Automated Software Testing of JavaScript Web Applications

Version of January 19, 2017

Martin Jorn Rogalla

Automated Software Testing of JavaScript Web Applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Martin Jorn Rogalla
born in Enschede, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Fujitsu Laboratories of America Inc.
Software Quality and Security Lab
1240 E Arques Avenue
Sunnyvale, CA, USA
`www.fujitsu.com/us/`

Automated Software Testing of JavaScript Web Applications

Author:	Martin Jorn Rogalla
Student id:	4173635
Email:	email@martinrogalla.com

Abstract

Modern software is becoming more and more complex and manual testing cannot keep up with the need for high-quality reliable software: often due to the complexity of event-driven software, manual testing is done. This comes with many disadvantages in comparison with automated testing.

The increased importance of having a secure, reliable online presence requires testing of JavaScript web applications. This thesis explores the current state of Automated Testing for JavaScript web applications, presents a new Automated Testing Framework and gives an outlook on future research. It intends to resolve some of the complexity issues to allow for automated testing.

Preface

This master thesis was written for the partial completion of my Master of Science Degree in Computer Science at Delft University of Technology. The research is a collaboration between Delft University and Fujitsu Laboratories of America Inc.

Automated Testing of Web Applications has gained an important role as systems have become more complex. As I spoke to Arie van Deursen regarding my interest in this area, he introduced me to Mukul Prasad. Mukul Prasad was so kind to provide me with a research internship opportunity in Sunnyvale, California. I am very grateful to Arie van Deursen and Mukul Prasad for the excellent opportunity they provided me and for all of their valuable feedback and discussions.

This is not a product created solely by one individual, but by an individual supported by many: I would like to thank my parents and my sister for supporting me through my life. I feel like the most lucky person on this planet to have such a supporting family. I hope to pass this on some day.

Thank you Judit for making every day a brighter day and supporting me. May our dreams come true!

Alex Horn, thanks for being such an amazing friend! Thank you for the adventures we shared and the life-encompassing discussions we had in California. *Propediem te videbo* (See you soon)!

Martin Jorn Rogalla
Delft, The Netherlands
January 19, 2017

Contents

List of Figures

Chapter 1

Introduction

Web applications play an ever-more important role in our lives. Shopping, studying, banking, job searching, finding a partner; practically anything can be done online. Having an online presence is seen as a necessity if a business wants to operate in the 21st century. One of the prominent mechanisms that facilitates this online presence are web applications. Usage of web applications facilitates user engagement, interaction and convenience between businesses and users. Providing web applications has become a business by itself: the Software as a Service (SaaS) delivery model allows companies to use their developed web application as a service to their customers. Usage of this delivery model is especially popular in the enterprise software world.

Web applications have been improved by making use of JavaScript. Before 2005, JavaScript was mainly used to enhance user-interface esthetics. In 2005, JavaScript gained a more prominent role with the proposal of Ajax [16]. Ajax allows for asynchronous communication in a web application. In this asynchronous communication, JavaScript is used for the client-side and is the actor that sends requests to a server and processes the responses back into the web application interface. JavaScript thus plays an important role in improving the user-experience of web applications; without it many of the web applications in use today could not exist.

The rise of web applications and our increased dependence on them has made us more vulnerable to system failures. Faulty code can cause security breaches and loss of business that can result in massive financial consequences. Many of these failures can be prevented if a correct test-suite is in place.

Perhaps the most straightforward way of testing a web application is by manually clicking through the System Under Test (SUT) and checking if the application behaves appropriately. While this does not require any prior setup, it is a time-intensive process that is prone to user-error.

Instead of manually clicking through a system, we can write a test-suite that uses a webdriver. The webdriver executes all the steps described in the test-suite and checks the system's behavior. Even though writing the test-suite is still a manual process, the execution of the steps and behavior checking of the system is automated, allowing for repetitive regression testing. This removes one level of user-error from the testing process, but leaves the possibility of an incomplete test-suite.

To eliminate all forms of user-errors and guarantee correct behavior of the system, the generation of test-suites for web applications can also be automated. The importance of the correctness of these systems has sparked interest for research in the field of automated testing of web applications by automatic test input generation.

Various projects have been set up to explore the automated testing of web applications. Artemis [1] is an Automated Testing Tool for JavaScript. Crawljax [37] is a crawler and testing tool for Ajax web applications. SymJS [27] is an Automated Testing Framework for JavaScript Web Applications. Artemis currently only focuses on single page applications and provides JavaScript constants or random values to generate input values. Crawljax allows for large-scale web application crawling, but only provides random input values. Both of these systems only allow limited input values as they do not analyze logic present in the SUT. SymJS however uses symbolic execution to generate input values that are based on logical constraints found in the client-side JavaScript code.

A shared problem of the previously mentioned Automated Web Applications Testing Tools is the existence of an infinite number of paths that can be taken through an application. Executing all possible paths is infeasible and therefore a technique is required to only select the paths that exhibit previously unseen behavior from the SUT. A so far unexplored technique that could relieve this problem and improve symbolic execution based automated testing of web applications is the use of 'model-learning'. A successful application of a combination of symbolic execution and model-learning in the area of protocol verification has been applied in MACE [11]. This gave us curiosity to explore the possibility of using model-learning in combination with symbolic execution in the Automated Testing of JavaScript Web Applications.

Model-learning allows us to extract a concise specification of the SUT, characterizing the state space of system behaviors. Capturing the system behavior in a model and only executing the paths leading to different states allows us to execute all the logic in an application while minimizing the set of paths that are needed to be executed.

The goal of this thesis is to explore the usefulness of model-learning in combination with symbolic execution and compare our new technique with state-of-the-art tools for the purpose of automated testing of web applications.

Specifically we want to implement model-learning in SymJS, a JavaScript Web Application Testing Framework that uses symbolic execution for its input generation. We extend the current framework to include model-learning for its event-sequence selection. We want to put a special focus on the testing of enterprise web applications. Since the testing of these often requires a specific parameterized events-sequence, making the exploration tougher.

We evaluate our proposed approach by applying it to two sets of web applications: Enterprise Applications and General Applications. We compare different exploration and input generation techniques using the 'model-learning-only', 'symbolic-execution-only' and 'model-learning+symbolic-execution' modes in our framework.

We compare the results of our benchmarks against two state-of-the-art tools: Crawljax and Artemis.

To conduct our experience we created a sophisticated infrastructure fully automating the process of running the benchmarks and aggregating the results.

Our results shows that model-learning by itself gives a substantial improvement in coverage, even more than is gained by the use of symbolic-execution. Although the speed at which coverage is reached is not improved, we do see a smaller test-suite size. A smaller test-suite which covers more code gives us a test-suite of higher quality. Furthermore, our results indicate that the model-learning+symbolic-execution technique is especially well applicable for enterprise applications. The resulting model-learning+symbolic-execution technique shows merit for the automated testing of enterprise web applications for Fujitsu.

Outline In this thesis we start off in Chapter 2 by explaining concepts in the area of Automated JavaScript Testing. After establishing a common understanding of these concepts, we continue in Chapter 3 with a list of prioritized objectives to understand the scope of the project. In Chapter 4 we present the architecutre that we have chosen to realize the objectives. As model-learning has played a big role in the architecure of the system, we elaborate in detail how model-learning is applied in Chapter 5. During the comparison of the state-of-the-art automated testing frameworks we have noticed a lack of a good tool to perform coverage analysis on large scale JavaScript web applications. We have created a tool that is presented in Chapter 6. We present two benchmark sets that allows us to evaluate our system in comparison with other methods and tools in Chapter 7. After the evaluation we discuss our results in Chapter 8. We present related work in Chapter 9 and close the thesis by presenting our conclusions and proposing future work in Chapter 10.

Chapter 2

Preliminaries

Dynamic software verification, also known as dynamic software testing, is a process that ensures software is in accordance with its set requirements by running the *System Under Test (SUT)* and analyzing its behavior. In this thesis we focus on the automatic creation and execution of tests in automated testing, in order to minimize or get rid of the required user interaction [20].

Although many of the techniques described in this thesis can be applied, a focus is put on the Automated Testing of JavaScript Web Applications. We limit our scope to ensure a complete, but non-diverting overview of the area. In the sections below, we give an introduction to various concepts, challenges and approaches relevant to the Automated Testing of JavaScript Web Applications.

2.1 Test Cases

The verification of the SUT is performed by executing a set of test-cases. Each test-case consists of *test-input* and a *test-oracle*. The test-input invokes the SUT (optionally with parameters), resulting in a specific execution path through the program. When this execution path has been fully executed, the program is in a state whose success is determined by the test-oracle.

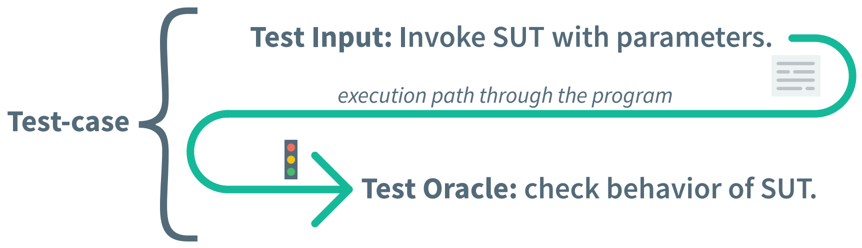

Test Input: Invoke SUT with parameters.
execution path through the program
Test-case
Test Oracle: check behavior of SUT.

Figure 2.1: An overview of a typical test-case used in Software Testing.

If according to the test-oracle the output meet the expectations, the test passes: indicating correct behavior of the program. If the output does not meet the expectations, the test-cases fails, indicating that the program is non-compliant with its requirements. Automated Software Testing aims to automate this *Dynamic Verification process* by automatically generating test-cases, executing them and reporting the results.

2.2 Input Generation

Since the objective is to test the whole program, the aim is to maximize coverage of the code in the SUT. In order to do so, we fire a sequence of parameterized events known as an execution path.

Each parameterized event consists of a set of *parameters* that contains DOM-elements selectors and the values they should have. For example:

```
#quantity_field="42", #remember_me="true", #password="D7rG3M"
```

The event itself consists of a DOM-element selector and the action that should be fired on that target. For example:

```
#quantity_update_btn::click
```

Firing a single execution path in the SUT will cover only part of the code. It is our goal to find a small subset out of the potential infinite set of execution paths that will cover as much code in the SUT as possible. In this section, the event parameter *input generation* is discussed, where in section 2.4, the *sequence of events* is discussed.

> The challenge of choosing the correct data inputs to maximize the coverage of the System Under Test (SUT) is known as the **Input Generation Problem**.

2.2.1 Randomized Testing

Randomized testing approaches [18] generate random inputs of the input domain, without analyzing the logic present in the function. Random testing is often used in combination with feedback. Various random testing approaches are presented below.

Feedback-directed Random Testing

Feedback-directed random testing uses random test generation in combination with feedback obtained from executing test inputs and uses this feedback to direct the generation of future input [46]. Some examples of feedback that can be used to direct the input generation are: coverage information (e.g. line, statement and branch coverage) and read/write-sets (dependency analysis of variables) [1, 27].

Differential Testing

Differential testing uses a reference and target system that are tested side-by-side. The systems are exhaustively tested using randomly generated tests. Differences in behavior of the reference and target system uncover potential problems [31].

Adaptive Random Testing

Adaptive random testing follows the idea that "test cases should be as evenly spread over the entire input domain as possible" [10]. By spreading over the entire input domain, adaptive random testing is able to detect failures quicker than ordinary random testing by using fewer test cases [10].

2.2.2 Systematic Testing using Symbolic Execution

Systematic testing approaches [6] causally generate inputs often based on internal program logic. Test generators that use this approach use some form of execution to derive the inputs necessary to trigger all the execution paths. The systematic testing approaches to test input generation we will be concerned with all make use of *symbolic execution*.

> **Symbolic Execution** supplies symbols rather than normal inputs for a program such that throughout the (virtual) execution of the program, we know how the program influences the symbols and how the program is influenced by the symbolic formulas [23].

To illustrate: imagine there exists an example program that takes an integer value. Assume that the program is designed to only succeed if the input value is equal to '42'. Instead of providing the program with an input value, the symbolic execution tool provides the program with a symbol, say 'α'. The symbolic execution tool executes the program and comes to a branch decision where the constraint of success or failure is described. It captures this path constraint '$\text{input} == 42$' and finishes its execution. Now that the whole execution is completed, the symbolic execution tool can provide concrete values with the help of a constraint solver that solves for the captured path conditions. It will solve for both sides of the constraint: '$\alpha == 42$' and thus provide the concrete value 42 and some other value that is not equal to 42.

