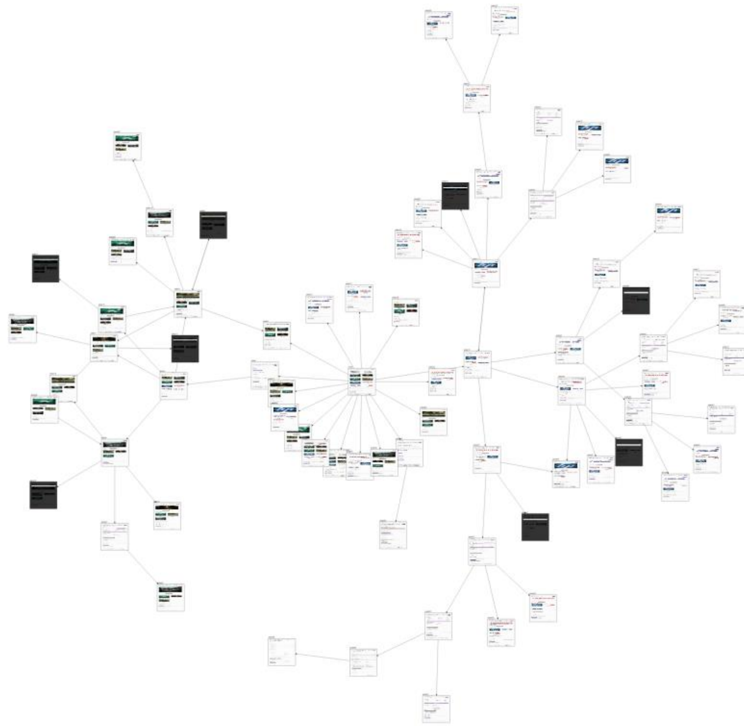


Figure 14 – State overview from Wordpress webshop application, by Crawljax.

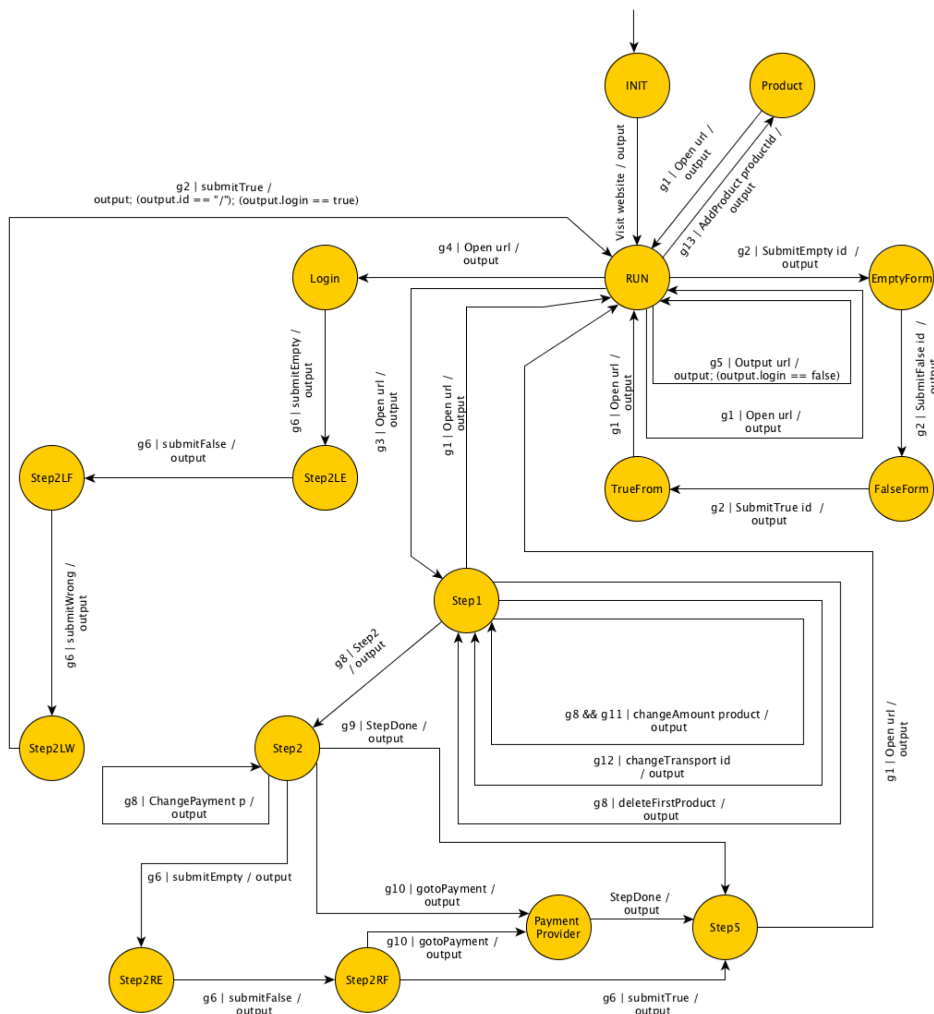


Figure 15 – Shop model applied on Wordpress webshop.