The success of using symbolic execution in combination with automated testing can be seen in KLEE [9]. Cadar et al. report that in the 452 tested applications KLEE has found 56 serious bugs, including three in COREUTILS, which had been missed for over 15 years [9].

By analyzing the source code, the symbolic execution engine can construct constraints on the input values. The set of all these conditions combined, form a *path condition*. The path condition is the all-encompassing condition that is required to traverse the corresponding path in the program. Satisfiability Modulo Theories (SMT) solvers can then be used to solve for these constraints in the path condition.

> "The research field concerned with the satisfiability of formulas with respect to some background theory is called Satisfiability Modulo Theories (SMT) [5]."

Examples of satisfiability modulo theories that are used in automated testing include: integer theory, real number theory, bit-vector theory and arithmetic theories [14].

> An **Satisfiability Modulo Theories (SMT) solver** is a tool for deciding the satisfiability (or dually the validity) of formulas in these (SMT) theories [14].

SMT Solvers are powerful tools that are not only limited to symbolic execution, but are mathematical tools that can be used in a variety of applications. Z3 [14], Yices [15] and CVC4 [4] are the leading SMT Solvers in the 10th International Satisfiability Modulo Theories Competition (SMT-COMP 2015)[1].

In the event that solvers are unable to reason about complex program structures, a combination of concrete execution and symbolic execution can be used. This is called *concolic execution*, but is also known as *dynamic symbolic execution*.

> **Concolic Testing** uses a combination of *symbolic* and *concrete* execution to analyze program logic and generates input for test cases [49].

Concolic testing allows for the automatic generation of test input by constructing symbolic formulas with the help of *symbolic execution*. If the symbolic execution engine is unable to set up a symbolic formula for a part of the program, *concrete execution* is used. Finally, input is created based on the symbolic formulas with the help of *SMT Solvers*. The generated input from the SMT Solvers and input from the concrete execution are used to maximize the coverage of the explored execution paths in the program.

2.3 Oracle Problem

After traversing the various execution paths, it is important to determine if the program successfully handled the input or if it failed. This problem is known as the *Oracle Problem*.

> Given an input for a system, the challenge of distinguishing the corresponding desired, correct behavior from potentially incorrect behavior is called the **Test Oracle Problem** [3].

There are three approaches to establishing automated oracles: [3]

- **Specification** of the test oracle: the conditions for the test are formally specified using model-based specification, transitions systems, assertions and contracts, and algebraic specifications.
- **Derivation** of the test oracle: the conditions for the test are derived from artifacts such as documentation, system executions, properties of the SUT or different versions of the SUT.
- Creation of the test oracle from **implicit** information: these oracles do not require any domain knowledge or formal specification and apply to nearly all programs.

[1]`http://smtcomp.sourceforge.net/2015/results-summary.shtml`

A trivial solution to the Oracle Problem is the checking of unexpected termination (*implicit*) or throwing of null pointer exceptions (*implicit*). While this of course is a correct assumption, it dismisses most of the functionality of the application.

One method to generate Test Oracles automatically is by *deriving* specifications from documentation, an API or by using previous versions of the program.

Further precision can be improved by manually defining invariants (*specified*). These invariants are properties, which should not have changed during the execution. If one of these invariants was modified, the test fails.

Instead of first generating the input and deciding correct execution afterwards, a program's functional description can also be used to generate input. An example of this is QuickCheck [19], which is a tool that requires specification of properties the program should satisfy. After providing these specifications, QuickCheck randomly generates test-cases and tries to make the program fail.

While the oracle problem is an important issue for an automated testing framework, it falls out of our scope for this thesis. The focus in this thesis falls on event-sequence construction. For this reason we only make use of *implicit* oracles that check for the throwing of null pointer exceptions.

2.4 Event-Sequence Construction

Not only inputs influence the path of execution in a program, the sequence of events influence the execution path as well. Analyzing a program to create meaningful event-sequences is a non-trivial task. It is important that all the different execution paths that exercise different parts in the code of the SUT are covered. In doing so however a problem arises, called the event-sequence explosion problem.

> The **Event-sequence Explosion** refers to the problem, where there exist a non-feasible potentially infinite number of executable parameterized event-sequences that result in a potentially infinite set of program states in the SUT.

The solution to this problem is to only execute a selection of the possible event-sequences.

Approaches to this problem include:

- **Combinatorial selection:** out of the set of all possible combinations only select the combinations that test t-way sequence coverage event-sequences [24].
- **Search-based selection:** use search-based algorithms to generate event-sequences that are maximally diverse and differ in length [30].
- **Learning-based selection:** use state abstraction and create a model of the SUT. Select event-sequences that are not described by the model [12, 11].

Of these three, our approach will focus on learning-based selection. This selection approach works by trying to discover a relation between our provided input and the page's output. By looking at series of execution traces it is possible to model relationships between the input and output.

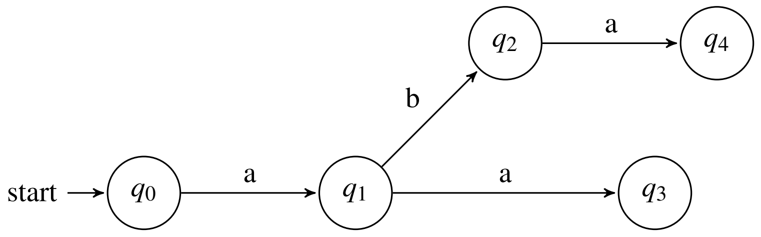
There are various ways in which a model can be represented. In the case of Hubble we decided to use a Deterministic Finite Automaton (DFA) for our model representation. DFA learning algorithms can be categorized into:

- *Active Model-Learning Algorithms* actively ask to execute specific traces to ensure the correctness of the model they are describing. The upside of this is that your model is guaranteed to be correct, the downside however is that this guarantee requires lots of traces to be executed.
- *Passive Model-Learning Algorithms* only use traces that were supplied to it to create a model. While the traces are described by the model, the model is not guaranteed to be correct.

Blue-Fringe

In order to construct a DFA we use a passive learning algorithm called the *Blue-Fringe* algorithm [25]. In this thesis we give an overview of the algorithm below. A thorough description and specification of the Blue-Fringe algorithm can be found in a paper by Verwer et al [51].

Prefix Tree Acceptor & State Abstraction The input for Blue-Fringe is a tree consisting of outputs and transitions between the outputs, also known as a Prefix Tree Acceptor (PTA). An example of a PTA can be found in Figure 2.2. For our model-learning purpose these outputs represent program states. In order to describe each program state, we use a *state abstraction*. Examples of the information on a page that can be used for state abstractions include the DOM-elements on a page, the JavaScript variables, or a set of enabled events. Transitions between the different states resemble the ways of getting from one program state to another program state. In our case these transitions are events fired to reach the different program states. These events are described by *event abstractions*. In our model, these event abstractions are described as parameterized events i.e. events with inputs used on the page.



$$\begin{tikzcd}
 & & q_2 \arrow[r, "a"] & q_4 \\
\text{start} \arrow[r] & q_0 \arrow[r, "a"] & q_1 \arrow[u, "b"] \arrow[rr, "a"] & & q_3
\end{tikzcd}$$

Figure 2.2: A Prefix Tree Acceptor (PTA) example for traces $\{a \rightarrow a, a \rightarrow b \rightarrow a\}$.

State Merging The PTA is passed to the Blue-Fringe algorithm to start the *state merging process*. In this process there are two operations:

- The MERGE operation merges two states with the same state abstraction into a super state and redirects the incoming and outgoing transitions via the newly created super state. After the merge, the algorithm checks for non-deterministic transitions and resolves the non-determinism by trying to merge the target states. The merge process is only halted once there are no more possible merges.
- Blue-Fringe has three sets of nodes: uncolored, blue and red nodes. The COLOR operation changes the color of uncolored nodes to blue nodes and blue nodes to red nodes. Initially all the nodes are uncolored. The first step is to color the root node red and it's children blue. Only blue states are considered merge candidates, red states are never modified. When a merge operation is applied to a blue node and there are no more possible merges for that node, the node's color is changed from blue to red and all of its uncolored children are colored blue.

After the above operations have been exhausted, there are no more possible merges and all the states are colored red. The result is a DFA that we can use as a model to guide the exploration.

Chapter 3

Objectives

To eliminate all forms of user-errors and guarantee correct behavior of the System Under Test (SUT), we aim to create an Automated Testing Framework. Our goal is to cover as much code of the SUT in a short amount of time. In order to keep the generated test-suite to a decent size and limit the time required to generate the test-suite, we need to avoid event-sequence explosions.

Throughout this document we refer back to these objectives to reason why certain decisions were made. In the Evaluation chapter we review the completion status of these objectives.

3.1 Primary Objectives

We aim to achieve our goal by using intelligent event-sequence generation to only execute event-sequences that increase coverage of the SUT. In order to reach our goals and keep a good overview we have set up the following primary, high priority objectives:

PO1 Explore a JavaScript Web Application and attempt to cover as much of the executable JavaScript code as possible.

- **PO1.1** Provide inputs that trigger different branches in functions.
- **PO1.2** Provide event-sequences that exercise event handlers and subsequently invoked methods.
- **PO1.3** Provide different event-sequence prefixes that cover uncovered branches inside of already partially covered functions.
- **PO1.4** Ensure that all the generated event-sequences can be triggered by a user.
- **PO1.5** Ignore external JavaScript libraries in the analysis.

PO2 Export event-sequences for easy replay of the generated sequences.

PO3 Avoid getting stuck in traversing possibly infinite sets of event-sequences.

- **PO3.1** Identify and ignore sequences that reach the same program state.

- **PO3.2** Only explore event-sequences of which the behavior is unknown.

PO4 Provide more targeted event-sequences compared to SymJS and competitors.

- **PO4.1** Perform coverage based analysis to establish event-handler targets.
- **PO4.2** Choose to target event-handlers depending on if they are uncovered due to different inputs vs. different event prefixes.

By completing these objectives, the created Automated Testing Framework will allow us to test as much of the JavaScript code of the SUT as possible. The primary objectives force our system to be competitive with state-of-the-art tools. Completing these objectives will result in a system that only executes uncovering event-sequences, terminates within a reasonable time-frame and provides a resulting test-suite that is of a decent size.

3.2 Secondary Objectives

Aside from the primary objectives, it is important that the development workflow is improved and that manual work is kept to a minimum. To give us more time to focus on the primary objectives, we have set up the following secondary, lower priority objectives:

SO1. Provide detailed and accurate coverage information.

- **SO1.1** Report Function, Branch and Line Coverage information.
- **SO1.2** Show the exact lines that were covered and how often they were hit.

SO2. Improve the development workflow.

- **SO2.1** Update Revision Control
- **SO2.2** Build System Setup
- **SO2.3** Set up Continuous Integration
- **SO2.4** Improve System Documentation

SO3. Provide an interface for Hubble.

- **SO3.1** Control the tool via a Web Interface.
- **SO3.2** Visualize run information to get a better overview.

SO4. Provide support for batch runs.

- **SO4.1** Easily configure Custom Login Information.
- **SO4.2** Easily run the tool on a large benchmark set.
- **SO4.3** Aggregate artifacts: logs, coverage results and event sequences.

The completion of these secondary objectives will allow us to quickly iterate and make new improvements to the system. Additionally it will provide us important information that can be analyzed for improvements and problems in the resulting framework. Providing support for batch runs will remove the focus from running the benchmarks on to thinking of the next improvements that can be made.

Chapter 4

HUBBLE's Architecture

The Automated JavaScript Testing Framework we created is called HUBBLE. In this chapter we describe its architecture. The chapter starts with a general overview of the project. It explains how we try to meet the objectives set in Chapter 3 and gives an explanation of the different modules, their roles in the system and how they interface.

4.1 Overview

HUBBLE is a corrected and improved continuation of the previous SYMJS [27] project. SYMJS is an Automated Testing Framework for JavaScript Web Applications created by Fujitsu Laboratories of America Inc. We build upon this work and aim to complete the primary objectives as follows:

- PO1: Existing work by SYMJS already fulfills most of this objective that consists of the basic functionality you would expect from an Automated Testing Framework.
- PO2: Although SYMJS had basic support for replay of generated test-cases, we added support in HUBBLE to run the test-cases directly in Selenium. This allows us to re-run generated test-cases for further analysis, such as in our coverage tool that is described in Chapter 6.
- PO3: SYMJS suffers from an explosion of the possible event sequences in a program. Resolving this issue has become one of the primary objectives of HUBBLE. HUBBLE aims to relieve this problem by selecting event-sequences based on a model that was created by a model learner.
- PO4: Using the learned model, function coverage information and fired events, we are able to provide more targeted event-sequences.

4.2 Components

In this section we present the different components of HUBBLE. An overview of these components can be found in Figure 4.1. We first present each individual component with a description of how the component helps HUBBLE achieve its objectives. After we have presented the individual components, we bring them into context by presenting a scenario and explaining how the components work together to create HUBBLE.

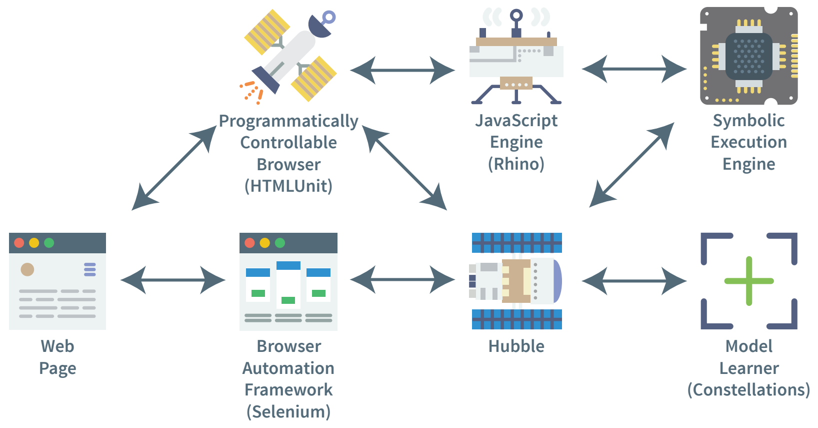

Programmatically Controllable Browser (HTMLUnit)
JavaScript Engine (Rhino)
Symbolic Execution Engine
Web Page
Browser Automation Framework (Selenium)
Hubble
Model Learner (Constellations)

Figure 4.1: HUBBLE Architecture - Overview

4.2.1 Programmatically Controllable Browser

In Figure 4.1 the Programmatically Controllable Browser on the top left is a browser that can be operated and queried from the code. Preferably we would use Selenium for this, however we need to add hooks and behavior that is deep inside of the JavaScript Engine. This prevents us from using Selenium, since it only has limited instrumentation capabilities.

A browser that satisfied our requirements is HTMLUnit[1]. HTMLUnit has methods to emulate different popular browsers such as Internet Explorer, Firefox and Chrome and provides an API to control the browser and query information. HTMLUnit is directly controllable by HUBBLE via a Java interface.

4.2.2 Browser Automation Framework

The Programmatically Controllable Browser is headless and thus does not physically render the page. It does not provide us the information regarding the visibility or interactibility of Document Object Model (DOM) elements. Reasons why the visibility or interactibility of an element can be obstructed include:

- Other elements that are overlapping the target element.
- The target element is hidden due to styling rules.
- The target element is outside of the window view-port.

[1]HTMLUnit: `http://htmlunit.sourceforge.net/`

- The element is disabled.

If we fire an event on the target element even though one of the above conditions is present, we enter a state that a user would not be able to reproduce. In other words, we would start to explore unrealistic state-space within the program. To remedy this problem, we use both a headless and a real browser, and play each scenario in both browsers, keeping them in sync.

The Programmatically Controllable Browser is needed to provide instrumentation capabilities for the coverage analysis and symbolic execution. Whereas the Browser Automation Framework is needed to ensure that the determine interactibility and visibility of elements to avoid unrealistic event-sequences (PO1.4). Both communicate directly with HUBBLE and do not interact with each other directly as can be seen in Figure 4.1.

The role of the real browser(s) in the Hubble architecture is fulfilled by the Browser Automation Framework. The tool we use is Selenium[2], since it is a well respected and widely used framework in the Automated Testing Community [8]. Selenium provides a Java interface via the Selenium-WebDriver[3]. HUBBLE uses this interface to control and query Selenium.

4.2.3 JavaScript Engine

A JavaScript Engine is a piece of software that is used to compile and execute JavaScript code. Every browser that supports JavaScript has a JavaScript Engine. To name a few: Mozilla Firefox uses SpiderMonkey[4], Google Chrome uses V8[5], Microsoft Edge uses Chakra[6].

Rhino[7] is the JavaScript Engine for HTMLUnit and is included as a Java dependency. We forked Rhino and made various modifications to the parser and interpreter to hook in our Symbolic Engine and perform coverage analysis (PO4.1).

4.2.4 Symbolic Engine

SymJS includes a custom-build symbolic execution engine for JavaScript [27]. This Symbolic Engine allows us to set up and solve path constraints used to generate inputs that trigger different branches in functions (PO1.1, PO1.2). The Symbolic Engine includes a *Parameterized Array String Solver (PASS)* [28] that allows us to reason about string constraints in Yices, an SMT Solver, since Yices 1.0 was not able to handle string constraints. We decided against updating to Yices 2.5.1 since the development effort involved would outweigh the benefits due to the large amounts of modifications this would require in SymJS's code-base.

[2]Selenium: `http://www.seleniumhq.org/`

[3]Selenium-WebDriver: `http://www.seleniumhq.org/docs/03_webdriver.jsp#java`

[4]SpiderMonkey: `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`

[5]V8: `https://developers.google.com/v8/`

[6]Chakra: `https://github.com/Microsoft/ChakraCore`

[7]Rhino: `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino`

The Symbolic Engine is a component within HUBBLE. It uses a combination of writing to files and Java Native Interfaces (JNI) for the communication with Yices, the SMT solver that is used to solve for constraints.

4.2.5 HUBBLE

HUBBLE is the orchestrator of the Automated Testing Framework. It controls the browsers, obtains information from the browsers, passes the information to the Symbolic Engine and forms constraints that are solved by an SMT Solver. HUBBLE keeps track of the event, function, branch and line coverage of the SUT. It passes the executed traces on to the model-learner. HUBBLE then guides the exploration based on all the coverage information of the non-library (PO1.4) code and asks the model-learner for new prefixes that are likely to cover uncovered code (PO3.1, PO3.2). HUBBLE is the entry-point and is completely written in Java and can be run from the terminal or web interface.

HUBBLE provides easy configuration for running batch runs (SO4). Aside from the exploration Uniform Resource Locator (URL), the user can specify a login URL with login details. (SO4.1) This allows us to run the tool on a large benchmark set (SO4.1). HUBBLE provides the aggregated coverage results, logs and executed sequences for each exploration in an output directory (SO4.3).

4.2.6 Model-Learner

The model-learner, Constellations, is written as a microservice that runs independently from HUBBLE. It communicates only with HUBBLE directly as can be seen in Figure 4.1. Constellations receives a set of traces, creates a model and provides newly created prefixes to reach specified target states (PO3 and PO1.3). The model learner exposes two interfaces:

- A REST service provides an interface for the control of the model-learner.
- A WebSockets service is used to provide a debugging interface to the user (SO3.1).

Constellations is discussed in detail in Chapter 5.

REST Service

The REST service in the model-learner is an interface that receives the traces from HUBBLE, runs the model-learning, generates different prefix paths and presents the created artifacts.

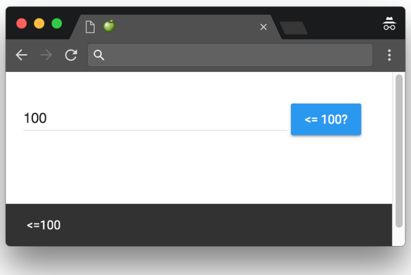
WebSockets Interface

The Model-Learner provides a WebSockets interface to accommodate live inspection of the created model during exploration (SO3.2). The choice for a WebSocket interface was made, since the inspection tool requires to be updated from the server side. A REST API would only provide a way if the updates were periodically requested by the inspection tool. With the WebSocket interface, the model-learner can send updates directly to the inspection tool, making the model inspection real-time.

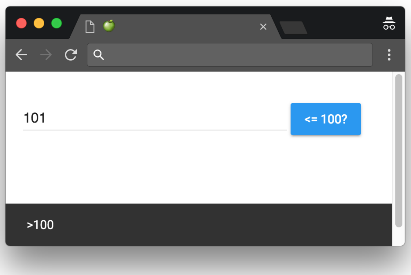
4.3 Example Execution Scenario

Loading a page and firing a parameterized event.
To get an understanding of how the system works, let us have a look what the modules do when HUBBLE loads a target page and fires an event. To follow along, a general overview of the different components is given in Figure 4.1.

Assume that the example page contains an input field and a single button that allows two possible program states as shown in Figure 4.2.



(a) Fill in a number that is less than or equal to 100 and push the button, which results in an alert displaying '$<=$ 100'.



(b) Fill in a number that is greater than 100 and push the button, which results in an alert displaying '$>$ 100'.

Figure 4.2: Possible reachable program states in the example page.

HUBBLE instructs both the Programmatically Controllable Browser and Browser Automation Framework to navigate to the target page. Both load the target page. During the loading, the Programmatically Controllable Browser instructs the JavaScript Engine to parse and interpret all the JavaScript code. In the JavaScript Engine we have made modifications to hook up a Symbolic Engine. The Symbolic Engine keeps track of all the different execution paths and sets up input constraints during the interpretation of the JavaScript code.

The Symbolic Engine then creates two symbolic states for the two different execution paths. It executes one of the symbolic states, where it solves for the constraint using an SMT solver and obtains an input value $v_1 = 100$. Now HUBBLE uses v_1 as an input value. HUBBLE instructs the Programmatically Controllable Browser and Browser Automation Framework to fill in v_1 in the input field.

Before it presses the button, HUBBLE queries the Browser Automation Framework to check for the visibility and interactibility of the element it wants to fire an event on, the button in this case. If the Browser Automation Framework returns that it is possible to fire such an event we fire the event simultaneously in the Programmatically Controllable Browser and Browser Automation Framework.

The result is that we have an executed symbolic state and we are able to send the corresponding trace to the model-learner. The model-learner incorporates the trace into the model and provide feedback back to HUBBLE to decide what symbolic state to execute next. The symbolic state selection prioritization using the model-learning is discussed in Chapter 5.

HUBBLE stops its exploration after one of the following pre-configured halt-conditions is met: time-limit, coverage-goal, maximum number of states, or number of iterations since the last coverage increase. After the exploration is stopped, HUBBLE exports all of the results in an output directory and terminates.

4.4 Development Pipeline

One of the secondary objectives was to improve the development workflow (SO2). In order to improve the development workflow we have made various changes to the project setup. SYMJS is a legacy Eclipse project written in Java. Part of the development team used Apache Subversion for revision control and another unit within the development team used Git. The tool would not run straight out of the box; it would take quite some configuration before the tool would be able to run.

The HUBBLE Project is now a Maven project (SO2.2) written in Java and hosted in a Git repository (SO2.1) on a private GitLab installation. The HUBBLE project uses an Automated Build Server (SO2.3), Jenkins, to perform regression-testing. Automatically before any pull request is merged, the build server makes sure that all tests pass.

SYMJS absorbed two external dependencies, HTMLUnit and Rhino, that have been merged into the project. This resulted in a very large code-base with non-test sources of the project amounting to 203.283 lines of Java code[8] out of the 431.374 total lines of Java code[9] in the project. Since the dependencies were subjected to partial manual updates, there is no reasonable cost-effective way of extracting these external dependencies. This makes it very challenging for a single person to get a good overview of the project and emphasizes the importance of having good regression tests. Having a continuous integration system set up greatly helps us to ensure that the changes made to the system were non-breaking.

[8]According to cloc, when invoked with '`cloc --exclude-dir=test src`'.

[9]According to cloc, when invoked with '`cloc .`'.

Chapter 5

Model-Learning

In this chapter we present the reasons why we use model-learning and how we apply the it in our model-learner, Constellations, for HUBBLE.

One of the Primary Objectives, PO3, is to avoid state-spaces with infinite possibilities of event-sequences, and execute only a small subset of sequences that suffice to exercise all the code for the given state-space.