8 Reflection

Looking back on the work we have done, we could have done things differently. The result of this research is based on certain choices made during this research. One thing we know for sure is that not everything that is testable within the field of web applications is covered. The field of web applications is just too broad.

Black-box testing and user flow

One of the first choices we made to reduce our scope of research is to determine on what level we want to test web applications. We have chosen to test web applications on the same level as a visitor would experience the use of the web application. We simulated actions that a user is able to perform on the web application through a web browser. We simulated those actions by finding the events that are triggered if a user performs a certain action and executes that manually. If we had chosen for a different approach, for example on a code level, we would ended up with different possibilities to test. We could have ended up with white-box testing instead of black-box testing.

HTML forms

In section 3.1.1, we discussed the different types of form elements there exist. We mentioned that we did not include the form elements in our test environment that are introduced in the HTML 5 standard. We have also chosen to rely on the annotations the .NET frameworks adds to the html code to validation the forms (Section 6.4). Now that we look back, we could have chosen to test forms based on the validation done in the HTML5 standard instead of relying on the annotated html made by the .NET framework. With this way of testing forms, we could include the new input types that are introduced in the standard. With the HTML5 standard, it is possible to validate the form on both the client-side and server-side. We implemented testing forms based on how .NET annotates form elements. However, it turns out that those annotations do not occur in other web applications developed in a different language. We had to rewrite a part of how we tested forms to deal with this issue. If we had instead chosen for testing forms using the validation of the HTML5 standard, we could have done things much easier.

Modelling languages

The choice of modelling languages that can be used to model the behaviour of the SUT is limited. This limit is set by the quality characteristics we wanted to cover but also by the abstraction we chose to communicate with. With a different set of requirements, other modelling languages could have been used, like an LTS. The choice of modelling language also depended on the suitable testing tools. Each tool deals with different modelling languages. We searched for open source and commercial model-based testing tools and what modelling language each tool could use. This way we could see what tools would work best with the model we created. Our preference went for an open source tool above a commercial one. In section 3.5, we explained why we have chosen for the MBT tool G∀ST and the modelling language ESM. The main decision to choose G∀ST instead of TestOptimal was because of the experience we have with the tool G∀ST and it is open source. We could have chosen for the tool TestOptimal instead. This could lead to other results, based on the approach TestOptimal uses to generate test cases. If we had chosen for a completely different modelling language like an LTS, we were forced to use a different model-based testing tool, probably ended up with the tool Torxakis.

Quality characteristics
In the next step, we determined the scope of our testing ground. One of the desired goals from Bluenotion is to test the functionality of web applications. All the functionality aspects are explained in section 5.2. We thought of testing the design of the web application but this aspect deserves a research on its own. We explained in section 5 that each browser has its own interpretation of rendering a webpage. Including browser compatibility in our testing scope would be work intensive but is worth investigating in the future. Not only design and functionality are important when testing web applications but security is important as well. We covered only a small part of this aspect. We only tested if users can access pages without proper credentials. There are a lot more aspects to test within the field of the security. The organization OWASP[70] is a non-profit organization that is focused on improving the security aspect of software. OWASP provides documents and a list of tools that can help organizations with delivering and maintaining applications that can be trusted. OWASP also maintains a list of the top 10 most critical web applications security risks. Definitely, there are more security risks than those mentioned in the OWASP top 10 but those are the most critical. We have chosen to only focus on security of credentials and access because this is especially relevant when you have a webshop application. Also many tools[71] already exist that scan for those top 10 security risks.

Web broswers
In our test architecture, we used the tool PhantomJS combined with NodeJS to communicate with G∀ST and the web application. At first PhantomJS worked fine until we started implementing the webshop model. PhantomJS only supports one kind of web browser so we are bound to the functionality of that browser. However, it is possible to start multiple instances of this browser. Because the simulation of user events is done in JavaScript, we were stuck to the JavaScript functions that PhantomJS supports within this browser. In some cases, we had to find a different way to fire an event, because the function we knew was not supported. In our search for a suitable tool to approach a web application, we found an additional tool named, Selenium. Selenium is also open source tool that is used in many cases of testing web applications. Selenium is implemented in the tools Crawljax and TestOptimal to open web applications. Multiple browsers are supported whereas PhantomJS only support one. Besides the support for multiple browsers, it has the ability to visualize what we are testing. This could have helped us with debugging challenges we encountered during the construction of our adapter. In section 4.2, we have chosen not to use Selenium because of previous struggle with installing the tool. However if we look back on how we used PhantomJS, it could be worth using Selenium in the future for its features mentioned above.

Generic model
In the beginning of our research to find an approach to model web applications, we discovered that certain parts of a web application were already being tested with model-based testing purposes[11]. The results in the literature showed us that it is possible to apply model-based testing techniques with a small side effect, which is a state explosion. We explained this downside in section 6.1. This state explosion was one of the behaviours we do not want to have in our new model.