5.1 State Abstraction

One of the sub-objectives that should be done to complete PO3 is PO3.1, which requires us to identify and ignore duplicate event-sequences that reach the same program state. In order to describe a program state, we need some form of state abstraction. This state abstraction is a representation that contains enough data to distinguish the different program states. Once a state abstraction has been created, we are able to create a cache of previously seen program states corresponding to that state abstraction. It is important to realize that this requires a balance:

- Using a too fine-grained state abstraction results in lots of states that do not differ in the code coverage that is executed, which results in a possibly infinite state-space.
- Using a too loosely defined state abstraction combines program states in the same abstracted state. This can result in an unexplored area of the state-space, due to the fact that the exploration device has the belief that it has already seen the state.

The possibility exists that a state abstraction does not describe the program state. For example:

Assume that a state abstraction consists of the set of available actions and a DOM representation. If an action changes the value of a JavaScript variable, one of the things that is not captured in the state abstraction, we have reached a different program state, but have no way to represent it in our model.

We can deal with undescribed behavior, but it would require us to describe the paths used to reach each of the program states. Rather than using a cache, what we would need to use here is a model that is described by a Deterministic Finite Automaton (DFA).

5.2 Model Creation

As we saw in the end of the previous section, using a simple cache of abstracted states is not sufficient to create a good representation of the program. In order to create a more complete representation, we need to know what traces lead up to the state so we can distinguish states not only by their abstraction, but also by the actions that were used to reach it. Adding the firing of parameterized events as transitions in a graph allows us to model program states that are a result of firing an action, but where there is no differences in the content of the state abstraction. This allows us to have states with the same state-abstraction content in our model, even though they represent different program states.

Equivalent states that are believed to reach the same program state are merged into a SuperState. The SuperState is a set of equivalent states. The nodes of the resulting automaton are the SuperStates and the edges are the fired parameterized events that are required to transition from one state to another state.

In Section 2.4 we discussed the Blue-Fringe Algorithm DFA inference algorithm. Blue-Fringe is a passive model-learning algorithm. Passive model-learning algorithms do not ask for the results of specific traces, like active model-learning algorithms do. They only use the provided traces to build a model. The choice for a passive model-learning algorithm was made since active model-learning algorithms want an exact model and therefore want to obtain traces that prove the correctness of the model. Using an active model-learning algorithm would require the execution of a much larger set of traces. Since we prefer a faster model-learning technique and because we only require an approximate model to guide the exploration, we choose to use a passive model-learning algorithm.

5.3 Integration with Hubble

Below are a few selection groups for the traces that HUBBLE wants to execute. The selection of the trace that should be executed next works on a priority basis: if there are no traces matching the top condition, HUBBLE tries to find trace that matches the subsequent conditions. The conditions are prioritized in decreasing priority as follows:

1. *Uncovered Events:* check a cache to see if a string representation tuple, consisting of an id and type of event (‘`value::event_type`’, eg. ‘`login::click`’), has been fired before.

2. *Model-Learning Targets*: obtain a function target that has the highest number of uncovered lines and has been pursued less than *n* times.

   a) *If the target function has branches depending on input:* try to execute symbolic states that provide different inputs that exercise the different branches.

   b) *If the target function has branches depending on event-sequences:* obtain all of the traces that previously exercised the function. These traces are sent to the model-learner. The model-learner then uses these traces to select the corresponding model states. The model learner provide prefixes that are likely to cover the branches. Symbolic states corresponding to these prefixes are then selected to be executed.

3. *States containing Symbolic Values:* select any state that contains a path condition and has symbolic variables that were solved.

4. *Remaining states:* execute any state that is still in the queue.

These four conditions allow us to prioritize which traces get execute. They allow us to select traces that have a high likelihood of covering thus-far-uncovered code.

Chapter 6

Coverage Analysis

During the development of HUBBLE we needed a coverage tool that would allow us to compare between different automated testing tools. At first JSCover seemed like a good option, however quickly it turned out that due to the method it measures coverage, by inserting statement around every line being executed. This slows down the JavaScript execution dramatically. For small benchmarks this is negligible. However, for the larger benchmarks we find that the execution slows down to the point that it halts and affects the JavaScript execution.

Therefore, we decided to create our own coverage tool with the following requirements:

CT1 Replay exported event-sequences from:

- **CT1.1** HUBBLE
- **CT1.2** SYMJS
- **CT1.3** Crawljax
- **CT1.4** Artemis

CT2 Provide detailed coverage information:

- **CT2.1** Branch coverage
- **CT2.2** Line coverage
- **CT2.3** Function coverage
- **CT2.4** Line execution count

CT3 Minimal effect on JavaScript execution.

CT4 Easily readable output.

With these requirements in mind we constructed a tool built on top of HTMLUnit. The coverage tool takes a JavaScript Object Notation (JSON) formatted file that has all the event-sequences from the generated test cases.

We have added modules to Hubble, Crawljax and Artemis such that we can export the generated event-sequences from the test-cases to a uniformly formatted file that can be read by the coverage tool. This exported file allows us to re-run all of tests exported from the tools and produces a coverage report in that can be read by LCOV's genhtml tool.

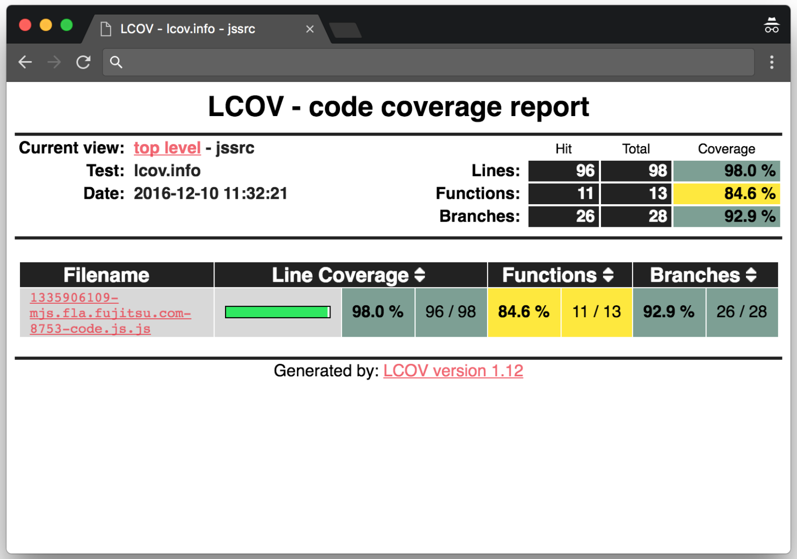
Exporting these sequences and re-running them on our own HTMLUnit coverage tool ensures that:

- Coverage is measured in the same way throughout all the tools.
- Coverage analysis does not affect the exploration.
- Ensure that tests-suites are correctly exported and coverage is not incorrectly reported.

Since traditional tools such as JSCover and Istanbul perform coverage analysis by inserting a statement after each line is executed, the to be instrumented code is modified which can cause side-effects during the analysis. One of these side-effects was a very noticeable slow-down in the JavaScript execution. Our HTMLUnit coverage tool instead works as a browser that captures coverage information inside the JavaScript interpreter. This not only takes less time to run all the test cases, but also ensures that no JavaScript code is affected or modified, as all the analysis is done on interpreter level, rather than intercepting each JavaScript function call.

In Figure 6.1 we get an overview of all the involved source files with their individual and aggregate coverage information. The branch, line and function coverage can be found in this overview.

In Figure 6.2 a detailed source-file view is shown. It shows the line count and branch coverage information.



LCOV - lcov.info - jssrc

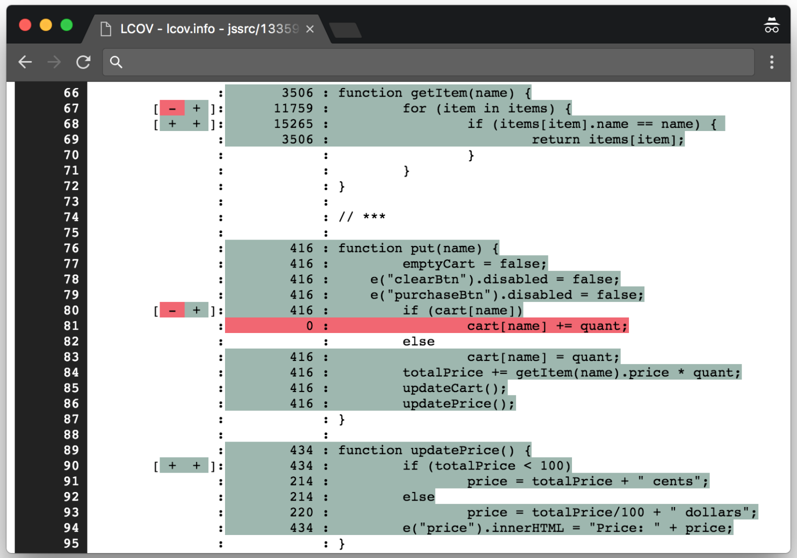
LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - jssrc				
Test:	lcov.info	**Lines:**	96	98	98.0 %
Date:	2016-12-10 11:32:21	**Functions:**	11	13	84.6 %
		Branches:	26	28	92.9 %

Filename	Line Coverage			Functions		Branches	
1335906109-mjs.fla.fujitsu.com-8753-code.js.js	🟩	98.0 %	96 / 98	84.6 %	11 / 13	92.9 %	26 / 28

Generated by: LCOV version 1.12

Figure 6.1: Coverage Overview



LCOV - lcov.info - jssrc/13359

```
                 :     3506 : function getItem(name) {
      [ -   + ]:    11759 :         for (item in items) {
      [ +   + ]:    15265 :                 if (items[item].name == name) {
                 :     3506 :                         return items[item];
                 :          :                 }
                 :          :         }
                 :          : }
                 :          :
                 :          : // ***
                 :          :
                 :      416 : function put(name) {
                 :      416 :         emptyCart = false;
                 :      416 :     e("clearBtn").disabled = false;
                 :      416 :     e("purchaseBtn").disabled = false;
      [ -   + ]:      416 :         if (cart[name])
                 :        0 :                 cart[name] += quant;
                 :          :         else
                 :      416 :                 cart[name] = quant;
                 :      416 :         totalPrice += getItem(name).price * quant;
                 :      416 :         updateCart();
                 :      416 :         updatePrice();
                 :          : }
                 :          :
                 :      434 : function updatePrice() {
      [ +   + ]:      434 :         if (totalPrice < 100)
                 :      214 :                 price = totalPrice + " cents";
                 :      214 :         else
                 :      220 :                 price = totalPrice/100 + " dollars";
                 :      434 :         e("price").innerHTML = "Price: " + price;
                 :          : }
```

Figure 6.2: Coverage Body

Chapter 7

Evaluation

In this chapter we evaluate HUBBLE by running different exploration techniques and comparing HUBBLE to other state-of-the-art automated testing tools. More specifically we evaluate the importance of symbolic execution, model-learning and the combination of both. Additionally we evaluate the applicability of these techniques on real-world large web applications and compare our results with the well-known Crawljax and Artemis automated testing frameworks.

The research question that will be answered is:

Does the combination of model-learning with symbolic execution improve the event-sequence generation used to create test-cases for JavaScript web applications?

Our main research question will be subdivided in the following sub-questions:

RQ 1 How does the performance of the model-learning technique compare to symbolic-execution in terms of coverage for Enterprise Web Applications and General Web Applications?

RQ 2 How do the size and total coverage of the generated test-suites using the new Hubble technique compare to the test-suites generated by the previous SymJS technique for Enterprise Web Applications and General Web Applications?

RQ 3 How do the size and coverage of test-suites generated by HUBBLE compare to the test-suites generated by state-of-the-art tools Artemis and Crawljax?

7.1 Execution Modes

RQ 1 and RQ 2 require us to compare different exploration and input-generation techniques. We have added three execution modes that only differ in event-sequence construction and input generation techniques that allow us to compare the different techniques. The three execution modes are:

- **SymJS**: provides the SymJS symbolic-execution functionality as is described in [27]. It includes development improvements that were made such as interactibility checking.
- **Model-only**: provides a technique that uses model-learning. Instead of using values produced with symbolic execution, it uses random values for all input available on a page whenever an event is fired.
- **Hubble**: provides trace selection using model-learning combined with symbolic execution as described in Chapter 5.

The 'SymJS' and 'Model-only' execution mode allows us to analyze the difference between symbolic-execution and model-learning. This allows us to evaluate the performance of symbolic-execution in terms of coverage for HUBBLE. We use this execution mode to answer RQ 1.