We focused in our search for a new model, on making this model as generic as possible, so that it is (ideally) applicable to all web applications that exist. With our model introduced in section 6.3, we are able to test web pages that are reachable within the web application. We tested this new model with the web application from Bluenotion. The results looked promising we found several errors that should not occur within this application. However, we found out that not every web page was tested with the generated test cases of our model-based testing tool. Forms that exists on a web application were not tested. Therefore, we extended our model in section 6.4 so it could test forms. The approach we took was testing forms with three different values in a following order. First, we tested a form with empty values, secondly with false values and at last with true values. This way a form should always be submitted successfully at the end of the test. Beforehand we knew the values each form field should have. We could have done this differently. For example, we could generate values for each field type randomly based on a certain set of values. This way we could test more values and combination of their values if we tested the form another time.

Results

The results of this form extend model looked fine. We deliberately changed values for certain fields in the form and they were indeed marked as invalid by the tool. However, with the second use case, the Bluenotion application, we test not all the web pages. We found out that G∀ST generated test cases randomly and not choosing other web pages that were not even tested. If we ran longer test cases than we currently did, we probably found more web pages. However, the time it takes to test was already a couple of hours for a small web application. With the tool Crawljax, we found more pages and more errors within less than half an hour.

Our goal at beginning of our research was to test webshop applications so we extended our model even further so it is able to test a webshop application. With the introduction of webshop applications more functionality is added to a normal web application. We extended our model, which is introduced in section 7. We found out that it is almost not possible to keep a generic model that is applicable to every webshop application that exists. With webshop applications, multiple variants exist. Each webshop application has its own interpretation of buying products. It was hard to model this in a generic solution. We do found out that the functionalities in every webshop is almost the same. However not every functionality is found on the same web page. We tested our webshop model with two use cases. With the first use case, we tested every transition in the model and even found unwanted behaviour in the webshop application. However, the downside of testing random was now even more noticeable. It is not possible to changes this behaviour of G∀ST very easy. Nevertheless, to achieve better and faster results we should change this random behaviour of G∀ST in to a much clever way of generating input actions.

9 Conclusion

In this section, we conclude this thesis. In section 9.1, we will state a conclusion about the testing tool we used and its usefulness. We discuss in section 9.2 the usefulness of the webshop model, defined in section 7, with regard to other webshop applications. We answer the research question we defined at the beginning of this thesis in section 9.3. We also state known issues that are still open in section 9.4. We end with the future work in section 9.5.

9.1 Compatibility of the test tool

We used G∀ST as a MBT tool to generate test cases. The tool turned out to be useful in our test architecture. However, the quality of the on-the-fly generated test cases were not that useful. The test cases were generated pseudo randomly and turned out to be a bit disappointing in the bigger use cases we used. With Bluenotion use case, roughly 65% of the existing pages were tested. This number even drops more when we tested the Matrascenter use case, where almost 40% of the existing pages were tested. To achieve a better page coverage we need a better solution than generating pseudo random test cases. This means we need to improve the test generation in G∀ST with a better solution than pseudo random generation or we have to switch to a different model-based testing tool that has a clever generation already implemented.

The adapter we created can be used in different model-based testing tools. Theoretically, it is possible to translate our current ESM model to an LTS model or an EFSM. With this possibility, we can use different MBT tools to test web applications. We only have to make sure that the output that is generated by the MBT tool will be accepted by the adapter. This means that we can easily switch to a better model-based testing tool than we current used. For example, the tool TorXakis can be used if the model is translated to an LTS and TestOptimal if the model is translated to an EFSM. By using other MBT tools, we can explore different approaches of generating test cases and analysing test results. It also gives users the ability to choose a preferred MBT tool that users have experience with.

9.2 Compatibility of webshop

As we know from our test cases, the current webshop model is not compatible enough to test every webshop application that exists. The main reason is that each webshop application differs from another in such a way that the current model is not applicable to all of them. We do find enough similarities between each webshop application. Most of the functionalities are the same, like adding a product to the shopping cart or having a shopping cart that shows what items are in there. Therefore, we can keep most of the functionalities we created in our adapter but we have to adjust our model to the behaviour of other webshops. This change of behaviour is only found in the checkout phase of the model.

In the two test cases of our webshop model, we indeed found the same functionalities but there are less steps used in the Wordpress webshop. If we look to other webshops like Amazon the checkout phase completely differs from the two test cases, but again the same functionalities hold.

9.3 Answer research questions

The goal of this thesis was to find an approach to test web applications with model-based testing. We wanted to formulate an abstract model that is capable of testing multiple web applications with the least amount of changes, preferably an out-of-the-box model. We defined the following research question to find out if we can improve the quality of web applications with model-based testing.

RQ1 How can we improve the quality of web applications by applying model-based testing techniques?

To come up with our final answer of the research question we first answer our sub research questions that should help us answer our main research question.

RQ1.1 What quality aspects are important with testing web applications?