The choice of execution mode is made before the benchmark run. SymJS and Hubble are used to compare the old SymJS technique to the new Hubble technique, thus allowing us to answer RQ 2.

7.2 Benchmarks

In order to evaluate our technique and compare our tool to other techniques, we have selected two benchmark sets to facilitate the evaluation:

- **Enterprise Web Applications** (Table 7.1): consists of web applications that are used in a business setting and often require specific input values and a series of steps to explore the full program. Our set consists of four open-source enterprise applications and two smaller shopping benchmarks.
- **General Web Applications** (Table 7.2): consists of web applications that are used by communities, individuals and businesses on the web. This includes blogging, bulletin-boards and other software. Our set contains three popular web applications: forum software, blogging software and a content-management system. Additionally we have added five systems that were used in the evaluation of Artemis [1].

Table 7.1: Business Application Benchmarks

Name	JS LOC	Total LOC	Description	URL
Shopping Cart	98	152	A single page shopping cart example that allows you to add and remove items from a cart and go through a checkout process.	n/a
Shopping List	364	42008	A multi-page shopping list application that allows you to keep track of your shopping list.	n/a
Snipe-It	15927	336104	An open-source asset management system.	https://github.com/snipe/snipe-it
IceHrm	8242	463006	A human resources management system.	https://github.com/gamonoid/icehrm
OrangeHRM	603	549096	A human resources management system.	https://sourceforge.net/projects/orangehrm/
Collins	306	67760	An infrastructure management system created by Tumblr.	https://tumblr.github.io/collins/

Table 7.2: General Application Benchmarks

Name	JS LOC	Total LOC	Description	URL
Ajax Poller	16	2313	An application that allows voting and administering of polls.	http://www.brics.dk/artemis/examples/
Ajax Tabs Content	160	418	An application that loads content depending on various ways of selecting tabs.	http://www.brics.dk/artemis/examples/
Dynamic Articles	154	382	An application where articles are loaded via Ajax depending on the article title pressed.	http://www.brics.dk/artemis/examples/
Fractal Viewer	755	4286	A fractal viewing application.	http://www.brics.dk/artemis/examples/
Homeostasis	2040	3311	A visualization tool to explain homeostasis.	http://www.brics.dk/artemis/examples/
phpBB	1740	260057	An open-source forum system.	https://github.com/phpbb/phpbb
Wordpress	203	311770	An open-source content management system.	https://github.com/WordPress/WordPress
Joomla	416	521972	An open-source content management system.	https://github.com/joomla/joomla-cms

7.3 Comparing HUBBLE Execution Modes

Enterprise Web Applications

Below we present the benchmarks results. Each graph shows the results for the benchmark mentioned in the graph title. The graph shows how the coverage increases over time. The differently colored lines indicate the different execution modes:
(*SymJS*: symbolic-execution-only, *Model-only*: model-learning-only, *Hubble*: symbolic+model).

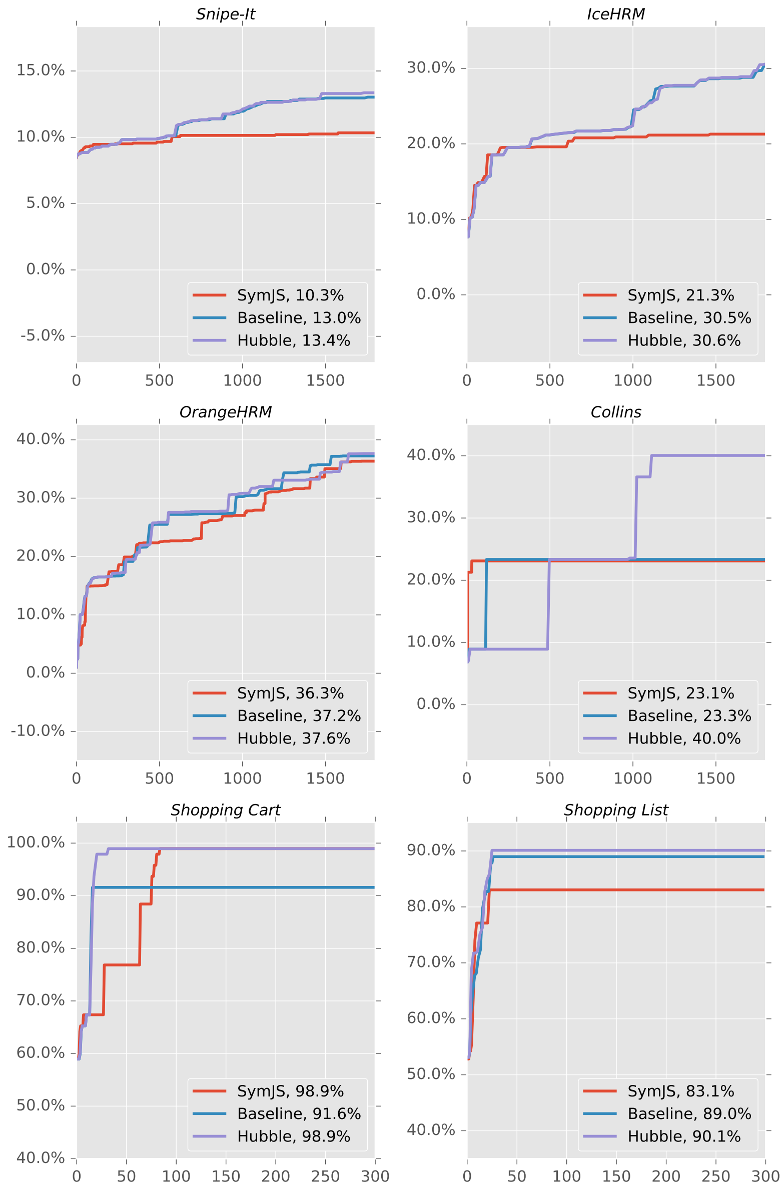


Figure 7.1: Coverage vs. Time - Enterprise Web Application Benchmarks

Snipe-It - Open Source Asset Management	**Line Coverage:**	**Trace Size:**
SymJS does not cover the management of status labels, while the Model-only and Hubble techniques do. Hubble is able to gain slightly more coverage due to using appropriate values.	SYMJS: 10.3%	426
	MODEL-ONLY: 13.0%	196
	HUBBLE: 13.4%	151
IceHrm - Online HR Software	**Line Coverage:**	**Trace Size:**
SymJS fails to discover a travel module that includes a file upload module. Model-only and Hubble have discovered this module due to the model-learning and started to cover its functionality.	SYMJS: 21.3%	151
	MODEL-ONLY: 30.5%	89
	HUBBLE: 30.6%	90
OrangeHRM - Human Resource Management	**Line Coverage:**	**Trace Size:**
SymJS fails to discover a time sheet and attendance module due to its focus on solving for input values rather than exploration. Hubble uses appropriate value types for the required inputs and has therefore slightly higher coverage than the Model-only method.	SYMJS: 36.3%	397
	MODEL-ONLY: 37.2%	217
	HUBBLE: 37.6%	45
Collins - Infrastructure Management	**Line Coverage:**	**Trace Size:**
Model-only and SymJS both fail to look at detailed asset pages that are harder to reach. Hubble uses an empty string as default if there are no constraints, which in this case allows it to find a list of modules which covers more code.	SYMJS: 23.1%	20
	MODEL-ONLY: 23.3%	446
	HUBBLE: 40.0%	152
Shopping Cart	**Line Coverage:**	**Trace Size:**
Model-only uses random values and is only able to cover part of the form validation code. SymJS and Hubble both cover the form validation code. Hubble reaches maximum coverage quicker due to the prioritization heuristics used with the model-learning.	SYMJS: 98.9%	86
	MODEL-ONLY: 91.6%	197
	HUBBLE: 98.9%	165
Shopping List	**Line Coverage:**	**Trace Size:**
SymJS stays on a single page trying to solve for input values. Hubble and Model-only explore the rest of the exploration. Hubble covers slightly more code by creating an item twice which triggers a merge operation in the system.	SYMJS: 83.1%	19
	MODEL-ONLY: 89.1%	94
	HUBBLE: 90.1%	17

Table 7.3: Case Studies - Enterprise Web Application Benchmarks

General Web Applications

In Figure 7.2 we present the general web application benchmark results for the different execution modes:
(*SymJS*: symbolic-execution-only, *Model-only*: model-learning-only, *Hubble*: symbolic+model).

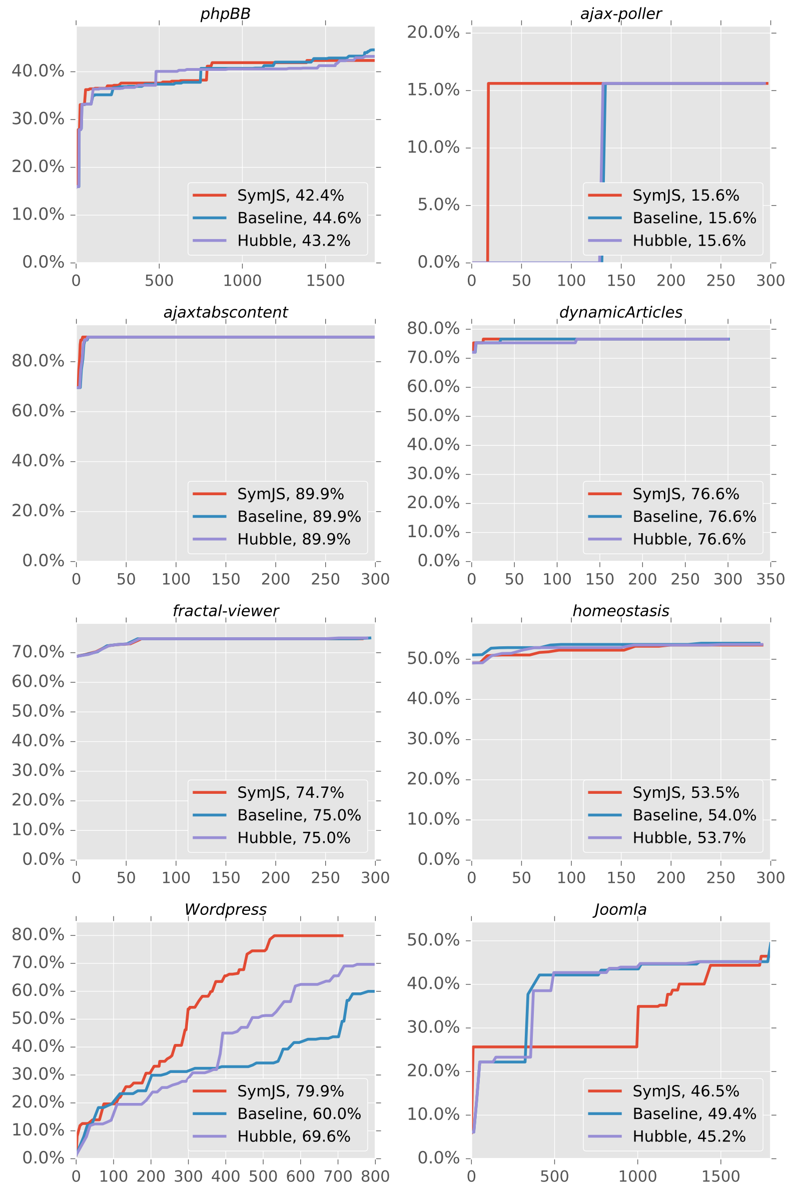


Figure 7.2: Coverage vs. Time - General Web Application Benchmarks

phpBB - Free and Open Source Forum Software	**Line Coverage:**	**Trace Size:**
Most of the code being covered is by executing events and does not require specific input values. The coverage plot shows that the techniques do not show a major differences in coverage.	SYMJS: 42.4%	312
	MODEL-ONLY: 44.6%	446
	HUBBLE: 43.2%	201
Ajax Poller	**Line Coverage:**	**Trace Size:**
This very small application just requires one specific event to be fired. Hubble and Baseline try to fire a range of other events before executing the right event, SymJS fires the event earlier.	SYMJS: 15.6%	168
	MODEL-ONLY: 15.6%	55
	HUBBLE: 15.6%	58
Ajax Tabs Content	**Line Coverage:**	**Trace Size:**
A small application with only a limited number of events. The techniques have fairly similar results.	SYMJS: 89.9%	470
	MODEL-ONLY: 89.9%	162
	HUBBLE: 89.9%	158
Dynamic Articles	**Line Coverage:**	**Trace Size:**
A small application with only a limited number of events. The techniques have fairly similar results.	SYMJS: 76.6%	625
	MODEL-ONLY: 76.6%	145
	HUBBLE: 76.6%	121
Fractal Viewer	**Line Coverage:**	**Trace Size:**
A small application with only a limited number of events. The techniques have fairly similar results.	SYMJS: 74.7%	27
	MODEL-ONLY: 75.0%	26
	HUBBLE: 75.0%	25
Homeostasis	**Line Coverage:**	**Trace Size:**
A small application with only a limited number of events. The techniques have fairly similar results. Model-only has slightly higher coverage since it fired more events.	SYMJS: 53.5%	30
	MODEL-ONLY: 54.0%	26
	HUBBLE: 53.7%	25
Joomla! - Content Management System	**Line Coverage:**	**Trace Size:**
Hubble and Model-only reach a plateau quicker after which it gets harder to increase coverage. Model-only in the last moment fires an event that covers more of the code.	SYMJS: 46.5%	53
	MODEL-ONLY: 49.4%	80
	HUBBLE: 45.2%	80
Wordpress - Blog Tool, Publishing Platform, and CMS	**Line Coverage:**	**Trace Size:**
In this benchmark it is important to fire a specific event sequence to uncover a large part of the code. SymJS does this slightly quicker than the other two. SymJS crashes at 700s when analyzing difficult constraints containing HTML elements.	SYMJS: 78.9%	229
	MODEL-ONLY: 79.3%	234
	HUBBLE: 72.4%	67

Table 7.4: Case Studies - General Web Application Benchmarks

7.3.1 Observations

Enterprise Web Applications

In Table 7.3 we have listed the test-suite results and our individual observations of the execution mode benchmark runs for Enterprise Web Applications.

Total Coverage: When comparing model-learning-only and SymJS in Figure 7.1, the shopping cart benchmark show a 7.2% increase in coverage when using symbolic-execution. However, this difference in not found in the other benchmarks. Snipe-it, IceHRM, Collins and Shopping List show a substantial (1-16.9%) improvement when using model-learning-only compared to SymJS.

When comparing Hubble and SymJS we see that Hubble outperforms SymJS in the Snipe-It, IceHRM, Collins and Shopping List benchmarks. In the OrangeHRM and Shopping Cart benchmarks the two techniques perform similarly. Thus in 4 out of 6 benchmarks, Hubble gets better coverage than SymJS. In all of the benchmarks, Hubble performs at least as well as SymJS.

Coverage increase over time: The coverage-time plots in Figure 7.1 show in 5 out of the 6 benchmarks that model-learning-using techniques reach coverage slightly quicker or in the same time as SymJS.

Test-suite Size: The results show that in 3 out of the 6 benchmarks Hubble covers more code with a smaller set of traces. For the other 2 out of 3, Hubble covers more code, but uses a larger set of traces. In the remaining shopping cart example Hubble uses a larger set of traces, while obtaining the same coverage.

General Web Applications

In Table 7.4 we have listed the test-suite results and our individual observations of the execution mode benchmark runs for General Web Applications.

Total Coverage: In Table 7.4 we see that there is not much difference in the Artemis benchmarks: ajax-poller, ajaxtabscontent, dynamicArticles, fractal-viewer and homeostasis. Looking at phpBB, all techniques perform similarly. In the Wordpress benchmark, SymJS gets better coverage than the model-learning techniques. In the Joomla benchmark the techniques perform similarly as well in terms of total coverage.

Coverage increase over time: In phpBB and the Artemis benchmarks, the three execution modes behave quite similarly. In Wordpress, SymJS has a greater coverage increase over time and in Joomla Hubble is performing better. There is no clear difference in coverage increase over time for the General Web Applications.

Test-suite Size: In 6 out of 8 benchmarks, Hubble has the same or better coverage with a smaller test-suite. In Joomla, Hubble has a larger test-suite and covers less code. In Wordpress, Hubble has a smaller test-suite and covers less code.

7.3.2 Evaluation

RQ 1 *How does the performance of the model-learning technique compare to symbolic-execution in terms of coverage for Enterprise Web Applications and General Web Applications?*

In terms of total coverage, we see a substantial improvement when using model-learning-only over SymJS. The only benchmarks that show better results using symbolic execution are Wordpress and Shopping Cart. We believe that this is due to the fact that a relatively small amount of real-world application logic is hard-coded in the JavaScript code. This type of logic often resides server-side and is accessed either via a page or Ajax request. As SymJS is unable to access these constraints, it is unable to create relevant input values and therefore random values work just as well. Due to the model-learning we are able to explore a larger part of the application and are thus able to cover more code.

In terms of coverage increase, we notice a negligible improvement for Enterprise Web Applications, but not for General Web Applications. This is probably due to the fact that most of the time is spent exploring the first few pages, which results in the same coverage increase. It is not until later in the exploration process that the difference in exploration techniques becomes noticeable.

The results indicate that although the coverage-increase over time is not significantly improved, the model-learning-only method has the same or better line-coverage than the symbolic-execution-only technique in 12 out of 14 benchmarks.

RQ 2 *How do the size and total coverage of the generated test-suites using the new Hubble technique compare to the test-suites generated by the previous SymJS technique for Enterprise Web Applications and General Web Applications?*

In 12 out of 14 benchmarks the new Hubble technique has obtained the same or better coverage compared to the previous SymJS technique. In 9 out of the 14 benchmarks, Hubble has the same or better coverage with a smaller test-suite compared to SymJS.

7.4 Comparing with State-of-the-Art

In the previous section we compared symbolic-execution, model-learning and the combination of both techniques. In this section we compare symbolic-execution combined with mode-learning in HUBBLE to two state-of-the-art Automated Testing Frameworks for JavaScript Web Applications. The two state-of-the-art frameworks are 'Crawljax' and 'Artemis'. Both of these frameworks were presented in the Related Work (Chapter 9.2).

7.4.1 Run Configuration

The Automated Testing Frameworks are run on the two benchmark sets and are only given enough information to log in to the web applications. This includes credentials and xpaths that are required for login.

Each of the frameworks has been configured to automatically login and clear its cookies before each trace. Every benchmark has been containerized in Docker. Before each run, the Docker container is reinitialized. This guarantees the same initial application state for each benchmark.

7.4.2 Coverage Measurement

Each of the frameworks gives a resulting trace file that allows us to replay the traces and uniformly measure the coverage using our coverage tool as described in Chapter 6.

7.4.3 Observations

In this section we analyze the results that are presented in Table 7.5.

Total Coverage

Table 7.5 shows that in 9 out of 14 benchmarks Hubble has the highest coverage or shares the highest coverage with one of the other tools. In 6 out of 14 Crawljax obtains the highest coverage. Artemis has the highest coverage or shares the highest coverage in 4 out of 14 benchmarks.

Enterprise Web Application Benchmarks

We see that in the smaller example Hubble performs best, then Artemis and then Crawljax. However, the test-suites created by Crawljax are much smaller. The test-suites created by Artemis are the largest.

Artemis Benchmarks

In Table 7.5 we see that for the Artemis benchmarks, HUBBLE performs similarly to Artemis in coverage. Crawljax performs the worst, as expected since most of these benchmarks are single page and Crawljax is more made to crawl larger web applications. This is the reason why out of the five benchmarks there are three where it only loads the page and terminates.

General Web Application Benchmarks

The General Web Applications show that Crawljax outperforms HUBBLE and Artemis with a reasonable set of traces. Most likely this is due to initialization code that is being covered by reaching new pages.

7.4.4 Evaluation

RQ 3 *How do the size and coverage of test-suites generated by* HUBBLE *compare to the test-suites generated by state-of-the-art tools Artemis and Crawljax?*

We see that Crawljax uses a low number of traces to provide decent coverage. Artemis uses a large number of traces to obtain its coverage. HUBBLE is in the middle, as it provides good coverage, but keeps it test-suite to a reasonable size.

Hubble obtains the highest coverage for most of the benchmarks, followed by Crawljax. Artemis contains the highest coverage for the least amount of benchmarks.

	Artemis				**Crawljax**				**Hubble**			
	Coverage				Coverage				Coverage			
	Line	Branch	Func.	TSize	Line	Branch	Func.	TSize	Line	Branch	Func.	TSize
Shopping Cart	92.9%	70.4%	84.6%	350	73.7%	51.9%	53.9%	5	97.0%	88.9%	84.6%	184
Ajax Tabs Content	89.3%	76.1%	80.0%	128	89.6%	79.1%	80.0%	18	89.0%	82.1%	76.0%	164
Dynamic Articles	82.3%	66.0%	58.3%	131	33.1%	8.0%	16.7%	1	82.3%	66.0%	58.3%	116
Fractal Viewer	72.6%	70.9%	63.2%	117	53.6%	45.0%	50.4%	1	69.2%	65.2%	61.6%	25
Homeostasis	58.2%	48.1%	37.1%	107	41.7%	27.3%	23.0%	1	59.6%	49.9%	38.9%	31
~~**Joomla**~~	2.6%	1.3%	0.5%	694	2.6%	1.3%	0.5%	68	45.4%	22.7%	17.4%	14
IceHrm	8.7%	4.8%	8.1%	921	23.1%	14.9%	14.1%	68	35.4%	20.2%	16.5%	118
phpBB	37.3%	28.0%	13.2%	336	37.1%	28.4%	12.3%	134	34.7%	25.5%	11.8%	68
Shopping List	26.1%	6.6%	14.3%	41	26.1%	6.6%	14.3%	10	26.1%	6.6%	14.3%	18
Wordpress	20.6%	11.0%	10.9%	1	86.3%	36.5%	18.5%	38	20.6%	11.0%	10.9%	27
OrangeHRM	17.9%	6.8%	8.2%	1363	16.6%	5.9%	8.2%	1	18.6%	6.8%	8.2%	45
Ajax Poller	0.0%	0.0%	0.0%	175	17.2%	10.0%	100.0%	12	17.2%	10.0%	100.0%	47
Collins	9.7%	0.0%	0.0%	800	30.7%	21.8%	5.5%	3	12.3%	0.0%	1.4%	33
~~**Snipe-It**~~	9.6%	3.6%	10.6%	1	9.6%	3.6%	10.6%	1	9.7%	3.7%	10.7%	151

Table 7.5: Benchmark results for state-of-the-art Automated Testing Frameworks.

Chapter 8

Discussion

In the previous section we evaluated HUBBLE. Now we discuss potential threats to the validity of our results. Followed by a reflection of potential issues that arose during the work on this thesis. Finally we discuss future work in development of Automated Testing Frameworks and suggest interesting areas for future research.

8.1 Threats to validity

In this section we discuss the factors that can influence our conclusions and affect the generalization of our results.

8.1.1 Internal Validity

We discuss factors that might affect the causal relation suggested in our conclusions. In other words: *what are factors that can influence our results that cannot be directly attributed to our technique?*

Coverage Measurement The coverage tool requires re-running of exported traces. This requires the exported traces of the state-of-the-art tools to be complete, since any unreported fired-events will not show up as covered.

Event-Types During our comparison of state-of-the-art tools, each framework supports a different set of event-types. Our internal comparison using execution modes is not affected by this issue. As our platform, HUBBLE, keeps the same technical capabilities and only modifies its event-sequence and input-generation techniques between execution modes.

Non-deterministic Behavior While we have solved the resetability between benchmark runs, it is possible that the execution of a single trace blocks access to the whole application. In principle this would mean that we would have to reset the benchmark after the execution of each trace. While this is possible, this would increase the runtime to the point that the tool becomes unusable.

Page Load Times Some web applications take more time to load than others. Even though the benchmarks and Automated Testing Tools were on the same local network, there is a possibility that the network traffic was affected during the analysis, therefore affecting the results.

8.1.2 External Validity

We discuss factors that affect the validity of generalizing our results in this subsection.

Benchmark Sets Due to some problems with an outdated JavaScript engine, we were unable to support a newer set of JavaScript applications. Another set of applications that we have not a analyzed contains JavaScript that is blocking. We have noticed that one or more tools would get stuck when running these applications. We therefore only have a relatively small set of benchmarks out of which we can draw our conclusions.

8.2 Reflection

A lot of work has been done in the area Automated Testing for Web Applications. In this section we present various issues for discussion and make suggestions for improvement.

8.2.1 Code Coverage

Analyzing the coverage of JavaScript Web Applications brings a few key questions that should be answered:

- Should we focus on JavaScript client-side code, server-side code or both?
- Should we analyze external libraries in our coverage analysis?
- What type of code coverage do we measure (e.g. line, branch, function)?
- How do we ensure a uniform representation of the code for the SUT?