The first sub question RQ 1.1, mentions what quality aspects are important while testing web applications. To answer this question we first required a definition of testing and quality, which are both mentioned in section 3.2. To improve quality we need to know what quality characteristics are important with testing applications we mentioned the important qualities characteristics in section 5.1. The list of important qualities was too big to cover in this thesis. We therefore selected a subset of this list where we want to focus on to improve the quality of web applications. This subset is mentioned in section 5.2, which consist of functionality of the web page, testing performance based on loading speed of the page, security based on testing login forms, usability based on the existence of files within a web page and finally accessibility based on the existence of html attributes.

RQ1.2 How can we apply model-based testing techniques to web applications?

Based on the answer in research question RQ 1.1 we could search for an answer to our second research question. In RQ 1.2 we asked the question: "How can we apply model-based testing techniques to web applications?". We searched for an approach to test web applications based on a user's approach. We want to test what users also experiences while using the web application. We found a way to access the information a user also sees through a web browser. We used the tool PhantomJS above Selenium to retrieve this information. The information we needed is stored in the DOM object and could be retrieved with certain search functions. We even could simulate user actions by using the events stored in this DOM object. The only thing that is left to be able to test web applications with model-based testing is a model and a suitable model-based testing tool. It turned out that with PhantomJS it is not possible to communicate with a model-based testing tool G∀ST we used. We added NodeJS for the communication between those two tools.

RQ1.3 What are the advantages and disadvantages of existing modelling languages, found in literature, with model-based testing of (web) applications?

In research question RQ 1.3, we asked the question what are the advantages and disadvantages of existing modelling languages, found in the literature, with model-based testing of (web) applications. We searched in existing literature for modelling languages that are used to test web applications.

We mainly found models of web applications that are modelled in an FSM or StateCharts[12]. Other literature talked about models in the modelling languages LTS (Section 3.3.2) and ESM (Section 3.3.4), but are not used to test web applications but other systems. The main disadvantage, which is also its advantage in an FSM(Section 3.3.3) is the state explosion. State explosion requires lot of time to implement, but is useful to test very detailed steps in system as done in[11]. In addition, the lack of variables and a finite set of states, input, and output makes this modelling language not a favourite language to model web applications.

An LTS, which we discussed in section 3.3.2, allows variables but the support of model-based testing tools is limited. The possibilities with using an ESM as modelling language is great. It support the use of variables, stored in a parametrized state and support deterministic and non-deterministic models. In section 3.5, we explain the reason why we have chosen for an ESM as modelling language combined with the model-based testing tool G∀ST.

RQ1.4 How can we create a new model that is capable of testing web applications that deals with the disadvantages of existing models?

In research question RQ 1.4, we defined the question: How can we create a new model that is capable of testing web applications that deals with the disadvantages of existing models? We found one real suitable model of a web application in [11]. This model of a web application is a state explosion and requires lots of time to model it and not even speaking of its implementation work. Other models that are introduced in papers are partial models that only cover the shopping cart functionalities of a web shop application[19] or testing forms on a web application [14]. With every model that we found usable to test web applications with, we could not reuse it in a different web application because it was only modelled for that specific web application.

We wanted to create an abstraction to this model so we do not have to model every web page that exists within a web application. By assuming that every pages is the same, not concerning its content we created this abstraction. Therefore, the model should also be applicable to test other web applications. To realize this abstraction we needed to store information about a page in the model. With using, an ESM as modelling language this is possible. We store information about each page, its url, links to other pages, forms, products, payment option, shipping method, login state, and the shopping cart in the parametrized state. This information is updated after every transition. With the tool PhantomJS, we can retrieve the information that we needed.

The new model is introduced in section 6.3 and an extension to test forms in section 6.4. With this model we saw that we were not able to model a webshop application so we introduced another extension to this model in section 7. We tried to continue the generic approach to model a webshop application. However, we were forced to created states for every step in the checkout phase. This is because we are interested in testing the shopping cart and modify the shopping cart in certain steps, by changing the amount, deleting products, adding products, changing shipping method, and change the payment option. Not every action is available in each page. Therefore, we had to make this distinction. The webshop model is kept as small and generic as possible but due to many different webshops, it is not a generic model to test web applications with.

A downside to the abstraction we made in the introduced models is the loss of detail. By keeping the model as generic as possible, we can use the model to test different applications. However, it is not possible to test every detail in the web application. Because our focus mainly lies with testing the functionality of the web application this level of detail is not required.

RQ1.5 How does the new model perform?

To answer our research question RQ 1.5, we have to look to the results of both models. Based on the results mentioned in the sections 6.5 we can say that the form extension model delivers interesting results for small web applications. However we are a bit disappointed in the performance and testing results from the bigger web application we tested, but lack of performance and results is achieved based on a random walk algorithm to generate test cases. It is possible to replace this algorithm with a smart algorithm that keeps track of already visited pages. This should probably lead to better performance and results.