Whenever code coverage is used as a metric to compare Automated Testing Frameworks, it is important that these questions are answered. Not providing the answers to these questions will give ambiguous potentially misleading results.

While coverage is an easily measurable metric, the ultimate goal is to support web application development efforts. According to our results, we believe that the use of symbolic execution for input generation does not have a significant effect on coverage in larger web applications. Does this mean that symbolic execution should not be applied in the automated testing of web applications or does this mean that code coverage is not the right metric?

8.2.2 Integration with the community

With a thriving open-source community on GitLab, GitHub and other platforms, it seems a natural progression for Automated Testing Tools to be developed closely with the community. There are many benefits to this approach including:

- Minimizing the gap between research and its application to the real world.
- Getting contributions from the community, not only in code improvements, but also in new ideas for research.
- Being more critical of code-quality, as it is publicly accessible.

Open-source software does not exclude commercial success. Potential profit models include providing professional assistance, selling proprietary extensions or providing the software as a service. Open-source software often qualifies for public funding.

8.2.3 Code-sharing

There are now at least three project that require the similar dependencies. Creating a single system that would fulfill the needs for each of these project can decrease maintenance time and would allow research effort to be focused on improving old techniques and creating new techniques. Sharing resources between research teams that allow us to focus on making new scientific leaps should be our goal.

8.3 Future Work

In this section we discuss potential ideas for future developments and research for Automated Testing Tools of JavaScript Web Applications.

8.3.1 Development

Web Browser and Code Instrumentation

In-depth JavaScript instrumentation cannot be performed by a browser automation tool such as Selenium without the use of code injection. Alternatively HTMLUnit does allow instrumentation, but has difficulties with visibility and interactibility checking as it is a headless browser. A feasible alternative would be to use Headless Chromium[1].

Developing a crawler that works with Headless Chromium directly without any interaction from Selenium, will greatly improve the speed and reliability compared to the current solution in HUBBLE.

Important to note here is the that any development in V8 or Chromium directly should be done in a modular plug-in fashion due to the fast paced nature of the Chromium and V8 projects. Adopting the projects by making modifications and then manually updating parts of the application is not doable for a project of this size. Keeping the dependency projects

[1]`https://chromium.googlesource.com/chromium/src/+/lkgr/headless/README.md`

in a given version should be avoided, as the whole system would become obsolete within a few years due to incompatibilities with newer versions of JavaScript.

Symbolic Execution in V8

A lot of time is used in HTMLUnit to execute the JavaScript. The JavaScript engine, Rhino, is a lot slower than the JavaScript engines that are in use by browsers today. The time that is currently spent on actually symbolic execution is a small fraction compared to the time it takes to execute the JavaScript code. Building a Symbolic Execution Engine for V8 not only provides a tool for JavaScript Web Applications, but also provides a tool that could be used for other applications that depend on V8 including Node.js.

8.3.2 Research

JavaScript Testing

Current JavaScript testing has mostly been performed on Web Applications. Creating a Symbolic Execution tool that is available for V8 as described in Section 8.3.1 allows us to expand from JavaScript Web Applications to general JavaScript applications.

Server-Side Analysis

Generating inputs for a web application may require server-side analysis. Some work has already been done as we discussed in the related work by Jensen et al [21]. This work should be expanded to work for large-scale web applications, as the scalability of these techniques is still under question.

Crawling vs. Coverage

Simply reaching a page often increases code coverage, however much of the code does not include any of the core-logic of the application, but focuses mostly on appearance. How do we solve this? Is it possible to automatically separate code that simply works on appearance from code such as form validation?

Model Developer Feedback

Using model-learning there should exist the possibility to visualize the difference between two versions of an application. Presenting the differences in the models and allowing inspection to a developer could reveal issues.

Chapter 9

Related Work

In the early 2000's JavaScript became an ever-more important technology to provide an instant and dynamic interface for users to interact with their applications. To assure the quality of these application, lots of manual testing was done. Manual testing is a time-intensive and thus costly process, sparking the interest for researchers to automate JavaScript web application testing. In this chapter we discuss related work to the field of automated JavaScript web application testing.

9.1 JavaScript Web Application Testing

Automated testing JavaScript applications has been an interesting topic for research in the automated testing and software engineering research community, as popular conferences in this field have seen large increases in publications for JavaScript analysis and testing. In this section we discuss related work in the field of JavaScript web application testing. It includes work related to manual testing and automated testing.

9.1.1 Dynamic Typing

JavaScript is a dynamically typed language. This means that every variable has a dynamic type, i.e. it can have a different type during runtime. Testing a dynamically typed language has consequences for the testing process: a developer is not necessarily warned during the development process, that there are type mismatches. One possibility is to use manual testing, to test every possible usage of the function. Manual testing is however labor-intensive and is prone to user-error. Another solution is to use dynamic analysis as has been done by Pradel et al. [47] Their tool, TypeDevil uses dynamic analysis to perform type observation. It combines these observations in a type graph, which it uses to warn the developer against inconsistent type usage.

9.1.2 Document Object Model (DOM)-element Locating

Uniquely locating elements on a page is a problem for automated testing. Element locating is needed in the page abstraction that is used to identify if a page has been visited before and to distinguish behavior-differing pages. Although this is still a largely open problem in automated testing, Jensen et al [22] have created a DOM-model that can represent the elements on a page. Even with their DOM-model, it is still a manual process to define the refinement of the abstraction level.

Some related work has been done in the development assistance by Bajaj et al. [2] Bajaj et al created LED, a tool that assists developers to identify which selectors can be used to reference targeted DOM-elements. Note however that this tool requires feedback from the developers to refine its DOM-element selection suggestions and thus has limited applicability to automated testing.

DOM-element locating is also a problem in the replaying of test-cases. Leotta et al [26] have suggested to use multi-locators to refer to DOM-elements in test-cases. These multi-locators chooses the best selector from a set of selectors produce by various algorithms. It is able to automatically repair the set of selectors once a mismatch is found. Potentially this could be used to describe the difference between two pages, giving a confidence interval of their similarity, greatly improving the page abstraction process.

9.1.3 Oracle Generation

In Section 2.3 we discussed the oracle problem and presented the different types of oracles. HTML validation [37] and the throwing of exceptions [44, 1, 37] are examples of techniques that use *implicit* oracles. Ocariza et al [44] found that 80% of the reported JavaScript errors were DOM-related. They therefore stress the importance of DOM-related oracles when performing automated testing of JavaScript web applications. In their automatic fault localizer, Ocariza et al make use of an *implicit* oracle by determining if JavaScript exceptions were thrown or not. Their tool locates the fault by backtracking through DOM-modifying or -querying functions.

Other techniques use *derived* oracles. Zaeem et al [52] and Mirshokraie et al [39] use human-written GUI test-cases to *derive* assertions to create unit oracles. Similarly Maezawa et al [29] try to establish oracles for user interaction, however their technique uses an automated approach with model-learning to model the user-system interaction and check if the model holds in regression testing. Jensen et al [21] use Server Interface Descriptions to derive invariants that can be used to generate oracles that test the JavaScript code in the context of server interactions.

Pythia [41] uses a mutation-based method to derive oracles in the form of assertions to unit-test the SUT. Pythia derives the oracles by creating faulty mutations of the unit and comparing the states of correct execution traces to the states of the mutated (incorrect) execution traces. It then derives function-level postcondition assertions from the resulting traces.

Mesbah et al [37] use a mixture of implicit and derived oracles. Their technique checks for DOM validity, exceptions, back-button compatibility (*implicit*) and use DOM-tree invariants [17] (*derived*) that can serve as oracles.

9.1.4 Input Generation

There are external actors that can influence the execution flow of the JavaScript code under test. For example logic that resides in the JavaScript code which constraints the input values, or communication that contains server-side logic of which we cannot analyze the code. Therefore it is difficult to come up with all possible feasible input values. To remedy this problem several approaches have been proposed. The most important ones are:

- **JavaScript Constants**: Artemis [1] has a input generation mode where it uses JavaScript constants: constants strings and values that can be found in the JavaScript to use as input values. An example of this would be to have a variable declaration in the code `var a = "lorem"`. Using the constants technique, 'lorem' would be put into the set of suggested inputs.

- **Symbolic Execution**: SymJS [27] has used symbolic execution to generate input values based on constraints defined in the JavaScript code. An example would contain a constraint such as `if(input==42)`. The symbolic execution would provide two values: one satisfying the condition ('42') and one not satisfying the condition (e.g. '7').

- **Server Interface Descriptions**: Jensen et al [21] can use Server Interface Descriptions to establish inputs to test the interaction between client and server. Their Server Interface Descriptions include relationships between requests and responses. An example would be if a server responds with 'true' if the number '42' is requested and otherwise returns 'false'. The Server Interface Description will describe this and allow us to extract two values '42' and e.g. '7' from the Server Interface Description that will allow us to receive all possible responses from the server.

- **Random Values**: Crawljax [42] uses a random approach for its input generation. An example for a name field on a page could result in having a value like '`&r93br3mr`'.

Mobile Applications

Similarly to JavaScript Web Applications, Mobile Applications also require input generation. The input generation approaches for Mobile Applications could also possibly be applied to JavaScript Web Applications.

In their survey [13], Choudhary et al categorized the input generation methods into the following categories:

- **Random Exploration Strategy**: random event and input generation which includes fuzzers that intend to generate invalid inputs to crash the application.

- **Model-based Exploration Strategy**: mostly tools that use finite state machines to guide the exploration. These tools are limited to the accuracy of their state abstraction. State changes may be undetected as the state abstraction in not fine-grained enough to distinguish the two program states.

- **Systematic Exploration Strategy**: these strategies include symbolic execution, evolutionary algorithms and other systematically input producing techniques.

Choudhary et al have run the different tools and showed that random exploration strategies work best on average. A cause for this might be the limited need for complex inputs. Randomly firing UI Events may seem to suffice in the context of Mobile Applications. The non-random tools should not be disregarded as if the techniques are combined used for the right situations, they can cause significant improvements in coverage.

9.2 Web Application Testing Tools

In this section we present three most popular JavaScript Web Application Testing Frameworks: SymJS, Crawljax and Artemis.

9.2.1 SymJS

SymJS is a proprietary automated JavaScript testing framework developed by Fujitsu Laboratories of America Inc. with its first publications by Guodong Li, Esben Andreasen and Indradeep Gosh in 2014. [27] SymJS uses symbolic execution to generate values that are required to uncover execution paths.

Since its conception in 2014, many improvements have been made to the symbolic execution engine. Additionally, the verification of the executability of the generated test cases has been added. Previously it was possible for SymJS to generate event sequences that would not be possible for a user to trigger. By verifying all of the traces in Selenium SymJS can filter out traces that would not be executable by a user. Code coverage is tracked by in the JavaScript execution engine directly.

Timeline

2014 Project started. [27]

2016 Hubble development started also leading to many improvements to SymJS itself.

2016 Most recent commit.

Technologies

SymJS is a Maven project written in Java and hosted on a GitLab installation. It uses HtmlUnit in combination with Rhino for the instrumentation, symbolic execution. SymJS uses Selenium to ensure that elements are visible and interactable. Yices is the SMT Solver used to solve the symbolic constraints.

Features

Crawl browser specification includes support for Firefox, Chrome, Internet Explorer and PhantomJS.

Visual overview provides an overview of all states in the inferred state-flow graph.

Detailed crawl configuration configurable in types of elements that should be examined or ignored during the crawling process.

Adaptive Interfaces provides command-line, web and programming interfaces.

Scalability has support for queuing and distributing crawling jobs.

Prioritized sequence construction provides functions and input feedback mechanisms for prioritization of event sequence construction.

Form input generators use random values or symbolic values based on constraints found in the JavaScript code.

Event sequence export provides the ability to export the constructed event-sequences, which can then be used to run in a browser automation framework.

9.2.2 Crawljax

Crawljax is a systematic exploration framework for Asynchronous JavaScript and XML (Ajax)-based web applications. Crawljax was conceived in 2007 to provide a way for search engines to crawl Ajax-based Web Applications. Since then it has been a widely-used research platform to perform automated cross-browser compatibility testing, regression testing, security testing and fault localization.