The form extension model found in section 6.4 is capable of testing web applications that contain forms, with very few modifications. We can say that this model is capable of testing every web application that is created by the company Bluenotion, excluding webshops and custom build applications. The only thing that is required to be changed are the values for the fields that can be found in each form within the application. We have to provide a correct and incorrect value for each field the model stays the same.

In a small web application like the one we tested in section 6.5.1, we can test every page and detect the errors that we are interested in. However when web applications grows in the amount of pages, we cannot guarantee that we can cover every page that the application holds, like the one tested in section 6.5.2. The reason we cannot guarantee this is because of the random walk G∀ST uses to generate test cases.

The same behaviour is found when we tested two webshops. The webshop application in section 7.3.2 was almost covered, except the search form, which was not tested. However, this use case contains fewer pages that use case we tested in section 7.3.1. If we look to the results the Matrascenter.nl use case, we can see that fewer pages are tested compared to the tool Crawljax. However, Crawljax did not tested every page either.

RQ1.6 What are the possibilities with the new modelling framework to test different web applications?

Research question RQ 1.6 states what the possibilities are with the new model to test web applications. With the model we created in section 6.4, we can test web applications that also contains forms. It is a generic solution to test informational web applications. Not every form element that can be used is implemented as explained in section 3.1.1, but the most occurring elements are supported. However, this model is not suitable for testing webshop applications.

We extended this model further so it is able to test webshop applications. With this model, we can add products to the shopping cart and follow the steps that are required to buy these products. It is even possible to pay (in a test environment) with an external payment provider. When testing this model with the use case mentioned in section 7.3.2, we found some troubles with the way we modelled. It turned out that not every webshop application is implemented the same way. They mainly differ in the checkout phase. We had to change the model so it was able to test a different webshop application. Nevertheless, the functionalities we test stays the same with this different model. We can say that our webshop model is not generic enough to test multiple webshop applications. Changes to the model are still required.

RQ1.7 What set of tools are required and available to test web applications?

Research question RQ 1.7 states what tools are required to test web applications. Based on our approach we used PhantomJS as a tool to communicate with web applications through a headless browser. Using this tool required an additional tool named NodeJS to communicate with a model-based testing tool. We used G∀ST as a model-based testing tool. G∀ST turned out to be useful but has a huge downside, which is its pseudo random generation. This tool requires improvement to deal with this downside. Our environment is created to be able to replace G∀ST is required to use a different model-based testing tool that may have better test selection.

PhantomJS combined with NodeJS is very usable to access web applications and retrieve information from it. There is only one downside we can think of to not use this tool, which is testing browser compatibility. Phantom uses only one browser engine and therefore we could not test browser compatibility. For testing this quality characteristic, we suggest using the tool Selenium.

RQ1.8 How does the tool perform in coverage with regard to other existing tools and manual testing?

We used four use cases, two use cases for each model to test our tool and a different testing tool Crawljax. We also tested each use case manually. The time it takes to test the web application is with Crawljax much faster than with our tool. The difference can be found in the approach taken to test the web application. Crawljax uses a crawl engine to find pages and tries to test every page it find during this search. G∀ST uses a pseudo random algorithm to generate test cases on-the-fly based on the given output from the previous transition. This is the reason why G∀ST take more time to test the web application.

If we look to the outcome of the results for each use case, we can say that our testing tool finds more errors, but this is because Crawljax is configured in such way that it does not detects these errors. In some cases, Crawljax found more page errors but this is because G∀ST tested less pages due to it random choices.

If we compare our testing tool with manual testing, our testing tool finds more errors. In particularly the error, we found in the shopping cart of the use case Matrascenter.nl. Some errors are overlooked in manual testing whereas our model-based testing tool did not. This shows that automated testing is most of the time better than manual testing.

Now that we answered the sub research questions, we can answer our main research question. We can say that we can improve the quality of web applications with model-based testing. However, the improvements to the quality are currently not that high. We say that our solution to test web applications is a first step to improve the quality and there is lots of room for improvement. We can improve this for example with a better test selection. We suggest several improvement in section 9.5.

The quality may not be improved that much, but testing is not only about the quality it is also about time and effort. The time it takes to test is currently high compared to manual testing and even with Crawljax. Improving the test generation will probably lead to quicker results. However, the effort that is required is less than that with testing manual. Our testing tool runs automated we only have to start it. This means we can run the tool overnight and see the testing results the next morning, without spending effort. With manual testing, we spent much more effort. We have to test everything and keep track of what we tested. If changes are made to the application, we can start all over again. If we want to test a different web application with the form extension model, we only need to change the values for the forms that exist within the web application. With a webshop application, more changes are required but still less than testing manually. Therefore, the effort and time to test a different web application is better with our tool.