More recently in 2015, Crawljax has been used for automated JavaScript unit test generation. The test generation requires the use of dynamic analysis, which is achieved using injection of JavaScript code. All the current research built on top of Crawljax executes JavaScript code or queries Selenium to perform dynamic analysis. This differs from the way instrumentation is done in HUBBLE as we present in Chapter 6. In HUBBLE no JavaScript code is executed on the page aside from the sources provided. All of the instrumentation is done on an interpreter level.

Coverage benchmarks in Crawljax are instrumented using JSCover or similar frameworks, which instrument the JavaScript code to obtain code coverage by injecting code before each line of execution. This approach greatly slows down JavaScript execution. In HUBBLE we avoid this by creating our own coverage analysis tool. An in-depth look at this tool can be found in Chapter 6.

Timeline

2007 Project started as a result of PhD Research by Ali Mesbah. [34]

2008 Publication on Inferring User Interface State Changes. [32]

2009 Publication on Invariant-Based Automatic Testing. [35]

2009 Publication on Automated Security Testing. [7]

2010 Publication on Regression Testing. [48]

2010 Publication on Automatic Invariant Detection. [37]

2011 Publication on Automated Cross-Browser Compatibility Testing.[33]

2011 Publication on Dynamic Analysis of User Interface State Changes. [36]

2012 Publication on Automatic Fault Localization. [45]

2012 Publication on Assertion-based Regression Testing. [38]

2012 Start of a major overhaul in the Crawljax codebase by Alex Nederlof.

2014 Publication with a large scale study regarding Software Engineering for the Web using Crawljax. [43]

2015 Publication on the prospects and challenges in Automated Crawling and Testing of Web Applications. [50]

2015 Publication on Automated JavaScript Unit Test Generation. [40]

2015 Most recent commit.[1]

Technologies

Crawljax is Maven project written in Java and is hosted on GitHub. Crawljax uses Selenium to query and control a previously specified browser.

Features

Open source allows reproducibility of benchmark results, and gives back to the community providing a good platform for further research.

Extensibility Crawljax has a plugin based architecture, making it easily extensible.

Crawl browser specification includes support for Firefox, Chrome, Internet Explorer and PhantomJS.

Visual overview provides an overview of all states in the inferred state-flow graph.

Detailed crawl configuration configurable in types of elements that should be examined or ignored during the crawling process.

Adaptive Interfaces provides command-line, web and programming interfaces.

Scalability has support for queuing and distributing crawling jobs.

Event sequence export provides the ability to export the constructed event-sequences, which can then be used to run in a browser automation framework.

[1] `https://github.com/crawljax/crawljax/commit/686863`

9.2.3 Artemis

Artemis is a tool written in C++ that performs automated, feedback-directed testing of JavaScript Web Applications. Started in late 2011 by Anders Møller, Artemis focused purely on automated testing and was not using feedback from the user in order to generate test-cases. More recently in 2013 Artemis is using Server Interface Descriptions to model client-server AJAX interactions. Test-cases are inferred from the description, but should be manually checked before being used. [1]

Timeline

2011 Project started by Anders Møller. [1]

2013 Publication on Server Interface Descriptions. [21]

2016 Most recent commit.[2]

Technologies

Artemis is a project written in C++ and hosted on GitHub. It uses a modified version of WebKit to perform instrumentation, concolic execution and gather feedback on the AUT. Constraint solver Z3, Kaluza and CVC4 can be used to solve for the symbolic path conditions given by the concolic execution engine.

Features

Open source allows reproducibility of benchmark results, and gives back to the community providing a good platform for further research.

Prioritized sequence construction provides functions and input feedback mechanisms for prioritization of event sequence construction.

Form input generators select one of the two strategies: use random values or use JavaScript constants seen previously in the code for form inputs.

Event sequence export provides the ability to export the constructed event-sequences, which can then be used to run in a browser automation framework.

[2] `https://github.com/cs-au-dk/Artemis/commit/ce3ca4`

Chapter 10

Conclusions and Contributions

10.1 Conclusions

In this thesis we explored the current state of Automated Testing for JavaScript web applications, presented a new Automated Testing Framework and gave an outlook on future research.

We demonstrated the importance of model-learning in combination with symbolic execution for the Automated Testing of JavaScript Web Applications. Our results indicate that model-learning by itself already provides an improvement over symbolic execution, not only in total coverage, but also in final size of the test-suite.

Although the total coverage is improved by using model-learning, our results do not indicate that the technique allows the tool to reach coverage quicker.

Combining symbolic execution and model-learning into a single technique has shown to obtain a higher total coverage for the applications in our benchmark sets. Comparing HUBBLE to two state-of-the-art automated web application testing tools has demonstrated it to be a competitive testing tool: HUBBLE obtains the highest coverage in most of the benchmarks.

10.2 Contributions

Throughout the thesis, we have made various contributions. A concise summary of these contributions can be found below.

10.2.1 HUBBLE

Fujitsu Laboratories of America Inc. has a large number of web applications in their portfolio. Testing each of these applications manually requires a lot of work from testers and does not guarantee the correct functioning of the application, as the manual testing process might overlook possible situations.

Creating HUBBLE such that it can be run on large web applications and thus is able to automatically generate test-sequences that can be used to test these applications greatly reduces the strain on the testing process and allows for more energy to be put into the development of the application itself.

The research presented in this thesis shows that for enterprise web applications, the model-learning has a significant impact on the coverage of the resulting test-suite. Furthermore, the results regarding symbolic execution can shape the way future research is done and on what development efforts should be focused.

10.2.2 Constellations

The BlueFringe passive model-learner, Constellations, is created in such a way that the inputs can easily be changed to fit the application. Constellations is set up as a micro-service and with minor modifications can be re-purposed for other applications.

10.2.3 Coverage Tool

The coverage tool that was created for the comparison between Artemis, Crawljax and HUBBLE can be used to easily replay test and give detailed coverage information in the form of lcov coverage reports.

This can be used to improve HUBBLE in the future, but also provides insights that can be used to check the coverage of the generated test-suite and manually add test-cases for the code that is still uncovered. Greatly reducing the development effort required to maintain test-suites.

Bibliography

[1] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 571–580.

[2] K. Bajaj, K. Pattabiraman, and A. Mesbah. LED: Tool for Synthesizing Web Element Locators. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 848–851.

[3] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[4] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 171–177. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[5] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, pages 737–797. IOS Press, Amsterdam, 2008.

[6] A. Bertolino. Software Testing Research: Achievements, Challenges, Dreams. In *2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[7] C. Bezemer, A. Mesbah, and A. van Deursen. Automated Security Testing of Web Widget Interactions. Technical report, Delft University of Technology, August 2009.

[8] A. Bruns, A. Kornstadt, and D. Wichmann. Web Application Tests with Selenium. *IEEE Softw.*, 26(5):88–91, September 2009.

[9] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th*

USENIX Conference on Operating Systems Design and Implementation, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[10] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive Random Testing. In *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, pages 320–329. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[11] C. Y. Cho, D. Babic, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song. MACE: Model-inference-assisted Concolic Exploration for Protocol and Vulnerability Discovery. In *Proceedings of the 20th USENIX Conference on Security*, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[12] W. Choi, G. Necula, and K. Sen. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pages 623–640, New York, NY, USA, 2013. ACM.

[13] S. R. Choudhary, A. Gorla, and A. Orso. Automated Test Input Generation for Android: Are We There Yet? In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440.

[14] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg, March 2008.

[15] B. Dutertre and L. de Moura. The YICES SMT Solver. In *Proceedings on 2nd SMT competition, SMT-COMP'06*, 2006.

[16] J. J. Garrett. AJAX: A new approach to web applications. 2005.

[17] F. Groeneveld, A. Mesbah, and A. van Deursen. Automatic invariant detection in dynamic web applications. Technical report, August 2010.

[18] R. Hamlet. *Random Testing*. John Wiley & Sons, Inc., Hoboken, NJ, USA, January 2002.

[19] J. Hughes and K. Claessen. QuickCheck: An Automatic Testing Tool for Haskell. Technical report.

[20] D. Huizinga and A. Kolawa. *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, 1st edition, September 2007.

[21] C. S. Jensen, A. Møller, and Z. Su. Server Interface Descriptions for Automated Testing of JavaScript Web Applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 510–520, New York, NY, USA, 2013. ACM.

[22] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 59–69, New York, NY, USA, 2011. ACM.

[23] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[24] D. R. Kuhn, J. M. Higdon, J. F. Lawrence, R. N. Kacker, and Y. Lei. Combinatorial Methods for Event Sequence Testing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 601–609. IEEE, April 2012.

[25] K. J. Lang, B. A. Pearlmutter, and R. A. Price. Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. *Grammatical Inference*, 1433(Chapter 1):1–12, 1998.

[26] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Using Multi-Locators to Increase the Robustness of Web Test Cases. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, April 2015.

[27] G. Li, E. Andreasen, and I. Ghosh. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 449–459, New York, NY, USA, 2014. ACM.

[28] G. Li and I. Ghosh. PASS: String Solving with Parameterized Array and Interval Automaton. In Valeria Bertacco and Axel Legay, editors, *Hardware and Software: Verification and Testing: 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*, pages 15–31. Springer International Publishing, Cham, 2013.

[29] Y. Maezawa, H. Washizaki, Y. Tanabe, and S. Honiden. Automated verification of pattern-based interaction invariants in AJAX applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE) IS - SN - VO - VL* -, pages 158–168.

[30] A. Marchetto and P. Tonella. Using search-based algorithms for AJAX event sequence generation during testing. *Empirical Software Engineering*, 16(1):103–140, December 2010.

[31] W. M. McKeeman. Differential Testing for Software. *Digital Technical Journal*, 10:100–107, 1998.

[32] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling AJAX by Inferring User Interface State Changes. In *2008 Eighth International Conference on Web Engineering*, pages 122–134, July 2008.

[33] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 561–570. ACM, May 2011.

[34] A. Mesbah and A. van Deursen. Exposing the Hidden-Web Induced by AJAX. Technical report, Delft University of Technology, January 2008.

[35] A. Mesbah and A. van Deursen. Invariant-based Automatic Testing of AJAX User Interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[36] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling AJAX-Based Web Applications Through Dynamic Analysis of User Interface State Changes. *ACM Trans. Web*, 6(1):3:1–3:30, March 2012.

[37] A. Mesbah, A van Deursen, and D. Roest. Invariant-Based Automatic Testing of Modern Web Applications. *IEEE Transactions on Software Engineering*, 38(1):35–53, January 2012.

[38] S. Mirshokraie and A. Mesbah. JSART: JavaScript Assertion-Based Regression Testing. In Marco Brambilla, Takehiro Tokuda, and Robert Tolksdorf, editors, *Web Engineering: 12th International Conference, ICWE 2012, Berlin, Germany, July 23-27, 2012. Proceedings*, pages 238–252. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[39] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Atrina: Inferring Unit Oracles from GUI Test Cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 330–340.

[40] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. JSEFT: Automated JavaScript Unit Test Generation. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10.

[41] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. PYTHIA: Generating test cases with oracles for JavaScript applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE) IS - SN - VO - VL -*, pages 610–615.

[42] A. Nederlof. Analyzing web applications: An empirical study. pages 1–89, 2013.

[43] A. Nederlof, A. Mesbah, and A. van Deursen. Software engineering for the web: the state of the practice. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 4–13. ACM, 2014.

[44] F. S. Ocariza Jr., K. Pattabiraman, and A. Mesbah. AutoFLox: An Automatic Fault Localizer for Client-Side JavaScript. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 31–40.

[45] F. S. Ocariza Jr., K. Pattabiraman, and A. Mesbah. AutoFLox: An Automatic Fault Localizer for Client-Side JavaScript. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 31–40, April 2012.

[46] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[47] M. Pradel, P. Schuh, and K. Sen. TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 314–324.

[48] D. Roest, A. Mesbah, and A. van Deursen. Regression Testing AJAX Applications: Coping with Dynamism. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 127–136, April 2010.

[49] K. Sen. Concolic Testing. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 571–572, New York, NY, USA, 2007. ACM.

[50] A. van Deursen, A. Mesbah, and A. Nederlof. Crawl-based analysis of web applications: Prospects and challenges. *Science of Computer Programming*, 97:173–180, 2015.

[51] S. Verwer, M. De Weerdt, and C. Witteveen. An algorithm for learning real-time automata. In *Benelearn 2007: Proceedings of the Annual Machine Learning Conference of Belgium and the Netherlands*, 2007.

[52] R. N. Zaeem, M. R. Prasad, and S. Khurshid. Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 183–192.