9.4 Known issues

One of the issues we faced was the use of third party libraries that are used in the application. For example the Google Maps API. Recently Google forced users of Google Maps to use a newer version API. Loading the map in the application works and no errors are given. However, testing a page that contains an older version of the Google Maps API result in the following error: “Unable to delete property”. We can argue if this is a proper error or just a bug in the testing software. After some digging, it turns out that it is indeed not an error in the web application, but a bug in the software we used to test. PhantomJS handles certain JavaScript code in a wrong way. This should be resolved in a newer version, but during the writing of this thesis the problem was not fixed yet.

9.5 Future work

We focussed on finding generic models to test web applications. The two models we have shown are useable but the last model we showed has still room for improvement. We did not succeed in developing a total generic solution to test webshop applications, but we have laid the foundation for an approach. We encourage to the keep searching for a more generic solution then the one we presented.

A great next step is improving the test tool to cover more qualities characteristics. Because some quality characteristics, like browser compatibility cannot yet be tested with our tool, because PhantomJS only supports one browser engine. One quality characteristic we could test more detailed is about testing forms. We now have fixed values for each field in each form. We can instead think of possibilities to generate each value based on their field property. In this way, it would be possible to cover more types of input fields, like the input types introduced in the HTML 5 standard and test different values if the form is tested twice or more.

We certainly suggest improving the test selection of G∀ST. We suggest to find a clever way to go through the state space of G∀ST instead of using the current pseudo random algorithm. Implementing a learning algorithm could be an additional feature to this. Currently random walk is used to generate test cases. With a learning algorithm, we can learn from previous test run and improve the test generation even better. With this improvement, we could diminish the amount of time it takes to discover pages and test a broader range of pages in the web application.

Furthermore, a good possibility is to continue testing while errors are found. Finding a way to recover from a page that cannot be found is desirable. A possible solution is introducing a "goback" function. If the page is not found, we can go back to a previous working page. This way it should be possible to recover from livelocks as mentioned in section 6.3.

A possible last step to make is combining Crawljax with our testing tool. We can think of using Crawljax as a pre-processing tool to find relevant paths or detect broken links in an earlier stage and using our model-based testing tool to find further unwanted behaviour.

10 References

1. Koopman, P. and R. Plasmeijer, *Fully Automatic Testing with Functions as Specifications*, in *Central European Functional Programming School*, Z. Horváth, Editor. 2006, Springer Berlin Heidelberg. p. 35-61.
2. Newell, D. *6 Reasons You Should Buy An Online Versus Offline Business*. 2014 [cited 2015 12-05-2015]; Available from: http://feinternational.com/blog/6-reasons-buy-online-versus-offline-business/.
3. Naseer, N., *4 reasons to shift your offline CRM to the cloud now.* 2014.
4. Post, H. *A Shift from Online to Offline: Adolescence, the Internet and Social Participation*. 2014 [cited 2015 12-05-2015]; Available from: http://www.huffingtonpost.com/undergraduate-awards/a-shift-from-online-to-of_b_4431523.html.
5. Catone, J. *Adobe Preparing Full Shift to Web Apps*. 2007 [cited 2015 12-5-2015]; Available from: http://readwrite.com/2007/10/18/adobe_preparing_full_shift_to_online_apps.
6. Microsoft. *ASP.NET MVC*. [cited 2015 6-4-2015]; Available from: http://www.asp.net/mvc.
7. Highsmith, J. and A. Cockburn, *Agile software development: the business of innovation.* Computer, 2001. **34**(9): p. 120-127.
8. *The Incredible Rate of Diminishing Returns of Fixing Software Bugs*. 2009 [cited 2015 14-5-2015]; Available from: http://superwebdeveloper.com/2009/11/25/the-incredible-rate-of-diminishing-returns-of-fixing-software-bugs/.
9. magazine, I.s. *Three-quarters of US small firms have not tested website cybersecurity*. 2011 [cited 2015 12-05-2015]; Available from: http://www.infosecurity-magazine.com/news/three-quarters-of-us-small-firms-have-not-tested/.
10. Pugh, T. *Contractors say late changes, lack of testing doomed health care website launch*. 2013 [cited 2015 12-05-2015]; Available from: http://www.mcclatchydc.com/2013/10/24/206431/contractors-say-late-changes-lack.html.
11. Achkar, H. *Model Based Testing Of Web Applications*. 2010.
12. Ogaard, K. and A. Malge, *A Model Based Testing Technique to Test Web Applications Using Statecharts*. Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on. 2008. 183--188.
13. Andrews, A.A., J. Offutt, and R.T. Alexander, *Testing Web applications by modeling with FSMs.* Software & Systems Modeling, 2005. **4**(3): p. 326-345.
14. Kung, D.C., L. Chien-Hung, and P. Hsia. *An object-oriented Web test model for testing Web applications*. in *Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International*. 2000.
15. Karam, M., W. Keirouz, and R. Hage, *An Abstract Model for Testing MVC and Workflow Based Web Applications*, in *Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services*. 2006, IEEE Computer Society. p. 206.
16. Ernits, J., et al., *Model-Based Testing of Web Applications Using NModel*, in *Testing of Software and Communication Systems*, M. Núñez, P. Baker, and M. Merayo, Editors. 2009, Springer Berlin Heidelberg. p. 211-216.

17. Marchetto, A., P. Tonella, and F. Ricca. *State-Based Testing of Ajax Web Applications*. in *Software Testing, Verification, and Validation, 2008 1st International Conference on*. 2008.
18. Mesbah, A., A. van Deursen, and D. Roest, *Invariant-Based Automatic Testing of Modern Web Applications.* Software Engineering, IEEE Transactions on, 2012. **38**(1): p. 35-53.
19. Shams, M., D. Krishnamurthy, and B. Far, *A model-based approach for testing the performance of web applications*, in *Proceedings of the 3rd international workshop on Software quality assurance*. 2006, ACM: Portland, Oregon. p. 54-61.
20. Hajiabadi, H. and M. Kahani. *An automated model based approach to test web application using ontology*. in *Open Systems (ICOS), 2011 IEEE Conference on*. 2011.
21. Bae, G., G. Rothermel, and D.-H. Bae, *Comparing model-based and dynamic event-extraction based GUI testing techniques: An empirical study.* Journal of Systems and Software, 2014. **97**(0): p. 15-46.
22. O'Reilly, T., *What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software.* 2005.
23. W3C. *Document Object Model (DOM)*. 2015 [cited 2015 14-5-2015]; Available from: http://www.w3.org/DOM/.
24. W3C. *Cascading Style Sheets*. 2015 [cited 2015 14-5-2015]; Available from: http://www.w3.org/Style/CSS/.
25. W3C, *HTML5*, in *A vocabulary and associated APIs for HTML and XHTML*. 2014, W3C.
26. W3Schools. *HTML Form Elements*. [cited 2015 6-4-2015]; Available from: http://www.w3schools.com/html/html_form_elements.asp.
27. W3. *W3 HTML Elements*. [cited 2015 6-5-2015]; Available from: http://www.w3.org/TR/html-markup/Overview.html#toc.
28. De Ryck, P., et al., *Automatic and Precise Client-Side Protection against CSRF Attacks*, in *Computer Security – ESORICS 2011*, V. Atluri and C. Diaz, Editors. 2011, Springer Berlin Heidelberg. p. 100-116.
29. ISO/IEC, *Guide 2, Standardization and related activities -- General vocabulary* 2004.
30. Myers, G.J., et al., *The Art of Software Testing*. 2004: Wiley.
31. Tretmans, J., *Testing Technique lecture notes.* Radboud University Nijmegen, 2012.
32. Tretmans, J., *Model Based Testing with Labelled Transition Systems*, in *Formal Methods and Testing*, R. Hierons, J. Bowen, and M. Harman, Editors. 2008, Springer Berlin Heidelberg. p. 1-38.
33. Bolton, C., *Adding Conflict and Confusion to CSP*, in *FM 2005: Formal Methods*, J. Fitzgerald, I. Hayes, and A. Tarlecki, Editors. 2005, Springer Berlin Heidelberg. p. 205-220.
34. Harel, D., *Statecharts: A visual formalism for complex systems.* Sci. Comput. Program., 1987. **8**(3): p. 231-274.
35. Veanes, M., et al., *Model-based testing of object-oriented reactive systems with spec explorer*, in *Formal methods and testing*, M.H. Robert, P.B. Jonathan, and H. Mark, Editors. 2008, Springer-Verlag. p. 39-76.
36. Bijl, M., A. Rensink, and G.J. Tretmans, *Component Based Testing with IOCO*. 2003: Centre for Telematics and Information Technology, University of Twente.

37. Tretmans, J., *Test generation with inputs, outputs, and quiescence*, in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Margaria and B. Steffen, Editors. 1996, Springer Berlin Heidelberg. p. 127-146.
38. Broy, M., et al., *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. 2005: Springer-Verlag New York, Inc.
39. El-Fakih, K., et al., *Fault Diagnosis in Extended Finite State Machines*, in *Testing of Communicating Systems*, D. Hogrefe and A. Wiles, Editors. 2003, Springer Berlin Heidelberg. p. 197-210.
40. Campbell, C., et al., *Testing Concurrent Object-Oriented Systems with Spec Explorer*, in *FM 2005: Formal Methods*, J. Fitzgerald, I. Hayes, and A. Tarlecki, Editors. 2005, Springer Berlin Heidelberg. p. 542-547.
41. Schmitt, M., M. Ebner, and J. Grabowski. *Test Generation with Autolink and TestComposer*. 2000.
42. Bohnenkamp, H. and A. Belinfante, *Timed Testing with TorX*, in *FM 2005: Formal Methods*, J. Fitzgerald, I. Hayes, and A. Tarlecki, Editors. 2005, Springer Berlin Heidelberg. p. 173-188.
43. Olsson, N. and K. Karl. *Graphwalker*. 2015 [cited 2015 15-01]; Available from: http://graphwalker.org/.
44. Hart, P.E., N.J. Nilsson, and B. Raphael, *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.* Systems Science and Cybernetics, IEEE Transactions on, 1968. **4**(2): p. 100-107.
45. Dijkstra, E.W., *A note on two problems in connexion with graphs.* Numerische Mathematik, 1959. **1**(1): p. 269-271.
46. Jones, S.P. *Haskell*. 1990 [cited 2015 14-01]; Available from: http://haskell.org.
47. Mostowski, W., et al., *Model-Based Testing of Electronic Passports*, in *Formal Methods for Industrial Critical Systems*, M. Alpuente, B. Cook, and C. Joubert, Editors. 2009, Springer Berlin Heidelberg. p. 207-209.
48. Koopman, P., et al., *Gast: Generic Automated Software Testing*, in *Implementation of Functional Languages*, R. Peña and T. Arts, Editors. 2003, Springer Berlin Heidelberg. p. 84-100.
49. Plasmeijer, R. and M. Eekelen van. *CLEAN*. 2002 [cited 2015 15-01]; Available from: http://clean.cs.ru.nl/Clean.
50. van Weelden, A., et al., *On-the-Fly Formal Testing of a Smart Card Applet*, in *Security and Privacy in the Age of Ubiquitous Computing*, R. Sasaki, et al., Editors. 2005, Springer US. p. 565-576.
51. Koopman, P., P. Achten, and R. Plasmeijer, *Model Based Testing with Logical Properties versus State Machines*, in *Implementation and Application of Functional Languages*, A. Gill and J. Hage, Editors. 2012, Springer Berlin Heidelberg. p. 116-133.
52. TestOptimal. *TestOptimal*. 2015 [cited 2015 13-05-2015]; Available from: http://testoptimal.com/.
53. SeleniumHQ. *Selenium HQ, Browser Automation*. 2015 [cited 2015 6-5-2015]; Available from: http://www.seleniumhq.org/.
54. PhantomJS. *PhantomJS, Full web stack no browser required*. 2015 [cited 2015 6-5-2015]; Available from: http://phantomjs.org/.
55. Inc., J. *NodeJS*. 2015 [cited 2015 6-5-2015]; Available from: https://nodejs.org/.
56. Microsoft. *ASP.NET Routing*. 2010 [cited 2014 8-10]; Available from: http://msdn.microsoft.com/en-us/library/vstudio/cc668201(v=vs.100).aspx.

57. Microsoft. *Models and Validation in ASP.Net MVC*. 2010 [cited 2014 8-10]; Available from: http://msdn.microsoft.com/en-us/library/dd410405(v=vs.100).aspx.
58. Köhler, R. *MVC, MVP, ASP.NET*. 2008 [cited 2014 8-10]; Available from: http://www.codeproject.com/Articles/30597/MVC-MVP-ASP-NET.
59. Guthrie, S., *Introducing "Razor" – a new view engine for ASP.NET*. 2010, Microsoft.
60. Microsoft. *Controllers and Action Methods in ASP.NET MVC Applications*. 2011 [cited 2014 8-10]; Available from: http://msdn.microsoft.com/en-us/library/dd410269(v=vs.100).aspx.
61. International, E., *The JSON Data Interchange Format*. 2013.
62. W3Schools. *AJAX Tutorial*. 2014 [cited 2014 8-10]; Available from: http://www.w3schools.com/ajax/.
63. MVC4Beginner. *ASP .NET MVC Accessing Models Data From Controller*. 2014 [cited 2014 8-10]; Available from: http://mvc4beginner.com/Tutorial/ASP-.NET-MVC-Accessing-Models-Data-From-Controller.html.
64. Olsina, L., G. Lafuente, and G. Rossi, *Specifying Quality Characteristics and Attributes for Websites*, in *Web Engineering*, S. Murugesan and Y. Deshpande, Editors. 2001, Springer Berlin Heidelberg. p. 266-278.
65. Olsina, L. and G. Rossi, *Measuring Web application quality with WebQEM.* MultiMedia, IEEE, 2002. **9**(4): p. 20-29.
66. W3. *Accessibility Evaluation Resources*. 2015 [cited 2015 11-5-2015]; Available from: http://www.w3.org/WAI/eval/Overview.html.
67. Crawljax. *Crawling Ajax-based Web Applications*. 2015 [cited 2015 19-4-2015]; Available from: http://crawljax.com/.
68. Mesbah, A., et al., *Exposing the Hidden-Web Induced by Ajax.* 2008.
69. Mesbah, A., E. Bozdag, and A.v. Deursen, *Crawling AJAX by Inferring User Interface State Changes*, in *Proceedings of the 2008 Eighth International Conference on Web Engineering*. 2008, IEEE Computer Society. p. 122-134.
70. OWASP. *OWASP*. [cited 2015 5-4-2015]; The Open Web Application Security Project]. Available from: https://www.owasp.org.
71. OWASP. *OWASP Scanning Tools*. 2015 [cited 2015 5-4-2015]; Available from: https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools.