Directed Test Generation and Analysis for Web Applications

by

Amin Milani Fard

BSc. Computer Engineering, Ferdowsi University of Mashhad, Iran, 2008

MSc. Computing Science, Simon Fraser University, Canada, 2010

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Electrical and Computer Engineering)

The University of British Columbia

(Vancouver)

January 2017

Abstract

The advent of web technologies has led to the proliferation of modern web applications with enhanced user interaction and client-side execution. JavaScript (the most widely used programming language) is extensively used to build responsive modern web applications. The event-driven and dynamic nature of JavaScript, and its interaction with the Document Object Model (DOM), make it challenging to understand and test effectively. The ultimate goal of this thesis is to improve the quality of web applications through automated testing and maintenance.

The work presented in this dissertation has focused on advancing the state-of-the-art in testing and maintaining web applications by proposing a new set of techniques and tools. We proposed (1) a feedback-directed exploration technique and a tool to cover a subset of the state-space of a given web application; the exploration is guided towards achieving higher functionality, navigational, and page structural coverage while reducing the test model size, (2) a technique and a tool to generate UI tests using existing tests; it mines the existing test suite to infer a model of the covered DOM states and event-based transitions including input values and assertions; it then expands the inferred model by exploring alternative paths and generates assertions for the new states; finally it generates a new test suite from the extended model, (3) the first empirical study on JavaScript tests to characterize their prevalence and quality metrics, and to find out root causes for the uncovered (missed) parts of the code under test, (4) a DOM-based JavaScript test fixture generation technique and a tool, which is based on dynamic symbolic execution; it guides the executing through different branches of a function by producing expected DOM instances, (5) a technique and a tool to detect JavaScript code smells using static and dynamic analysis.

We evaluated the presented techniques by conducting various empirical studies and comparisons. The evaluation results point to the effectiveness of the proposed techniques in terms of fault detection capability and code coverage for test generation, and in terms of accuracy for code smell detection.

Preface

Research projects included in this dissertation have been either published or currently under review. I have conducted the research described in this work in collaboration with my supervisor, Ali Mesbah. I was the main contributor for the research projects, including the idea, tool development, and evaluations. I also had the collaboration of Mehdi Mirzaaghaei, and Eric Wohlstadter in two of the presented work.

The following list presents publications for each chapter.

- Chapter 2:
  - Feedback-Directed Exploration of Web Applications to Derive Test Models [123]: A. Milani Fard, A. Mesbah, IEEE International Symposium on Software Reliability Engineering (ISSRE'13), 278–287, 2013, ©IEEE, Reprinted by permission;
- Chapter 3:
  - Leveraging Existing Tests in Automated Test Generation for Web Applications [126]: A. Milani Fard, M. Mirzaaghaei, A. Mesbah, IEEE /ACM International Conference on Automated Software Engineering (ASE'14), 67–78, 2014, ©ACM, Inc., Reprinted by permission;
  - Leveraging Existing Tests in Automated Web Test Generation: A. Milani Fard, A. Mesbah, Submitted to a software engineering journal;
- Chapter 4:
  - JavaScript: The (Un)covered Parts [125]: A. Milani Fard, A. Mesbah, Accepted at IEEE International Conference on Software Testing, Verification and Validation (ICST'17), 11 pages, 2017;
- Chapter 5:
  - Generating Fixtures for JavaScript Unit Testing [127]: A. Milani Fard, A. Mesbah, and E. Wohlstadter, IEEE/ACM International Conference on Automated Software Engineering (ASE'15), 190–200, 2015, ©IEEE, Reprinted by permission;

- Chapter 6:
  - JSNose: Detecting JavaScript Code Smells [124]: A. Milani Fard, A. Mesbah, IEEE International Conference on Source Code Analysis and Manipulation (SCAM'13), 116–125, 2013, ©IEEE, Reprinted by permission;

Table of Contents

List of Tables

List of Figures

List of Algorithms

Glossary

List of acronyms and abbreviations.

AJAX	Asynchronous JavaScript and XML
API	Application Program Interface
AST	Abstract Syntax Tree
BFS	Breadth-First Search
CFG	Control-Flow Graph
CSS	Cascading Style Sheets
DFS	Depth-First Search
DOM	Document Object Model
DTD	Document Type Definition
HTML	HyperText Markup Language
JS	JavaScript
SFG	State-Flow Graph
SLOC	Source Lines of Code
UI	User Interface
URL	Uniform Resource Locator
XHR	XMLHttpRequest
XML	Extensible Markup Language

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Ali Mesbah, for his exemplary supervision and guidance in the past four and a half years. He introduced me to the field of web software testing and analysis, and had a profound influence on my research. I have always benefited from his insightful comments and discussions. This thesis would not have been completed without his support.

I collaborated with Eric Wohlstadter and Mehdi Mirzaaghaei in two of my research projects, whom I would like to thank. I am also thankful to Karthik Pattabiraman and Ivan Beschastnikh for providing valuable comments and suggestions on my PhD proposal. Moreover, I would like to thank Philippe Kruchten, Mieszko Lis, Hasan Cavusoglu, and Guy-Vincent Jourdan for reading my dissertation and providing valuable comments.

Thanks also go to all faculty members, staff, and friends in the Electrical and Computer Engineering at UBC for providing such a nice academic environment. In particular, I have enjoyed the time with all the members in the Software Analysis and Testing Lab. I also gratefully acknowledge the financial support by the Natural Sciences and Engineering Research Council of Canada (NSERC) through its Canada Graduate Scholarship and Strategic Project Grants, as well as the Four Year Doctoral Fellowship of UBC, that helped me to focus full time on my research.

Finally, I would like to thank my family and friends for their love and support over the years. I am specially indebted to my wife, Hoda, for her patience, love, understanding, and encouragement since we started our life journey ten years ago. I wish to dedicate this thesis to her. Fatima, my beloved daughter, thank you also for coming into our life in the middle of my PhD career and bringing joy to our family. And last but not the least, I would like to thank the Almighty for giving me the strength and patience to work through all these years.

Chapter 1

Introduction

The advent of web technologies has led to the proliferation of modern web applications, such as Gmail, Google Drive, Facebook, and YouTube, with enhanced user interaction and client-side execution. Due to the considerable impact of web applications on economic, technical, and social activities, it is important to ensure their quality. Testing techniques aim at increasing the quality of *functional* properties of a software to ensure they meet the requirements. On the other hand, the goal of maintenance techniques is to increase the quality of *non-functional* (structural) properties. In this work, we aim at improving the quality of web applications through both *testing* and *maintenance*.

Web applications are often written in multiple languages such as JavaScript, HTML, and CSS. JavaScript is known to be the most widely used programming language[1], which is extensively used to build responsive modern web applications. The event-driven dynamic nature of JavaScript and its interaction with the Document Object Model (DOM[2]) [180], make it challenging for developers to understand [62], test [66, 130], and debug [141] effectively. Testing frameworks, such as SELENIUM [42] for DOM-based user interface (UI) testing and QUNIT [40] for JavaScript unit testing, help to automate test execution. However, test cases are written manually, which is a tedious process with an incomplete result. Maintaining web applications is also difficult due to intricate dependencies among the components written in different programming languages.

[1] According to a recent survey of more than 56K developers conducted by Stack Overflow [167], and also exploration of the programming languages used across GitHub repositories [91].

[2] The DOM is a tree-like structure that provides APIs for accessing, traversing, and mutating the content and structure of HTML elements at runtime.

The ultimate goal of this thesis is improving the quality of web applications through automated testing and maintenance. Such automated techniques reduce the time and effort of manual testing and maintenance. Towards this objective, we consider three perspectives that are complementary to each other: (1) *UI testing*, (2) *unit testing*, and (3) *code maintenance*. The results of each perspective will benefit the goal of this thesis in a different way.

1.1 UI Testing

To avoid dealing separately with the complex interactions between multiple web languages, many developers treat the web application as a black-box and test it via its manifested DOM and perform UI testing. Such DOM-based UI tests bring the application to a particular DOM state through a sequence of actions, such as filling a form and clicking on an element, and subsequently verify the existence or properties of particular elements in that DOM state.

1.1.1 Test Model Generation

Model-based testing uses models of program behaviour to generate test cases [175]. These *test models* are either specified manually or derived automatically. Dynamic analysis and exploration (also know as *crawling*) plays a significant role in automated test model derivation for many automated testing techniques [68, 71, 74, 113, 119, 121, 128, 129, 171]. Unlike traditional static hyper-linked web pages, crawling web applications is challenging due to event-driven user interface changes caused by client-side code execution.

In the recent past, web crawlers have been proposed to explore event-driven DOM mutations in web applications. For instance, CRAWLJAX [120] uses dynamic analysis to exercise and crawl client-side states of web applications. It incrementally reverse engineers a model, called *State-Flow Graph* (SFG), where each vertex represents a runtime DOM state and each edge represents an event-driven transition between two states. Such an inferred SFG can be utilized in DOM-based UI test case generation [121], by adopting different graph coverage methods (e.g., all states/edges/paths/transitions coverage).

Challenge. Most industrial web applications have a huge state-space and exhaus-

tive crawling – e.g., breadth-first search (BFS), depth-first search (DFS), or random search – lead to *state explosion*[177]. In addition, given a limited amount of time, exhaustive crawlers can become mired in specific parts of the application, yielding poor functionality coverage. Since exploring the whole state-space can be infeasible (state explosion) and undesirable (time constrains), it is challenging to automatically derive an incomplete test model but with an adequate functionality coverage in a timely manner.

Related Work. Efficient strategies for web application crawling [70, 73, 79, 134] try to discover as many states as possible in the shortest amount of time. However, discovering more state does not necessarily result in higher functionality coverage. Thummalapenta et al. [169] proposed a guided test generation technique for web applications. Although their approach directs test generation towards higher business logic coverage, the application is crawled in an exhaustive manner that can lead to poor coverage.

1.1.2 UI Test Generation

Manually writing DOM-based UI test cases is inefficient with limited coverage, therefore crawling-based techniques [74, 121, 159, 169] have been proposed to automate exploration and test generation.

Challenges. Such test generation methods are limited in three areas:

1. *Input values:* Valid input values is required for proper state-space coverage. Automatic input generation is challenging since many applications require a specific type, value, or combination of inputs to expose the hidden states behind input fields and forms.

2. *Paths to explore:* Since covering the whole state-space of many industrial web applications is infeasible in practice, crawlers are limited to a crawl depth, exploration time or number of states. Not knowing which paths are important to explore results in obtaining a partial coverage of a specific region of the application.

3. *Assertions:* Any generated test case needs to assert the application behaviour. However, generating proper assertions, known as the *oracle problem* [184],

without human knowledge is challenging. Thus, many web testing techniques rely on generic invariants [121] or standard validators [66].

Related work. Researchers have proposed techniques to use existing artifacts for testing. Elbaum et al. [81] and Sprenkle et al. [165] leverage user-sessions data for web application test generation. Schur et al. [159] infer behaviour models from enterprise web applications via crawling and generate test cases simulating possible user inputs. Similarly, Xu et al. [188] mine executable specifications of web applications from SELENIUM test cases to create an abstraction of the system. Yuan and Memon [192] propose an approach to iteratively rerun automatically generated test cases for generating alternating test cases. Artzi et al. [66] present Artemis, which uses the gathered information while random testing to generated new test inputs. These approaches, however, do not use information in the existing tests and do not address test oracle generation. Yoo and Harman [190] propose a search-based approach to reuse and regenerate existing test data for primitive data types. Pezze et al. [150] present a technique to generate integration test cases from existing unit test cases. Test suite augmentation techniques, such as [189], are used in regression testing to generate new test cases for the changed parts of the application. Wang et at.[181] propose aggregating tests generated by different approaches using a unified test case language. However, these techniques also do not consider the oracle generation problem.

1.2 Unit Testing

Writing DOM-based UI tests does not generally require an understanding of the client side code under test as it is a black-box testing process, thus easier to write for testers. However, DOM-based tests may miss code-level bugs that do not propagate to the DOM [130]; therefore JavaScript unit testing is important for proper testing of web applications.

1.2.1 JavaScript Test Quality Assessment

To assist developers with writing tests, there exist number of JavaScript unit testing frameworks, such as Mocha [36], Jasmine[31], QUnit [40], and Nodeunit [37], each having its own advantages and disadvantages [52]. However, even by us-

ing a testing framework, some JavaScript features, such as DOM interactions, event-dependent callbacks, asynchronous callbacks, and closures (hidden scopes), are considered to be harder to test [28, 46, 48, 53, 127, 130, 173]. Thus the research community have proposed some automated test generation techniques for JavaScript programs to partially assist with hard to test code [66, 95, 127, 130, 131], though are not considerably used by testers and developers yet. Currently there is no (large scale) study on JavaScript (unit) tests in the wild to evaluate difficulties in writing such tests. Moreover, studying the quality of JavaScript tests can reveal some shortcomings and difficulties of manual testing, which provides insights on how to improve existing JavaScript test generation tools and techniques.

Challenges. Test quality assessment is, however, not a trivial task and can be subjective. For instance a good test quality metric that considers test effectiveness, is fault/bug detection capability. Ideally, this requires manual investigation of JavaScript bug reports, which confines the study to JavaScript projects that have bug reports associated to some test cases. As an alternative, mutation score, i.e., the percentage of killed mutants over total non-equivalent mutants, is often used as an estimate of defect detection capability of a test suite. However, running test suites on many generated mutated versions of JavaScript programs can be very time consuming to apply for a large scale study. Consequently, some other test quality metrics, such as code coverage, and average number of assertions per test, can be used. While code coverage does not directly imply a test suite effectiveness [97], it is a widely accepted test quality indicator. Thus finding the root cause of why a particular statement is not covered by a test suite, can help in writing higher quality tests.

Related work. Ocariza et al. [142] performed study to characterize root causes of client-side JavaScript bugs. Researchers also studied test cases and mining test suites in the past. Inozemtseva et al. [97] found that code coverage does not directly imply the test suite effectiveness. Zhang et al. [199] analyzed test assertions and showed that existence of assertions is strongly correlated with test suite effectiveness. These work, however, did not study JavaScript tests. Mirshokraie et al. [129] presented a JavaScript mutation testing approach and as part of their evaluation, assessed mutation score for test suites of two JavaScript libraries.

1.2.2 JavaScript Unit Test Generation

Unit testing is a software testing method to test individual units of a source code. In generic unit testing, a *test fixture* is a fixed state of the software under test that is set up to provide all the required resources in order to properly run a test. In client-side JavaScript unit testing, test fixtures can be in the form of a partial HTML for the expected DOM structure before calling the JavaScript function under tests. If such a DOM-based fixture is not provided in the exact structure as expected by the function, a DOM API method (e.g., `getElementById()`) returns `null`, thus the execution of the function fails due to the null exception and the test case terminates prematurely. Therefore proper DOM fixtures are required for effective JavaScript unit testing. Since manual construction of test fixtures is tedious and costly, automated fixture generation is of great value.

Challenges. Automated generation of proper test fixtures is challenging as it requires inferring and solving constraints in the code. This is even more complicated when DOM structure is involved. There are two main challenges in generating proper DOM-based fixtures:

1. *DOM-related variables:* JavaScript is a weakly-typed and highly-dynamic language, which makes static code analysis quite challenging. Moreover, its interactions with the DOM can become difficult to follow [62, 141]. A fixture generator needs to determine DOM-dependent variables that refer to values/properties of DOM elements.

2. *Hierarchical DOM relations:* Unlike most test fixtures that deal only with primitive data types, DOM-based test fixtures require a tree structure. In fact, DOM fixtures not only contain proper DOM elements with attributes and their values, but also hierarchical parent-child relations that can be difficult to reconstruct.

Related work. Most JavaScript test generation techniques [66, 95, 158, 161] do not consider DOM fixture generation. Li et al. [109] proposed SymJS, which applies symbolic execution with a limited support for the DOM. They consider substituting DOM element variables with integer or string values and using a solver, rather than actually generating the hierarchical DOM structure. Mirshokraie et al. [130] proposed JSeft that captures the full DOM tree during the execution of an application

before executing the function under test, and uses that DOM as a test fixture. This approach, assumes that the captured DOM contains the DOM structure as expected by the function, which is not always the case. As such, the code coverage achieved with such a DOM can be quite limited. Moreover, the DOM captured this way, can be too large and difficult to read as a fixture in a test case.

1.3 Code Maintenance

Web applications must continually evolve and be maintained to remain functional. Maintenance may involve refactoring to improve the internal logical structure of the code without changing its external behaviour.

1.3.1 JavaScript Code Smell Detection

JavaScript code maintenance is particularly challenging because (1) there is typically no compiler in the development cycle that would help developers to spot erroneous or unoptimized code; (2) JavaScript has a dynamic, weakly-typed, asynchronous nature, and supports intricate features such as prototypes, first-class functions, and closures; and (3), it interacts with the DOM through a complex event-based mechanism [180]. As a result JavaScript applications tend to contain many *code smells*, i.e., patterns in the code that indicate potential comprehension and maintenance issues [85]. Code smells, once detected, need to be refactored to improve the design and quality of the code.

Challenges. Manual JavaScript code smell detection is time consuming and error-prone, thus automated tools can be very helpful. Detecting code smells is dependent on identifying objects, classes, and functions in the code, which is not straightforward in JavaScript due to its very flexible model of objects and functions. JavaScript code refactoring is also challenging due to dynamism, wide use of global variables and first-class functions, which can cause unexpected side-effects when making code changes.

Related work. WebScent [137] is a tool to detect client-side smells (mixing of HTML, CSS, and JavaScript, duplicate code in JavaScript, and HTML syntax errors) in embedded code within server-side code. However it requires manual navigation of the application, and also does not support inferring dynamic cre-

ation/change of objects, properties, and functions at runtime. WARI [17] finds unused and duplicated JavaScript functions. JSLint [9] and PMD [12] are static code analysis tools that validate JavaScript code against a set of good coding practices. The Google Closure Compiler [27] rewrites JavaScript code to make it faster by removing comments and unreachable code. Tool support for refactoring JavaScript are in their early stages. Feldthaus et al. [83, 84] proposed static pointer analysis techniques to perform renaming of variables or object properties, and some JavaScript-specific refactorings, such as encapsulation of properties and extraction of modules, targeting programming idioms [77].

1.4 Research Questions

The overarching goal of this dissertation is *to develop automated testing and maintenance techniques that improve the quality of web applications*. To achieve this goal and in accordance to the aforementioned three perspectives, we designed five research questions listed below.

RQ1.4.1 *How can we effectively derive test models for web applications?*

RQ1.4.1 is in line with UI testing. In response to RQ1.4.1, we propose a feedback-directed exploration technique [123] and tool, called FEEDEX [25], that selectively covers a subset of the state-space of a given web application. The exploration is guided towards achieving higher functionality, navigational, and page structural coverage of a given web application while reducing the size of our test model i.e., a *State-Flow Graph* (SFG) [120, 121]. Our results show that FEEDEX is capable of generating test models that are enhanced in all aspects compared to the traditional exploration methods.

RQ1.4.2. *Can we utilize the knowledge in existing UI tests to generate new tests?*

RQ1.4.2 is also with respect to UI testing. To address RQ1.4.2, we proposed a technique [126] and a tool, called TESTILIZER [47], to generate DOM-based UI test cases using existing tests. TESTILIZER mines the existing test suite to infer a model (in the form of a SFG) of the covered DOM states and event-based transitions including input values and assertions. It then expands the inferred model by exploring alternative paths and generates assertions for the new states. Finally it

generates a new test suite from the extended model. Our results supports that leveraging input values and assertions from human-written test suites can be helpful in generating more effective UI tests.

RQ1.4.3. *What is the quality of JavaScript tests in practice and which part of the code is hard to cover and test?*

RQ1.4.3 is in line with unit testing. To address RQ1.4.3, we present the first empirical study of JavaScript tests to characterize their prevalence, quality metrics, and shortcomings [125]. We perform our study across a large corpus of JavaScript projects and found that about 22% of the studied subjects do not have test code. About 40% of projects with JavaScript at client-side do not have a test, while this is only about 3% for the purely server-side JavaScript projects. Also tests for server-side code have high quality, while tests for client-side code have moderate to low quality. On average JavaScript tests lack proper coverage for event-dependent callbacks (36%), asynchronous callbacks (53%), and DOM-related code (63%).

RQ1.4.4. *How can we automate fixture generation for JavaScript unit testing?*

RQ1.4.4 is also with respect to unit testing. In response to RQ1.4.4, we proposed a DOM-based test fixture generation technique [127] and a tool, called CONFIX [24], which is based on concolic execution. Conceptually our approach infers a *control flow graph* (CFG) by guiding the executing through different branches of a function. Our empirical results show that the generated fixtures substantially improve code coverage compared to test suites without these fixtures, with a negligible overhead.

RQ1.4.5. *Which JavaScript code smells are prevalent in practice and what maintenance issues they cause?*

RQ1.4.5 is regarding code maintenance. In response to RQ1.4.5, we proposed a tool and technique, called JSNOSE [34], to detect JavaScript code smells [124] that could benefit from refactoring. It uses static and dynamic analysis to infer *program entities model* containing objects, functions, variables, and code blocks. We collected 13 JavaScript code smells by studying various online development resources and books that discuss bad JavaScript coding patterns. Our results show that lazy object, long method/function, closure smells, coupling JS/HTML/CSS, and excessive global variables, are the most prevalent JavaScript smells.

1.5 Publications

In response to the research questions in Section 1.4, the following peer-reviewed papers have been published:

- Feedback-Directed Exploration of Web Applications to Derive Test Models [123]: A. Milani Fard, A. Mesbah, IEEE International Symposium on Software Reliability Engineering (ISSRE'13), 278–287, 2013;
- Leveraging Existing Tests in Automated Test Generation for Web Applications [126]: A. Milani Fard, M. Mirzaaghaei, A. Mesbah, IEEE/ACM International Conference on Automated Software Engineering (ASE'14), 67–78, 2014;
- JavaScript: The (Un)covered Parts [125]: A. Milani Fard, A. Mesbah, IEEE International Conference on Software Testing, Verification and Validation (ICST'17), 11 pages, 2017;
- Generating Fixtures for JavaScript Unit Testing [127]: A. Milani Fard, A. Mesbah, and E. Wohlstadter, IEEE/ACM International Conference on Automated Software Engineering (ASE'15), 190–200, 2015;
- JSNose: Detecting JavaScript Code Smells [124]: A. Milani Fard, A. Mesbah, IEEE International Conference on Source Code Analysis and Manipulation (SCAM'13), 116–125, 2013.

I have also contributed to the following testing related publication:

- An Empirical Study of Bugs in Test Code [176]: A. Vahabzadeh, A. Milani Fard, and A. Mesbah, International Conference on Software Maintenance and Evolution (ICSME'15), 101–110, 2015.

Chapter 2

Feedback-Directed Exploration of Web Applications to Derive Test Models

Summary[3]

Dynamic exploration techniques play a significant role in automated web application testing and analysis. However, a general web application crawler that exhaustively explores the states can become mired in limited specific regions of the web application, yielding poor functionality coverage. In this chapter, we propose a feedback-directed web application exploration technique to derive test models. While exploring, our approach dynamically measures and applies a combination of code coverage impact, navigational diversity, and structural diversity, to decide a-priori (1) which state should be expanded, and (2) which event should be exercised next to maximize the overall coverage, while minimizing the size of the test model. Our approach is implemented in a tool called FEEDEX. We have empirically evaluated the efficacy of FEEDEX using six web applications. The results show that our technique is successful in yielding higher coverage while reducing the size of the test model, compared to classical exhaustive techniques such as depth-first, breadth-first, and random exploration.

[3] An initial version of this chapter has been published in the IEEE International Symposium on Software Reliability Engineering (ISSRE), 2013 [123].

2.1 Introduction

Modern web applications make extensive use of JavaScript to dynamically mutate the DOM in order to provide responsive interactions within the browser. Due to this highly dynamic nature of such applications, dynamic analysis and exploration (also know as crawling) play a significant role [88] in many automated web application testing techniques [64, 68, 74, 114, 119, 121, 128, 129, 171]. Such testing techniques depend on data and models generated dynamically through web application crawling.

Most web application exploration techniques used for testing apply an exhaustive search in order to achieve a "complete" coverage of the application state-space and functionality. An assumption often made is that the state-space of the application is completely coverable in a reasonable amount of time. In reality, however, most industrial web applications have a huge dynamic state-space and exhaustive crawling – e.g., through breadth-first search (BFS), depth-first search (DFS), or random search – can cause the state explosion problem [177]. In addition, a generic crawler that exhaustively explores the states can become mired in irrelevant regions of the web application [144], producing large test models that yield poor functionality coverage.

Because exploring the whole state-space can be infeasible (state explosion) and undesirable (time constrains), the challenge we are targeting in this chapter is to automatically derive an incomplete test model but with adequate functionality coverage, in a timely manner.

To that end, we propose a novel feedback-directed exploration technique, called FEEDEX, which is focused on efficiently covering a web application's functionality to generate test models. We propose four metrics to capture different aspects of a test model, namely *code coverage impact*, *navigational diversity*, *page structural diversity*, and *test model size*. Using a combination of these four metrics, our approach dynamically monitors the exploration and its history. It uses the feedback obtained to decide a-priori (1) which states should be expanded, and (2) which events should be exercised next, so that a subset of the total state-space is effectively captured for adequate test case generation.

The main contributions of our work include:

- We present a feedback-directed exploration technique that selectively covers a subset of the state-space of a given web application to generate an adequate test model;
- We propose the notions of coverage impact and diversity – i.e., navigational and page structural diversity – to capture different aspects of derived test models;
- We describe an event execution method, which prioritizes events based on their historical productivity in producing relevant states;
- We implement our approach in a tool called FEEDEX, which is freely available;
- We empirically evaluate the efficacy of our approach on six web applications. The results show that our method yields much better code coverage (10% at the minimum), and diversity (23% at the minimum), compared to traditional exhaustive methods. In addition, our approach is able to reduce the size of the derived test model and test suite by at least 38% and 42%, respectively.

2.2 Background and Motivation

Web Applications. The advent of recent Web and browser technologies has led to the proliferation of modern – also known as web 2.0 – web applications, with enhanced user interaction and more efficient client-side execution. Building on web technologies such as AJAX, web applications execute a significant amount of JavaScript code in the browser. A large portion of this code is dedicated to interact with the Document Object Model (DOM) to update the user interface at runtime. In turn, these incremental updates lead to dynamically generated execution states within the browser.

Automated Test Model Generation. Model-based testing uses models of program behaviour to generate test cases. These *test models* are either specified manually or derived automatically.

Tool support for exploring (or "crawling") dynamic states of web applications enables automated test model derivation. Unlike traditional static hyper-linked

web pages, automatically exploring web applications is challenging because many of the user interface state changes are (1) event-driven, (2) caused by the execution of client-side application code, and (3) represented by dynamically mutating the internal DOM tree in the browser. In the recent past, web crawlers have been proposed that can explore event-driven DOM mutations in web applications. For instance, CRAWLJAX [120] uses dynamic analysis to exercise and crawl client-side states of web applications. It incrementally reverse engineers a model, called *State-Flow Graph* (SFG), which captures dynamic DOM states and event-driven transitions connecting them. This SFG model is formally defined as follows:

Definition 1 *A **state-flow graph** SFG for a web application* **A***, is a labeled directed graph, denoted by a 4 tuple* $< \boldsymbol{r}, \boldsymbol{V}, \boldsymbol{E}, \mathscr{L} >$ *where:*

1. $\boldsymbol{r}$ *is the root node (called Index) representing the initial state after* **A** *has been fully loaded into the browser.*
2. $\boldsymbol{V}$ *is a set of vertices representing the states. Each* $\boldsymbol{v} \in \boldsymbol{V}$ *represents a runtime DOM state in* **A**.
3. $\boldsymbol{E}$ *is a set of (directed) edges between vertices. Each* $(\boldsymbol{v}_1, \boldsymbol{v}_2) \in \boldsymbol{E}$ *represents a clickable* $\mathbf{c}$ *connecting two states if and only if state* $\boldsymbol{v}_2$ *is reached by executing* $\mathbf{c}$ *in state* $\boldsymbol{v}_1$.
4. $\mathscr{L}$ *is a labelling function that assigns a label, from a set of event types and DOM element properties, to each edge.*
5. *SFG can have multi-edges and can be cyclic.* □

To infer this SFG, events (e.g., clicks) are automatically generated on *candidate clickables*, i.e., user interface elements of the web application that can potentially change the state of the application. An example is a `DIV` element, dynamically bound to an event-listener, which when clicked calls a JavaScript function, which in turn mutates the DOM inside the browser (see Figures 2.3 and 2.4). Only when the event generated results in a state change, the candidate clickable is seen as a real *clickable* and the new state and the clickable are added to the SFG.

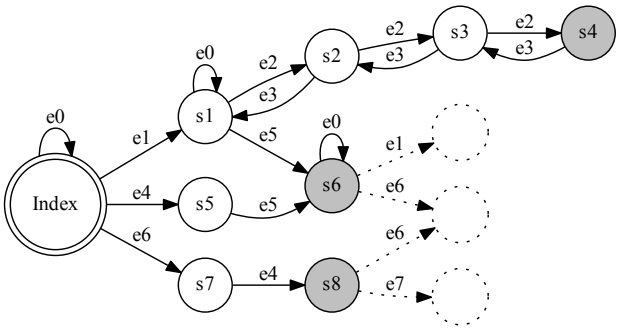


Figure 2.1: State-flow graph of the running example. The shaded nodes represent partially expanded states, edges with dashed lines are candidate events, and nodes shown in dashed circle are states that might be discovered after event execution.

An example of such a SFG is shown in Figure 2.1. Such an automatically inferred test model can be utilized in test case generation [121], by adopting different graph coverage methods (e.g., all states/edges/paths/transitions coverage).

Motivation. Given that (1) most industrial web applications have a huge dynamic state-space, which can cause the state explosion problem, (2) dynamic exploration is time-consuming, and (3) in any realistic software development environment, the amount of time dedicated to testing is limited, opting for the exploration of a partial subset of the total state-space of a given web application seems like a pragmatic feasible solution.

Given a specific amount of time, there are, however, different ways of exploring this partial subset. For example, assume that the SFG in Figure 2.1, including the dashed nodes and edges, represents the complete state model of a web application. Also assume that our allowed crawling time is limited to 4 event executions and our crawler applies a depth-first search. If the explored sequence of exercised clickables is $e1$, $e2$, $e2$, $e2$, we retrieve states $s1$, $s2$, $s3$, and $s4$, all of which belong to same crawling path. It could also be the case that the crawler has a predetermined order for event execution. Consider a crawler in which $e3$ always executes before $e2$. The sequence of the resulting clicks would then become $e1$, $e2$, $e3$, $e2$ which results in discovering only 3 states, namely $s1$, $s2$, and $s3$. Another example, depicted in Figures 2.3–2.4, includes exploring states behind *next* and *previous* links. This is a typical scenario in which exhaustive crawlers can become mired in

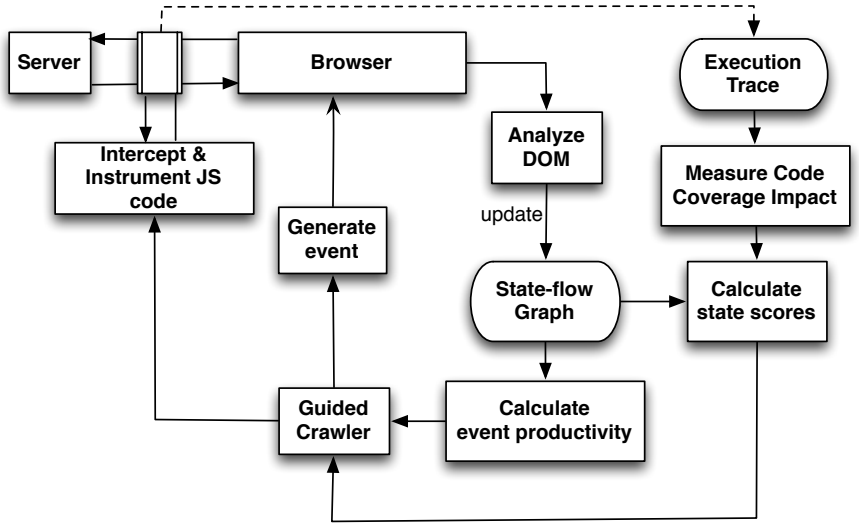


Figure 2.2: Processing view of FEEDEX, our feedback-directed exploration approach.

specific parts of the web application, yielding poor functionality coverage.

Because a complete exploration of the whole application can be infeasible (state explosion), undesirable (time constraint), and inefficient (large test model derived that yields low coverage), the challenge we are targeting in this chapter is to derive an incomplete test model that can adequately provide functionality coverage.

2.3 Approach

The goal of our work is to automatically derive a test model that captures different aspects of the given web application's client-side functionality. In the next subsections, we present these desired aspects of a test model, followed by a salient description of our *feedback-directed exploration* technique, called FEEDEX.

2.3.1 Deriving Test Models

A web crawler is generally evaluated on its ability to retrieve relevant and desirable content [61, 144, 166]. In this work, we propose metrics that target relevance and desirability in the context of web application testing. We believe that a test

model derived automatically from a web application should possess the following properties to effectively cover different aspects of the application under test:

- **Functionality Coverage.** A test suite generated from a derived test model can only cover the maximum functionality contained in the test model, and not more. For instance consider Figure 2.1. If the inferred test model does not capture events $e6$ and $e7$ the generated test suite will not be able to cover the underlying functionality behind those two events. Therefore, it is important to derive a test model that possesses adequate coverage of the web application when the end goal is test suite generation.

- **Navigational Coverage.** The navigational structure of a web application allows its users to navigate it in various directions. For instance $s3$ and $s4$ are both on the same navigational path, whereas $s3$ and $s8$ are on different branches (Figure 2.1). To cover the navigational structure adequately, we believe that a test model should cover different navigational branches of the web application.

- **Page Structural Coverage.** The structure of a webpage in terms of its internal DOM elements, attributes, and values, provides the main interaction interface with end users. Each page structure provides a different degree of content and functionality. To capture this structural functionality adequately, we believe a test model should cover heterogeneous DOM structures of the web application.

- **Size.** The size of the derived test model has a direct impact on the number of generated test cases and event executions within each test case. Reducing the size of the test model can decrease both the cost of maintaining the generated test suite and the number of test cases that must be rerun after changes are made to the software [94]. Thus, while deriving test models, the size should be optimized as long as the reduction does not adversely influence the coverage. We consider the number of events (edges) in the SFG as an indicator of the model size since test case generation is done by traversing the sequence of events.

Algorithm 1: Feedback-directed Exploration

```
   input : A web application A, the maximum exploration time t, the maximum
           number of states to explore n, exploration strategy STR
   output: The inferred state-flow graph SFG
 1 SFG ← AddInitialState(A)
   Procedure Explore() begin
 2     while ConstraintSatisfied(t,n) do
 3         PES ← GetPartiallyExpandedStates(SFG)
 4         for s_i ∈ PES do
 5             for s_j ∈ PES & s_j ≠ s_i do
 6                 Score(s_i,s_j) ← GetScore(s_i,s_j,STR)
               end
 7             MinScore(s_i) ← GetMinScore(s_i)
           end
 8         s ← GetNextToExploreState(PES,MinScore)
 9         C ← PrioritizeEvents(s)
10         for c ∈ C do
11             browser.Goto(SFG.GetPath(s))
12             dom ← browser.GetDOM()
13             robot.FireEvent(c)
14             new_dom ← browser.GetDOM()
15             if dom.HasChanged(new_dom) then
16                 SFG.Update(c,new_dom)
17                 Explore()
               end
           end
       end
18     return SFG
   end
```

2.3.2 Feedback-directed Exploration Algorithm

In this chapter, we refer to the process of executing candidate clickables of a state as *state expansion*. Figure 2.2 depicts an overview of our approach. At a high level, it dynamically analyzes the exploration history at runtime, including previously covered states and events, to anticipate and decide a-priori (1) which *state* should be expanded next, and (2) from the state that is selected for expansion, which *event* (i.e., clickable element) should be exercised.

Given a web application, a maximum exploration time t, and a maximum number of states to explore n, the intent is to maximize the test model coverage while

reducing the model size within t. The rationale behind our technique is to reward states and events that have a substantial impact on different aspects of a test model (and penalize those that do not). Our exploration strategy, shown in Algorithm 1, applies a greedy approach to select and expand a partially expanded state.

Definition 2 *A **partially expanded state** is a state, during exploration, which still contains one or more unexercised candidate clickables.* □

In Figure 2.1, both $s6$ and $s8$ are partially expanded states. To expand the next partially expanded state, while exploring, we calculate a *state score* (explained in Section 2.3.3) for each state that needs to be expanded, at runtime; this score prioritizes partially expanded states selectively so that a test model with enhanced aspects can be inferred for testing. Our key insight is that by dynamically measuring and expanding states with the highest scores, we can construct a state-flow graph of an application, yielding higher overall coverage. In addition to the state score, our approach prioritizes events based on their productivity ratio (described in Section 2.3.4).

Algorithm 1 repeats the following main steps until the given time limit t, or state-space limit n is reached:

- For each partially expanded state, compute the *state score* with respect to other unexpanded states (Lines 4-6). The score of a state is the minimum pair-wise state score of that state (Line 7);
- Choose the state with the highest fitness (score) value as the next state to expand (Line 8);
- On the chosen state for expansion, prioritize the events based on their *event score* (Line 9);
- Take the browser to the chosen state and execute the highest ranked event according to the prioritized order (Lines 10-3);
- If the DOM state is changed after the event execution, update the SFG accordingly (Lines 14-16) and call the `Explore` procedure (line 17).

2.3.3 State Expansion Strategy

In this section, we describe our state expansion strategy, which is used to prioritize partially expanded states based on their overall contributions towards the different desired properties of the test model.

Code Coverage Impact. One primary objective for generating a test model is to achieve an adequate code coverage of the core functionality of the system. Note that we only consider the client-side code coverage and the server-side code coverage is not the goal of this work. The following example shows how state expansion can affect the code coverage.

Example 1 *Figure 2.3 depicts a simple JavaScript code. Figure 2.4 shows the corresponding DOM state. Assume that the state-flow graph as shown in the Figure 2.1 was generated by exploring this simple example. If the crawler finishes after clicking on the next and the previous clickables multiple times, only the function* `show` *and the first two lines of the script code will be covered, i.e., 9 lines in total. Assuming that the total number of JavaScript lines of code is 27 (not all the code is shown in the figure), coverage percentage would be* $\frac{9}{27} = 33.33\%$*. This indicates that no matter what test case generation method we apply on the inferred SFG, we can not achieve a higher code coverage than 33.33%. However, if the crawler was able to click on the Update clickable before termination, the function* `load` *would be executed as well, yielding a coverage of* $\frac{15}{27} = 55.55\%$*.* □

The *code coverage impact* for a state s, denoted by $CI(s)$, is defined as the amount of increase in the code coverage after s is created. Considering the example again, when the *Index* page is loaded, the first two JavaScript lines are executed and thus initial coverage is $\frac{2}{27} = 7.4\%$, and the $CI(Index)$=0.074. After executing `onclick="show()"`, coverage would reach 33.33% with a 25.93% increase and thus $CI(s1) = 0.259$. For each newly discovered state, we calculate the *CI* in this manner. If a resulting state of an event is an already discovered state in the SFG, the *CI* value will be updated if the new value is larger. The *CI* score is taken into consideration when making decisions about expanding partially expanded states.

Path Diversity. Given a state-flow graph, states located on different branches are

```
1 myImg = new Array("1.jpg","2.jpg","3.jpg","4.jpg");
2 curIndex = 1;
3 ...
4 function show (offset) {
5   curIndex = curIndex + offset;
6   if (curIndex > 4) {
7     curIndex = 1; }
8   if (curIndex == 0) {
9     curIndex = 4 ; }
10  document.imgSrc.src = myImg[curIndex - 1];
11 }
12 ...
13 function load () {
14   var xhr = new XMLHttpRequest();
15   xhr.open('GET', '/update/', true);
16   xhr.onreadystatechange = function(e) {
17     document.getElementById("container").innerHTML = this.responseText
18   };
19   xhr.send();
20 }
```

Figure 2.3: Simple JavaScript code snippet.

```
<body>
  <img name="imgSrc" id="imgSrc" src="1.jpg">
  <span onclick="show(-1)">previous</span>
  <span onclick="show(1)">next</span>
  ...
  <a href="#" onclick="load(); return false;">Update!</a>
  <div id="container"></div>
</body>
```

Figure 2.4: A simple DOM instance.

more likely to cover different navigational functionality (as in Section 2.3.1) than those on a same path. Thus, guiding the exploration towards diversified paths can yield a better navigational functionality coverage.

Definition 3 *A **simple event path** P_{s_i} of state* $\mathbf{s}_i$ *is a path on SFG from the Index node to* $\mathbf{s}_i$*, without repeated nodes.* □

Note that in our definition we only consider "simple paths" without repeated nodes to avoid the ambiguity of having cycles and loops, such as those shown in Figure 2.1, where for instance, there exists an infinite number of paths from *Index*

to $s2$. We formulate the *path similarity* of two states s_i and s_j as:

$$PathSim(s_i, s_j) = \frac{2 \times |P_{s_i} \cap P_{s_j}|}{|P_{s_i}| + |P_{s_j}|} \tag{2.1}$$

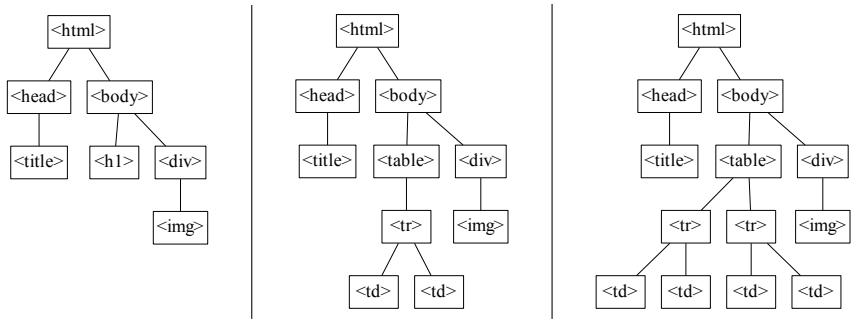
where $|P_{s_i}|$ is the length of P_{s_i}, and $|P_{s_i} \cap P_{s_j}|$ denotes the number of shared events between P_{s_i} and P_{s_j}. The path similarity notion captures the percentage of events shared by two simple paths. The fewer events two states share in their paths, the more diverse their navigational functionality. Thus, let $MaxPathSim(s_i, s_j)$ be the maximum path similarity of s_i and s_j considering all possible simple paths, from Index to s_i and s_j, respectively. The *path diversity* of s_i and s_j, denoted by $PD(s_i, s_j)$, is then calculated as:

$$PD(s_i, s_j) = 1 - MaxPathSim(s_i, s_j) \tag{2.2}$$

Example 2 *Consider the running example of Figure 2.1. We calculate pair-wise path diversity scores for states s4, s6, and s8.* $PathSim(s4, s6) = \frac{2 \times |P_{s4} \cap P_{s6}|}{|P_{s4}| + |P_{s6}|}$ *can be computed in two ways as there exist two simple event paths from Index to s6: For* P_{s6} *through s1,* $PathSim(s4, s6) = \frac{2 \times 1}{4+2} = \frac{2}{6} = \frac{1}{3}$*, and for* P_{s6} *through s5,* $PathSim(s4, s6) = \frac{2 \times 0}{4+2} = 0$*. Thus* $MaxPathSim(s4, s6) = \frac{1}{3}$ *and* $PD(s4, s6) = 1 - \frac{1}{3} = \frac{2}{3}$*. Similarly,* $PD(s4, s8)$*=1-0=1, and* $PD(s6, s8)$*=1-0=1. This indicates that s4 and s6 are not that diverse with respect to each other, but they are very diverse with respect to s8.* □

DOM Diversity. In many web applications, the DOM is mutated dynamically through JavaScript to reflect a state change, without requiring a URL change. Many of such dynamic DOM mutations are, however, incremental in nature and correspond to small delta changes, which might not be interesting from a testing perspective, especially given the space and time constraints. Therefore, guiding the exploration towards diversified DOM states can result in a better page structural coverage.

In most current AJAX crawlers, string representations of the DOM states are used for state comparison. The strings are compared by either calculating the edit distance [120], a strip and compare method [120], or by computing a hash of the



(a) DOM tree t_1 **(b)** DOM tree t_2 **(c)** DOM tree t_3

Figure 2.5: DOM tree comparison.

content [80]. These approaches ignore the tree structure of DOM trees. To account for the actual structural differences, we adopt the *tree edit distance* between two ordered labeled trees, which was proposed [168] and implemented [149] as the minimum cost of a sequence of edit operations that transforms one tree into another. The operations include deleting a node and connecting its children to the parent, inserting a node between a node and the children of that node, and relabelling a node.

We define state DOM diversity as the normalized DOM tree edit distance. Let t_i and t_j be the corresponding DOM trees of two states s_i and s_j. The DOM diversity of s_i and s_j, denoted by $DD(s_i, s_j)$, is defined as:

$$DD(s_i, s_j) = \frac{TED(t_i, t_j)}{max(|t_i|, |t_j|)} \tag{2.3}$$

where $TED(t_i, t_j)$ is the tree edit distance between t_i and t_j, and $max(|t_i|, |t_j|)$ is the maximum number of nodes in t_i and t_j.

Example 3 *Figure 2.5 depicts three DOM trees with $|t_1|$=7, $|t_2|$=10, and $|t_3|$=13. t_2 can be produced from t_1 by (1) relabelling <h1> in t_1 to <table>, and (2) inserting three nodes under <table>. Thus $TED(t_1, t_2) = 4$ and their DOM diversity equals $\frac{4}{10}$=0.4. Similarly $TED(t_1, t_3) = 7$ and thus their DOM diversity equals $\frac{7}{13}$=0.53. This shows t_3 is more DOM diverse than t_2 with respect to t_1. $TED(t_2, t_3) = 3$ and their DOM diversity equals $\frac{3}{13}$=0.23* □

Overall State Score. The state score is a combination of code coverage impact, path diversity, and DOM diversity. Our state expansion fitness function is a linear combination of the three metrics as follows:

$$Score(s_i, s_j) = w_{CI} \cdot CI(s_i, s_j) + w_{PD} \cdot PD(s_i, s_j) + w_{DD} \cdot DD(s_i, s_j) \quad (2.4)$$

where, w_{CI}, w_{PD}, and w_{DD} are user-defined weights (between 0 and 1) for code coverage impact, path diversity, and DOM diversity, respectively.

2.3.4 Event Execution Strategy

The goal of our event execution strategy is to reduce the size of events sequences (edges) in the SFG, while preserving the coverage. Reducing the size of events is important since it reduces the size of generated test cases, which in turn minimizes the time overhead of test rerun [94].

Intuitively, we try to minimize the execution of events that are not likely to produce new states. We categorize web application user interface events into four groups based on their impact on the application state transitional behaviour: (1) An event that does not change the DOM state is called a *self-loop*, e.g., events that replace the DOM tree with an exact replica, e.g., refresh with no changes, or clear data in a form; (2) A *state-independent event* is an event that always causes the same resulting state, e.g., events that always result in the Index page; (3) A *state-dependent event* is an event that after its first execution, always causes the same state, when triggered from the same state. (4) A *Nondeterministic event* is an event that may results in a new state, regardless of where it is triggered from. Such events can result in different states when triggered from the same state. In Figure 2.1, for instance, $e0$ is a self-loop event, $e5$ is a state-independent event, and $e4$ is a state-dependent event.

A crawler that distinguishes between these different events can avoid self-loops, minimize state-independent and nondeterministic event executions, and emphasize state-dependent events to explore uncovered states. To that end, we define *event productivity* (EP) as follows.

Let $RS_i(e)$ denote the resulting state of the i-th execution of the event e, and n be the total number of executions of e (including the last execution). The event

productivity ratio of e, denoted by $EP(e)$, is defined as:

$$EP(e) = \begin{cases} 1 & ; \text{if } n = 0 \\ \frac{\sum_{i=1}^{n} MinDD(RS_i(e))}{n} & ; \text{otherwise} \end{cases} \tag{2.5}$$

where $MinDD(RS_i(e)) = \min_{s \in SFG} \{DD(RS_i(e), s)\}$, i.e., the minimum diversity of $RS_i(e)$ and all existing states in the SFG. Note that $0 \leq EP(e) \leq 1$ and its value can change after each execution of e, while exploring.

The above definition captures three properties. Firstly, it gives the highest ratio to the unexecuted events (in case $n = 0$) since the resulting state is more likely to be a new state compared to already executed events. Naturally, this also helps in covering more of the JavaScript code, since the event-listeners typically trigger the execution of one or more JavaScript function(s). Secondly, it penalizes events that result in an already discovered state, such as self-loops and state-independent events, with $MinDD(RS_i(e))$=0. Thirdly, the productivity ratio is proportional to the structural diversity of the resulting state with respect to previously discovered states. This gives a higher productivity ratio to events that have resulted in more diverse structures, guiding the exploration towards more DOM diverse states.

Remark 1. We do not consider path-diversity (*PD*) in the calculation of EP. This is because when the execution of an event results in a new state, the resulting state shares much of its navigational path with the source state that leads to *PD* close to 0, which discourages new state discovery. On the other hand, if the resulting state is an already discovered state in the SFG, its shortest event path may not share much with other paths and therefore might get a high *PD*. This is also contrary to penalizing events causing already discovered states.

The next example shows how this definition is applied on self-loops, state-independent events, and forward-backward events.

Example 4 *Consider Figure 2.1 again. For simplicity assume DOM diversity is 1 for new states and 0 for existing states. An example of a self-loop event is* $e0$*. Assume that the first observation of* $e0$ *is at state Index. Since we have never executed* $e0$ *before,* $EP(e0) = 1$*. After the first execution,* $EP(e0) = \frac{0}{1} = 0$ *because* $MinDD(Index) = 0$ *as the diversity of the source state (Index) and the resulting*

*state (Index) is 0. By the second execution, say at s*1*, EP*($e0$) $= \frac{0+0}{1+1} = 0$*. Now consider e*5 *as a state-independent event. For the first time, say at s*1*, EP*($e5$) $= 1$*. After its first execution, since s*6 *is a new state (we assume diversity is 1), the productivity ratio will be EP*($e5$) $= \frac{1}{1} = 1$*. However, the second execution results in a duplicate (s*6 *again) and thus EP*($e5$) $= \frac{1+0}{1+1} = 0.5$*. Events e*2 *and e*3 *are examples of previous-next events (see Figure 2.4). After the first execution of e*3 *at state s*2*, EP*($e3$) $= \frac{0}{1} = 0$*, because s*1 *was already in the SFG before executing e*3 *and thus MinDD*($s1$)*=0.* □

Remark 2. Prioritizing events only makes sense when we allow the crawler to exercise the same clickable multiple times. If the objective of the test model generation is to merely achieve high code coverage, then clicking on the same clickable again is unlikely to increase the code coverage; however, if the event is *state-dependent* or *nondeterministic*, multiple execution of the same clickable can have an impact on DOM and path diversity.

2.3.5 Implementation

Our proposed approach is implemented in a tool called FEEDEX, which is publicly available [25].

To explore web applications, we build on top of CRAWLJAX [120, 121]. The default engine supports both depth-first and breadth-first (multi-threaded, multi-browser) crawling strategies. FEEDEX replaces the default crawling strategy of CRAWLJAX (as described in [120]) with our feedback-directed exploration algorithm. The tool can be configured to constrain the state space, by setting parameters such as the maximum number of states to crawl, maximum crawling time, and search depth level.

As shown in Figure 2.2, FEEDEX intercepts and instruments the JavaScript code to collect execution traces while crawling. To parse and instrument the JavaScript code, we use Mozilla Rhino [41]. After each event execution, FEEDEX analyzes the collected execution trace and measures the code coverage impact, which in turn is used in our overall exploration decision making.

State path diversity is calculated according to Equation 2.2 by finding simple paths (Definition 3) from *Index* state to each state in the SFG. In order to compute

Table 2.1: Experimental objects (statistics excluding blank/comment lines, and JavaScript libraries).

ID	Name	JS LOC	Description
1	ChessGame [20]	198	A JavaScript-based simple game
2	TacirFormBuilder [45]	209	A JavaScript-based simple HTML form builder
3	TuduList [51]	782	An AJAX-based todo lists manager in J2EE and MySQL
4	FractalViewer [26]	1245	A JavaScript-based fractal zoomer
5	PhotoGallery [39]	1535	An AJAX-based photo gallery in PHP without MySQL
6	TinyMCE [50]	26908	A JavaScript-based WYSIWYG editor

each simple event path, we apply a DFS traversal from the *Index* state to a state node in the SFG and disregard already visited ones to avoid cycles or loops. For the computation of the tree edit distance for DOM state diversity as in Equation 2.3, we take advantage of the *Robust Tree Edit Distance* (RTED) algorithm [149], which has optimal $O(n^3)$ worst case complexity and is robust, where n is the maximum number of nodes of the two given trees. Considering that the number of nodes in a typical DOM tree is relatively small, the overhead of the DOM diversity computation is negligible.

In order to calculate the productivity ratio for an event, we store, for each event, a set of tuples comprising of source and target states, corresponding to all the previous executions of that event.

2.4 Empirical Evaluation

To assess the efficacy of the proposed feedback-directed exploration, we have conducted a controlled experiment. Our main research question is *whether our proposed exploration technique is able to derive a better test model compared to traditional exhaustive methods*.

Our experimental data along with the implementation of FEEDEX are available for download [25].

2.4.1 Experimental Objects

Because our approach is targeted towards web applications, our selection criteria included applications that (1) use extensive client-side JavaScript, (2) are based on dynamic DOM manipulation and AJAX interactions, and (3) fall under different

Table 2.2: State-space exploration methods evaluated.

Method	Exploration Criteria
DFS	Expand the last partially expanded state
BFS	Expand the first partially expanded state
RND	Randomly expand a partially expanded state
FEEDEX	Expand the partially expanded state with the highest score ($w_{CI} = 1, w_{PD} = 0.5, w_{DD} = 0.3$), and prioritize events

domains. Based on these criteria, we selected six open source applications from different domains which are shown in Table 2.1. We use CLOC [22] to count the JavaScript lines of code (JS LOC). The reported LOC in Table 2.1 is excluding blank lines, comment lines, and JavaScript libraries such as jQuery, DWR, Scriptaculous, Prototype, Bootstrap, and google-analytics. Note that we also exclude these libraries in the instrumentation step.

2.4.2 Experimental Setup

All our experiments are performed on a core-2 Duo 2.40GHz CPU with 3.46 GB memory, running Windows 7 and Firefox web browser.

Independent Variables

We compare our feedback-directed exploration approach with different traditional exploration strategies.

Exploration Constraints. We confine the designated exploration time for deriving a test model to 300 seconds (5 minutes) for all the experimental runs.[4] We set no limits on the crawling depth nor the maximum number of states to be discovered. We configure the tool to allow multiple executions of the same clickable element, as the same clickable can cause different resulting states.

State-space Exploration. Table 2.2 presents the different state expansion strategies we evaluate in the first part of our experiment. The first three (DFS, BFS, RND) are exhaustive crawling methods. DFS expands states in a depth-first fashion, i.e., the last discovered state would be expanded next. BFS applies breath-first

[4]Dedicating 5–10 minutes to test generation is acceptable in most testing environments [96].

exploration by expanding the first discovered state next. RND performs random exploration in which a partially expanded state is chosen uniformly at random to be expanded next. Note that for these traditional exhaustive exploration methods we consider the original event execution strategy, i.e., a user-defined order for clicking on elements. For this experiment, the order is defined as: `A`, `DIV`, `SPAN`, `IMG`, `BUTTON`, `INPUT`, and `TD`.The last method is an instantiation of our feedback-directed exploration score (See Equation 2.4) where $w_{CI} = 1$, $w_{PD} = 0.5$, and $w_{DD} = 0.3$. We empirically evaluated different weightings and found this setting among other settings can generally produce good results. FEEDEX prioritizes clickable elements based on the event productivity ratio EP (Equation 2.5) and executes them in that order (see Section 2.3.4).

Dependent Variables

We analyze the impact of different state-space exploration strategies on the code coverage, the overall average path diversity and DOM diversity, as well as the size of the derived test model.

Code Coverage Score. We measure the final statement code coverage achieved by each method, after 5 minutes of exploration. In order to measure the JavaScript code coverage, we instrument the JavaScript code as explained in Section 2.3.5.

Diversity Scores. In order to measure the path diversity of a SFG, we measure average pair-wise navigational diversity of leaf nodes (states without any outgoing edges) since the position of the leaf nodes in the graph is an indication of the diversity of its event paths (i.e., paths from the Index node to the leaves). The *AvgPD* is defined as:

$$AvgPD(SFG) = \frac{\sum_{\forall s_i, s_j \in L(V)} PD(s_i, s_j)}{2 \cdot m \cdot (m-1)} \tag{2.6}$$

where $L(V)$ denotes the set of leaf nodes in the SFG and $m = |L(V)|$, i.e., the number of leaf nodes. This value is in the range of 0 to 1.

To assess the page structural diversity of the derived test models from each method, we compute the overall average pair-wise structural diversity (*AvgDD*) in

Table 2.3: Results of different exploration methods.

Exploration Method	Statement Coverage	Navigational Path Diversity	Page Structural Diversity	Test Model Size	Test Suite Size
DFS	37.55%	0.010	0.035	578	247
BFS	43.82%	0.410	0.065	475	165
RND	40.44%	0.369	0.066	450	241
FEEDEX	48.13%	0.443	0.081	280	95
Improvement (min–max%)	10–28%	7–4000%	23–130%	38–86%	42–61%

the derived SFG as:

$$AvgDD(SFG) = \frac{\sum_{\forall s_i, s_j \in V} DD(s_i, s_j)}{2 \cdot n \cdot (n-1)} \tag{2.7}$$

where $n = |V|$, i.e., the number of states in the SFG and *AvgDD* is in the range of 0 to 1.

Test Model and Test Suite Size. As discussed in Section 2.3.1, the derived test model (SFG) can be used to generate test cases through different graph traversal and coverage methods. The event size of the derived test model has a direct impact on the number and size of test cases that can be generated, regardless of the test generation method used.

We consider (1) the number of edges (events) in the SFG, as the *size of test model*, and (2) the number of "distinct" simple event paths in the SFG, as the *size of test suite* (e.g. the number of all possible Selenium [42] test cases generated from the test model). Two simple event paths (Definition 3) are distinct if they visit different sequence of states. Note that simple event paths can not have cycles or loops. As described in Section 2.3.5, simple paths are generated using DFS traversal from the Index state to every other state in the SFG.

2.4.3 Results and Findings

Table 2.3 shows the results of different exploration methods. We report the average values obtained from the six experimental objects. For the random state expansion method (RND), we report the average values over five runs. The table shows statement code coverage, navigational path diversity (*AvgPD*), page structural diversity

(*AvgDD*), number of edges (events) as well as the number of distinct paths in the derived test model.

Results present that for FEEDEX, there is on average between (min–max%) 10–28% improvement on the final statement coverage (after 5 minutes), 7–4000% on path diversity, 23–130% on page structural diversity, 38–86% reduction in the number of edges, and 42–61% reduction in the distinct paths in the test model. It is evident from the results that FEEDEX is capable of generating test models that are enhanced in all aspects compared to the traditional exploration methods. Test models created using FEEDEX have smaller size (thus need less time to execute test cases) and higher code coverage and state diversity. The simultaneous improvement in all the evaluation criteria points to the the effectiveness of our state expansion and event execution strategy.

Given the limited amount of exploration time, we believe the achieved improvements for code coverage is substantial. Our approach also significantly increases the average DOM diversity compared to the RND method. Note that the main reason for having small *AvgDD* values for all the methods, is the normalization by the large number of possible pair-wise links in the computation of *AvgDD* (Equations 2.7). There is a low improvement of 7% achieved by FEEDEX over BFS with respect to the navigational path diversity (*AvgPD*). This is expected due to the fact that BFS, by its nature, generates models with many branches that do not share too many event paths, and are more sparse, compared to the other methods. The amount of shared paths also contribute to reducing the number of distinct paths, thus reducing the size of test suite. Therefore, although there is some improvement, FEEDEX can not improve *AvgPD* significantly compared to BFS. Among traditional methods, DFS is the least effective. Specifically, *AvgPD* achieved by DFS is much lower than others due to the fact that it keeps expanding states in the same branch in most cases, and FEEDEX can improve the navigational diversity significantly (4000%).

While evaluating different weights in FEEDEX scoring function, we found that assigning 1 to w_{DD} and 0 to the rest, is effective in producing more structural diverse models, if an application has many different DOM states, such as PhotoGallery, and not that effective for applications with minimum DOM changes, such as ChessGame (a chess board that remains relatively the same). Similarly,

assigning 1 to w_{PD} and 0 to the rest, is more effective in increasing *AvgPD* in applications with many navigational branches (features) such as TinyMCE. In general, feedback-directed exploration technique, with the settings we used, is superior over the exhaustive methods with respect to all aspects of generated test models. Considering the significant improvements achieved by using FEEDEX, the computational overhead of our method is negligible.

It worth to mention that if the exploration time is unlimited, the final generated test model for all the exploration methods will converge to a same model. This is, however, infeasible for many industrial web applications with huge state space and short release time. Thus we believe the given 5 minute exploration limit can show how FEEDEX can outperform exhaustive search methods in different aspects.

2.5 Discussion

Limitations. A limitation of FEEDEX is that the maximum effectiveness depends on the weights used in the scoring function (Equation 2.4) and on the application state-space and functionality, which is not known in advance. For example in our other experiments we observed that setting $w_{DD}, w_{PD} = 0$ and $w_{CI} = 1$, can generate test models with higher code coverage but less diversity and larger test model size. In general, the setting we used in this work can generate a test model which enhances all aspects of a test model compared to traditional methods. We do not claim that Equation 2.4 is the best way to combine CI, PD, and DD, or the weights that we empirically found, always outperform previous work. Instead, we rely on the intuition of the feedback-directed heuristic which we believe effectively works most of the time.

Applications. The main application of our technique is in automated testing of web applications. The automatically derived test model, for instance, can be used to generate different types of test cases used for invariant-based assertions after each event, regression testing, cross-browser testing [68, 74, 114, 119, 121, 128]. Our exploration technique towards higher client-side code coverage can also help with a more accurate detection of unused/dead JavaScript code [124]. Moreover, FEEDEX is generic in terms of its scoring function, thus by changing the fitness function (line 6 of Algorithm 1), it can generate a test model based on other user-

defined criteria.

Threats to Validity. A threat to the external validity of our experiment is with regard to the generalization of the results to other web applications. We acknowledge that more web applications should be evaluated to support the conclusions. To mitigate this threat we selected our experimental objects from four different domains: gallery, task management, game, and editor. The selected web applications exhibit variations in functionality and structure and we believe they are representative of modern web applications.

One threat to the internal validity of our experiment is related to the evaluation metrics including *AvgDD* and *AvgPD* proposed by the authors of this work based on which the effectiveness of FEEDEX is evaluated. However, we believe these metrics capture different properties of a test model as described in Section 2.3.

With respect to reliability of our results, FEEDEX and all the web-based systems are publicly available, making the experiment reproducible.

2.6 Related Work

Crawling Web Applications. Web crawling techniques for traditional hypertext-based websites have been extensively studied in the past two decades. More related to our work is the idea behind "scoped crawling" [144] to constrain the exploration to webpages that fall within a particular scope; this way, obtaining *relevant content* becomes more efficient than through a comprehensive crawl. Well-known examples of scoped crawling include hidden-web crawling [110] and focused crawling [117]. Scope can also be defined in terms of geography, language, genre, format, and other content-based properties of webpages. All these techniques focus on the *textual contents* of webpages. Our approach, on the other hand, is geared towards functionality coverage of a web application under test.

Crawling modern Web 2.0 applications engenders more challenges due to the dynamic client-side processing (through JavaScript and AJAX calls), and is relatively a new research area [70, 80, 120, 134]. Many web testing techniques [64, 68, 74, 114, 119, 121, 128, 129, 169, 171] depend on data gathered and models generated through crawling. However, such techniques rely on exhaustive search methods. To the best of our knowledge, this work is the first to propose a feedback-

directed exploration of web applications that enhances different aspects of a test model.

Some metrics have been proposed to measure crawling effectiveness and diversity. Marchetto et al. [64] propose crawlability metrics that capture the degree to which a crawler is able to explore the web application. These metrics combine dynamic measurements such as code coverage, with static indicators, such as lines of code and cyclomatic complexity. Their crawlibility metrics are geared towards the server-side of web applications. Regarding our diversity metrics, although the notion of diversity has been used for classifying search query results [61, 156, 157], we propose new metrics for capturing state and navigational diversity as two aspects of a web application test model.

More related to our work are guided test generation [169] and efficient AJAX crawling techniques [70, 73, 79, 134]. Thummalapenta et al. [169] recently proposed a guided test generation technique for web applications, called WATEG. Their work crawls the application in an exhaustive manner, but directs test generation towards higher coverage of specific business rules, i.e., business logic of an application's GUI. Our work differs from WATEG in two aspects. Firstly, we guide the exploration and not the test generation, and secondly, our objective is to increase code coverage and state diversity, and at the same time decrease test model size. Efficient strategies for AJAX crawling [70, 73, 79, 134] try to discover as many states as possible in the shortest amount of time. The goal of our work is, however, not to crawl the complete state-space as soon as possible (which is infeasible for many industrial web applications), but to drive a partial model, which adequately covers different desired properties of the application.

Static analysis. Researchers have used static analysis to detect bugs and security vulnerabilities in web apps [93, 200]. Jensen et al. [101] model the DOM as a set of abstract JavaScript objects. However, they acknowledge that there are substantial gaps in their static analysis, which can result in false-positives. Because JavaScript is a highly dynamic language, such static techniques typically restrict themselves to a subset of the language. In particular, they ignore dynamic JavaScript interactions with the DOM, which is error-prone and challenging to detect and resolve [143].

Dynamic Analysis. Dynamic analysis and automated testing of JavaScript-based

web applications has gained more attention in the recent past. Marchetto et al. [114] proposed a state-based testing technique for AJAX applications in which semantically interacting event sequences are generated for testing. Mesbah et al. proposed [121] an automated technique for generating test cases with invariants from models inferred through dynamic crawling. JSArt [128] and DoDOM [148] dynamically derive invariants for the JavaScript code and the DOM respectively. Such inferred invariants are used for automated regression and robustness testing.

Artemis [66] is a JavaScript testing framework that uses feedback-directed random testing to generate test inputs. Compared to FEEDEX, Artemis randomly generates test inputs, executes the application with those inputs and uses the gathered information to generate new input. FEEDEX on the other hand, explores the application by dynamically running it and building a state-flow graph that can be used for test generation. The strength of FEEDEX is that it guides the application at runtime towards a better code and more diverse navigational and structural coverage.

Kudzu [158] combines symbolic execution of JavaScript with random test generation to explore sequence of events and produce input values depending on the execution of the control flow paths. Their approach uses a complex string constraint solver to increase the code coverage by producing acceptable input string values. In our work we did not intend to increase the code coverage by considering the problem of input generation for the crawler and only focused on improving the crawling strategy. Compared to Kudzu, FEEDEX is much simpler, more automated, and does not require such heavy computation.

2.7 Conclusions

An enabling technique commonly used for automating web application testing and analysis is crawling. In this work we proposed four aspects of a test model that can be derived by crawling, including *functionality coverage*, *navigational coverage*, *page structural coverage*, and *size of the test model*. We have presented a generic feedback-directed exploration approach, called FEEDEX, to guide the exploration towards client-side states and events that produce a test model with enhanced aspects. Our experiment on six Web 2.0 applications shows that FEEDEX yields higher code coverage, diversity, and smaller test models, compared to traditional

exhaustive methods (DFS, BFS, and random).

In this work, we only measure the client-side coverage. One possible future work can be including the server-side code coverage in the feedback loop to our guided exploration engine. This might help deriving some new states that are dependent on particular server-side code execution. Our state expansion method is currently based on a memory-less greedy algorithm, which might benefit from a learning mechanism to improve its effectiveness.

Chapter 3

Leveraging Existing Tests in User Interface Test Generation for Web Applications

Summary[5]

Current test generation techniques start with the assumption that there are no existing test cases for the system under test. However, many software systems nowadays are already accompanied with an existing test suite. We believe that existing human-written test cases are a valuable resource that can be leveraged in test generation. We demonstrate our insight by focusing on testing modern web applications. To test web applications, developers currently write test cases in popular frameworks such as SELENIUM. On the other hand, most web test generation techniques rely on a crawler to explore the dynamic states of the application. The former requires much manual effort, but benefits from the domain knowledge of the developer writing the test cases. The latter is automated and systematic, but lacks the domain knowledge required to be as effective. We believe combining the two can be advantageous. In this work, we propose a technique to (1) mine the human knowledge present in the form of input values, event sequences, and assertions, in the human-written test suites, (2) combine the inferred knowledge with the power of automated crawling and machine learning, and (3) extend the test suite for uncovered/unchecked portions of the web application under test. Our approach is

[5]This chapter is the extended version of the paper appeared at the IEEE/ACM International Conference on Automated Software Engineering (ASE), 2014 [126] that is in preparation for submission to a software engineering journal.

implemented in a tool called TESTILIZER. An evaluation of our approach indicates that, on average TESTILIZER can generate test suites with improvements by 105% in mutation score (fault detection rate) and by 22% in code coverage, compared to the original human-written test suite.

3.1 Introduction

Testing modern web applications is challenging since multiple languages, such as HTML, JavaScript, CSS, and server-side code, interact with each other to create the application. The final result of all these interactions at runtime is manifested through the DOM and presented to the end-user in the browser. To avoid dealing with all these complex interactions separately, many developers treat the web application as a black-box and test it via its manifested DOM, using testing frameworks such as SELENIUM [42]. These DOM-based test cases are written manually, which is a tedious process with an incomplete result.

On the other hand, many automated testing techniques [74, 121, 159, 169] are based on crawling to explore the state space of the application. Although crawling-based techniques automate the testing to a great extent, they are limited in three areas:

- **Input values:** Having valid input values is crucial for proper coverage of the state space of the application. Generating these input values automatically is challenging since many web applications require a specific type, value, and combination of inputs to expose the hidden states behind input fields and forms.

- **Paths to explore:** Industrial web applications have a huge state space. Covering the whole space is infeasible in practice. To avoid unbounded exploration, which could result in state explosion, users define constraints on the depth of the path, exploration time or number of states. Not knowing which paths are important to explore results in obtaining a partial coverage of a specific region of the application.

- **Assertions:** Any generated test case needs to assert the application behaviour. However, generating proper assertions automatically without hu-

man knowledge is known to be challenging. As a result, many web testing techniques rely on generic invariants [121] or standard validators [66] to avoid this problem.

These two approaches work at the two extreme ends of the spectrum, namely, fully manual or fully automatic. We believe combining the two can be advantageous. In particular, humans may have the domain knowledge to see which interactions are more likely or important to cover than others; they may be able to use domain knowledge to enter valid data into forms; and, they might know what elements on the page need to be asserted and how. This knowledge is typically manifested in manually-written test cases.

In this work, we propose to (1) mine the human knowledge existing in manually-written test cases, (2) combine that inferred knowledge with the power of automated crawling, and (3) extend the test suite for uncovered/unchecked portions of the web application under test. We present our technique and tool called TESTILIZER, which given a set of SELENIUM test cases TC and the URL of the application, automatically infers a model from TC, feeds that model to a crawler to expand by exploring uncovered paths and states, generates assertions for newly detected states based on the patterns learned from TC, and finally generates new test cases.

To the best of our knowledge, this work is the first to propose an approach for extending a web application test suite by leveraging existing test cases. The main contributions of our work include:

- A novel technique to address limitations of automated test generation techniques by leveraging human knowledge from existing test cases.
- An algorithm for mining existing test cases to infer a model that includes (1) input data, (2) event sequences, (3) and assertions, and feeding and expanding that model through automated crawling.
- An algorithm for reusing human-written assertions in existing test cases by exact/partial assertion matching as well as through a learning-based mechanism for finding similar assertions.
- An open source implementation of our technique, called TESTILIZER [47].

- An empirical evaluation of the efficacy of the generated test cases on 8 web applications. On average, TESTILIZER can generate test suites that improve the mutation score (fault detection rate) by 105% and the code coverage by 22%, compared to the original test suite.
- An empirical evaluation of the impact of the original tests effectiveness on the generated tests. Our results suggest that, there is a very strong correlation between the effectiveness of the original tests and the effectiveness of the generated tests.

3.2 Background and Motivation

In practice, web applications are largely tested through their DOM using frameworks such as SELENIUM. The DOM is a dynamic tree-like structure representing user interface elements in the web application, which can be dynamically updated through client-side JavaScript interactions or server-side state changes propagated to the client-side. DOM-based testing aims at bringing the application to a particular DOM state through a sequence of actions, such as filling a form and clicking on an element, and subsequently verifying the existence or properties (e.g., text, visibility, structure) of particular DOM elements in that state. Figure 3.1 depicts a snapshot of a web application and Figure 3.2 shows a simple DOM-based (SELENIUM) test case for that application.

For this work, a DOM state is formally defined as:

Definition 4 (DOM State) *A DOM State $\mathscr{D}S$ is a rooted, directed, labeled tree. It is denoted by a 5-tuple, $< D, Q, o, \Omega, \delta >$, where D is the set of vertices, Q is the set of directed edges, $o \in D$ is the root vertex, Ω is a finite set of labels and $\delta : D \rightarrow \Omega$ is a labelling function that assigns a label from Ω to each vertex in D.* □

The DOM state is essentially an *abstracted* version of the DOM tree of a web application, displayed on the web browser at runtime. This abstraction is conducted through the labelling function δ, the implementation of which is discussed in Section 3.3.1 and Section 3.4.

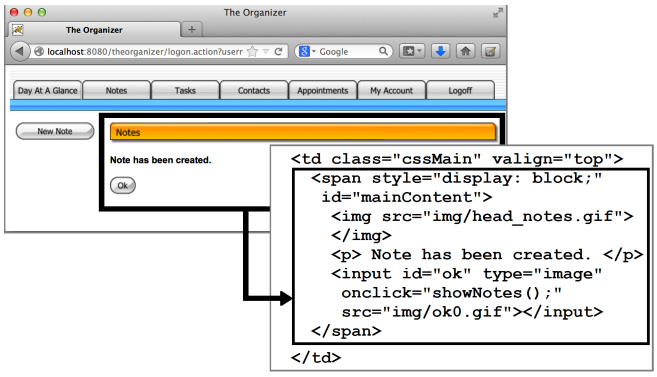


Figure 3.1: A snapshot of the running example (an organizer application) and its partial DOM structure.

Motivation. Overall, our work is motivated by the fact that a human-written test suite is a valuable source of domain knowledge, which can be exploited for tackling some of the challenges in automated web application test generation. Another motivation behind our work is that manually written test cases typically correspond to the most common *happy-paths* of the application that are covered. Automated analysis can subsequently expand these to cover unexplored *bad-weather* application behaviour.

Running example. Figure 3.1 depicts a snapshot of the Organizer [38], a web application for managing notes, contacts, tasks, and appointments, which we use as a running example to show how input data, event paths, and assertions can be leveraged from the existing test cases to generate effective test cases.

Suppose we have a small test suite that verifies the application's functionality for "adding a new note" and "adding a new contact". Due to space constraints, we only show the `testAddNote` test case in Figure 3.2. The test case contains valuable information regarding how to log onto the Organizer (Lines 4–5), what data to insert (Lines 9–10), where to click (Lines 6, 8, 11, 13), and what to assert (Lines 7, 12).

We believe this information can be extracted and leveraged in automated test generation. For example, the paths (i.e., sequence of actions) corresponding to these covered functionalities can be used to create an abstract model of the application, shown in thick solid lines in Figure 3.3. By feeding this model that contains the event sequences and input data leveraged from the test case to a crawler, we can

```
1 @Test
2 public void testAddNote(){
3   get("http://localhost:8080/theorganizer/");
4   findElement(By.id("logon_username")).sendKeys("user");
5   findElement(By.id("logon_password")).sendKeys("pswd");
6   findElement(By.cssSelector("input type="image"")).click();
7   assertEquals("Welcome to The Organizer!", closeAlertAndGetItsText());
8   findElement(By.id("newNote")).click();
9   findElement(By.id("noteCreateShow_subject")).sendKeys("Running ↩
        Example");
10  findElement(By.id("noteCreateShow_text")).sendKeys("Create a simple ↩
        running example");
11  findElement(By.cssSelector("input type="image"")).click();
12  assertEquals("Note has been created.", driver.findElement(By.id("↩
        mainContent")).getText());
13  findElement(By.id("logoff")).click();
14 }
```

Figure 3.2: A human-written DOM-based (SELENIUM) test case for the Organizer.

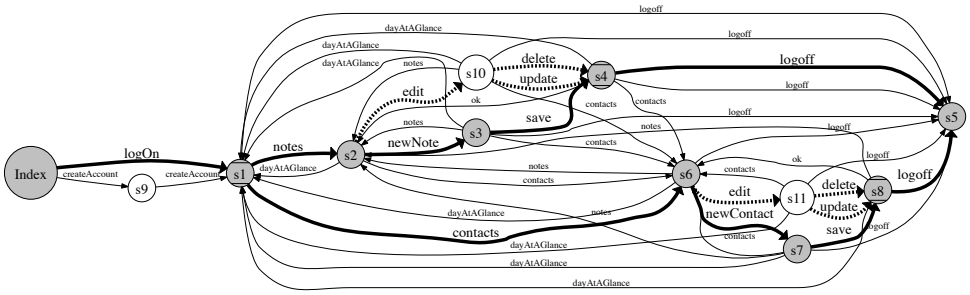


Figure 3.3: Partial view of the running example application's state-flow graph. Paths in thick solid lines correspond to the covered functionalities in existing tests. Alternative paths, shown in thin lines, can be explore using a crawler. Alternative paths through newly detected states (i.e., s10 and s11) are highlighted as dashed lines.

explore *alternative paths* for testing, shown as thin lines in Figure 3.3; alternative paths for deleting/updating a note/contact that result in newly detected states (i.e., s10 and s11) are highlighted as dashed lines.

Further, the assertions in the test case can be used as guidelines for generating new assertions on the newly detected states along the alternative paths. These original assertions can be seen as parallel lines inside the nodes on the graph of Figure 3.3. For instance, line 12 of Figure 3.2 verifies the existence of the text "Note has been created" for an element (`span`) with `id="mainContent"`, which can be

assigned to the DOM state $s4$ in Figure 3.3.

By exploring alternative paths around existing paths and learning assertions from existing assertions, new test cases can be generated. For example the events corresponding to states $\langle Index, s1, s2, s10, s4, s5 \rangle$ can be turned into a new test method `testUpdateNote()`, which on state $s4$, verifies the existence of a `<span>` element with `id="mainContent"`. Further, patterns found in existing assertions can guide us to generate similar assertions for newly detected states (e.g., $s9$, $s10$, $s11$) that have no assertions.

3.3 Approach

Figure 3.4 depicts an overview of our approach. At a high level, given the URL of a web application and its human-written test suite, our approach mines the existing test suite to infer a model of the covered DOM states and event-based transitions including input values and assertions (blocks 1, 2, and 3). Using the inferred model as input, it explores alternative paths leading to new DOM states, thus expanding the model further (blocks 3 and 4). Next it regenerates assertions for the new states, based on the patterns found in the assertions of the existing test suite (block 5), and finally generates a new test suite from the extended model, which is a superset of the original human-written test suite (block 6). We discuss each of these steps in more details in the following subsections.

Scope. The technique presented in this work is applicable only in the context of user interface regression testing. We assume the current version of the given application is correct and thus augmented test suites are not useful to test the same version of the software under test.

3.3.1 Mining Human-Written Test Cases

To infer an initial model, in the first step, we (1) instrument and execute the human-written test suite T to mine an intermediate dataset of test operations. Using this dataset, we (2) run the test operations to infer a state-flow graph (3) by analyzing DOM changes in the browser after the execution of each test operation.

Instrumenting and executing the test suite. We instrument the test suite (block 1 Figure 3.4) to collect information about DOM interactions such as elements ac-

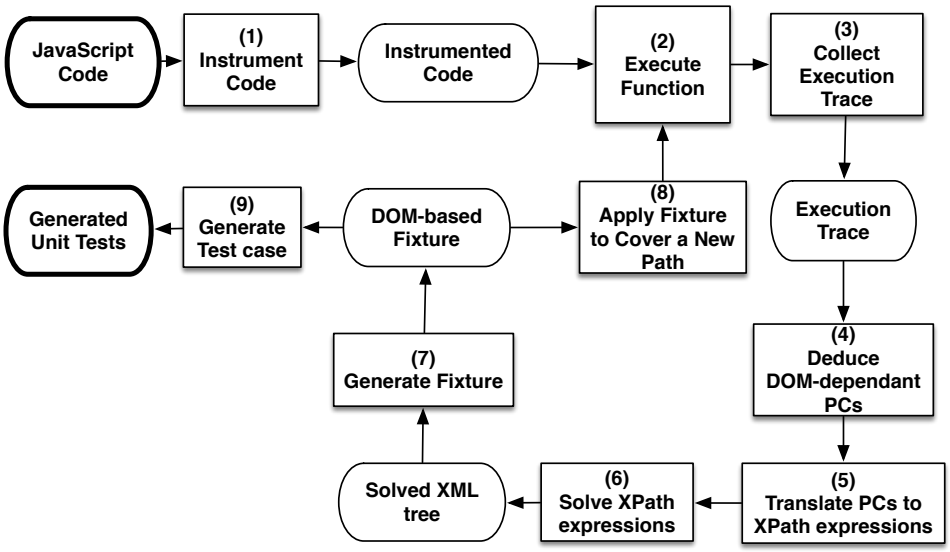


Figure 3.4: Processing view of our approach.

cessed in actions (e.g., clicks) and assertions as well as the structure of the DOM states covered.

Definition 5 (Manual-test Path) *A manual-test path is the sequence of event-based actions performed while executing a human-written test case $t \in T$.* □

Definition 6 (Manual-test State) *A manual-test state is a DOM state located on a manual-test path.* □

The instrumentation hooks into any code that interacts with the DOM in any part of the test case, such as test setup, helper methods, and assertions. Note that this instrumentation does not affect the functionality of the test cases (more details in Section 3.4). By executing the instrumented test suite, we store all observed manual-test paths as an intermediate dataset of test operations:

Definition 7 (Test Operation) *A test operation is a triple <action, target, input>, where action specifies an event-based action (e.g., a click), or an assertion (e.g., verifying a text), target pertains to the DOM element to perform the action on, and input specifies input values (e.g., data for filling a form).* □

The sequence of these test operations forms a dataset that is used to infer the initial model. For a test operation with an assertion as its action, we refer to the target DOM element as a *checked element*, defined as follows:

Definition 8 (Checked Element) *A checked element $ce \in v_i$ is an element in the DOM tree in state v_i, which its existence, its attributes, or its value are checked in an assertion of a test case $t \in T$.* □

For example in line 12 of the test case in Figure 3.2, the text value of the element with ID `"mainContent"` is asserted and thus that element is a checked element. Part of the DOM structure at this state is shown in Figure 3.1, which implies that `<span id="mainContent">` is that checked element.

For each checked element we record the element location strategy used (e.g., XPath, ID, tagname, linktext, or cssselector) as well as the access values and inner-HTML text. This information is later used in the assertion generation process (in Section 3.3.3).

Constructing the initial model. We model a web application as a *State-Flow Graph* (SFG) [120, 121] that captures the dynamic DOM states as nodes and the event-driven transitions between as edges.

Definition 9 (State-flow Graph) *A state-flow graph SFG for a web application $\mathscr{W}$ is a labeled, directed graph, denoted by a 4 tuple $< r, \mathscr{V}, \mathscr{E}, \mathscr{L} >$ where:*

1. *r is the root node (called Index) representing the initial DOM state after $\mathscr{W}$ has been fully loaded into the browser.*
2. *$\mathscr{V}$ is a set of vertices representing the states. Each $v \in \mathscr{V}$ represents an abstract DOM state $\mathscr{DS}$ of $\mathscr{W}$, with a labelling function $\Phi : \mathscr{V} \rightarrow \mathscr{A}$ that assigns a label from $\mathscr{A}$ to each vertex in $\mathscr{V}$, where $\mathscr{A}$ is a finite set of DOM-based assertions in a test suite.*
3. *$\mathscr{E}$ is a set of (directed) edges between vertices. Each $(v_1, v_2) \in \mathscr{E}$ represents a clickable c connecting two states if and only if state v_2 is reached by executing c in state v_1.*
4. *$\mathscr{L}$ is a labelling function that assigns a label, from a set of event types and DOM element properties, to each edge.*
5. *SFG can have multi-edges and be cyclic.* □

Algorithm 2: State-Flow Graph Inference

```
   input : A Web application url URL, a DOM-based test suite TS, crawling constraints CC
   output: A state-flow graph SFG
   Procedure INFERSFG(URL,TS,CC)
   begin
1      TS_inst ← INSTRUMENT(TS)
2      EXECUTE(TS_inst)
3      TOP ← READTESTOPERATIONDATASET()
4      SFG_init ← ∅
5      browser.GOTO(URL)
6      dom ← browser.GETDOM()
7      SFG_init.ADDINITIALSTATE(dom)
8      for top ∈ TOP do
9          C ← GETCLICKABLES(top)
10         for c ∈ C do
11             assertion ← GETASSERTION(top)
12             dom ← browser.GETDOM()
13             robot.FIREEVENT(c)
14             new_dom ← browser.GETDOM()
15             if dom.HASCHANGED(new_dom) then
16                 SFG_init.UPDATE(c,new_dom,assertion)
               end
           end
17         browser.GOTO(URL)
       end
18     SFG_ext ← SFG_init
19     EXPLOREALTERNTIVEPATHS(SFG_ext,CC)
20     return SFG_ext
   end
   Procedure EXPLOREALTERNTIVEPATHS(SFG,CC) begin
21     while CONSTRAINTSATISFIED(CC) do
22         s ← GETNEXTTOEXPLORESTATE(SFG)
23         C ← GETCANDIDATECLICKABLES(s)
24         for c ∈ C do
25             browser.GOTO(SFG.GETPATH(s))
26             dom ← browser.GETDOM()
27             robot.FIREEVENT(c)
28             new_dom ← browser.GETDOM()
29             if dom.HASCHANGED(new_dom) then
30                 SFG.UPDATE(c,new_dom)
31                 EXPLOREALTERNTIVEPATHS(SFG,CC)
               end
           end
       end
   end
```

An example of such a partial SFG is shown in Figure 3.3. The abstract DOM state is an abstracted version of the DOM tree of a web application, displayed on the web browser at runtime. This abstraction can be conducted by using a DOM string edit distance, or by disregarding specific aspects of a DOM tree (such as irrelevant

attributes, time stamps, or styling issues) [121]. The state abstraction plays an important role in reducing the size of SFG since many subtle DOM differences do not represent a proper state change, e.g., when a row is added to a table.

Algorithm 2 shows how the initial SFG is inferred from the manual-test paths. First the initial *index* state is added as a node to an empty SFG (Algorithm 2, lines 5–7). Next, for each test operation in the mined dataset (TOP), it finds DOM elements using the locator information and applies the corresponding actions. If an action is a DOM-based assertion, the assertion is added to the set of assertions of the corresponding DOM state node (Algorithm 2, lines 8–17). The state comparison to determine a new state (line 15) is carried out via a state abstraction function (more explanation in Section 3.4).

3.3.2 Exploring Alternative Paths

At this stage, we have a state-flow graph that represents the covered states and paths from the human-written test suite. In order to further explore the web application to find alternative paths and new states, we seed the graph to an automated crawler (block 4 Figure 3.4).

The exploration strategy can be conducted in various ways: (1) remaining close to the manual-test paths, (2) diverging [123] from the manual-test paths, or (3) randomly exploring. However, in this work, we have opted for the first option, remaining close to the manual-test paths. The reason is to maximize the potential for reuse of and learning from existing assertions. Our insight is that if we diverge too much from the manual-test paths and states, the human-written assertions will also be too disparate and thus less useful. We evaluate this exploration approach in Section 3.5.

To find alternative paths, events are generated on DOM elements and if such an event mutates the DOM state, the new state and the corresponding event transition are added to the SFG. Note that the state comparison to determine a new state (line 29) is done via the same state abstraction function used before (line 15). The procedure `ExploreAlternativePaths` (Algorithm 2, lines 21–31) repeatedly crawls the application until a crawling constraint (e.g., maximum time, or number of states) is reached. The exploration is guided by the manual-test states while ex-

ploring alternative paths (Line 22); `GetNexToExploreState` decides which state should be expanded next. It gives the highest priority to the manual-test states and when all manual-test states are fully expanded, the next immediate states found are explored further. More specifically, it randomly selects a manual-test state that has some unexercised candidate clickables and navigates the application through that state. The `GetCandidateClickable` method (Line 23) returns a set of candidate clickables that can be applied on the selected state. This process is repeated until all manual-test states are fully expanded.

For example, consider the manual-test sates shown in grey circles in Figure 3.3. The alternative paths exploration method starts by randomly selecting a state such as $s2$, navigating the application to reach to that state from the *Index* state, and firing an event on $s2$ which can result in an alternative state $s10$.

3.3.3 Regenerating Assertions

The next step is to generate assertions for the new DOM states in the extended SFG (block 5 Figure 3.4). In this work, we propose to leverage existing assertions to regenerate new ones. By analyzing human-written assertions we can leverage information about (1) portions of the page that are more important for testing; for example, a banner section or decoration parts of a page might not be as important as an inner content that changes according to a main functionality, (2) patterns in the page that are part of a template. Therefore, extracting important DOM patterns from existing assertions may help us in generating new but similar assertions.

We formally define a DOM-based assertion as a function $\mathscr{A} : (s, c) \rightarrow \{True, False\}$, where s is a DOM state, and c is a DOM condition to be checked. It returns *True* if s matches/satisfies the condition c, denoted by $s \models c$, and *False* otherwise. We say that an assertion A subsumes (implies) assertion B, denoted by $A \Longrightarrow B$, if $A \rightarrow True$, then $B \rightarrow True$. This means that B can be obtained from A by weakening A's condition. In this case, A is more specific/constrained than B. For instance, an assertion verifying the existence of a checked element `<span id="mainContent">` can be implied by an assertion which verifies both the existence of that element and its attributes/textual values.

Algorithm 3 shows our assertion regeneration procedure. We consider each

manual-test state s_i (Definition 6) in the SFG and try to reuse existing associated assertions in s_i or generate new ones based on them for another state s_j. We extend the set of DOM-based assertions in three forms: (1) reusing the *same* assertions from manual-test states for states without such assertions, (2) regenerating assertions with the *exact* assertion pattern structure as the original assertions but adapted for another state, and (3) learning assertions structures of the original assertions, and generate *similar* ones for another state.

Assertion Reuse

As an example for the assertion reuse, consider Figure 3.3 and the manual-test path with the sequence of states $\langle Index, s1, s2, s3, s4, s5\rangle$ for adding a note. Assertions in Figure 3.2 line 7 and 12 are associated to states $s1$ and $s4$, respectively. Suppose that we explore an alternative path for deleting a note with the sequence $\langle Index, s1, s2, s10, s4, s5\rangle$, which was not originally considered by the developer. Since the two test paths share a common path from $Index$ to $s1$, the assertion on $s1$ can be reused for the new test case (note deletion) as well. This is a simple form of assertion reuse on new test paths.

Assertion Regeneration

We regenerate two types of precondition assertions namely *exact element-based assertions*, and *exact region-based assertions*. By "exact" we mean repetition of the same structure of an original assertion on a checked element.

The rationale behind our technique is to use the location and properties of checked elements and their close-by neighbourhood in the DOM tree to regenerate assertions, which focus on the exact repeated structures and patterns in other DOM states. This approach is based on our intuition that checking the close-by neighbour of checked elements is similarly important.

Exact element assertion generation. We define assertions of the form $\mathscr{A}(s_j, c(e_j))$ with a condition $c(e_j)$ for element e_j on state s_j. Given an existing checked element (Definition 8) e_i on a DOM state s_i, we consider 2 conditions as follows:

1. *ElementFullMatched*: If a DOM state s_j contains an element with exact tag, attributes, and text value as e_i, then *reuse* assertion on e_i for checking e_j on

Algorithm 3: Assertion Regeneration

```
    input : An extended state-flow graph SFG = < r,V,E,L >
    Procedure REGENERATEASSERTIONS(SFG)
    begin
        /*Learn from DOM elements in the manual-test states*/
1       dataset ←MAKEDATASET(SFG.GETMANUALTESTSTATES())
2       TRAIN(dataset)
3       for s_i ∈ V do
4           for ce ∈ s_i.GETCHECKEDELEMENTS() do
5               assert ← ce.GETASSERTION()
6               cer ← ce.GETCHECKEDELEMENTREGION()
7               s_i.ADDREGFULLASSERTION(cer)
8               for s_j ∈ V & s_j ≠ s_i do
9                   dom ← s_j.GETDOM()
                    /*Generate exact element assertion for s_j*/
10                  if ELEMENTFULLMATCHED(ce,dom) then
11                      s_j.REUSEASSERTION(ce,assert)
                    end
12                  else if ELEMENTTAGATTMATCHED(ce,dom) then
13                      s_j.ADDELEMTAGATTASSERTION(ce)
                    end
                    /*Generate exact region assertion for s_j*/
14                  if REGIONFULLMATCHED(cer,dom) then
15                      s_j.ADDREGFULLASSERTION(cer)
                    end
16                  else if REGIONTAGATTMATCHED(cer,dom) then
17                      s_j.ADDREGTAGATTASSERTION(cer)
                    end
18                  else if REGIONTAGMATCHED(cer,dom) then
19                      s_j.ADDREGTAGASSERTION(cer)
                    end
                end
            end
            /* Generate similar region assertions for s_i */
20          for be ∈ s_i.GETBLOCKELEMENTS() do
21              if PREDICT(be) == 1 then
22                  s_i.ADDREGTAGATTASSERTION(be.GETREGION())
                end
            end
        end
    end
```

s_j.

2. *ElementTagAttMatched*: If a DOM state s_j contains an element e_j with exact tag and attributes, but different text value as e_i, then *generate* assertion on e_j for checking its tag and attributes.

Table 3.1 summarizes these conditions. An example of a *generated* assertion is

Table 3.1: Summary of the assertion reuse/regeneration conditions for an element e_j on a DOM state s_j, given a checked element e_i on state s_i.

Condition	Description
ElementFullMatched	$Tag(e_i)$=$Tag(e_j)$ $\wedge$ $Att(e_i)$=$Att(e_j)$ $\wedge$ $Txt(e_i)$=$Txt(e_j)$
ElementTagAttMatched	$Tag(e_i)$=$Tag(e_j)$ $\wedge$ $Att(e_i)$=$Att(e_j)$
RegionFullMatched	$Tag(R(e_i,s_i))$=$Tag(R(e_j,s_j))$ $\wedge$ $Att(R(e_i,s_i))$=$Att(R(e_j,s_j))$ $\wedge$ $Txt(R(e_i,s_i))$=$Txt(R(e_j,s_j))$
RegionTagAttMatched	$Tag(R(e_i,s_i))$=$Tag(R(e_j,s_j))$ $\wedge$ $Att(R(e_i,s_i))$=$Att(R(e_j,s_j))$
RegionTagMatched	$Tag(R(e_i,s_i))$=$Tag(R(e_j,s_j))$

`assertTrue(isElementPresent(By.id("mainContent")))` which checks the existence of a checked element with ID `"mainContent"`. Such an assertion can be evaluated in any state in the SFG that contains that DOM element (and thus meets the precondition). Note that we could also propose assertions in case of mere tag matches, however, such assertions are not generally considered useful as they are too generic.

Exact region assertion generation. We define the term *checked element region* to refer to a close-by area around a checked element:

Definition 10 (Checked Element Region) *For a checked element e on state s, a checked element region $\mathscr{R}(e,s)$, is a function $\mathscr{R}:(e,s)\rightarrow\{e,\mathscr{P}(e),\mathscr{C}h(e)\}$, where $\mathscr{P}(e)$ and $\mathscr{C}h(e)$ are the parent node, and children nodes of e respectively.* □

For example, for the element e = `<span id="mainContent">` (Figure 3.1), which is in fact a checked element in line 12 of Figure 3.2 (at state $s4$ in Figure 3.3), we have $\mathscr{R}(e,s4)=\{e,\mathscr{P}(e),\mathscr{C}h(e)\}$, where $\mathscr{P}(e)$=`<td class="cssMain" valign="top">`, and $\mathscr{C}h(e)$={`<img src="img/head_notes.gif">`, `<p>`, `<input id="ok" src="img/ok0.gif">`}.

We define assertions of the form $\mathscr{A}(s_j,c(\mathscr{R}(e_j,s_j)))$ with a condition $c(\mathscr{R}(e_j,s_j))$ for the region $\mathscr{R}$ of an element e_j on state s_j. Given an existing checked element e_i on a DOM state s_i, we consider 3 conditions as follows:

1. *RegionFullMatched*: If a DOM state s_j contains an element e_j with exact tag, attributes, and text values of $\mathscr{R}(e_j,s_j)$ as $\mathscr{R}(e_i,s_i)$, then *generate* assertion on $\mathscr{R}(e_j,s_j)$ for checking its tag, attributes, and text values.

2. *RegionTagAttMatched*: If a DOM state s_j contains an element e_j with exact tag, and attributes values of $\mathscr{R}(e_j, s_j)$ as $\mathscr{R}(e_i, s_i)$, then *generate* assertion on $\mathscr{R}(e_j, s_j)$ for checking its tag and attributes values.

3. *RegionTagMatched*: If a DOM state s_j contains an element e_j with exact tag value of $\mathscr{R}(e_j, s_j)$ as $\mathscr{R}(e_i, s_i)$, then *generate* assertion on $\mathscr{R}(e_j, s_j)$ for checking its tag value.

Note that the assertion conditions are relaxed one after another. In other words, on a DOM state s, if $s \models RegionFullMatched$, then $s \models RegionTagAttMatched$; and if $s \models RegionTagAttMatched$, then we have $s \models RegionTagMatched$. Consequently it suffices to use the most constrained assertion. We use this property for reducing the number of generated assertions in Section 22.

Table 3.1 summarizes these conditions. Assertions that we generate for a checked element region, are targeted around a checked element. For instance, to check if a DOM state contains a checked element region with its tag, attributes, and text values, an assertion will be *generated* in the form of `assertTrue(isElementRegionFullPresent(parentElement,element,childrenElements))`, where `parentElement`, `element`, and `childrenElements` are objects reflecting information about that region on the DOM.

For each checked element *ce* on s_i, we also generate a `RegionFull` type of assertion for checking its region, i.e., verifying *RegionFullMatched* condition on s_i (Algorithm 3 lines 6–7). Lines 10–13 perform exact element assertion generation. The original assertion can be reused in case of *ElementFullMatched* (line 11). Lines 14–19 apply exact region assertion generation based on the observed matching. Notice the hierarchical selection which guarantees generation of more specific assertions.

Learning Assertions for Similar Regions

The described exact element/region assertion regeneration techniques only consider the exact repetition of a checked element/region. However, there might be many other DOM elements that are similar to the checked elements but not exactly the same. For instance, consider Figure 3.2 line 12 in which a `<span id="mainContent">` element was checked in an assertion. If in another state,

a `<div id="centreDiv">` element exists, which is similar to the `<span>` element in certain aspects such as content and position on the page, we could generate a DOM-based assertion for the `<div>` element in the form of `assertTrue(isElementPresent(By.id("centreDiv")));`.

We view the problem of generating similar assertions as a classification problem which decides whether a block level DOM element is *important to be checked* by an assertion or not. To this end, we apply machine learning to train a classifier based on the features of the checked elements in existing assertions. More specifically, given a training dataset $\mathscr{D}$ of n DOM elements in the form $\mathscr{D} = \{(\mathbf{x}_i, y_i) \mid \mathbf{x}_i \in \mathbb{R}^p, y_i \in \{-1, 1\}\}_{i=1}^{n}$, where each $\mathbf{x}_i$ is a p-dimensional real vector representing the features of a DOM element e_i, and y_i indicates whether e_i is a checked element $(+1)$ or not (-1), the classification function $\mathscr{F} : \mathbf{x_j} \rightarrow y_i$ maps a feature vector $\mathbf{x}_j$ to its class label y_j. To do so, we use Support Vector Machine (SVM) [179] to find the max-margin hyperplane that divides the elements with $y_i = 1$ from those with $y_i = -1$. We chose SVM because it outperforms other classification algorithms that we tried. In the rest of this subsection, we describe our used features, how to label the feature vectors, and how to generate similar region DOM-based assertions.

DOM element features. We present a set of features for a DOM element to be used in our classification task. A feature extraction function $\psi : e \rightarrow \mathbf{x}$ maps an element e to its feature set $\mathbf{x}$. Many of these features are based on and adapted from the work in [164], which performs page segmentation ranking for adaptation purpose. The work presented a number of spatial and content features that capture the importance of a webpage segment based on a comprehensive user study. Although they targeted a different problem than ours, we gained insight from their empirical work and use that to reason about the importance of a page segment for testing purposes. Our proposed DOM features are presented in Table 3.2. We normalize feature values between [0–1] as explained in Table 3.2, to be used in the learning phase. For example, consider the element e = `<span>` in Figure 3.1, then $\psi(e)$ = $\langle$0.5, 0.7, 0.6, 0.5, 1, 0.2, 0, 0.3$\rangle$ corresponding to features `BlockCenterX`, `BlockCenterY`, `BlockWidth`, `BlockHeight`, `TextImportance`, `InnerHtmlLength`, `LinkNum`, and `ChildrenNum`, respectively.

Table 3.2: DOM element features used to train a classifier.

Feature Name	Definition	Rationale
ElementCenterX, ElementCenterY	The (x,y) coordinates of the centre of a DOM element. BlockCenterX and BlockCenterY are normalized by dividing by PageWidth and PageHeight (i.e., the width and height of the whole page) respectively.	Web designers typically put the most important information (main content) in the centre of the page, the navigation bar on the header or on the left side, and the copyright on the footer [164]. Thus, if the (x,y) coordinate of the centre of a DOM block is close to the (x,y) coordinate of the web page centre, that block is more likely to be part of the main content.
ElementWidth, ElementHeight	These are the width and height of the DOM element, which are also normalized by dividing by PageWidth and PageHeight, respectively.	The width and height of an element can be an indication for an important segment. Intuitively, large blocks typically contain much irrelevant noisy content [164].
TextImportance	This binary value feature indicates whether the block element contains any visually important text.	Text in bold/italic style, or header elements (such as h1, h2,..., h5) to highlight and emphasize textual content usually imply importance in that region.
InnerHtmlLength	The innerHtmlLength is the length of all HTML code string (without whitespace) in the element block. We normalize this value by dividing it by InnerHtmlLength of the whole page.	The normalized feature value can indicate the block content size. Intuitively, blocks with many sub-blocks and elements are considered to be less important than those with fewer but more specific content [164].
LinkNum	The LinkNum is the number of anchor (hyperlink) elements inside the DOM element and is normalized by the link number of the whole page.	If a DOM region contains clickables, it is likely part of a navigational structure (menu) and not part of the main content [164].
ChildrenNum	The ChildrenNum is the number of child nodes under a DOM node. We normalize this value by dividing it by a constant number (10 in our implementation) and setting the normalized value to 1 if it exceeds 1.	We have observed in many DOM-based test cases that checked elements do not have a large number of children nodes. Therefore, this feature can be used to discourage elements with many children to be selected for a region assertion, to enhance test readability.

Labelling the feature vectors. For the training phase, we need a dataset of feature vectors for DOM elements annotated with +1 (important to be checked in assertion) and -1 (not important for testing) labels. After generating a feature vector for each "checked DOM element", we label it by +1. For some elements with label -1, we consider those with "most frequent features" over all the manual-test states. Unlike previous work that focuses on DOM invariants [148], our insight is that DOM subtrees that are invariant across manual-test states, are less important to be checked in assertions. In fact, most modern web applications execute a significant amount of client-side code in the browser to mutate the DOM at runtime; hence DOM elements that remain unchanged across application execution are more likely to be related to fixed (server-side) HTML templates. Consequently, such elements are less likely to contain functionality errors. Thus, for our feature vectors we consider all block elements (such as `div`, `span`, `table`) on the manual-test states and rank them in a decreasing order based on their occurrences. In order to have a balanced dataset of items belonging to $\{$-1,+1$\}$, we select the k-top ranked (i.e., k most frequent) elements with label -1, were k equals the number of label +1 samples.

Predicting new DOM elements. Once the SVM is trained on the dataset, it is used to predict whether a given DOM element should be checked in an assertion (algorithm 3, Lines 20–23). If the condition $\mathscr{F}(\psi : e \rightarrow \mathbf{x})$=1 holds, we generate a `RegionTagAtt` type assertion (i.e., checking tag and attributes of a region). We do not consider a `RegionFull` (i.e., checking tag, attributes, and text of a region) assertion type in this case because we are dealing with a *similar* detected region, not an exact one. Also, we do not generate a `RegionTag` assertion type because a higher priority should be given to the similar region-based assertions.

Assertion Minimization

The proposed assertion regeneration technique can generate many DOM-based assertions per state, which in turn can make the generated test method hard to comprehend and maintain. Therefore, we (1) avoid generating redundant assertions, and (2) prioritize assertions based on their constraints and effectiveness.

Avoiding redundant assertions. A new reused/generated assertion for a state

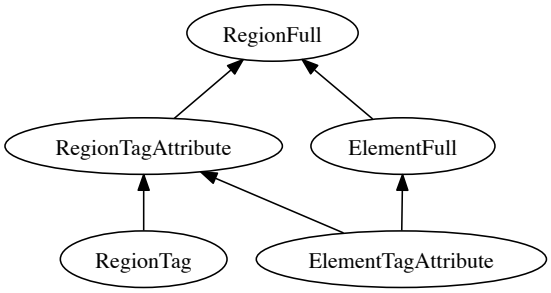


Figure 3.5: The subsumption lattice showing the is-subsumed-by relation for generated assertions.

(Algorithm 3, lines 5, 11, 13, 15, 17, 19, and 22), might already be subsumed by, or may subsume other assertions, in that state. For example an exact element assertion which verifies the existence of a checked element `<span id="mainContent">` can be subsumed by an exact region assertion which has the same `span` element in either its checked element, parent, or its children nodes. Assertions that are subsumed by other assertions are redundant and safely eliminated to reduce the overhead in testing time and increase the readability and maintainability of test cases. For a given state s with an existing assertion B, a new assertion A generated for s is treated as follows:

$$\begin{cases} \text{Discard } A & \text{; if } B \Longrightarrow A \\ \text{Replace } B \text{ with } A & \text{; if } A \Longrightarrow B \text{ and} \\ & \quad B \notin \text{original assertions} \\ \text{Add } A \text{ to } s & \text{; otherwise} \end{cases}$$

Figure 3.5 depicts the subsumption lattice in which arrows indicate is-subsumed-by relation. For instance, assertions of type *RegionFull* (checking the tag, attributes, and text of a region) subsume all other weaker assertions such as *RegionTagAttribute* (checking the tag and attributes of a region), or *ElementFull* (checking the tag, attributes, and text of an element).

Prioritizing assertions. We prioritize the generated assertions such that given a maximum number of assertions to produce per state, the more effective ones are ranked higher and chosen. We prioritize assertions in each state in the following or-

der; the highest priority is given to the original human-written assertions. Next are the reused, the `RegionFull`, the `RegionTagAtt`, the `ElementTagAtt`, and the `RegionAtt` assertions. This ordering gives higher priorities to more specific/constrained assertions first.

3.3.4 Test Suite Generation

In the final step, we generate a test suite from the extended state-flow graph. Each path from the *Index* node to a sink node (i.e., node without outgoing edges) in the SFG is transformed into a unit test. Loops are included once. Each test case captures the sequence of events as well as any assertions for the target states. To make the test case more readable for the developers, information (such as tag name and attributes) about related DOM elements is generated as code comments.

Filtering unstable assertions. After generating the extended test suite, we make sure that the reused/regenerated assertions are stable, i.e., do not falsely fail, when running the test suite on an unmodified version of the web application. Some of these assertions are not only DOM related but also depend on the specific path through which the DOM state is reached. Our technique automatically identifies and filters these false positive cases from the generated test suite. This is done through executing the generated test suite and eliminating failing assertions form the test cases iteratively, until all tests pass successfully.

3.4 Implementation

The approach is implemented in a tool, called TESTILIZER, which is publicly available [47]. The state exploration component is built on top of CRAWLJAX [120]. TESTILIZER requires as input the source code of the human-written test suite and the URL of the web application. Our tool does not modify the web application thus the source code of web application is not required. TESTILIZER currently supports SELENIUM tests, however, our approach can be easily applied to other DOM-based tests as well.

To instrument the test cases, we use JavaParser [32] to get an abstract syntax tree. We instrument all DOM related method calls and calls with arguments that have DOM element locaters. The identified DOM related method call expressions,

such as `findElement` in SELENIUM tests, will be then wrapped by our own method call that enables us to collect information such as page source, test case name, elements under test, element locator, and assertions, from test execution at runtime. We also log the DOM state after every event in the tests, capable of changing the DOM. Note that this instrumentation does not affect the functionality of the test cases.

For the state abstraction function (as defined in Definition 4), we generate an abstract DOM state by ignoring recurring structures (patterns such as table rows and list items), textual content (such as ignoring the text node "Note has been created" in the partial DOM shown in Figure 3.1), and contents in the `<script>` tags. For the classification step, we use LIBSVM [72], which is a popular library for support vector machines.

3.5 Empirical Evaluation

To assess the efficacy of our proposed technique, we have conducted a controlled experiment to address the following research questions:

RQ1. How much of the information (input data, event sequences, and assertions) in the original human-written test suite is leveraged by TESTILIZER?

RQ2. How successful is TESTILIZER in regenerating effective assertions?

RQ3. How does the effectiveness of the original test suite affect the effectiveness of the generated tests?

RQ4. Does TESTILIZER improve code coverage?

Our experimental data along with the implementation of TESTILIZER are available for download [47].

3.5.1 Experimental Objects

We selected eight open-source web applications that contain existing SELENIUM test suites and fall under different application domains. Moreover, since our effectiveness assessment (RQ2) is based on the client-side mutation analysis (as ex-

Table 3.3: Experimental objects.

ID	Name	SLOC	# Test Methods	# Assertions	# Mutants
1	Claroline (v 1.11.7)	PHP (295K) JS (36K)	23	35	38
2	PhotoGallery (v 3.31)	PHP (5.6K) JS (1.5K)	7	18	26
3	WolfCMS (v 0.7.8)	PHP (35K) JS (1.3K)	12	42	28
4	EnterpriseStore (v 1.0.0)	Java (3K) JS (57K)	19	17	52
5	CookeryBook	PHP (25K) JS (0.4K)	6	25	22
6	AddressBook (v 8.2.5)	PHP (30.2K) JS (1.1K)	13	13	30
7	StudyRoom	PHP (1.6K) JS (10.6K)	12	23	38
8	Brotherhood (v 0.4.5)	PHP (212K) JS (0.8K)	10	10	41

plained in Section 3.5.2), we chose applications that make extensive use of client-side JavaScript.

The experimental objects and their properties are shown in Table 3.3. *Claroline* [21] is a collaborative e-learning environment, which allows instructors to create and administer courses. *Phormer* [39] is a photo gallery equipped with upload, comment, rate, and slideshow functionalities. *WolfCMS* [18] is a content management system. *EnterpriseStore* [6] is an enterprise asset management web application. *CookeryBook* [5] is an adaptable cookery book that customizes the presentation of recipes. *AddressBook* [1] is an address/phone book. *SimulatedStudyRoom* [14] is a a web interface for the projector system of a study room, which simulates an outdoor study environment. *Brotherhood* [2] is an online social networking platform for sharing interests, activities, and updates, sending messages, and uploading photos.

3.5.2 Experimental Setup

Our experiments are performed on Mac OS X, running on a 2.3GHz Intel Core i7 CPU with 8 GB memory, and FireFox 38.0.

Table 3.4: Test suite generation methods evaluated.

Test Suite Generation Method	Exploration Strategy	Action Sequence Generation Method	Assertion Generation Method
ORIG	Manual	Manual	Manual
EXND+AR (TESTILIZER)	Explore around manual-test states in the initial SFG (EXND)	Traversing paths in the extended SFG generated from the original tests	Assertion regeneration (AR)
EXND*+AR (TESTILIZER*)	Explore around newly discovered states in the extended SFG (EXND*)	Traversing paths in the extended SFG generated from the original tests	Assertion regeneration (AR)
EXND+RND	Explore around states in the initial SFG (EXND)	Traversing paths in the extended SFG generated from the original tests	Random (RND)
RAND+RND	Random crawling (RAND)	Traversing paths in the SFG generated by random crawling	Random (RND)

Independent Variables

We compare the original human-written test suites with the test suites generated by TESTILIZER.

Test suite generation method. We evaluate different test suite generation methods for each application as presented in Table 3.4. We compare EXND+AR (TESTILIZER) with 4 baselines, (1) ORIG: original human-written test suite, (2) EXND*+AR (TESTILIZER*): differs from TESTILIZER in the alternative path exploration approach, i.e., EXND* explores around the newly discovered states rather than exploring around states in the initial SFG. In other word, paths in the extended SFG are more diverse from manual-test paths. This is to evaluate our proposed exploration approach (Section 3.3.2). (3) EXND+RND: test suite generated by traversing the extended SFG, equipped with random assertion generation, and (4) RAND+RND: random exploration and random assertion generation.

In random assertion generation, for each state we generate element/region assertions by randomly selecting from a pool of DOM-based assertions. These random assertions are based on the existence of an element/region in a DOM state. Such assertions are expected to pass as long as the application is not modified. However, due to our state abstraction mechanism, this can result in unstable assertions, which are also automatically eliminated following the approach explained in Section 3.3.4.

We further evaluate various instantiations of our assertion generation in EXND+AR:

- (a) original assertions,
- (b) reused assertions (Section 22),
- (c) exact generated (Section 22),
- (d) similar region generated (Section 22),
- (e) a combination of all these types.

Exploration constraints. We confine the max exploration time to five minutes in our experiments to evaluate generated models of different sizes. Suppose in the EXND/EXND* approaches, TESTILIZER/TESTILIZER* spend time t to generate the initial SFG for an application. To make a fair comparison, we add this time t to the 5 minutes for the RAND exploration approach. We set no limits on the crawling depth nor the maximum number of states to be discovered while looking for alternative paths. Note that for EXND/EXND* and RAND crawling, after a clickable element on a state was exercised, the crawler resets to the index page and continues crawling from another chosen state.

Maximum number of generated assertions. We set the maximum number of generated assertions for each state to five. To have a fair comparison, also for the EXND+RND and RAND+RND methods, we perform the same assertion prioritization used in TESTILIZER and select the top ranked.

Learning parameters. We set the SVM's kernel function to the Gaussian RBF, and use 5-fold cross-validation for tuning the model and feature selection.

Dependent Variables

Original coverage. To assess how much of the information including input data, event sequences, and assertions of the original test suite is leveraged (RQ1), we measure the state and transition coverage of the initial SFG (i.e., SFG mined from the original test cases). We also measure how much of the unique assertions and unique input data in the original test cases has been utilized.

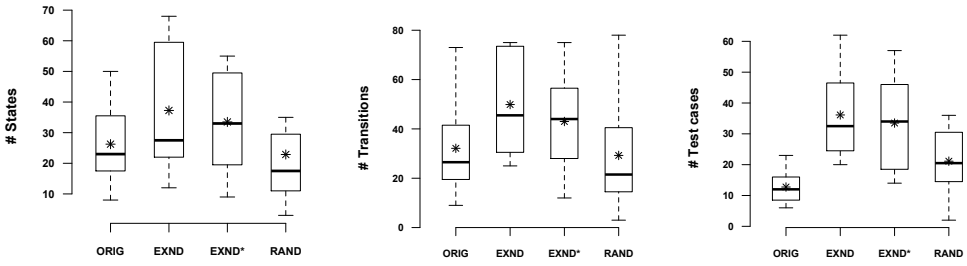
Table 3.5: DOM mutation operators.

ID	Mutation Operator Description
1	Changing the id/tag name in `getElementById` and `getElementByTagName` methods.
2	Changing the attribute name/value in `setAttribute`, `getAttribute` and `removeAttribute` methods.
3	Removing $ sign that returns a jQuery object.
4	Changing the `innerHTML` assignment of an element to an empty string.
5	Swapping `innerHTML` and `innerText` properties.
6	Changing the name of the property/class/element in the `addClass`, `removeClass`, `removeAttr`, `remove`, `attr`, and `css` methods in jQuery.
7	Changing request type (`Get`/`Post`), URL, and the value of the boolean asynch argument in the `request.open` method.

Mutation score. In the context of regression testing, a realistic evaluation of the effectiveness of generated tests requires different versions of the same subject applications. However, we did not have such multiple versions that work fine with the existing tests. Thus, to answer RQ2 (assertions effectiveness), we evaluate the DOM-based fault detection capability of TESTILIZER through automated first-order mutation analysis. The test suites are evaluated based on the number of detected (or killed) mutants, which is known as the *mutation score*. Mutation score is used to measure the effectiveness of a test suite in terms of its ability to detect faults [186]. The mutation score is computed as the percentage of killed mutants over all (non-equivalent) mutants.

We apply the DOM, jQuery, and XHR mutation operators at the JavaScript code level, shown in Table 3.5, which are based on a study of common mistakes made by web developers [129]. For each experimental object we generate as much possible mutated (faulty) versions, each containing one injected artificial defect into the JavaScript code. To do so, we carefully searched within JavaScript files of each subject system for target mutation locations based on the defined mutation operators. The last column of Table 3.3 presents the number of injected mutants for each application with an average of 34 generated mutants.

Correlation. For RQ3 (the impact of original tests effectiveness), we use the non-parametric Spearman's correlation to quantitively study the relationship between the effectiveness of the original and the generated test suites. The Spearman's correlation coefficient measures the monotonic relationship between two continuous random variables. The Spearman correlation coefficient does not require the data



(a) Number of states. **(b)** Number of transitions. **(c)** Number of test cases.

Figure 3.6: Box plots of number of states, transitions, and test cases for different test suites. Mean values are shown with (*).

to be normally distributed [103].

In order to measure the influence of the effectiveness of the original tests on the effectiveness of the generated test, we compute the correlation between the mutation score of the original tests and the generated tests. For this purpose, we generate different versions of a test suite by removing some test cases, which can affect the mutation score of the original test suite as well as the generated test suite. Since applying mutation analysis on many different variants of test suites for all of the experimental objects is very costly, we only considered the *Phormer* photo gallery application in this study and executed test suites 780 times. The original written test suite contains 7 test cases. We generate different combinations on test suites with 1 test case (7 variants) and 6 test cases (7 variants) out of 7 tests, resulting in 14 test suites. For each of the 14 test suites plus the original one (15 in total) we use TESTILIZER to generate an extended test suite, resulting in 30 test suites (15 original and 15 generated). Finally we calculate the mutation score for each of the 30 test suites on 26 mutants, i.e., 780 test suite executions, and measure the correlation between the mutation score of the 15 original and the 15 generated tests.

Code coverage. Code coverage has been commonly used as an indicator of the quality of a test suite by identifying under-tested parts, while it does not directly imply the effectiveness of a test suite [97]. Although TESTILIZER does not target code coverage maximization, to address RQ4, we compare the JavaScript code coverage of the different test suites using JSCover [33].

3.5.3 Results

Box plots in Figure 3.6 depict characteristics of the evaluated test suites with respect to the exploration strategies. For the random exploration (RAND), we report the average values over three runs. As expected, the number of states, transitions, and generated test cases are higher in TESTILIZER/TESTILIZER* (shown as EXND/EXND*). RAND on average generates fewer states and transitions, but more test cases compared to the original test suite. This is mainly due to the fact that in the SFG generated by RAND, there are more paths from the `Index` to the sink nodes than in the SFG mined from the original test suite. Results indicate that the number of states, transitions, and test cases are not that different in EXND or EXND* exploration methods.

Figure 3.7 presents the average number of assertions per state before and after filtering the unstable ones (as explained in Section 3.3.4). The difference between the number of actual generated assertions and the stable ones reveals that our generated assertions (combined, similar/exact generated) are more stable than the random approach. The reduction percentage is 17%, 48%, 30%, 9%, 13%, 15%, 38%, and 35% for the original, reused, exact generated, similar generated, combined (TESTILIZER), TESTILIZER*, EXND+RND and RAND+RND, respectively.

A major source of this instability is the selection of dynamic DOM elements in the generated assertions. For instance, RND (random assertion generation) selects many DOM elements with dynamic time-based attributes. Also the more restricted an assertion is, the less likely it is to remain stable in different paths. This is the case for some of the (1) reused assertions that replicate the original assertions and (2) exact generated ones specially `FullRegionMatchs` type. On the other hand, learned assertions are less strict (e.g., `AttTagRegionMatchs`) and are thus more stable.

***Finding 1**: Test suites generated by* TESTILIZER *compared to randomly generated tests are more stable, i.e., they have fewer false failures.*

Overall, the test suite generated by TESTILIZER, on average, consists of 9% original assertions, 14% reused assertions, 33% exact generated assertions, and 44% of similar learned assertions.

Original SFG coverage (RQ1). Table 3.6 shows the usage of original test suite

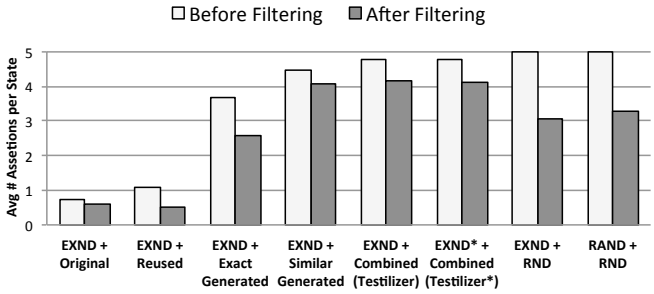


Figure 3.7: Average number of assertions per state, before and after filtering unstable assertions.

Table 3.6: Statistics of the original test suite information usage, average over experimental objects.

Test Suite Generation	Original State Coverage	Original Transition Coverage	Original Input Data Usage	Original Assertion Usage
EXND/EXND*+AR	98%	96%	100%	100%
EXND+RND	98%	96%	100%	0%
RAND+RND	70%	63%	0%	0%

information (RQ1) by different test suites. As expected TESTILIZER, which leverages the event sequences and inputs of the original test suite, has on average almost full state (98%) and transition (96%) coverage of the initial model. This is the same for EXND and EXND* exploration strategies. The few cases missed are due to the traversal algorithm we used, which has limitations on dealing with cycles in the graph that do not end with a sink node and thus are not generated. Note that we can select the missing cases from the original manual-written test suite and add them to the generated test suite.

On average, the RAND exploration approach covered 70% of the states and of 63% the transitions, without using input data except for the login data, which was provided to the crawler. By analyzing the generated test suites, we found that the missing of original states and transitions are mainly due to the lack of proper input data.

***Finding 2**: Randomly generated tests are unable to produce many of the states and transitions of the model extracted from the original written tests due to the lack of proper inputs.*

Test suite effectiveness (RQ2). Figure 3.8 depicts a comparison of mutation scores

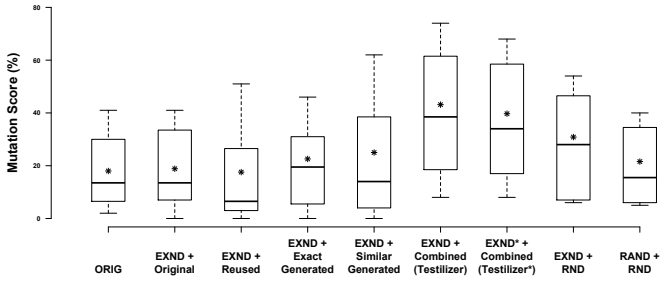


Figure 3.8: Box plots of mutation score using different test suite generation methods. Mean values are shown with (*).

for the different methods. It is evident that exact and similar generated assertions are more effective than original and reused ones. The effectiveness of each assertion generation technique solely is not more than the random approach. However, the results show that their combination (TESTILIZER) outperforms fault detection capability of the original test suite by 105% (21 percentage point increase) and the random methods by 22% (8 percentage point increase). This supports our insight that leveraging input values and assertions from human-written test suites can be helpful in generating more effective test cases.

***Finding 3**: On Average, tests generated by* TESTILIZER *outperform the fault detection capability of the original test suite by 105% (21 percentage point increase) and the random methods by 22% (8 percentage point increase).*

Test suites generated using TESTILIZER* are almost as effective as those generated by TESTILIZER. Applying EXND and EXND* can result in extended SFGs with different states and transitions. Based on our observations of failed test cases, i.e. detected mutants, we can give two reasons for this similar effectiveness: (1) the injected mutants can be reflected in multiple DOM states and thus assertions may fail on states that can be different in the two extended SFGs. (2) most of the detected mutants were reflected on common initial SFG states.

Impact of the original tests effectiveness (RQ3). The Spearman correlation analysis reveals that there exists a strong positive correlation (r= 0.91, p=0) between the effectiveness of the original tests and the effectiveness of the generated tests. This is expected as more manual-test paths/states and assertions helps TESTILIZER

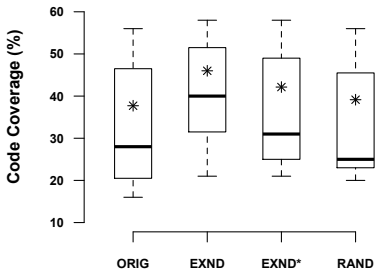


Figure 3.9: Box plots of JavaScript code coverage achieved using different test suites. Mean values are shown with (*).

to generate more tests with diverse assertions.

Finding 4*: Our results suggest that, there is a strong correlation between the effectiveness of the original tests and the effectiveness of the generated tests. Thus the quality of tests generated by* TESTILIZER *is directly influenced by the original test suite.*

Code coverage (RQ4). Although code coverage improvement is not the main goal of TESTILIZER in this work, the generated test suite has a slightly higher code coverage. As shown in Figure 3.9, on average there is a 22% improvement (8 percentage point increase) over the original test suite and 19% improvement (7 percentage point increase) over the RAND test suite. Note that the original test suites were already equipped with proper input data, but not many execution paths (thus the slight increase). On the other hand, the random exploration considered more paths in a blind search, but without proper input data (except for login data). Code coverage is slightly higher for EXND compared to EXND*. Our observations reveal that giving higher priority for expanding manual-test states, i.e., EXND, resulted in executing lines of code and consequently covering new functionalities that could be missed by applying EXND*. The difference is, however, not significant and can not imply that EXND can always achieve higher code coverage compared to EXND*.

Finding 5*: Test suites generated by* TESTILIZER *have slightly higher code coverage compared to the original ones and the ones generated by random exploration.*

3.6 Discussion

3.6.1 Applications

TESTILIZER can be used in automated testing of web applications by generating DOM-based tests given an existing test suite. The presented technique is applicable only in the context of regression testing, which assumes the current version of the given application is correct. This assumption is the basis for assertion generation and the filtration process for unstable assertions.

Although we proposed this work in the context of testing rich web interfaces, a similar approach can be adapted to GUI applications in general such as desktop and mobile apps. For instance, Memon [116] proposed dynamic analysis techniques to reverse engineer an event-flow graph (EFG) of desktop GUI applications. Similar to the SFG used in our work, EFG can be used for test case generation.

The results of this work support the usefulness of leveraging existing UI tests to generate new ones. However, TESTILIZER is limited to applications that already have human-written tests, which may not be so prevalent in practice. An interesting research question is whether human-written tests of similar applications can be leveraged to generate effective tests for applications without existing tests.

3.6.2 Generating Negative Assertions

In our experiments we observed that the majority of DOM-based assertions verify either the existence of an element, its properties, or its value on the DOM tree. In this work for fewer instances of negative assertions, which assert the absence of an element, we only considered *reusing* such assertions on a same state. We believe that its is possible to generate negative assertions using the existing manually-written tests for similar DOM states. One possible approach to calculate the similarity – between 0 and 1 – of two abstract DOM state in terms of similarity of their corresponding string representation. Given a *source state* containing a negative assertion, and a *target state*, if the similarity between *source* and *target* is above 0.5, we can add the negative assertion from *source* to the set of assertions for the *target* state. Generating such negative assertions might in some cases produce unstable assertions that fail on the unmodified version of the application. Such unstable

assertions can be removed through the filtering process described in Section 3.3.4.

3.6.3 Test Case Dependencies

An assumption made in TESTILIZER is that the original test suite does not have any test case dependencies. Generally, test cases should be executable without any special order or dependency on previous tests. However, it has been shown that dependent tests exist in practice, and are sometimes difficult to identify [198]. While conducting our evaluation, we also came across multiple test suites that violated this principle. For such cases, although TESTILIZER can generate test cases, failures can occur due to these dependencies.

3.6.4 Effectiveness

The effectiveness of the generated test suite depends on multiple factors. First, the size and the quality of the original test suite is very important; if the original test suite does not contain paths with effective assertions, it is not possible to generate an effective extended test suite. Second, the learning-based approach can be tuned in various ways (e.g., selecting other features, changing the SVM parameters, and choosing sample dataset size) to obtain better results. Also other learning techniques can be used and we used SVM due to the ease of use. Third, the size of the DOM subtree (region) to be checked can be increased to detect changes more effectively, however, it might come at the cost of making the test suite more brittle. Forth, the crawling time directly affects the size of the extended SFG and thus size of generated test suites, which may result in producing more effective tests.

3.6.5 Efficiency

The larger a test suite, the more time it takes to test an application. Since in many testing environments time is limited, not all possible paths of events should be generated in the extended test suite. The challenge is finding a balance between effectiveness and efficiency of the test cases. The current graph traversal method in TESTILIZER may produce test cases that share common paths, which do not contribute much to fault detection or code coverage. An optimization could be realized by guiding the test generation algorithm towards sates that have more constrained

DOM-based assertions.

3.6.6 Threats to Validity

A threat to the external validity of our experiment is with regard to the generalization of the results to other web applications. We acknowledge that more web applications should be evaluated to support the conclusions. To mitigate this threat, however, we selected our experimental objects from different domains with variations in functionality and structure, which we believe they are representative of real-world web applications. Although SELENIUM is widely used in industry for testing commercial web applications, unfortunately, very few open source web applications are publicly available that have (working) SELENIUM test suites. Moreover, since our effectiveness evaluation is based on the client-side mutation analysis, as shown in Table 3.5, the chosen applications should have a considerable amount of JavaScript code. However, many available web applications with SELENIUM tests are mainly server-side containing embedded JavaScript code. Therefore, we were able to include a limited number of applications with not large test suites in our study.

One threat to the internal validity is related to the mutation score analysis for the generated tests, for which the mutants were manually produced for the studied subjects. To mitigate the bias towards producing good results for TESTILIZER, we carefully searched within all JavaScript files of each subject system for target mutation locations based on the defined mutation operators on DOM, jQuery, and XHR, and produced a mutant based on each observed instance similarly for all cases.

With respect to reproducibility of our results, TESTILIZER, the test suites, and the experimental objects are publicly available, making the experiment reproducible.

3.7 Related Work

Elbaum et al. [81] leverage user-sessions for web application test generation. Based on this work, Sprenkle et al. [165] propose a tool to generate additional test cases based on the captured user-session data. McAllister et al. [115] leverage

user interactions for web testing. Their method relies on prerecorded traces of user interactions and requires instrumenting one specific web application framework. These techniques do not consider leveraging knowledge from existing test cases.

Yuan and Memon [192] propose an approach to iteratively rerun automatically generated test cases for generating alternating test cases. This is inline with feedback-directed testing [146], which leverages dynamic data produced by executing the program using previously generated test cases. For instance, Artemis [66] is a feedback-directed tool for automated testing of JavaScript applications that uses generic oracles such as HTML validation. Our previous work, FeedEx [123], applies a feedback-directed exploration technique to guide the exploration at runtime towards more coverage and higher navigational and structural diversity. These approaches also do not use information in existing test cases, and they do not address the problem of test oracle generation.

A generic approach used often as a test oracle is checking for thrown exceptions and application crashes [191]. This is, however, not very helpful for web applications as they do not crash easily and the browser continues the execution even after exceptions. Current web testing techniques simplify the test oracle problem in the generated test cases by using soft oracles, such as generic user-defined oracles, and HTML validation [66, 121]. Mirshokraie and Mesbah [128], and Pattabiraman and Zorn [148] dynamically derive invariants for the JavaScript code and the DOM respectively. Such inferred invariants are used for automated regression and robustness testing. Mirshokraie et al. [130] perform mutation analysis to generate test cases with oracles using the dynamic event-driven model of JavaScript. They generate oracle for their test cases using mutation testing. However, they do not reuse the existing test inputs or oracles.

Xie and Notkin [187] infer a model of the application under test by executing the existing test cases. Dallmeier et al. [78] mine a specification of desktop systems by executing the test cases. Schur et al. [159] infer behaviour models from enterprise web applications via crawling. Their tool generates test cases simulating possible user inputs. Similarly, Xu et al. [188] mine executable specifications of web applications from SELENIUM test cases to create an abstraction of the system. Yoo and Harman [190] propose a search-based approach to reuse and regenerate existing test data for primitive data types. They show that the knowledge of existing

test data can help to improve the quality of new generated test data. Alshahwan and Harman [63] generate new sequences of HTTP requests through a def-use analysis of server-side code. Pezze et al. [150] present a technique to generate integration test cases from existing unit test cases. Mirzaaghaei et al. [132] use test adaptation patterns in existing test cases to support test suite evolution.

Leveraging unit tests has been studied in the context of mining API usage [201], and test recommendation [67, 82, 90, 99, 107] where testers can get inspired by recommended tests written for other similar applications. For instance Janjic and Atkinson [99] introduced recommendation techniques for JUnit tests using previously written test cases and the knowledge extracted from them. Landhäußer and Tichy [107] explored the possibilities of test reuse for recommendation based on clones. Such techniques can be utilized in JavaScript unit test generation tools, such as JSeft [130] or ConFix [127], to generate test for a similar code.

Adamsen et al. [60] propose a technique for testing Android mobile apps by systematically exposing existing tests to adverse conditions. Zhang et al. [197] reuse existing test cases for security testing by analyzing the security vulnerabilities in the execution traces. Tonella et al. [172] reuse existing DOM-based tests to generate visual web tests. This work is also related to test suite augmentation techniques [155, 189] used in regression testing. In test suite augmentation the goal is to generate new test cases for the changed parts of the application. More related to our work is [181], which aggregates tests generated by different approaches using a unified test case language. They propose a test advice framework that extracts information in the existing tests to help improve other tests or test generation techniques.

Our work is different from these approaches in that we (1) reuse knowledge in existing human-written test cases in the context of web application testing, (2) reuse input values and event sequences in test cases to explore alternative paths and news states of web application, and (3) reuse oracles of the test cases for regenerating assertions to improve the fault finding capability of the test suite.

3.8 Conclusions

This work is motivated by the fact that a human-written test suite is a valuable source of domain knowledge, which can be used to tackle some of the challenges in automated web application test generation. Given a web application and its DOM-based (such as SELENIUM) test suite, our tool, called TESTILIZER, utilizes the given test suite to generate effective test cases by exploring alternative paths of the application, and regenerating assertions for new detected states. Our empirical results on 8 real-world applications show that TESTILIZER easily outperforms a random test generation technique, provides substantial improvements in the fault detection rate compared with the original test suite, while slightly increasing code coverage too.

The results support the usefulness of leveraging existing DOM-based tests to generate new ones. However, TESTILIZER is limited to applications that already have human-written tests, which may not be so prevalent in practice. On the other hand, many web applications are similar to each other in terms of design and code base, such as being built on top of the same content management system. We propose an open research problem whether human-written tests can be leveraged to generate effective tests for applications without existing tests. This is, however, challenging particularly for assertion generation based on learned patterns. DOM-based assertions on abstract DOM states of an application may require some changes to be applied on similar abstract DOM state of another application.

Chapter 4

JavaScript: The (Un)covered Parts

Summary[6]

Testing JavaScript code is important. JavaScript has grown to be among the most popular programming languages and it is extensively used to create web applications both on the client and server. We present the first empirical study of JavaScript tests to characterize their prevalence, quality metrics (e.g. code coverage), and shortcomings. We perform our study across a representative corpus of 373 JavaScript projects, with over 5.4 million lines of JavaScript code. Our results show that 22% of the studied subjects do not have test code. About 40% of projects with JavaScript at client-side do not have a test, while this is only about 3% for the purely server-side JavaScript projects. Also tests for server-side code have high quality (in terms of code coverage, test code ratio, test commit ratio, and average number of assertions per test), while tests for client-side code have moderate to low quality. In general, tests written in *Mocha*, *Tape*, *Tap*, and *Nodeunit* frameworks have high quality and those written without using any framework have low quality. We scrutinize the (un)covered parts of the code under test to find out root causes for the uncovered code. Our results show that JavaScript tests lack proper coverage for event-dependent callbacks (36%), asynchronous callbacks (53%), and DOM-related code (63%). We believe that it is worthwhile for the developer and research community to focus on testing techniques and tools to achieve better coverage for difficult to cover JavaScript code.

[6] An initial version of this chapter has been accepted to be published in the IEEE International Conference on Software Testing, Verification and Validation (ICST), 2017 [125].

4.1 Introduction

JavaScript is currently the most widely used programming language according to a recent survey of more than 56K developers conducted by Stack Overflow [167], and also exploration of the programming languages used across GitHub repositories [91]. JavaScript is extensively used to build responsive modern web applications, and is also used to create desktop and mobile applications, as well as server-side network programs. Consequently, testing JavaScript applications and modules is important. However, JavaScript is quite challenging to test and analyze due to some of its specific features. For instance, the complex and dynamic interactions between JavaScript and the DOM, makes it hard for developers to test effectively [66, 127, 130].

To assist developers with writing tests, there exist number of JavaScript unit testing frameworks, such as Mocha [36], Jasmine[31], QUnit [40], and Nodeunit [37], each having its own advantages [52]. Also the research community have proposed some automated testing tools and test generation techniques for JavaScript programs [66, 95, 127, 130, 131], though they are not considerably used by testers and developers yet.

Some JavaScript features, such as DOM interactions, event-dependent callbacks, asynchronous callbacks, and closures (hidden scopes), are considered to be harder to test [28, 46, 48, 53, 127, 173]. However, there is no evidence that to what extent this is true in real-world practice.

In this work, we study JavaScript (unit) tests in the wild from different angles. The results of this study reveal some of the shortcomings and difficulties of manual testing, which provide insights on how to improve existing JavaScript test generation tools and techniques. We perform our study across a representative corpus of 373 popular JavaScript projects, with over 5.4 million lines of JavaScript code. To the best of our knowledge, this work is the first study on JavaScript tests. The main contributions of our work include:

- A large-scale study to investigate the prevalence of JavaScript tests in the wild;
- A tool, called TESTSCANNER, which statically extracts different metrics in

our study and is publicly available [59];

- An evaluation of the quality of JavaScript tests in terms of code coverage, average number of assertions per test, test code ratio, and test commit ratio;
- An analysis of the uncovered parts of the code under test to understand which parts are difficult to cover and why.

4.2 Methodology

The goal of this work is to study and characterize JavaScript tests in practice. We conduct quantitative and qualitative analyses to address the following research questions:

RQ1: How prevalent are JavaScript tests?

RQ2: What is the quality of JavaScript tests?

RQ3: Which part of the code is mainly uncovered by tests and why?

4.2.1 Subject Systems

We study 373 popular open source JavaScript projects. 138 of these subject systems are the ones used in a study for JavaScript callbacks [87] including 86 of the most depended-on modules in the *NPM repository* [55] and 52 JavaScript repositories from *GitHub Showcases*[7] [56]. Moreover, we added 234 JavaScript repositories from Github with over 4000 stars. The complete list of these subjects and our analysis results, are available for download [59]. We believe that this corpus of 373 projects is representative of real-world JavaScript projects as they differ in domain (category), size (lines of code), maturity (number of commits and contributors), and popularity (number of stars and watchers).

We categorize our subjects into 19 categories using topics from *JSter* JavaScript Libraries Catalog [54] and *GitHub Showcases* [56] for the same or similar projects. Table 4.1 presents these categories with average values for the number

[7] GitHub Showcases include popular and trending open source repositories organized around different topics.

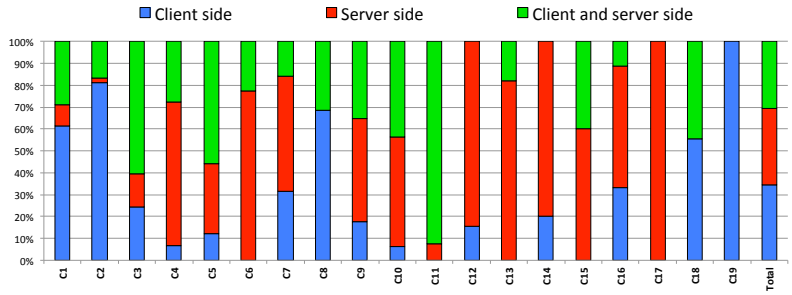


Figure 4.1: Distribution of studied subject systems.

of JavaScript files (production code), source lines of code (SLOC) for production and test code, number of test cases, and number of stars in Github repository for each category. We used SLOC [58] to count lines of source code excluding libraries. Overall, we study over 5.4 million (3.7 M production and 1.7 M test) source lines of JavaScript code.

Figure 4.1 depicts the distribution of our subject systems with respect to the client or server side code. Those systems that contain server-side components are written in Node.js[8], a popular server-side JavaScript framework. We apply the same categorization approach as explained in [87]. Some projects such as MVC frameworks, e.g. Angular, are purely client-side, while most NPM modules are purely server-side. We assume that client-side code is stored in directories such as *www*, *public*, *static*, or *client*. We also use code annotations such as `/* jshint browser:true, jquery:true */` to identify client-side code.

The 373 studied projects include 128 client-side, 130 server-side, and 115 client&server-side code. While distributions in total have almost the same size, they differ per project category. For instance subject systems in categories C1 (UI components), C2 (visualization), C8 (touch and drag&drop), C19 (MVC frameworks), and C18 (multimedia) are mainly client-side and those in categories C4 (software dev tools), C6 (parsers and compilers), C12 (I/O), C13 (package and build managers), C14 (storage), C16 (browser utils), and C17 (CLI and shell) are mainly server-side.

[8] https://nodejs.org

Table 4.1: Our JavaScript subject systems (60K files, 3.7 M production SLOC, 1.7 M test SLOC, and 100K test cases).

ID	Category	# Subject systems	Ave # JS files	Ave Prod SLOC	Ave Test SLOC	Ave # tests	Ave # assertions	Ave # stars
C1	UI Components, Widgets, and Frameworks	52	41	4.7K	2.8K	235	641	9.8K
C2	Visualization, Graphics, and Animation Libraries	48	53	10.2K	3.8K	425	926	7.5K
C3	Web Applications and Games	33	61	10.6K	1.4K	61	119	4K
C4	Software Development Tools	29	67	12.7K	7.8K	227	578	6.9K
C5	Web and Mobile App Design and Frameworks	25	91	22.3K	6.9K	277	850	14.4K
C6	Parsers, Code Editors, and Compilers	22	167	27K	9.5K	701	1142	5.5K
C7	Editors, String Processors, and Templating Engines	19	26	4.3K	1.9K	102	221	6.5K
C8	Touch, Drag&Drop, Sliders, and Galleries	19	10	1.9K	408	52	72	7.9K
C9	Other Tools and Libraries	17	93	9.1K	7.6K	180	453	8.5K
C10	Network, Communication, and Async Utilities	16	19	4.1K	7.6K	279	354	7.6K
C11	Game Engines and Frameworks	13	86	17K	1.2K	115	293	3.5K
C12	I/O, Stream, and Keyboard Utilities	13	8	0.6K	1K	40	61	1.5K
C13	Package Managers, Build Utilities, and Loaders	11	47	3.4K	5.4K	200	300	8.5K
C14	Storage Tools and Libraries	10	19	4K	7K	222	317	5.5K
C15	Testing Frameworks and Libraries	10	28	2.8K	3.6K	271	632	5.7K
C16	Browser and DOM Utilities	9	45	5.6K	7.1K	76	179	5.2K
C17	Command-line Interface and Shell Tools	9	9	2.8K	1K	26	244	2.6K
C18	Multimedia Utilities	9	11	1.6K	760	17	97	6.2K
C19	MVC Frameworks	9	174	40.1K	15.2K	657	1401	14.2K
	Client-side	128	39	8.2K	3.2K	343	798	7.9K
	Server-side	130	63	9.4K	7.2K	231	505	6.7K
	Client and server-side	115	73	12.7K	4.7K	221	402	7.4K
	Total	373	57	10.1K	4.5K	263	644	7.3K

4.2.2 Analysis

To address our research questions, we statically and dynamically analyze test suites of our subject programs. To extract some of the metrics in our study, we develop a static analyzer tool, called TESTSCANNER [59], which parses production and test code into an abstract syntax tree using Mozilla Rhino [41]. In the rest of this section we explain details of our analysis for each research question.

Prevalence of tests (RQ1)

To answer RQ1, we look for presence of JavaScript tests written in any framework (e.g. Mocha, Jasmine, or QUnit). Tests are usually located at folders namely *tests*, *specs*[9], or similar names.

We further investigate the prevalence of JavaScript tests with respect to subject categories, client/server-side code, popularity (number of stars and watchers), maturity (number of commits and contributors), project size (production SLOC), and testing frameworks. To distinguish testing frameworks, we analyze package management files (such as `package.json`), task runner and build files (such as `grunt.js` and `gulpfile.js`), and test files themselves.

Quality of tests (RQ2)

To address RQ2, for each subject with test we compute four quality metrics as following:

Code coverage. Coverage is generally known as an indicator of test quality. We compute statement, branch, and function coverage for JavaScript code using JSCover [33] (for tests that run in the browser), and Istanbul [30]. To calculate coverage of the minified JavaScript code, we beautify them prior to executing tests. We also exclude dependencies, such as files under the *node_modules* directory, and libraries (unless the subject system is itself a library).

Average number of assertions per test. Code coverage does not directly imply a test suite effectiveness [97], while assertions have been shown to be strongly correlated with it [199]. Thus, TESTSCANNER also computes average number of

[9]For instance Jasmine and Mocha tests are written as specs and are usually located at folders with similar names.

assertions per test case as a test suite quality metric. Our analysis tools detects usage of well-known assertion libraries such as assert.js, should.js, expect.js, and chai.

Test code ratio. This metric is defined as the ratio of test SLOC to production and test SLOC. A program with a high test code ratio may have a higher quality test suite.

Test commit ratio. This metric is the ratio of test commits to total commits. Higher test commit ratio may indicate more mature and higher quality tests. We assume that every commit that touches at least one file in a folder named *test*, *tests*, *spec*, or *specs* is a test commit. In rare cases that tests are stored elsewhere, such as the root folder, we manually extract number of test commits by looking at its github repository page and counting commits on test files.

We investigate these quality metrics with respect to subject categories, client/server-side code, and testing frameworks.

(Un)covered code (RQ3)

Code coverage is a widely accepted test quality indicator, thus finding the root cause of why a particular statement is not covered by a test suite, can help in writing higher quality tests. Some generic possible cases for an *uncovered* (missed) statement s, are as following:

1. s belongs to an *uncovered* function f, where

   (a) f has no calling site in both the production and the test code. In this case, f could be (1) a callback function sent to a callback-accepting function (e.g., `setTimeout()`) that was never invoked, or (2) an unused utility function that was meant to be used in previous or future releases. Such unused code can be considered as code smells [124]. Consequently we cannot pinpoint such an uncovered function to a particular reason.

   (b) the calling site for f in the production code was never executed by a test. Possible root causes can be that (1) f is used as a callback (e.g. event-dependent or asynchronous) that was never invoked, (2) the call

to f statement was never reached because of an earlier `return` statement or an exception, or the function call falls in a never met condition branch.

(c) f is an *anonymous* function. Possible reasons that f was not covered can be that (1) f is used as a callback that was never invoked (e.g. an event-dependent callback while the required event was not triggered, or an asynchronous callback while did not wait for the response), (2) f is a self-invoking function that was not executed to be invoked, or (3) f is set to a variable and that variable was never used or its usage was not executed.

2. s belongs to a *covered* function f, where

   (a) the execution of f was terminated, by a `return` statement or an exception, prior to reaching s.

   (b) s falls in a never met condition in f (e.g. browser or DOM dependent statements).

3. The test case responsible for covering s was not executed due to a test execution failure.

4. s is a dead (unreachable) code.

Uncovered statement in uncovered function ratio. If an uncovered statement s belongs to an uncovered function f, making f called could possibly cover s as well. This is important specially if f needs to be called in a particular way, such as through triggering an event.

In this regard, our tool uses coverage report information (in json or lcov format) to calculate the ratio of the uncovered statements that fall within uncovered functions over the total number of uncovered statements. If this value is large it indicates that the majority of uncovered statements belong to uncovered functions, and thus code coverage could be increased to a high extent if the enclosing function is called by a test case.

```
1   function setFontSize(size) {
2       return function() {
3           // this is an anonymous closure
4           document.body.style.fontSize = size + 'px';
5       };
6     }
7     var small = setFontSize(12);
8     var large = setFontSize(16);
9     ...
10    function showMsg() {
11       // this is an async callback
12       alert("Some message goes here!");
13    }
14    ...
15    $("#smallBtn").on("click", small);
16    $("#largeBtn").on("click", large);
17    $("#showBtn").on("click", function() {
18       // this is an event-dependent anonymous callback
19       setTimeout(showMsg, 2000);
20       $("#photo").fadeIn("slow", function() {
21         // this is an anonymous callback
22         alert("Photo animation complete!");
23       });
24    });
25    ...
26    checkList = $("#checkList");
27    checkList.children("input").each(function () {
28       // this is an DOM-related code
29       if (this.is(':checked')) {
30       ...
31       }else{
32       ...
33       }
34    });
```

Figure 4.2: A hard to test JavaScript code snippet.

Hard-to-test JavaScript code. Some JavaScript features, such as DOM interactions, event-dependent callbacks, asynchronous callbacks, and closures (hidden scopes), are considered to be harder to test [28, 46, 48, 53, 127, 173]. In this section we explain four main hard-to-test code with an example code snippet depicted in Figure 4.2. Also we fine-grain statement and function coverage metrics to investigate these hard-to-test code separately in detail. To measure these coverage metrics, TESTSCANNER maps a given coverage report to the locations of hard-to-test code.

DOM related code coverage. In order to unit test a JavaScript code with DOM read/write operations, a DOM instance has to be provided as a test fixture in the ex-

act structure expected by the code under test. Otherwise, the test case can terminate prematurely due to a null exception. Writing such DOM based fixtures can be challenging due to the dynamic nature of JavaScript and the hierarchical structure of the DOM [127]. For example, to cover the `if` branch at line 29 in Figure 4.2, one needs to provide a DOM instance such as `<div id="checkList"><input type="checkbox" checked></input></div>`. To cover the `else` branch, a DOM instance such as `<div id="checkList"><input type="checkbox"></input></div>` is required. If such fixtures are not provided, `$("checkList")` returns `null` as the expected element is not available, and thus `checkList.children` causes a null exception and the test case terminates.

DOM related code coverage is defined as the fraction of number of covered over total number of DOM related statements. A DOM related statement is a statement that can affect or be affected by DOM interactions such as a DOM API usage. To detect DOM related statements TESTSCANNER extracts all DOM API usages in the code (e.g. `getElementById`, `createElement`, `appendChild`, `addEventListener`, `$`, and `innerHTML`) and their forward slices. Forward slicing is applied on the variables that were assigned with a DOM element/attribute. For example the forward slice of `checkList` at line 26 in Figure 4.2 are lines 27–34. A DOM API could be located in a (1) `return` statement of a function f, (2) conditional statement, (3) function call (as an argument), (4) an assignment statement, or (5) other parts within a scope. In case (1), all statements that has a call to f are considered DOM related. In case (2), the whole conditional statements (condition and the body of condition) are considered DOM related. In case (3) the statements in the called function, which use that DOM input will be considered DOM related. In other cases, the statement with DOM API is DOM related.

Event-dependent callback coverage. The execution of some JavaScript code may require triggering an event such as clicking on a particular DOM element. For instance it is very common in JavaScript client-side code to have an (anonymous) function bound to an element's event, e.g. a click, which has to be simulated. The anonymous function in lines 17–24 is an *event-dependent callback* function. Such callback functions would only be passed and invoked if the corresponding event is triggered. In order to trigger an event, testers can use methods such as jQuery's

`.trigger(event, data, ...)` or `.emit(event, data, ...)` of Node.js EventEmitter. Note that if an event needs to be triggered on a DOM element, a proper fixture is required otherwise the callback function cannot be executed.

Event-dependent callback coverage is defined as the fraction of number of covered over total number of event-dependent callback functions. In order to detect event-dependent callbacks, our tool checks if a callback function is an event method such as `bind`, `click`, `focus`, `hover`, `keypress`, `emit`, `addEventListener`, `onclick`, `onmouseover`, and `onload`.

Asynchronous callback coverage. Callbacks are functions passed as an argument to another function to be invoked either immediately (synchronous) or at some point in the future (asynchronous) after the enclosing function returns. Callbacks are particularly useful to perform non-blocking operations. Function `showMsg` in lines 10–13 is an *asynchronous callback* function as it was passed to the `setTimeout()` asynchronous API call. Testing *asynchronous callbacks* requires waiting until the callback is called, otherwise the test would probably finish unsuccessfully before the callback is invoked. For instance QUnit's `asyncTest` allows tests to wait for asynchronous callbacks to be called.

Asynchronous callback coverage is defined as the fraction of number of covered over total number of asynchronous callback functions. Similar to a study of callbacks in JavaScript [87], if a callback argument is passed into a known deferring API call we count it as as an asynchronous callback. TESTSCANNER detects some asynchronous APIs including network calls (e.g. `XMLHTTPRequest.open`), DOM events (e.g., `onclick`), timers (`setImmediate`, `setTimeout`, `setInterval`, and `process.nextTick`), and I/O (e.g. APIs of `fs`, `http`, and net).

Closure function coverage. Closures are nested functions that make it possible to create hidden scope to privatize variables and functions from the global scope in JavaScript. A closure function, i.e., the inner function, has access to all parameters and variables – except for `this` and `argument` variables – of the outer function, even after the outer function has returned [77]. The anonymous function in lines 2–5 is an instance of a *closure*.

Such hidden functions cannot be called directly in a test case and thus testing them is challenging. In fact writing a unit test for a closure function without code modification is impossible. Simple solutions such as making them public or putting the test code inside the closure are not good software engineering practices. One approach to test such private functions is adding code inside the closure to store references to its local variables inside objects and return it to the outer scope [48]. *Closure function coverage* is defined as the fraction of number of covered over total number of closure functions.

Average number of function calls per test. Some code functionalities depend on the execution of a sequence of function calls. For instance in a shopping application, one needs to add items to the cart prior to check out. We perform a correlation analysis between average number of unique function calls per test and code coverage. We also investigate whether JavaScript unit tests are mostly written at single function level or they execute sequence of function calls.

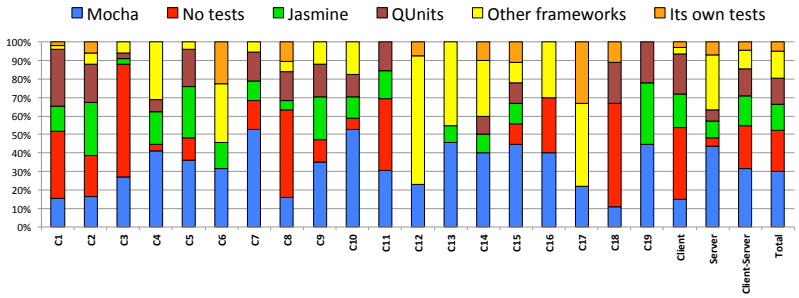
4.3 Results

4.3.1 Prevalence of Tests

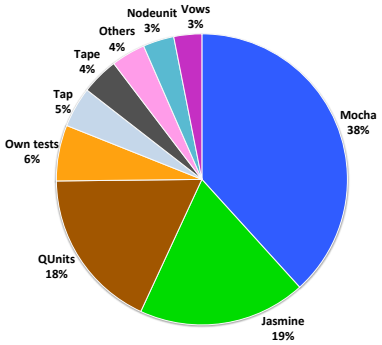
The stacked bar charts in Figure 4.3a depicts the percentage of JavaScript tests, per system category (Table 4.1), per client/server side, and in aggregate. The height of each bar indicates the percentage of subjects in that category. In total, among the 373 studied subjects, 83 of them (i.e., 22%) do not have JavaScript tests. The majority (78%) of subjects have at least one test case.

Finding 1*: 22% of the subject systems that we studied do not have any JavaScript test, and 78% have at least one test case.*

As shown in figure 4.3b, amongst subjects with test, the majority of tests are written in *Mocha* (38%), *Jasmine* (19%), and *QUnit* (18%). 6% does not follow any particular framework and have their own tests. Minor used frameworks are *Tap* (5%), *Tape* (4%), *Nodeunit* (3%), *Vows* (3%) and others (4%) including *Jest*, *Evidence.js*, *Doh*, *CasperJS*, *Ava*, *UTest*, *TAD*, and *Lab*. We also observe that 3 repositories have tests written in two testing frameworks: 2 projects (server and client-server) with *Nodeunit+Mocha* test, and one (client-server) with *Jasmine+QUnit*



(a) Distribution within all subjects.



(b) Testing frameworks distribution.

Figure 4.3: Distribution of JavaScript tests.

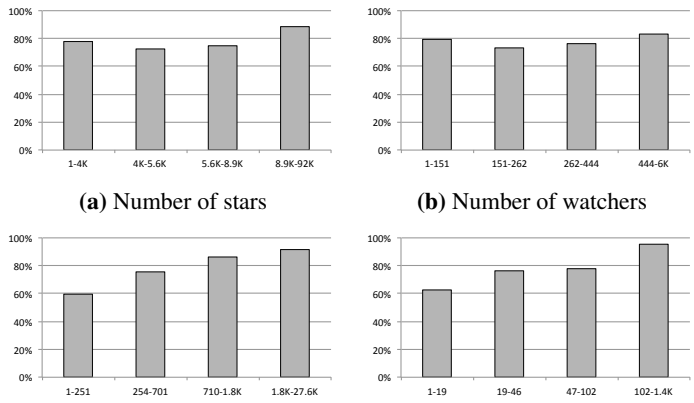
test.

Finding 2*: The most prevalent used test frameworks for JavaScript unit testing are Mocha (38%), Jasmine (19%), and QUnit (18%).*

We also investigate the prevalence of UI tests and observe that only 12 projects (i.e., 3%) among all 373 ones have UI tests for which 9 are written using *Webdriverio* and *Selenium webdriver*, and 3 uses *CasperJS*. 7 of these projects are client and server side, 3 are client-side, and 2 are server-side. One of these subjects does not have any JavaScript test.

Finding 3*: Only 3% of the studied repositories have functional UI tests.*

Almost all (95%) of purely server-side JavaScript projects have tests, while this is 61% for client-side and 76% for client&server-side ones. Note that the number of subjects in each category are not very different (i.e., 128 client-side, 130 server-



(a) Number of stars **(b)** Number of watchers

(c) Number of commits **(d)** Number of contributors

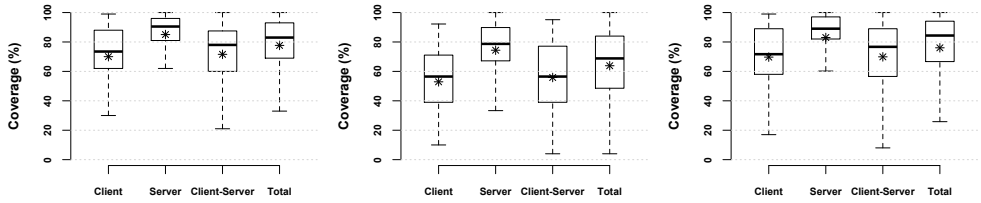
Figure 4.4: Percentage of subjects with test per each quartile with respect to popularity (number of stars and watchers) and maturity (number of commits and contributors).

side, and 115 client and server-side code). Interestingly the distribution of test frameworks looks very similar for client-side and client-server side projects.

As shown in Figure 4.3a, all subjects systems in categories C6 (parsers and compilers), C12 (I/O), C13 (package and build managers), C14 (storage), C19 (MVC frameworks), and C17(CLI and shell), have JavaScript unit tests. Projects in all of these categories, except for C19, are mainly server-side as depicted in Figure 4.1. In contrast, many of subjects in categories C1 (UI components), C3 (web apps), C8 (touch and drag&drop), and C18 (multimedia) do not have tests, which are mainly client-side. Thus we can deduce that JavaScript tests are written more for server-side code than client-side, or client and server-side code.

***Finding 4**: While almost all subjects (95%) in the server-side category have tests, about 40% of subjects in client-side and client-server side categories do not have tests.*

We believe the more prevalence of tests for server-side code can be attributed to (1) the difficulties in testing client-side code, such as writing proper DOM fixtures or triggering events on DOM elements, and (2) using time-saving test scripts for most *Node.js* based projects, such as `npm test` that is included by default when initializing a new `package.json` file. This pattern is advocated in the *Node.js* community [57] and thus many server-side JavaScript code, such as NPM modules,



(a) Statement coverage **(b)** Branch coverage **(c)** Function coverage

Figure 4.5: Boxplots of the code coverage of the executed JavaScript tests. Mean values are shown with (*).

have test code.

We also consider how popularity (number of stars and watchers) and maturity (number of commits and contributors) of subject systems are related to the prevalence of unit tests. Figure 4.4a shows the percentage of subjects with tests in each quartile. As popularity and maturity increase, the percentage of subjects with test increases as well.

4.3.2 Quality of Tests

Code coverage. Calculating the code coverage requires executing tests on a properly deployed project. In our study, however, we faced number of projects with failure in build/deployment or running tests. We tried to resolve such problems by quick changes in build/task configuration files or by retrieving a later version (i.e., some days after fetching the previous release). In most cases build failure was due to errors in dependent packages or their absence. We could finally calculate coverage for 231 out of 290 (about 80%) subjects with tests. We could not properly deploy or run tests for 44 subject systems (41 with test run failure, freeze, or break, and 3 build and deployment error), and could not get coverage report for 15 projects with complex test configurations.

Boxplots in Figure 4.5 show that in total tests have a median of 83% statement coverage, 84% function coverage, and 69% branch coverage. Tests for server-side code have higher coverage in all aspects compared to those for client-side code. We narrow down our coverage analysis into different subject categories. As depicted in Table 4.2, subjects in categories C6 (parsers and compilers), C10 (Network and

Table 4.2: Test quality metrics average values.

		Statement coverage	Branch coverage	Function coverage	Ave # assertions per test	Test code ratio	Test commit ratio
Subject category	C1	77%	57%	76%	2.83	0.41	0.16
	C2	67%	52%	65%	2.72	0.28	0.14
	C3	60%	38%	58%	3.75	0.88	0.14
	C4	79%	68%	78%	2.50	0.58	0.24
	C5	75%	63%	75%	2.53	0.52	0.21
	C6	87%	79%	88%	2.53	0.47	0.24
	C7	80%	67%	72%	2.51	0.46	0.22
	C8	64%	47%	60%	2.04	0.35	0.12
	C9	73%	58%	69%	2.67	0.49	0.23
	C10	91%	79%	90%	2.73	0.72	0.24
	C11	64%	45%	57%	3.41	0.18	0.11
	C12	90%	77%	89%	2.36	0.59	0.20
	C13	86%	67%	84%	2.27	0.59	0.18
	C14	88%	77%	87%	2.74	0.62	0.26
	C15	81%	69%	79%	5.79	0.59	0.25
	C16	78%	67%	79%	1.67	0.49	0.29
	C17	67%	54%	63%	8.32	0.47	0.21
	C18	60%	31%	62%	4.42	0.31	0.16
	C19	81%	67%	80%	3.58	0.53	0.21
Testing framework	Mocha	82%	70%	79%	2.39	0.49	0.20
	Jasmine	74%	60%	75%	1.93	0.41	0.21
	QUnit	71%	54%	71%	3.93	0.41	0.16
	Own test	61%	41%	58%	5.99	0.30	0.16
	Tap	89%	80%	89%	1.56	0.58	0.21
	Tape	93%	81%	94%	2.93	0.70	0.18
	Others	80%	65%	77%	5.60	0.46	0.24
	Nodeunit	74%	63%	72%	6.20	0.57	0.24
	Vows	74%	66%	72%	1.92	0.55	0.27
	Client	70%	53%	70%	2.71	0.36	0.16
	Server	85%	74%	83%	3.16	0.58	0.23
	C&S	72%	56%	70%	2.9	0.4	0.18
	Total	78%	64%	76%	2.96	0.46	0.2

Async), C12 (I/O), C13 (package and build managers), C14 (storage), C15 (testing frameworks), and C19 (MVC frameworks) on average have higher code coverage. Projects in these categories are mainly server-side. In contrast, subjects in categories C2 (visualization), C3 (web apps), C8 (touch and drag&drop), C11 (game engines), C17 (CLI and shell), and C18 (multimedia), have lower code coverage. Note that subjects under these categories are mainly client-side.

Finding 5*: The studied JavaScript tests have a median of 83% statement coverage, 84% function coverage, and 69% branch coverage. Tests for server-side code have higher coverage in all aspects compared to those for client-side code.*

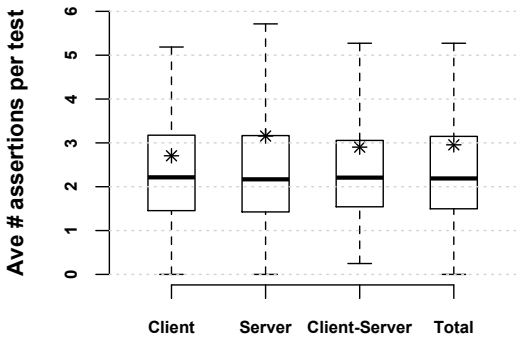


Figure 4.6: Average number of assertions per test.

Table 4.2 also depicts the achieved coverage per testing framework. Tests written in *Tape*, *Tap*, and *Mocha* have generally higher code coverage. The majority of server-side JavaScript projects are tested using these frameworks. On the other hand, tests written in *QUnit*, which is used more often for the client-side than the server-side, has generally lower code coverage. Developers that used their own style of testing without using popular frameworks write tests with the poorest coverage.

Finding 6*: Tests written in Tape, Tap, and Mocha frameworks, generally have higher coverage compared to those written in QUnit, Nodeunit, and those without using any test framework.*

Average number of assertions per test. Figure 4.6 depicts boxplots of average number of assertions per test case. While median values are very similar (about 2.2) for all cases, server-side code has a slightly higher mean value (3.16) compared to client-side (2.71). As shown in Table 4.2, subjects in categories C3 (web apps), C11 (game engines), C15 (testing frameworks), C17 (CLI and shell), C18 (multimedia), and C19 (MVC frameworks) on average have higher average number of assertions per test compared to others. Interestingly among these categories only for C15 and C19 code coverage is also high while it is low for the rest.

Finding 7*: The studied test suites have a median of 2.19 and a mean of 2.96 for the average number of assertions per test. These values do not differ much among server-side and client-side code.*

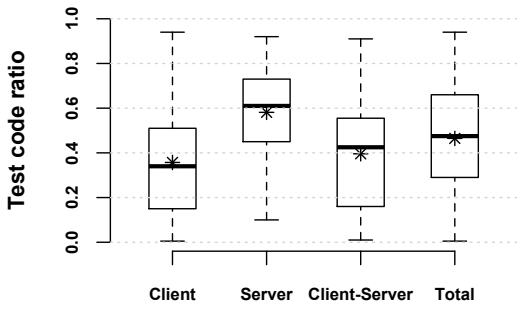


Figure 4.7: Test to total code ratio.

Also results shown in Table 4.2 indicate that tests written in *QUnit*, *Tape*, *Nodeunit*, other frameworks (e.g. *Jest*, *CasperJS*, and *UTest*), and those without using a framework, have on average more assertions per test. The majority of server-side JavaScript projects are tested using these frameworks. Again we observe that only for tests written in Tape framework code coverage is also high while it is low for the rest.

Test code ratio. Figure 4.7 shows test to total (production and test) code ratio comparison. The median and mean of this ratio is about 0.6 for server-side projects and about 0.35 for client-side ones. As shown in Table 4.2, on average subjects with higher test code ratio belongs to categories C3, C4, C5, C10, C12, C13, C14, C15, and C19 while those in C2, C8, C11, and C18 have lower test code ratio. Also tests written in *Tap*, *Tape*, *Nodeunit*, and *Vows* have higher test code ratio while tests written without using any framework have lower test code ratio.

We further study the relationship between test code ratio and total code coverage (average of statement, branch, and function coverage) through the Spearman's correlation analysis[10]. The result shows that there exists a moderate to strong correlation ($\rho = 0.68$, $p = 0$) between test code ratio and code coverage.

***Finding 8**: Tests for server-side code have higher test code ratio (median and mean of about 0.6) compared to client-side code (median and mean of about 0.35). Also there exists a moderate to strong correlation ($\rho = 0.68$, $p = 0$) between test code ratio and code coverage.*

[10]The non-parametric Spearman's correlation coefficient measures the monotonic relationship between two continuous random variables and does not require the data to be normally distributed.

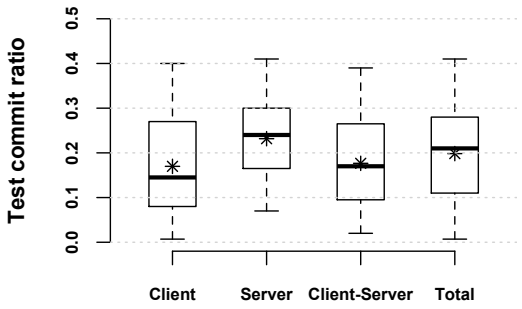


Figure 4.8: Test to total commits ratio.

Test commit ratio. Figure 4.8 depicts test to total (production and test) commit ratio comparison. The median and mean of this ratio is about 0.25 for server-side projects and about 0.15 for client-side ones. As shown in Table 4.2, on average subjects with higher test commit ratio belongs to categories C4, C6, C9, C10, C14, C15, and C16 while those in C1, C2, C3, C8, C11, and C18 have lower test commit ratio. Also tests written in *Nodeunit*, *Vows*, and other frameworks (e.g. *Jest*, *CasperJS*, and *UTest*) have higher test commit ratio while tests written in *QUnit* or without using any framework have lower test commit ratio.

Similar to the correlation analysis for test code ratio, we study the relationship between test commit ratio and total code coverage. The result indicates that there exists a moderate to low correlation ($\rho = 0.49$, $p = 0$) between test commit ratio and code coverage.

***Finding 9**: While test commit ratio is relatively high for server-side projects (median and mean of about 0.25), it is moderate in total and relatively low for client-side projects (median and mean of about 0.15). Also there exists a moderate to low correlation ($\rho = 0.49$, $p = 0$) between test commit ratio and code coverage.*

4.3.3 (Un)covered Code

As explained earlier in Section 4.2.2, one possible root cause for uncovered code is that the responsible test code was not executed. In our evaluation, however, we observed that for almost all the studied subjects, test code had very high coverage meaning that almost all statements in test code were executed properly. Thus the

Table 4.3: Statistics for analyzing uncovered code. The "–" sign indicates no instance of a particular code.

		Function coverage				Statement coverage			
		All	**Async callback**	**Event dependent callback**	**Closure**	**All**	**DOM related**	**Ave # func calls per test**	**USUF ratio**
Subject category	C1	76%	65%	33%	79%	77%	73%	2.91	0.59
	C2	65%	43%	17%	62%	67%	61%	2.82	0.73
	C3	58%	21%	10%	38%	60%	27%	3.94	0.82
	C4	79%	49%	48%	70%	81%	75%	2.89	0.53
	C5	75%	52%	33%	65%	75%	62%	3.05	0.72
	C6	88%	60%	32%	87%	87%	57%	3.30	0.33
	C7	72%	34%	28%	81%	80%	52%	3.11	0.52
	C8	60%	40%	39%	80%	64%	78%	2.18	0.77
	C9	69%	14%	8%	80%	73%	23%	2.89	0.59
	C10	90%	65%	60%	95%	91%	81%	4.98	0.5
	C11	57%	33%	7%	68%	64%	51%	2.79	0.85
	C12	89%	85%	68%	98%	90%	85%	3.56	0.32
	C13	84%	71%	74%	60%	86%	85%	2.86	0.49
	C14	87%	66%	36%	89%	88%	–	2.98	0.62
	C15	79%	70%	39%	62%	81%	58%	2.16	0.59
	C16	79%	40%	5%	43%	78%	48%	2.69	0.39
	C17	63%	7%	5%	56%	67%	–	2.42	0.65
	C18	62%	–	0%	89%	60%	40%	2.19	0.86
	C19	81%	61%	47%	76%	82%	62%	2.92	0.53
Testing framework	Mocha	79%	50%	34%	71%	82%	58%	3.62	0.56
	Jasmine	75%	65%	34%	69%	74%	62%	2.28	0.71
	QUnit	71%	53%	28%	76%	71%	68%	3.35	0.66
	Own test	58%	45%	26%	66%	61%	51%	1.78	0.63
	Tap	89%	68%	87%	94%	89%	–	2.52	0.24
	Tape	94%	79%	65%	92%	93%	88%	3.19	0.22
	Others	77%	33%	30%	66%	80%	79%	2.14	0.48
	Nodeunit	72%	53%	63%	74%	74%	52%	4.08	0.62
	Vows	72%	60%	38%	79%	74%	0%	1.60	0.6
	Client	70%	46%	25%	69%	70%	66%	2.96	0.68
	Server	83%	64%	48%	82%	85%	67%	3.19	0.45
	C&S	70%	48%	29%	69%	72%	57%	2.93	0.69
	Total	76%	53%	36%	74%	78%	63%	3.05	0.57

test code coverage does not contribute in the low coverage of production code.

Uncovered statement in uncovered function (USUF) ratio. If an uncovered code c belongs to an uncovered function f, making f called could possibly cover c as well. As described in Section 4.2.2, we calculate the ratio of uncovered statements that fall within uncovered functions over the total number of uncovered statements.

Table 4.3 shows average values for this ratio (USUF). The mean value of USUF ratio is 0.57 in total, 0.45 for server-side projects, and about 0.7 for client-side ones. This indicate that the majority of uncovered statements in client-side code belong

to uncovered functions, and thus code coverage could be increased to a high extent if the enclosing function could be called during test execution.

***Finding 10**: A large portion of uncovered statements fall in uncovered functions for client-side code (about 70%) compared to server-side code (45%).*

Hard-to-test-function coverage. We measure coverage for hard-to-test functions as defined in Section 4.2.2. While the average function coverage in total is 76%, the average *event-dependent callback coverage* is 36% and the average *asynchronous callback coverage* is 53%. The average value of *closure function coverage* in total is 74% and for server-side subjects is 82% while it is 69% for client-side ones.

***Finding 11**: On average, JavaScript tests have low coverage for event-dependent callbacks (36%) and asynchronous callbacks (53%). Average values for client-side code are even worse (25% and 46% respectively). The average, closure function coverage is 74%.*

We measure the impact of tests with event triggering methods on event-dependent callback coverage, and writing async tests on asynchronous callback coverage through correlation analysis. The results show that there exists a weak correlation ($\rho = 0.22$) between number of event triggers and event-dependent callback coverage, and a very weak correlation ($\rho = 0.1$) between number of asynchronous tests and asynchronous callback coverage.

***Finding 12**: There is no strong correlation between number of event triggers and event-dependent callback coverage. Also number of asynchronous tests and asynchronous callback coverage are not strongly correlated.*

This was contrary to our expectation for higher correlations, however, we observed that in some cases asynchronous tests and tests that trigger events were written to merely target specific parts and functionalities of the production code without covering most asynchronous or event-dependent callbacks.

DOM related code coverage. On average, JavaScript tests have a moderately low coverage of 63% for DOM-related code. We also study the relationship of existence of DOM fixtures and DOM related code coverage through correlation analysis. The result shows that there exists a correlation of $\rho = 0.4$, $p = 0$ between having DOM fixtures in tests and DOM related code coverage. Similar to the cases

for event-dependent and async callbacks, we also observed that DOM fixtures were mainly written for executing a subset of DOM related code.

***Finding 13**: On average, JavaScript tests lack proper coverage for DOM-related code (63%). Also there exists a moderately low correlation (ρ = 0.4) between having DOM fixtures in tests and DOM related code coverage.*

Average number of function calls per test. As explained in Section 4.2.2, we investigate number of unique function calls per test. The average number of function calls per test has a mean value of about 3 in total and also across server-side and client-side code. We further perform a correlation analysis between the average number of function calls per test and total code coverage. The result shows that there exists a weak correlation (ρ = 0.13, p = 0) between average number of function calls per test and code coverage.

***Finding 14**: On average, there are about 3 function calls to production code per test case. The average number of function calls per test is not strongly correlated with code coverage.*

4.3.4 Discussion

Implications. Our findings regarding RQ1 indicate that the majority (78%) of studied JavaScript projects and in particular popular and trending ones have at least one test case. This indicates that JavaScript testing is getting attention, however, it seems that developers have less tendency to write tests for client-side code as they do for the server-side code. Possible reasons could be difficulties in writing proper DOM fixtures or triggering events on DOM elements. We also think that the high percentage of test for server-side JavaScript can be ascribed to the testing pattern that is advocated in the *Node.js* community [57]. To assist developers with testing their JavaScript code, we believe that it is worthwhile for the research community to invest on developing test generation techniques in particular for the client-side code, such as [127, 130, 131].

For RQ2, the results indicate that in general, tests written for mainly client-side subjects in categories C2 (visualization), C8 (touch and drag&drop), C11 (game engines), and C18 (multimedia) have lower quality. Compared to the client-side

projects, tests written for the server-side have higher quality in terms of code coverage, test code ratio, and test commit ratio. The branch coverage in particular for client-side code is low, which can be ascribed to the challenges in writing tests for DOM related branches. We investigate reasons behind the code coverage difference in Section 4.3.3. The higher values for test code ratio and test commit ratio can also be due to the fact that writing tests for server-side code is easier compared to client-side.

Developers and testers could possibly increase code coverage of their tests by using existing JavaScript test generator tools, such as Kudzu [158], ARTEMIS [66], JALANGI [161], SymJS [109], JSEFT [130], and CONFIX [127]. Tests written in *Mocha*, *Tap*, *Tape*, and *Nodeunit* generally have higher test quality compared to other frameworks and tests that do not use any testing framework. In fact developers that do not write their test by leveraging an existing testing framework write low quality tests almost in all aspects. Thus we recommend JavaScript developers community to use a well-maintained and mature testing framework to write their tests.

As far as RQ3 is concerned, our study shows that JavaScript tests lack proper coverage for event-dependent callbacks, asynchronous callbacks, and DOM-related code. Since these parts of code are hard to test they can be error prone and thus requires effective targeted tests. For instance a recent empirical study [141] reveals that the majority of reported JavaScript bugs and the highest impact faults are DOM-related.

It is expected that using event triggering methods in tests, increase coverage for event-dependent callbacks, asynchronous callbacks, and DOM-related statements. However, our results do not show a strong correlation to support this. Our manual analysis revealed that tests with event triggering methods, async behaviours, and DOM fixtures are mainly written to cover only particular instances of event-dependent callbacks, asynchronous callbacks, or DOM-related code. This again can imply difficulties in writing tests with high coverage for such hard-to-test code.

We believe that there is a research potential in this regard for proposing test generation techniques tailored to such uncovered parts. While most current test generation tools for JavaScript produce tests at single function level, in practice developers often write tests that invoke about 3 functions per test on average. It

might also worth for researchers to develop test generation tools that produce tests with a sequence of function calls per test case.

Finally, we observed that UI tests are much less prevalent in the studied JavaScript projects. Our investigation of the coverage report did not show a significant coverage increase on the uncovered event-dependent callbacks or DOM-related code between UI and unit tests. Since UI tests do not need DOM fixture generation, they should be able to trigger more of the UI events, compared to code level unit tests. It would be interesting to further investigate this in JavaScript applications with large UI tests.

Test effectiveness. Another test quality metric that is interesting to investigate is test effectiveness. An ideal effective test suite should fail if there is a defect in the code. *Mutation score*, i.e., the percentage of killed mutants over total non-equivalent mutants, is often used as an estimate of defect detection capability of a test suite. In fact it has been shown that there exists a significant correlation between mutant detection and real fault detection [102]. In this work, however, we did not consider mutation score as a quality metric as it was too costly to generate mutants for each subject and execute the tests on each of them. We believe that it is worthwhile to study the effectiveness of JavaScript tests using mutation testing techniques, such as Mutandis [129], which guides mutation generation towards parts of the code that are likely to affect the program output. This can help to find out which aspects of code are more error-prone and not well-tested. Apart from test quality evaluation based on mutation score, studying JavaScript bug reports [142] and investigating bug locations, can give us new insights for developing more effective test generation tools.

Threats to validity. With respect to reproducibility of the results, our tool and list of the studied subjects are publicly available [59]. Regarding the generalizability of the results to other JavaScript projects, we believe that the studied set of subjects is representative of real-world JavaScript projects as they differ in domain (category), size (SLOC), maturity (number of commits and contributors), and popularity (number of stars and watchers). With regards to the subject categorization, we used some existing categories proposed by *JSter* Catalog [54] and *GitHub Showcases* [56].

There might be case that TESTSCANNER cannot detect a desired pattern in the code as it performs complex static code analysis for detecting DOM-related statements, event-dependent callbacks, and asynchronous APIs. To mitigate this threat, we made a second pass of manual investigation through such code patterns using `grep` with regular expressions in command line and manually validated random cases. Such a textual search within JavaScript files through `grep` was especially done for a number of projects with parsing errors in their code for which TESTSCANNER cannot generate a report or the report would be incomplete. Since our tool statically analyzes test code to compute the number of function calls per test, it may not capture the correct number of calls that happen during execution. While dynamic analysis could help with this regard, it can not be used for the unexecuted code and thus is not helpful to analyze uncovered code.

4.4 Related Work

There are number of previous empirical studies on JavaScript. Ratanaworabhan et al. [152] and Richards et al. [153] studied JavaScript's dynamic behavior and Richards et al. [154] analyzed security issues in JavaScript projects. Ocariza et al. [142] performed study to characterize root causes of client-side JavaScript bugs. Gallaba et al. [87] studied the use of callback in client and server-side JavaScript code. Security vulnerabilities in JavaScript have also been studied on remote JavaScript inclusions [140], [193], cross-site scripting (XSS) [183], and privacy violating information flows [98]. Milani Fard et al. [124] studied code smells in JavaScript code. Nguyen et al. [139] performed usage patterns mining in JavaScript web applications.

Researchers also studied test cases and mining test suites in the past. Inozemtseva et al. [97] found that code coverage does not directly imply the test suite effectiveness. Zhang et al. [199] analyzed test assertions and showed that existence of assertions is strongly correlated with test suite effectiveness. Vahabzadeh et al. [176] studied bugs in test code. Milani Fard et al. proposed Testilizer [126] that mines information from existing test cases to generate new tests. Zaidman et al. [194] investigated co-evolution of production and test code.

These work, however, did not study JavaScript tests. Related to our work,

Mirshokraie et al. [129] presented a JavaScript mutation testing approach and as part of their evaluation, assessed mutation score for test suites of two JavaScript libraries. To the best of our knowledge, our work is the first (large scale) study on JavaScript tests and in particular their quality and shortcomings.

4.5 Conclusions

JavaScript is heavily used to build responsive client-side web applications as well as server-side projects. While some JavaScript features are known to be hard to test, no empirical study was done earlier towards measuring the quality and coverage of JavaScript tests. This work presents the first empirical study of JavaScript tests to characterize their prevalence, quality metrics, and shortcomings.

We found that a considerable amount of JavaScript projects do not have any test and this is in particular for projects with JavaScript at client-side. On the other hand almost all purely server-side JavaScript projects have tests and the quality of those tests are higher compared to tests for client-side. On average JavaScript tests lack proper coverage for event-dependent callbacks, asynchronous callbacks, and DOM-related code. The result of this study can be used to improve JavaScript test generation tools in producing more effective test cases that target hard-to-test code.

It would be interesting to evaluate effectiveness of JavaScript test by measuring their mutation score, which reveals the quality of written assertions. Another possible direction could be designing automated JavaScript code refactoring techniques towards making the code more testable and maintainable.

Chapter 5

Generating Fixtures for JavaScript Unit Testing

Summary[11]

In today's web applications, JavaScript code interacts with the Document Object Model (DOM) at runtime. This runtime interaction between JavaScript and the DOM is error-prone and challenging to test. In order to unit test a JavaScript function that has read/write DOM operations, a DOM instance has to be provided as a test fixture. This DOM fixture needs to be in the exact structure expected by the function under test. Otherwise, the test case can terminate prematurely due to a null exception. Generating these fixtures is challenging due to the dynamic nature of JavaScript and the hierarchical structure of the DOM. We present an automated technique, based on dynamic symbolic execution, which generates test fixtures for unit testing JavaScript functions. Our approach is implemented in a tool called CONFIX. Our empirical evaluation shows that CONFIX can effectively generate tests that cover DOM-dependent paths. We also find that CONFIX yields considerably higher coverage compared to an existing JavaScript input generation technique.

5.1 Introduction

To create responsive web applications, developers write JavaScript code that dynamically interacts with the DOM. As such, changes made through JavaScript code

[11] An initial version of this chapter has been published in the IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015 [127].

via these DOM API calls become directly visible in the browser.

This complex interplay between two separate languages, namely JavaScript and the HTML, makes it hard to analyze statically [101, 111], and particularly challenging for developers to understand [62] and test [66, 130] effectively. As revealed in a recent empirical study [141], the majority (65%) of reported JavaScript bugs are *DOM-related*, meaning the fault pertains to a DOM API call in JavaScript code. Moreover, 80% of the highest impact JavaScript faults are DOM-related.

In order to unit test a JavaScript function that has DOM read/write operations, a DOM instance needs to be provided in the exact structure as expected by the function. Otherwise, a DOM API method (e.g., `var n = getElementById("news")`) returns `null` because the expected DOM node is not available; any operations on the variable pointing to this non-existent DOM node (e.g., `n.firstChild`) causes a null exception and the test case calling the function terminates prematurely. To mitigate this problem, testers have to write *test fixtures* for the expected DOM structure before calling the JavaScript function in their unit tests. The manual construction of proper DOM fixtures is, however, tedious and costly.

Despite the problematic nature of JavaScript-DOM interactions, most current automated testing techniques ignore the DOM and focus on generating events and function arguments [66, 161]. JSeft [130] applies an approach in which the application is executed and the DOM tree is captured just before executing the function under test. This DOM tree is used as a test fixture in generated test cases. This heuristic-based approach, however, assumes that the DOM captured at runtime contains all the DOM elements, values, and relations as expected by the function, which is not always the case. Thus, the code coverage achieved with such a DOM can be quite limited. Moreover, the DOM captured this way, can be too large and difficult to read as a fixture in a test case. SymJS [109] applies symbolic execution to increase JavaScript code coverage, with limited support for the DOM, i.e., it considers DOM element variables as integer or string values and ignores the DOM structure. However, there exist complex DOM structures and element relations expected by the JavaScript code in practice, which this simplification cannot handle.

In this work, we provide a technique for automatically generating DOM-based fixtures and function arguments. Our technique, called ConFix, is focused on cov-

ering DOM-dependent paths inside JavaScript functions. It operates through an iterative process that (1) dynamically analyses JavaScript code to deduce DOM-dependent statements and conditions, (2) collects path constraints in the form of symbolic DOM constraints, (3) translates the symbolic path constraints into XPath expressions, (4) feeds the generated XPath expressions into an existing structural constraint solver [89] to produce a satisfiable XML structure, (5) generates a test case with the solved structure as a test fixture or function argument, runs the generated test case, and continues recursively until all DOM-dependent paths are covered.

To the best of our knowledge, our work is the first to consider the DOM as a test input entity, and to automatically generate test fixtures to cover DOM-dependent JavaScript functions.

Our work makes the following main contributions:

- A novel dynamic symbolic execution engine to generate DOM-based test fixtures and inputs for unit testing JavaScript functions;
- A technique for deducing DOM structural constraints and translating those to XPath expressions, which can be fed into existing structural constraint solvers;
- An implementation of our approach in a tool, called ConFix, which is publicly available [24];
- An empirical evaluation to assess the coverage of ConFix on real-world JavaScript applications. We also compare ConFix's coverage with that of other JavaScript test generation techniques.

The results of our empirical evaluation show that ConFix yields considerably higher coverage — up to 40 and 31 percentage point increase on the branch, and the statement coverage, respectively — compared to tests generated without DOM fixtures/inputs.

```
1 function dg(x){
2   return document.getElementById(x);
3 }

5 function sumTotalPrice(){
6   sum = 0;
7   itemList = dg("items");
8   if (itemList.children.length == 0)
9     dg("message").innerHTML = "List is empty!";
10  else {
11    for (i = 0; i < itemList.children.length; i++){
12      p = parseInt(itemList.children[i].value);
13      if (p > 0)
14        sum += p;
15      else
16        dg("message").innerHTML += "Wrong value for the price of item "↩
              + i;
17    }
18    dg("total").innerHTML = sum;
19  }
20  return sum;
21 }
```

Figure 5.1: A JavaScript function to compute the items total price.

5.2 Background and Motivation

The majority of reported JavaScript bugs are caused by faulty interactions with the DOM [141]. Therefore, it is important to properly test DOM-dependent JavaScript code. This work is motivated by the fact that the execution of some paths in a JavaScript function, i.e., unique sequences of branches from the function entry to the exit, depends on the existence of specific DOM structures. Such DOM structures have to be provided as test fixtures to effectively test such DOM-dependent paths.

5.2.1 DOM Fixtures for JavaScript Unit Testing

A test fixture is a fixed state of the system under test used for executing tests [122]. It pertains to the code that initializes the system, brings it into the right state, prepares input data, or creates mock objects, to ensure tests can run and produce repeatable results. In JavaScript unit testing, a fixture can be a partial HTML that the JavaScript function under test expects and can operate on (read or write to), or a fragment of JSON/XML to mock the server responses in case of XMLHttpRequest

(XHR) calls in the code.

Running Example.. Figure 5.1 depicts a simple JavaScript code for calculating the total price of online items. We use this as a running example to illustrate the need for providing proper DOM structures as test input for unit testing JavaScript code.

Lets assume that a tester writes a unit test for the `sumTotalPrice` function without any test fixture. In this case, when the function is executed, it throws a `null` exception at line 8 (Figure 5.1) when the variable `itemList` is accessed for its `children` property. The reason is that the DOM API method `getElementById` (line 2) returns `null` since there is no DOM tree available when running the unit test case. Consequently, `dg` (line 2) returns `null`, and hence the exception at line 8. Thus, in order for the function to be called from a test case, a DOM tree needs to be present. Otherwise, the function will terminate prematurely. What is interesting is that the mere presence of the DOM does not suffice in this case. The DOM tree needs to be in a particular structure and contain attributes and values as expected by the different statements and conditions of the JavaScript function.

For instance, in the case of `sumTotalPrice`, in order to test the calculation, a DOM tree needs to be present that meets the following constraints:

1. A DOM element with id `"items"` must exist (line 7)
2. That element needs to have one or more child nodes (lines 8, 10)
3. The child nodes must be of a DOM element type that can hold *values* (line 12), e.g., `<input value="..."/>`
4. The values of the child nodes need to be positive (line 13) integers (lines 12)
5. A DOM element with id `"total"` must exist (line 18).

A DOM subtree that satisfies these constraints is depicted in Figure 5.2, which can be used as a test fixture for unit testing the JavaScript function. There are currently many JavaScript unit testing frameworks available, such as QUnit [40], JSUnit [35], and Jasmine [31]. We use the popular QUnit framework to illustrate the running example in this work. QUnit provides a `$"qunit-fixture"` variable to which DOM test fixtures can be appended in a test case. A QUnit unit test

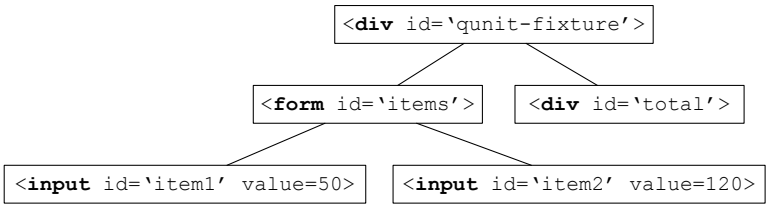


Figure 5.2: A DOM subtree for covering a path (lines 6-8, 10-14, and 18-20) of `sumTotalPrice` in Figure 5.1.

```
1 test("A test for sumTotalPrice", function() {
2   $("#qunit-fixture").append('<form id="items"><input type="text" id="↩
        item1" value=50><input type="text" id="item2" value=120></form><↩
        div id="total"/>');
3   sum = sumTotalPrice();
4   equal(sum, 170, "Function sums correctly.");
5 });
```

Figure 5.3: A QUint test case for the `sumTotalPrice` function. The DOM subtree of Figure 5.2 is provided as a fixture before calling the function. This test with this fixture covers the path going through lines 6-8, 10-14, and 18 in `sumTotalPrice`.

is shown in Figure 5.3. The execution of the test case with this particular fixture results in covering a path (lines 6-8, 10-14, and 18). If the fixture lacks any of the provided DOM elements that is required in the execution path, the test case fails and terminates before reaching the assertion.

Other DOM fixtures are required to achieve branch coverage. For example, to cover the true branch of the `if` condition in line 8, the DOM must satisfy the following constraints:

1. A DOM element with id `"items"` must exist (line 7)
2. That element must have no child nodes (line 8)
3. A DOM element with id `"message"` must exist (line 9).

Yet another DOM fixture is needed for covering the `else` branch in line 16:

1. A DOM element with id `"items"` must exist (line 7)
2. That element needs to have one or more child nodes (lines 8, 10)
3. The child nodes must be of a DOM element type that can hold *values* (line 12) e.g., `input`

4. The child nodes values need to be integers (lines 12)
5. The value of a child node needs to be zero or negative (line 15)
6. A DOM element with id `"message"` must exist (line 16).
7. A DOM element with id `"total"` must exist (line 18).

5.2.2 Challenges

As illustrated in this simple example, different DOM fixtures are required for maximizing JavaScript code coverage. Writing these DOM fixtures manually is a daunting task for testers. Generating these fixtures is not an easy task either. There are two main challenges in generating proper DOM-based fixtures that we address in our proposed approach.

Challenge 1: DOM-related variables.. JavaScript is a weakly-typed and highly-dynamic language, which makes static code analysis quite challenging [153, 161, 182]. Moreover, its interactions with the DOM can become difficult to follow [62, 141]. For instance, in the condition of line 13 in Figure 5.1, the value of the variable `p` is checked. A fixture generator needs to determine that `p` is DOM-dependent and it refers to the value of a property of a DOM element, i.e., `itemList.children[i].value`.

Challenge 2: Hierarchical DOM relations.. Unlike most test fixtures that deal only with primitive data types, DOM-based test fixtures require a tree structure. In fact, DOM fixtures not only contain proper DOM elements with attributes and their values, but also hierarchical parent-child relations that can be difficult to reconstruct. For example the DOM fixture in Figure 5.2 encompasses the parent-child relation between `<form>` and `<input>` elements, which is required to evaluate `itemList.children.length` and `itemList.children[i].value` in the code (lines 8, 11, and 12 in Figure 5.1).

5.2.3 Dynamic Symbolic Execution

Our insight in this work is that the problem of generating expected DOM fixtures can be formulated as a constraint solving problem, to achieve branch coverage.

Symbolic execution [106] is a static analysis technique that uses symbolic values as input values instead of concrete data, to determine what values cause each

branch of a program to execute. For each decision point in the program, it infers a set of symbolic constraints. Satisfiability of the conjunction of these symbolic constraints is then checked through a constraint solver. *Concolic execution* [92, 160], also known as *dynamic symbolic execution* [170], performs symbolic execution while systematically executing all feasible program paths of a program on some concrete inputs. It starts by executing a program with random inputs, gathers symbolic constraints at conditional statements during execution, and then uses a constraint solver to generate a new input. Each new input forces the execution of the program through a new uncovered path; thus repeating this process results in exploring all feasible execution paths of the program.

5.3 Approach

We propose a DOM-based test fixture generation technique, called CONFIX. To address the highly dynamic nature of JavaScript, is based on a dynamic symbolic execution approach.

Scope. Since primitive data constraints can be solved using existing input generators for JavaScript [109, 158, 161], in this work, we focus on collecting and solving DOM constraints that enable achieving coverage for DOM dependent statements and conditions in JavaScript code. Thus, is designed to generate DOM-based test fixtures and function arguments for JavaScript functions that are *DOM-dependent*.

Definition 11 (DOM-Dependent Function) *A DOM dependent function is a JavaScript function, which directly or indirectly accesses DOM elements, attributes, or attribute values at runtime using DOM APIs.* □

An instance of a direct access to a DOM element is `document.getElementById("items")` in the function `dg` (Line 2, Figure 5.1). An indirect access is a call to another JavaScript function that accesses the DOM. For instance, the statement at line 7 of Figure 5.1 is an indirect DOM access through function `dg`.

Overview. Figure 5.4 depicts an overview of CONFIX. At a high level, instruments the JavaScript code (block 1), and executes the function under test to collect a trace (blocks 2 and 3). Using the execution trace, it deduces DOM-dependent path

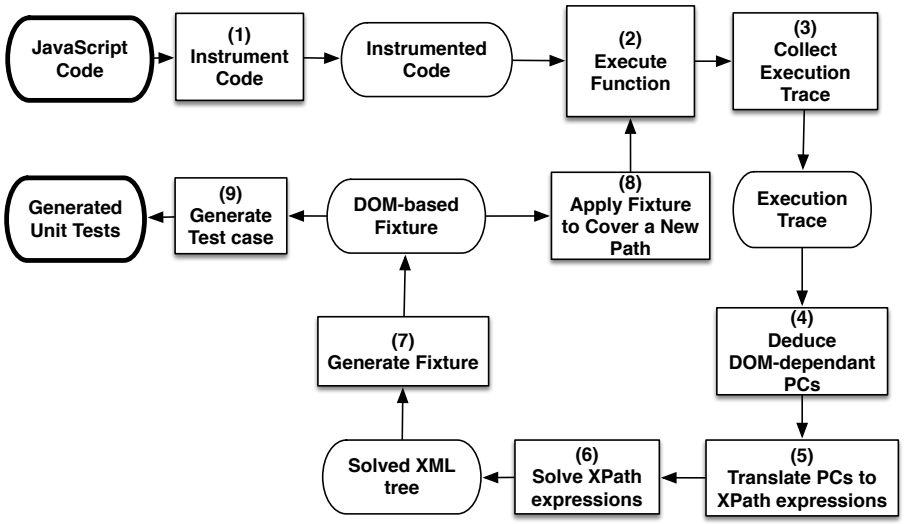


Figure 5.4: Processing view of our approach.

constraints (block 4), translates those constraints into XPath expressions (block 5), which are fed into an XML constraint solver (block 6). The solved XML tree is then used to generate a DOM-based fixture (block 7), which subsequently helps in covering unexplored paths (block 8). Finally, it generates a test suite by adding generated test fixtures into a JavaScript unit testing template such as QUnit (block 9). In the following subsections we discuss each of these steps in more details.

Algorithm. Algorithm 4 demonstrates our DOM fixture generation technique. The input to our algorithm is the JavaScript code, the function under test f, and optionally its function arguments (provided by a tester or a tool [158, 161]). The algorithm concolically generates DOM fixtures required for exploring all DOM-dependent paths.

5.3.1 Collecting DOM-based Traces

We instrument the JavaScript code under test (algorithm 4 line 3) with wrapper functions to collect information pertaining to DOM interactions, which include statements or conditional constructs that depend on the existence of a DOM structure, element, attribute, or value. The instrumentation is non-intrusive meaning that it does not affect the normal functionality of the original program.

The instrumentation augments function calls, conditionals (in `if` and loop constructs), infix expressions, variable initializations and assignments, and return statements with inline wrapper functions to store an execution trace (see Section 5.3.5 for more details). Such a trace includes the actual statement, type of statement (e.g. infix, condition, or function call), list of variables and their values in the statement, the enclosing function name, and the actual concrete values of the statement at runtime. CONFIX currently supports DOM element retrieval patterns based on tag names, IDs, and class names, such as `getElementById`, `getElementsByTagName`, `children`, `innerHTML`, `parentNode`, and `$()` for jQuery-based code.

After instrumenting the source code, the modified JavaScript file is used in a runner HTML page that is loaded inside a browser (line 7, algorithm 4) and then a JavaScript driver (e.g., WebDriver [42]) executes the JavaScript function under test (line 8). The initial DOM fixture is an empty HTML runner having a `div` element with id `qunit-fixture`. This execution results in an execution trace, which is collected from the browser for further analysis (line 9).

5.3.2 Deducing DOM Constraints

An important phase of dynamic symbolic execution is gathering *path constraints* (PCs). In our work, path constraints are conjunctions of constraints on symbolic DOM elements.

Definition 12 (Symbolic DOM Element) *A symbolic DOM element d is a data structure representing a DOM element in terms of its symbolic properties and values. d is denoted by a 4 tuple $< \mathscr{P}, \mathscr{C}, \mathscr{A}, \mathscr{T} >$ where:*

1. *$\mathscr{P}$ is d's parent symbolic DOM element.*
2. *$\mathscr{C}$ is a set of child symbolic DOM elements of d.*
3. *$\mathscr{A}$ is a set of $\langle$`att, val`$\rangle$ pairs; each pair stores an attribute `att` of d with a symbolic value `val`.*
4. *$\mathscr{T}$ is the element type of d.* □

Algorithm 4: Test Fixture Generation

```
   input  : JavaScript code JS, the function under test f, function arguments for f
   output: A set of DOM fixtures fixtureSet for f
 1 negatedConstraints ← ∅
 2 DOMRefTrackList ← ∅
   Procedure GENERATEFIXTURE(JS, f)
   begin
 3     JS_inst ← INSTRUMENT(JS)
 4     fixtureSet ← ∅
 5     fixture ← ∅
 6     repeat
 7         browser.LOAD(JS_inst, fixture)
 8         browser.EXECUTE(f)
 9         t ← browser.GETEXECUTIONTRACE()
10         fixture ← SOLVECONSTRAINTS(t)
11         fixtureSet ← fixtureSet ∪ fixture
       until fixture ≠ ∅;
12     return fixtureSet
   end
   Procedure SOLVECONSTRAINTS(t)
   begin
13     DOMRefTrackList ← GETDOMREFERENCETRACKS(t)
14     pc ← GETPATHCONSTRAINT(t, DOMRefTrackList)
15     fixture ← UNSAT
16     while fixture = UNSAT do
17         fixture ← ∅
18         c ← GETLASTNONNEGCONST(pc, negatedConstraint)
19         if c ≠ null then
20             negatedConstraint ← negatedConstraint ∪ c
21             pc ← NEGATECONSTRAINT(pc, c)
22             xp ← GENERATEXPATH(pc, DOMRefTrackList)
               /* SOLVEXPATHCONSTRAINT returns UNSAT if xp is not
                  solvable.                                          */
23             fixture ← SOLVEXPATHCONSTRAINT(xp)
           end
       end
24     return fixture
   end
```

Note that keeping the parent-children relation for DOM elements is sufficient to recursively generate the DOM tree.

DOM constraints in the code can be conditional or non-conditional. A *non-conditional* DOM constraint is a constraint on the DOM tree required by a DOM accessing JavaScript statement. A *conditional* DOM constraint is a constraint on the DOM tree used in a conditional construct.

Example 5 *Consider the JavaScript code in Figure 5.1. In line 2,*

`document.getElementById(x)` is a non-conditional DOM constraint, i.e., an element with a particular ID is targeted. On the other hand, `itemList.children.length == 0` (line 8) is a conditional DOM constraint, i.e., the number of child nodes of a DOM element is checked to be zero.

Non-conditional DOM constraints evaluate to `null` or a not-null object, while, conditional DOM constraints evaluate to `true` or `false`. For example, if we execute `sumTotalPrice()` with a DOM subtree void of an element with `id="items"`, the value of `itemList` at line 7 will be `null` and the execution will terminate at line 8 when evaluating `itemList.children.length == 0`. On the other hand, if that element exists and has a child, the condition at line 8 evaluates to `false`.

In this work, the input to be generated is a DOM subtree that is accessed via DOM APIs in the JavaScript code. As we explained in Section 5.2.2, due to the dynamic nature of JavaScript, its interaction with the DOM can be difficult to follow (challenge 1). Tracking DOM interactions in the code is needed for extracting DOM constraints. Aliases in the code add an extra layer of complexity. We need to find variables in the code that refer to DOM elements or element attributes. To address this challenge, we apply an approach similar to dynamic backward slicing, except that instead of slices of the code, we are interested in relevant DOM-referring variables. To that end, we use dynamic analysis to extract DOM referring variables from the execution trace, by first searching for DOM API calls, their arguments, and their actual values at runtime. The process of collecting DOM referring variables (Algorithm 4 line 13) is outlined further in subsection 5.3.5.

Once DOM referring variables are extracted, constraints on their corresponding DOM elements are collected and used to generate constraints on symbolic DOM elements (see Definition 12). DOM constraints can be either *attribute-wise* or *structure-wise*. *Attribute-wise* constraints are satisfied when special values are provided for element attributes. For example, `parseInt(itemList.children[i].value) > 0` (line 13 of Figure 5.1) requires the value of the `i`-th child node to be an integer greater than zero. The *value* is an attribute of the child node in this example. *Structure-wise* constraints are applied to the element type and its parent-children relations. For example, in

`itemList.children.length == 0` (line 8 of Figure 5.1 to cover the `else` branch (lines 10–19) an element with id `"items"` is needed with at least one child node.

The conjunction of these symbolic DOM constraints in an iteration forms a path constraint. For instance, the structure-wise constraint in `parseInt(itemList.children[i].value) > 0` (line 13 of Figure 5.1) requires the child nodes to be of an element type that can hold values, e.g., `input` type, and the attribute-wise constraint requires the value to be a positive integer.

Our technique reasons about a collected path constraint and constructs symbolic DOM elements needed. For each symbolic DOM element, (1) infers the type of the parent node, (2) determines the type and number of child nodes, and (3) generates, through a constraint solver, satisfied values that are used to assign attributes and values (i.e., ⟨`att, val`⟩ pairs). The default element type for a symbolic DOM element is `div` — the `div` is a placeholder element that defines a division or a section in an HTML document. It satisfies most of the element type constraints and can be parent/child of many elements — unless specific attributes/-values are accessed from the element, which would imply that a different element type is needed. For instance, if the `value` is read/set for an element, the type of that element needs to change to, for instance `input`, because per definition, the `div` type does not carry a `value` attribute (more detail in subsection 5.3.5). These path constraints with satisfied symbolic DOM elements are used to generate a corresponding XPath expression, as presented in the next subsection.

5.3.3 Translating Constraints to XPath

The problem of DOM fixture generation can be formulated as a decision problem for the emptiness test [69] of an XPath expression, in the presence of XHTML meta-models, such as Document Type Definitions (DTD) or XML Schemas. These meta-models define the hierarchical tree-like structure of XHTML documents with the type, order, and multiplicity of valid elements and attributes. XPath [75] is a query language for selecting nodes from an XML document. An example is the expression `/child::store/child::item/child::price` which navigates the root through the top-level “store” node to its “item” child nodes and

on to its "price" child nodes. The result of the evaluation of the entire expression is the set of all the "price" nodes that can be reached in this manner. XPath is a very expressive language. We propose a restricted XPath expression grammar, shown in Figure 5.5, which we use to model our DOM constraints in.

⟨*XPath*⟩ ::= ⟨*Path*⟩ | /⟨*Path*⟩

⟨*Path*⟩ ::= ⟨*Path*⟩/⟨*Path*⟩ | ⟨*Path*⟩[⟨*Qualifier*⟩] | child::⟨*Name*⟩ | ⟨*Name*⟩

⟨*Qualifier*⟩ ::= ⟨*Qualifier*⟩ and ⟨*Qualifier*⟩ | ⟨*Path*⟩ | @⟨*Name*⟩

⟨*Name*⟩ ::= ⟨*HTMLTag*⟩ | ⟨*Attribute*⟩=⟨*Value*⟩

⟨*HTMLTag*⟩ ::= a | b | button | div | form | frame | h1 - h6 | iframe | img | input | i | li | link | menu | option | ol | p | select | span | td | tr | ul

⟨*Attribute*⟩ ::= id | type | name | class | value | src | innerHTML | title | selected | checked | href | size | width | height

Figure 5.5: Restricted XPath grammar for modeling DOM constraints.

We transform the deduced path constraints, with the symbolic DOM elements, into their equivalent XPath expressions conforming to this specified grammar. These XPath expressions systematically capture the hierarchical relations between elements. Types of common constraints translated to expressions include specifying the existence of a DOM element/attribute/value, properties of style attributes, type and number of child nodes or descendants of an element, or binary properties such as selected/not selected.

Example 6 *Table 5.1 shows examples of collected DOM constraints that are translated to XPath expressions, for the running example. For example, in the first row, the DOM constraint* `document.getElementById("items")` $\neq$ `null` *is translated to the XPath expression* `div[@id="qunit-fixture"][div[@id="items"]]`*, which expresses the desire for the existence of a* `div` *element with id "items" in the fixture. The last row shows a more complex example, including six DOM constraints in a path constraint, which are translated into a corresponding XPath expression.*

Table 5.1: Examples of DOM constraints, translated XPath expressions, and solved XHTML instances for the running example.

DOM constraints	**Corresponding XPath expressions**	**Solved XHTML**
document.getElementById(“items”) $\neq$ null	div[@id=“qunit-fixture”][div[@id=“items”]]	
document.getElementById(“items”) $\neq$ null $\wedge$ itemList.children.length == 0 $\wedge$ document.getElementById(“message”) $\neq$ null	div[@id=“qunit-fixture”][div[@id=“items”] and child::div[@id=“message”]]	
document.getElementById(“items”) $\neq$ null $\wedge$ itemList.children.length $\neq$ 0 $\wedge$ 0 < itemList.children.length $\wedge$ document.getElementById(“message”) $\neq$ null	div[@id=“qunit-fixture”][div[@id=“items” and child::div[@id=“Confix1”]] and child::div[@id=“message”]]	
document.getElementById(“items”) $\neq$ null $\wedge$ itemList.children.length $\neq$ 0 $\wedge$ 0 < itemList.children.length $\wedge$ parseInt(itemList.children[0].value) > 0 $\wedge$ document.getElementById(“message”) $\neq$ null $\wedge$ document.getElementById(“total”) $\neq$ null	div[@id=“qunit-fixture”][div[@id=“items” and child::input[@id=“Confix1” and @value=”1”]] and child::div[@id=“message”] and child::div[@id=“total”]]	 <input id=“Confix1” value=“1”/>

5.3.4 Constructing DOM Fixtures

Next, the XPath expressions are fed into a structural XML solver [89]. The constraint solver parses the XPath expressions and compiles them into a logical representation, which is tested for satisfiability. If satisfiable, the solver generates a solution in the XML language. Since an XHTML meta-model (i.e., DTD) is fed into the solver along with the XPath expressions, the actual XML output is an instance of valid XHTML. The last column of Table 5.1 shows solved XHTML instances that satisfy the XPath expressions, for the running example. These solved XHTML instances are subsequently used to construct test fixtures.

Each newly generated fixture forces the execution of the JavaScript function under test along a new uncovered path. This is achieved by systematically negating the last non-negated conjunct in the path constraint and solving it to obtain a new fixture, in a depth first manner. If a path constraint is unsatisfiable, the technique chooses a different path constraint and this process repeats until all constraints are negated.

In the main loop of Algorithm 4 (lines 6–11), fixtures are iteratively generated and added to the `fixtureSet`. In the `SolveConstraints` procedure, a fixture is initialized to `UNSAT`; the loop (lines 16–23) continues until the fixture is set either to a solved DOM subtree[12] (line 23), or to $\varnothing$ (line 17) if there exist no non-negated constraints in the PCs. When a $\varnothing$ fixture returns from `SolveConstraints`, the loop in the main procedure terminates, and the `fixtureSet` is returned.

[12]Note that `SolveXpathConstraint` returns `UNSAT` when it fails to solve the given path constraint.

Table 5.2: Constraints table for the running example. The "Next to negate" field refers to the last non-negated constraint.

Iteration	Current fixture	Current DOM constraints	Negated	Next to negate	Fixture for the next iteration	Paths covered
1	∅	document.getElementById("items") = null	-	✓		Lines 1–8
2		document.getElementById("items") ≠ null ∧	✓	-		Lines 1–9
		itemList.children.length = 0 ∧	-	-	<div id="message"/ >	
		document.getElementById("message") = null	-	✓		
3		document.getElementById("items") ≠ null ∧	✓	-		Lines 1–9 and 20
	<div id="message"/ >	itemList.children.length = 0 ∧	-	✓		
		document.getElementById("message") ≠ null	✓	-		
4		document.getElementById("items") ≠ null ∧	✓	-		Lines 1–8, 10–13,
		itemList.children.length ≠ 0 ∧	✓	-		and 15–18
		0 < itemList.children.length ∧	-	-		
		parseInt(itemList.children[0].value) ≯ 0 ∧	-	-		
		document.getElementById("message") ≠ null ∧	✓	-		
		document.getElementById("total") = null	-	✓		
5		document.getElementById("items") ≠ null ∧	✓	-		Lines 1–8, 10–13,
		itemList.children.length ≠ 0 ∧	✓	-	<input id="Confix1" value="1"/>	and 15–20
		0 < itemList.children.length ∧	-	-		
		parseInt(itemList.children[0].value) ≯ 0 ∧	-	✓		
		document.getElementById("message") ≠ null ∧	✓	-		
		document.getElementById("total") ≠ null	✓	-		
6		document.getElementById("items") ≠ null ∧	✓	-	UNSAT ⇒ Negate last non-negated	Lines 1–8, 10–14,
	<input id="Confix1" value="1"/>	itemList.children.length ≠ 0 ∧	✓	-	constraint	and 18–20
		0 < itemList.children.length ∧	-	✓		
		parseInt(itemList.children[0].value) > 0 ∧	✓	-		
		document.getElementById("message") ≠ null ∧	✓	-		
		document.getElementById("total") ≠ null	✓	-		
		document.getElementById("items") ≠ null ∧	✓	-	No non-negated constraint exists ⇒	
	<input id="Confix1" value="1"/>	itemList.children.length ≠ 0 ∧	✓	-	Fixture = ∅	
		0 < itemList.children.length ∧	✓	-		
		parseInt(itemList.children[0].value) > 0 ∧	✓	-		
		document.getElementById("message") ≠ null ∧	✓	-		
		document.getElementById("total") ≠ null	✓	-		

Example 7 *Table 5.2 shows the extracted path constraints and their values, as well as the current and next iteration fixtures at each iteration of the concolic execution for the running example. Since there is no fixture available in the first iteration, the constraint obtained is* `document.getElementById("items")=null`. *This means the execution terminates at line 8 of the JavaScript code (Figure 5.1) due to a null exception. Our algorithm negates the last non-negated constraint (*`document.getElementById("items")`$\neq$`null`*), and generates the corresponding fixture (*`<div id="items">`*) to satisfy this negated constraint. This process continues until the solver fails at producing a satisfiable fixture in the sixth iteration.* `UNSAT` *is returned because the constraints* `itemList.children.length`$\neq$`0` *and the newly negated one (*`0`$\not<$`itemList.children.length`*) require that the number of child nodes be negative, which is not feasible in the DOM structure. It then tries to generate a fixture by negating the last non-negated constraint without applying any fixtures, i.e., the path constraint extracted in the sixth iteration. However, in this case there are no non-negated constraints left since the last one (*`0<itemList.children.length`*) had already been negated and the result was not satisfiable. Consequently the algorithm terminates with an empty fixture in the last iteration. The table also shows which paths of the running example are covered in terms of lines of JavaScript code.*

5.3.5 Implementation Details

ConFix currently generates QUnit test cases with fixtures, however, it can be easily adapted to generate test suites in other JavaScript testing frameworks. To parse the JavaScript code into an abstract syntax tree for instrumentation, we build up on Mozilla Rhino [41]. To collect execution traces while executing a function in the browser, we use WebDriver [42].

XML Solver.. To solve *structure-wise* DOM constraints using XPath expressions, we use an existing XML reasoning solver [89]. A limitation with this solver is that it cannot generate XML structures with valued attributes (i.e., attributes are supported but not their values). To mitigate this, we developed a transformation technique that takes our generic XPath syntax (Figure 5.5) and produces an XPath format understandable by the solver. More specifically, we merge attributes and

their values together and feed them to the solver as single attributes to be generated. Once the satisfied XML is generated, we parse and decompose it to the proper attribute-value format as expected in a valid XHTML instance. Another limitation is that it merges instances of the same tag elements at the same tree level when declared as children of a common parent. We resolved this by appending an auto-increment number to the end of each tag and remove it back once the XML produced.

Handling asynchronous calls. Another challenge we encountered pertains to handling asynchronous HTTP requests that send/retrieve data (e.g., in JSON/XML format) to/from a server performed by using the `XMLHttpRequest` (XHR) object. This feature makes unit-level testing more challenging since the server-side generated data should also be considered in a test fixture as an XHR response if the function under test (in)directly uses the XHR and expects a response from the server. Existing techniques [100] address this issue by mocking the server responses, but they require multiple concrete executions of the application to learn the response. This is, however, not feasible in our case because we generate JavaScript unit tests in isolation from other dependencies such as the server-side code. As a solution, our instrumentation replaces the XHR object of the browser with a new object and redefines the XHR `open()` method in a way that it always uses the `GET` request method to synchronously retrieve data from a local URL referring to our mocked server. This helps us to avoid null exceptions and continue the execution of the function under test. However, if the execution depends on the actual value of the retrieved data (and not merely their existence), our current approach can not handle it. In such cases, a string solver [105] may be helpful.

Tracking DOM-referring variables. To detect DOM-referring variables — used to generate constraints on symbolic DOM elements (Definition 12) — we automatically search for DOM API calls, their arguments, and their actual values at runtime, in the execution trace. Algorithm 4 keeps track of DOM references (line 13) by storing information units, called DOM Reference Track (DRT), in a data structure.

Definition 13 (DOM Reference Track (DRT)) *A DOM reference track is a data structure capturing how a DOM tree is accessed in the JavaScript code. It is de-*

Table 5.3: DRT data structure for the running example.

Iteration	DOMVariable	ParentVariable	Type Element	AttributeVariables	Exists?
1	itemList	document	div	⟨id:items, -⟩	✗
2	itemList	document	div	⟨id:items,-⟩	✓
	-	itemList	div	⟨id:Confix1,-⟩	✗
	-	document	div	⟨id:message,-⟩	✗
3	itemList	document	div	⟨id:items,-⟩	✓
	-	itemList	div	⟨id:Confix1,-⟩	✗
	-	document	div	⟨id:message,-⟩	✓
...	...	...	...	...	...
6	itemList	document	div	⟨id:items,-⟩	✓
	-	itemList	input	⟨id:Confix1,-⟩, ⟨value:1, p⟩	✓
	-	document	div	⟨id:message,-⟩	✓
	-	document	div	⟨id:total,-⟩	✓

noted by a 4 tuple $< \mathscr{D}, \mathscr{P}, \mathscr{A}, \mathscr{T} >$ *where:*

1. $\mathscr{D}$ *(DOMVariable) is a JavaScript variable v that is set to refer to a DOM element d.*
2. $\mathscr{P}$ *(ParentVariable) is a JavaScript variable (or the* `document` *object) that refers to the parent node of d.*
3. $\mathscr{A}$ *(AttributeVariables) is a set of* $\langle$`att:val, var`$\rangle$ *pairs; each pair stores the variable* `var` *in the code that refers to an attribute* `att` *of d with a value* `val`*.*
4. $\mathscr{T}$ *(ElementType) is the node type of d.* □

When JavaScript variables are evaluated in a condition, the DRT entries in this data structure are examined to determine whether they refer to the DOM. If the actual value of a JavaScript variable at runtime contains information regarding a DOM element object, and it does not exist in our DRT data structure, we add it as a new DOM referring variable. Table 5.3 presents an example of the DRT for the running example.

We implemented a constraint solver that reasons about some common symbolic DOM constraints such as string/integer attribute values, and number of children nodes. Specifically the solver infers conditions on DOM referring variables by examining DRT entries. If the constraint is on an attribute of a DOM element, then the `AttributeVariables` property of the corresponding DRT will be updated with a satisfying value. In case the constraint is a structural constraint, such as number of child nodes, a satisfying number of DRT entities would be added to the table. Table 5.3 depicts the process of constructing the DRT during different iterations of the concolic execution. The `Exists` field indicates whether the element exists in the DOM fixture.

Example 8 *Consider the running example of Figure 5.1. When* `sumTotalPrice()` *is called in the first iteration,* `dg("items")` *returns* `null` *as no DOM element with ID* `items` *exists. Table 5.3 would then be populated by adding the first row:* `DOMVariable` *is* `itemList`*, the* `ParentVariable` *points to* `document`*, the default element type is set to* `div`*, and the attribute* `id` *is set to* `items`*; and this particular element does not exist yet. The execution terminates with a null exception at line 8. In the next iteration,* ConFix *updates the DOM fixture with a* `div` *element with id* `items`*. Therefore* `dg("items")` *returns a DOM element and line 8 evaluates the number of child nodes under* `itemList`*. This would then update the table with a new entry having* `ParentVariable` *point to* `itemList` *and attribute* `id` *set to an automatically incremented id* `"Confix1"` *(in the second row). This process continues as shown in Table 5.3.*

Generating DOM-based function arguments. Current tools for JavaScript input generation (e.g., [158, 161]) only consider primitive data types for function arguments and thus cannot handle functions that take DOM elements as input. Consider the following simple function:

```
1  function foo(elem) {
2    var price = elem.firstChild.value;
3    if (price > 200) {
4        ...
5    }}
```

The `elem` function parameter is expected to be a DOM element, whose first child node's value is read in line 2 and used in line 3. The problem of generating DOM function arguments is not fundamentally different from generating DOM fixtures. Thus, we propose a solution for this issue in CONFIX. The challenge here, however, is that JavaScript is dynamically typed and since `elem` in this example does not reveal its type statically nor when `foo` is executed in isolation in a unit test (because `elem` does not exist to log its type dynamically), it is not possible to determine that `elem` is a DOM element. To address this challenge, CONFIX first computes a forward slice of the function parameters. If there is a DOM API call in the forward slice, CONFIX deduces constraints and solves them similarly to how DOM fixtures are constructed. The generated fixture is then parsed into a DOM element object and the function is called with that object as input in the test case. In the example above, there is a DOM API call present, namely `firstChild` in the forward slice of `elem`. Therefore, CONFIX would know that `elem` is a DOM element and would generate it accordingly. Then `foo` is called in the test case with that object as input.

5.4 Empirical Evaluation

To assess the efficacy of our proposed technique, we have conducted a controlled experiment to address the following research questions:

RQ1 (Coverage) Can fixtures generated by CONFIX effectively increase code coverage of DOM-dependent functions?

RQ2 (Performance) What is the performance of running ? Is it acceptable?

The experimental objects and our results, along with the implementation of are available for download [24].

5.4.1 Experimental Objects

To evaluate CONFIX, we selected four open source web applications that have many DOM-dependent JavaScript functions. Table 5.4 shows these applications, which fall under different application domains and have different sizes. *ToDoList*

Table 5.4: Characteristics of experimental objects excluding blank/comment lines and external JavaScript libraries.

Name	JS LOC	# Branches	# Functions	% DOM-Dependent Functions	# DOM Constraints (DC)	% Non-Conditional DC	% Conditional DC
ToDoList	82	10	7	100	19	84	16
HotelReserve	106	88	9	56	13	69	31
Sudoku	399	344	18	78	66	67	33
Phormer	1553	464	109	71	194	70	30
Total	2140	906	143	72	292	70	30

[43] is a simple JavaScript-based todo list manager. *HotelReserve* [29] is a reservation management application. *Sudoku* [44] is a web-based implementation of the Sudoku game. And *Phormer* [39] is an Ajax photo gallery. As presented in Table 5.4, about 70% of the functions in these applications are DOM-dependent. The table also shows the lines of JavaScript code, and number of branches and functions in each application.

5.4.2 Experimental Setup

Our experiments are performed on Mac OS X, running on a 2.3GHz Intel Core i7 with 8 GB memory, and FireFox 37.

Independent variables

To the best of our knowledge, there exists no DOM test fixture generation tool to compare against; the closest to is JSeft [130], which generates tests that use the entire DOM at runtime as test fixtures. However, JSeft does not *generate* DOM fixtures, and it requires a deployed web application before it can be used.

Therefore, we construct baseline test suites to compare against. We compare different types of test suites to evaluate the effectiveness of test fixtures generated by ConFix. Table 5.5 depicts different JavaScript unit test suites. We classify test suites based on the type of test input they have support for, namely, (1) DOM

Table 5.5: Evaluated function-level test suites.

Test Suite	Function Arguments	DOM Fixture	DOM Input	# Test Cases
NoInput	No input	✗	✗	98
Jalangi	Generated by JALANGI	✗	✗	98+4
Manual	Manual inputs	✗	✗	98+55
ConFix + NoInput	No input	✓	✗	98+125
ConFix + Jalangi	Generated by JALANGI	✓	✗	98+129
ConFix + Manual	Manual inputs	✓	✓	98+236

fixtures, and/or (2) DOM function arguments.

Test suites without DOM fixtures. *NoInput* is a naive test suite that calls each function without setting any fixture or input for it. *Jalangi* produces (non-DOM) function arguments using the concolic execution engine of JALANGI [161]. *Manual* is a test suite that uses manually provided (non-DOM) inputs.

Test suites with DOM fixtures. To assess the effect of DOM fixtures and inputs generated by CONFIX, we consider different combinations: *ConFix + NoInput* has DOM fixtures generated by but no function arguments, *ConFix + Jalangi* has DOM fixtures generated by but uses the inputs generated by JALANGI for non-DOM function arguments, and *ConFix + Manual* uses DOM fixtures and DOM function arguments generated by and manual inputs for non-DOM function arguments. Table 5.5 shows all these combinations along with the number of test cases in each category.

Note that since our approach is geared toward generating DOM-based test fixtures/inputs, we only consider test generation for DOM-dependent functions and thus for all categories we consider the same set of 98 DOM-dependent functions under test, but with different inputs/fixtures. The 55 manual non-DOM function arguments were written by the authors through source code inspection.

Dependent variables

Our dependent variables are code coverage and generation time.

Code coverage. Code coverage is commonly used as a test suite adequacy criterion. To address RQ1, we compare the JavaScript code coverage of the different

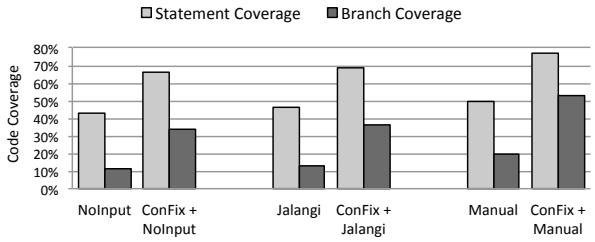


Figure 5.6: Comparison of statement and branch coverage, for DOM-dependent functions, using different test suite generation methods.

test suites, using JSCover [33]. Since our target functions in this work are DOM-dependent functions (Definition 11), code coverage is calculated by considering only DOM-dependent functions.

Fixture generation time. To answer RQ2 (performance), we measure the time (in seconds) required for to generate test fixtures for a test suite, divide it by the number of generated tests, and report this as the average fixture generation time.

5.4.3 Results

Coverage (RQ1). Figure 5.6 illustrates the comparison of code coverage achieved by each test suite category. We report the total statement and branch coverage of the JavaScript code obtained from the experimental objects.

Table 5.4 shows that in total about 14% of the code (i.e., 292 out of 2140 LOC) contains DOM constraints. However, as shown in Figure 5.6, this relatively small portion of the code has a remarkable impact on the code coverage when comparing test suites with and without DOM test fixtures. This is due to the fact that if a DOM constraint is not satisfied, the function terminates as a result of a null exception in most cases. Such constraints may exist at statements near the entrance of functions (as shown in Figure 5.1) and thus, proper DOM test fixtures are essential to achieve proper coverage. Table 5.6 shows the coverage increase for the test suites.

Our results, depicted in Figure 5.6 and Table 5.6, show that *Manual* and *Jalangi* cannot achieve a much higher code coverage than *NoInput*. This again relates to the fact that if expected DOM elements are not available, then the execution of DOM-dependent functions terminates and consequently code coverage cannot be

Table 5.6: Coverage increase (in percentage point) of test suites on rows over test suites on columns. Statement and branch coverage are separated by a slash, respectively.

	NoInput	*Jalangi*	*Manual*	*ConFix* + *NoInput*	*ConFix* + *Jalangi*
Jalangi	3 / 2	—	—	—	—
Manual	7 / 9	4 / 7	—	—	—
ConFix + NoInp	23 / 23	20 / 21	16 / 14	—	—
ConFix + Jalangi	26 / 25	23 / 23	19 / 16	3 / 2	—
ConFix + Manual	**34 / 42**	**31 / 40**	**27 / 33**	11 / 19	8 / 17

increased much, no matter the quality of the function arguments provided.

The considerable coverage increase for *ConFix + NoInput* vs. *Jalangi* and *Manual* indicates that test suites generated by CONFIX, even without providing any arguments, is superior over other test suites with respect to the achieved code coverage. The coverage increases even more when function arguments are provided; for example, for *ConFix + Manual* compared to *Jalangi*, there is a 40 percentage point increase (300% improvement) in the branch coverage, and a 31 percentage point increase (67% improvement) in the statement coverage.

The coverage increase for *ConFix + Manual* vs. *ConFix + NoInput* and *ConFix + Jalangi* is more substantial in comparison with the coverage increase for *Manual* vs. *NoInput* and *Jalangi*. This is mainly due to (1) the DOM fixtures generated, which are required to execute paths that depend on manually given arguments; and (2) the DOM arguments generated, which enable executing paths that depend on DOM elements provided as function arguments.

Although DOM fixtures generated by can substantially improve the coverage compared to the current state-of-the-art techniques, we discuss why does not achieve full coverage in Section 5.5.

Performance (RQ2). The execution time of is mainly affected by its concolic execution, which requires iterative code execution in the browser and collecting and solving constraints. Our results show that, on average, CONFIX requires 0.7 second per test case and 1.6 second per function to generate DOM fixtures. This amount of time is, however, negligible considering the significant code coverage increase. Since the number of DOM constraints in typical DOM-dependent JavaScript func-

tions is not large (2.9 on average in our study), concolic execution can be performed in a reasonable time.

5.5 Discussion

Applications. Given the fact that JavaScript extensively interacts with the DOM on the client-side, and these interactions are highly error-prone [141], we see many applications for our technique. CONFIX can be used to automatically generate JavaScript unit tests with DOM fixtures that could otherwise be quite time consuming to write manually. It can also be used in combination with other existing test generation techniques [130, 161] to improve the code coverage. In case a DOM constraint depends on a function argument, we can perform concolic execution i.e., beginning with an arbitrary value for the argument, capturing the DOM constraint during execution, and treating the DOM referring variable and the argument as symbolic variables. In addition to DOM fixtures, can also generate DOM-based function arguments, i.e., DOM elements as inputs, as explained in subsection 5.3.5. Currently there is no tool that supports DOM input generation for JavaScript functions.

Limitations. We investigated why does not achieve full coverage. The main reasons that we found reveal some of the current limitations of our implementation, which include: (1) we implemented a simple integer/string constraint solver to generate XPath expressions with proper structure and attribute values, which cannot handle complex constraints currently; (2) we do not support statements that require event-triggering; (3) the XML solver used in our work cannot efficiently solve long lists of constraints, (4) some paths are browser-dependent, which is out of the scope of CONFIX; (5) the execution of some paths are dependent on global variables that are set via other function calls during the execution, which is also out of the scope of ; and (6) does not analyze dynamically generated code using `eval()` that interacts with the DOM.

Threats to validity. A threat to the external validity of our experiment is with regard to the generalization of the results to other JavaScript applications. To mitigate this threat, we selected applications from different domains (task management, form validation, game, gallery) that exhibit variations in functionality, and

we believe they are representative of JavaScript applications that use DOM APIs for manipulating the page; although we do need more and large applications to make our results more generalizable. With respect to the reproducibility of our results, CONFIX, the test suites, and the experimental objects are all available [24], making the experiment repeatable.

5.6 Related Work

Most current web testing techniques focus on generating sequences of events at the DOM level, while we consider unit test generation at the JavaScript code level. Event-based test generation techniques [121, 123, 126] can miss JavaScript faults that do not propagate to the DOM [130].

Unit testing. Alshraideh [65] generates unit tests for JavaScript programs through mutation analysis by applying basic mutation operators. Heidegger et al. [95] propose a test case generator for JavaScript that uses contracts (i.e., type signature annotations that the tester has to include in the program manually) to generate inputs. ARTEMIS [66] is a framework for automated testing of JavaScript, which applies feedback-directed random test generation. None of these techniques consider DOM fixtures for JavaScript unit testing.

More related to our work, JSEFT [130] applies a heuristic-based approach by capturing the full DOM tree during the execution of an application just before executing the function under test, and uses that DOM as a test fixture in a generated test case. This approach, however, cannot cover all DOM dependent branches.

Symbolic and concolic execution. Nguyen et al. [138] present a technique that applies symbolic execution for reasoning about the potential execution of client-side code embedded in server-side code. KUDZU [158] performs symbolic reasoning to analyze JavaScript security vulnerabilities, such as code injections in web applications. JALANGI [161] is a dynamic analysis framework for JavaScript that applies concolic execution to generate function arguments; however, it does not support DOM-based arguments nor DOM fixtures, as CONFIX does. SYMJS [109] contains a symbolic execution engine for JavaScript, as well as an automatic event explorer. It extends HTMLUnit's DOM and browser API model to support symbolic execution by introducing symbolic values for specific elements, such as text

inputs and radio boxes. However, it considers substituting DOM element variables with integer or string values and using a traditional solver, rather than actually generating the hierarchical DOM structure. CONFIX on the other hand has support for the full DOM tree-structure including its elements and their hierarchical relations, attributes, and attribute values.

To the best of our knowledge, CONFIX is the first to address the problem of DOM test fixture construction for JavaScript unit testing. Unlike most other techniques, we consider JavaScript code in isolation from server-side code and without the need to execute the application as a whole.

5.7 Conclusions

Proper test fixtures are required to cover DOM-dependent statements and conditions in unit testing JavaScript code. However, generating such fixtures is not an easy task. In this chapter, we proposed a concolic technique and tool, called CONFIX, to automatically generate a set of unit tests with DOM fixtures and DOM function arguments. Our empirical results show that the generated fixtures substantially improve code coverage compared to test suites without these fixtures.

Chapter 6

Detecting JavaScript Code Smells

Summary[13]

JavaScript is a powerful and flexible prototype-based scripting language that is increasingly used by developers to create interactive web applications. The language is interpreted, dynamic, weakly-typed, and has first-class functions. In addition, it interacts with other web languages such as CSS and HTML at runtime. All these characteristics make JavaScript code particularly error-prone and challenging to write and maintain. Code smells are patterns in the source code that can adversely influence program comprehension and maintainability of the program in the long term. We propose a set of 13 JavaScript code smells, collected from various developer resources. We present a JavaScript code smell detection technique called JSNOSE. Our metric-based approach combines static and dynamic analysis to detect smells in client-side code. This automated technique can help developers to spot code that could benefit from refactoring. We evaluate the smell finding capabilities of our technique through an empirical study. By analyzing 11 web applications, we investigate which smells detected by JSNOSE are more prevalent.

6.1 Introduction

JavaScript is a flexible popular scripting language that is used to offload core functionality to the client-side web browser and mutate the DOM tree at runtime to

[13] An initial version of this chapter has been published in the IEEE International Conference on Source Code Analysis and Manipulation (SCAM), 2013 [124].

facilitate smooth state transitions. Because of its flexibility JavaScript is a particularly challenging language to write code in and maintain. The challenges are manifold: First, it is an interpreted language, meaning that there is typically no compiler in the development cycle that would help developers to spot erroneous or unoptimized code. Second, it has a dynamic, weakly-typed, asynchronous nature. Third, it supports intricate features such as prototypes [151], first-class functions, and closures [77]. And finally, it interacts with the DOM through a complex event-based mechanism [180].

All these characteristics make it difficult for web developers who lack in-depth knowledge of JavaScript, to write maintainable code. As a result, web applications written in JavaScript tend to contain many *code smells* [85]. Code smells are patterns in the source code that indicate potential comprehension and maintenance issues in the program. Code smells, once detected, need to be refactored to improve the design and quality of the code.

Detecting code smells manually is time consuming and error-prone. Automated smell detection tools can lower long-term development costs and increase the chances for success [178] by helping to make the code more maintainable.

Current work on web application code smell detection is scarce [137] and tools [9, 17, 27, 137] available to web developers to maintain their code are mainly static analyzers and thus limited in their capabilities.

In this work, we propose a list of code smells for JavaScript applications. In total, we consider 13 code smells: 7 are existing well-known smells adapted to JavaScript, and 6 are specific JavaScript code smell types, collected from various JavaScript development resources. We present an automated technique, called JSNOSE, which performs a metric-based static with dynamic analysis to detect these smells in JavaScript code.

Our work makes the following main contributions:

- We propose a list of JavaScript code smells, collected from various web development resources;
- We present an automated metric-based approach to detect JavaScript code smells;

- We implement our approach in a tool called JSNose, which is freely available;
- We evaluate the effectiveness of our technique in detecting code smells in JavaScript applications;
- We empirically investigate 11 web applications using JSNOSE to find out which smells are more prevalent.

Our results indicate that amongst the smells detected by JSNOSE, lazy object, long method/function, closure smells, coupling between JavaScript, HTML, and CSS, and excessive global variables, are the most prevalent code smells. Further, our study indicates that there exists a strong and significant positive correlation between the types of smells and lines of code, number of functions, number of JavaScript files, and cyclomatic complexity.

6.2 Motivation and Challenges

Although JavaScript is increasingly used to develop modern web applications, there is still lack of tool support targeting code quality and maintenance, in particular for automated code smell detection and refactoring.

JavaScript is a dynamic weakly typed and prototype-based scripting language, with first-class functions. Prototype-based programming is a class-free style of object-oriented programming, in which objects can inherit properties from other objects directly. In JavaScript, prototypes can be redefined at runtime, and immediately affect all the referring objects.

The detection process for many of the traditional code smells [85, 104] in object-oriented languages is dependent on identifying objects, classes, and functions in the code. Unlike most object-oriented languages such as Java or C++, identification of such key language items is not straightforward in JavaScript code. Bellow we explain the major challenges in identifying objects and functions in JavaScript.

JavaScript has a very flexible model of objects and functions. Object properties and their values can be created, changed, or deleted at runtime and accessed via first-class functions. For instance, in the following piece of code a call to the

function `foo()` will dynamically create a property *prop* for the object *obj* if *prop* does not already exist.

```
1 function foo(obj, prop, value){
2       obj.prop = value;
3 }
```

Due to such dynamism, the set of all available properties of an object is not easily retrievable through static analysis of the code alone. Empirical studies [153] reveal that most dynamic features in JavaScript are frequently used by developers and cannot be disregarded in code analysis processes.

Furthermore, functions in JavaScript are first-class values. They can (1) be objects themselves, (2) contain properties and nested function closures, (3) be assigned dynamically to other objects, (4) be stored in variables, objects, and arrays, (5) be passed as arguments to other functions, and (6) be returned from functions. JavaScript also allows the creation (through `eval()`) and execution of new code at runtime, which again makes static analysis techniques insufficient.

Manual analysis and detection of code smells in JavaScript is time consuming, tedious, and error-prone in large code bases. Therefore, automated techniques are needed to support web developers in maintaining their code. Given the challenging characteristics of JavaScript, our goal in this work is to propose a technique that can handle the highly dynamic nature of the language to detect potential code smells effectively.

6.3 Related Work

Fowler and Beck [85] proposed 22 code smells in object-oriented languages and associated each of them with a possible refactoring. Although code smells in object-oriented languages have been extensively studied in the past, current work on smell detection for JavaScript code is scarce [137]. In this work we study a list of code smells in JavaScript, and propose an automated detection technique. The list of proposed JavaScript code smells in this chapter is based on a study of various discussions in online development forums and JavaScript books [86, 136, 137, 145, 147].

Many tools and techniques have been proposed to detect code smells automat-

ically in Java and C++ such as Checkstyle [3], Decor [133], and JDeodorant [174]. A common heuristic-based approach to code smell detection is the use of code metrics and user defined thresholds [76, 108, 133, 135, 162]. Similarly we adopt a metric-based smell detection strategy. Our approach is different from such techniques in the sense that due to the dynamic nature of JavaScript, we propose a code smell technique that combines static with dynamic analysis.

Cilla [118] is a tool that similar to our work applies dynamic analysis but for detecting unused CSS code in relation to dynamically mutated DOM elements. A case study conducted with Cilla revealed that over 60% of CSS rules are unused in real-world deployed web applications, and eliminating them could vastly improve the size and maintainability of the code.

A more closely related tool is WebScent [137], which detects client-side smells that exist in embedded code within scattered server-side code. Such smells can not be easily detected until the client-side code is generated. After detecting smells in the generated client-side code, WebScent locates the smells in the corresponding location in the server-side code. WebScent primarily identifies mixing of HTML, CSS, and JavaScript, duplicate code in JavaScript, and HTML syntax errors. Our tool, JSNOSE, can similarly identify JavaScript code smells generated via server-side code. However, we propose and target a larger set of JavaScript code smells and unlike the manual navigation of the application in WebScent, we apply automated dynamic exploration using a crawler. Another advantage of JSNOSE is that it can infer dynamic creation/change of objects, properties, and functions at runtime, which WebScent does not support.

A number of industrial tools exist that aim at assisting web developers with maintaining their code. For instance, WARI [17] examines dependencies between JavaScript functions, CSS styles, HTML tags and images. The goal is to statically find unused images as well as unused and duplicated JavaScript functions and CSS styles. Because of the dynamic nature of JavaScript, WARI cannot guarantee the correctness of the results. JSLint [9] is a static code analysis tool written in JavaScript that validates JavaScript code against a set of good coding practices. The code inspection tends to focus on improving code quality from a technical perspective. The Google Closure Compiler [27] is a JavaScript optimizer that rewrites JavaScript code to make it faster and more compact. It helps to reduce the size of

JavaScript code by removing comments and unreachable code.

6.4 JavaScript Code Smells

In this section, we propose a list of code smells for JavaScript-based applications. Of course a list of code smells can never be complete as the domain and projects that the code is used in may vary. Moreover, code smells are generally subjective and imprecise, i.e., they are based on opinions and experiences [178]. To mitigate this subjective nature of code smells, we have collected these smells by studying various online development resources [10, 86, 136, 145, 147, 163] and books [77, 195, 196] that discuss bad JavaScript coding patterns.

In total, we consider 13 code smells in JavaScript. Although JavaScript has its own specific code smells, most of the generic code smells for object-oriented languages [85, 104] can be adapted to JavaScript as well. Since JavaScript is a class-free language and objects are defined directly, we use the notion of "object" instead of "class" for these generic smells. The generic smells include the following 7: *Empty catch blocks* (poor understanding of logic in the try), *Large object* (too many responsibilities), *Lazy object* (does too little), *Long functions* (inadequate decomposition), *Long parameter list* (need for an object), *Switch statements* (duplicated code and high complexity), and *Unused/dead code* (never executed or unreachable code).

In addition to the generic smells, we propose 6 types of JavaScript code smells in this section as follows.

6.4.1 Closure Smells

In JavaScript, it is possible to declare nested functions, called `closures`. Closures make it possible to emulate object-oriented notions such as `public`, `private`, and `privileged` members. Inner functions have access to the parameters and variables — except for `this` and `argument` variables — of the functions they are nested in, even after the outer function has returned [77]. We consider four smells related to the concept of function closures in JavaScript.

Long scope chaining. Functions can be multiply-nested, thus closures can have multiple scopes. This is called "scope chaining" [10], where inner functions have

access to the scope of the functions containing them. An example is the following code:

```
1 function foo(x) {
2     var tmp = 3;
3     function bar(y) {
4         ++tmp;
5         function baz(z) {
6           document.write(x + y + z + tmp);
7         }
8         baz(3);
9     }
10    bar(10);
11  }
12  foo(2);  // writes 19 i.e., 2+10+3+4
```

This nested function style of programming is useful to emulate privacy, however, using too many levels of nested closures over-complicates the code, making it hard to comprehend and maintain. Moreover, identifier resolution performance is directly related to the number of objects to search in the scope chain [195]. The farther up in the scope chain an identifier exists, the longer the search goes on and the longer time it takes to access that variable.

Closures in loops. Inner functions have access to the actual variables of their outer functions and not their copies. Therefore, creating functions within a loop can cause confusion and be wasteful computationally [77]. Consider the following example:

```
1 var addTheHandler = function (nodes) {
2     for (i = 0; i < nodes.length; i++) {
3         nodes[i].onclick = function (e) {
4             document.write(i);
5         };
6     }
7 };
8 addTheHandler(document.getElementsByTagName("div"));
```

Assume that there are three `div` DOM elements present. If the developer's actual intention is to display the ordinal of the `div` nodes when a node is clicked, then the result will not be what she expected as the length of nodes, 3, will be returned instead of the node's ordinal position. In this example, the value of `i` in the `document.write` function is assigned when the `for` loop is finished and

the inner anonymous function is created. Therefore, the variable `i` has the value of `nodes.length`, which is 3 in this case. To avoid this confusion and potential mistake, the developer can use a helper function outside of the loop that will deliver a function binding to the current local value of `i`.

Variable name conflict in closures. When two variables in the scopes of a closure have the same name, there is a name conflict. In case of such conflicts, the inner scope takes precedence. Consider the following example [10]:

```
1 function outside() {
2      var a = 10;
3      function inside(a) {
4         return a;
5      }
6      return inside;
7   }
8   result = outside()(20);  \\ result: 20
```

In this example, there is a name conflict between the variable `a` in `outside()` and the function parameter `a` in `inside()` that takes precedence. We consider this a code smell as it makes it difficult to comprehend the actual intended value assignment. Due to scope chain precedence, the value of `result` is now 20. However, it is not evident from the code whether this is the intended result (or perhaps 10).

Moreover, dynamic typing in JavaScript makes it possible to reuse the *same* variable for *different* types at runtime. Similar to the variable name conflict issue, this style of programming reduces readability and in turn maintainability. Thus, declaring a new variable with a dedicated unique name is the recommended refactoring. This issue is not restricted to closures, or to nested functions.

Accessing the `this` reference in closures. Due to the design of JavaScript language, when an inner function in a closure is invoked, `this` becomes bounded to the global object and not to the `this` variable of the outer function [77]. We consider the usage of `this` in closures a code smell as it is a potential symptom for mistakes. As a refactoring workaround, the developer can assign the value of the `this` variable of the outer function to a new variable `that` and then use `that` in the inner function [77].

6.4.2 Coupling between JavaScript, HTML, and CSS

In web applications, HTML is meant for presenting content and structure, CSS for styling, and JavaScript for functional behaviour. Keeping these three entities separate is a well-known programming practice, known as *separation of concerns*. Unfortunately, web developers often mix JavaScript code with markup and styling code [137], which adversely influences program comprehension, maintenance and debugging efforts in web applications. We categorize the tight coupling of JavaScript with HTML and CSS code into the following three code smells types:

JavaScript in HTML. One common way to register an event listener in web applications is via inline assignment in the HTML code. We consider this inline assignment of event handlers a code smell as it tightly couples the HTML code to the JavaScript code. An example of such a coupling is shown below:

```
1 <button onclick="foo();" id="myBtn"/>
```

This smell can be refactored by removing the `onclick` attribute from the button in HTML and using the `addEventListener` function of DOM Level 2 [180] to assign the event handler through JavaScript:

```
1 <button id="myBtn"/>

3 function foo() {
4       // code
5 }
6 var btn = document.getElementById("myBtn");
7 btn.addEventListener("click", foo, false);
```

This could be further refactored using the jQuery library as `$("#myBtn").on("click", foo);`.

Note that JavaScript code within the `<script>` tag in HTML code can be seen as a code smell [137]. We do not consider this a code smell as it does not affect comprehension nor maintainability, although separating the code to a JavaScript file is preferable.

HTML in JavaScript. Extensive DOM API calls and embedded HTML strings in JavaScript complicate debugging and software evolution. In addition, editing markup is believed to be less error prone than editing JavaScript code [196]. The following code is an example of embedded HTML in JavaScript [86]:

```
1 // add book to the list
2 var book = doc.createElement("li");
3 var title = doc.createElement("strong");
4 titletext = doc.createTextNode(name);
5 title.appendChild(titletext);
6 var cover = doc.createElement("img");
7 cover.src = url;
8 book.appendChild(cover);
9 book.appendChild(title);
10 bookList.appendChild(book);
```

To refactor this code smell, we can move the HTML code to a template (`book_tpl.html`):

```
1 <li><strong>TITLE</strong><img src="COVER"/></li>
```

The JavaScript code would then be refactored as:

```
1 var tpl = loadTemplate("book_tpl.html");
2 var book = tpl.substitute({TITLE: name,  COVER: url});
3 bookList.appendChild(book);
```

Another example this smell is using long strings of HTML in jQuery function calls [147]:

```
1 $('\#news')
2     .append('<div class="gall"><a href="javascript:void(0)">Linky</a></↩
          div>')
3     .append('<button onclick="app.doStuff()">Button</button>');
```

CSS in JavaScript. Setting the presentation style of DOM elements by assigning their `style` properties in JavaScript is a code smell [86]. Keeping styling code inside JavaScript is asking for maintenance problems. Consider the following example:

```
1 div.onclick = function(e) {
2     var clicked = this;
3     clicked.style.border = "1px solid blue";
4 }
```

The best way to change the style of an element in JavaScript is by manipulating CSS classes properly defined in CSS files [86, 196]. The above code smell can be refactored as follows:

```
\\ CSS file:
.selected{border: 1px solid blue;}
\\ JavaScript:
div.onclick = function(e) {
  this.setAttribute("class","selected");
}
```

6.4.3 Excessive Global Variables

Global variables are accessible from anywhere in JavaScript code, even when defined in different files loaded on the same page. As such, naming conflicts between global variables in different JavaScript source files is common, which affects program dependability and correctness. The higher the number of global variables in the code, the more dependent existing modules are likely to be; and dependency increases error-proneness, and maintainability efforts [145]. Therefore, we see the excessive use of global variables as a code smell in JavaScript. One way to mitigate this issue is to create a single global object for the whole application that contains all the global variables as its properties [77]. Grouping related global variables into objects is another remedy.

6.4.4 Long Message Chain

Long chaining of functions with the dot operator can result in complex control flows that are hard to comprehend. This style of programming happens frequently when using the jQuery library. One extreme example is shown bellow [147]:

```
$('a').addClass('reg-link').find('span').addClass('inner').end().find('↩
    div').mouseenter(mouseEnterHandler).mouseleave(mouseLeaveHandler).↩
    end().explode();
```

Long chains are unreadable specially when a large amount of DOM traversing is taking place [147]. Another instance of this code smell is too much cascading. Similar to object-oriented languages such as Java, in JavaScript many methods calls can be cascaded on the same object sequentially within a single statement. This is possible when the methods return the `this` object. Cascading can help to produce expressive interfaces that perform much work at once. However, the code

written this way tends to be harder to follow and maintain. The following example is borrowed from [77]:

```
1 getElement('myBoxDiv').move(350, 150).width(100).height(100).color('red↩
      ').border('10px outset').padding('4px').appendText("Please stand ↩
      by").on('mousedown', function (m) {
2   this.startDrag(m, this.getNinth(m));}).on('mousemove', 'drag').on('↩
        mouseup', 'stopDrag').tip("This box is resizeable");
```

A possible refactoring to shorten the message chain is to break the chain into more general methods/properties for that object which incorporate longer chains.

6.4.5 Nested Callback

A callback is a function passed as an argument to another (parent) function. Callbacks are executed after the parent function has completed its execution. Callback functions are typically used in asynchronous calls such as timeouts and XMLHttpRequests (XHRs). Using excessive callbacks, however, can result in hard to read and maintain code due to their nested anonymous (and usually asynchronous) nature. An example of a nested callback is given below [163]:

```
1 setTimeout(function () {
2     xhr("/greeting/", function (greeting) {
3         xhr("/who/?greeting=" + greeting, function (who) {
4             document.write(greeting + " " + who);
5         });
6     });
7 }, 1000);
```

A possible refactoring to nested callbacks is to split the functions and pass a reference to another function [19]. The above code can be rewritten as below:

```
1 setTimeout(foo,1000);
2 function foo() {
3   xhr("/greeting/", bar);
4 }
5 function bar(greeting) {
6   xhr("/who/?greeting=" + greeting, baz);
7 }
8 function baz(who) {
9   document.write(greeting + " " + who);
10 }
```

6.4.6 Refused Bequest

JavaScript is a class-free prototypal inheritance language, i.e., an object can inherit properties from another object, called a prototype object. A JavaScript object that does not use/override many of the properties it inherits from its prototype object is an instance of a refused bequest [85] soft code smell. In the following example, the `student` object inherits from its prototype parent `person`. However, `student` only uses one of the five properties inherited from `person`, namely `fname`.

```
1 var person={fname:"John", lname:"Smith", gender:"male", age:28, ↩
      location:"Vancouver"};
2 var student = Object.create(person);
3 ...
4 student.university = "UBC";
5 document.write(student.fname + " studies at " + student.university);
```

A simple refactoring, similar to the push down field/method proposed by Fowler [85], could be to eliminate the inheritance altogether and add the required property (`fname`) of the prototype to the object that refused the bequest.

6.5 Smell Detection Mechanism

In this section, we present our JavaScript code smell detection mechanism, which is capable of detecting the code smells discussed in the previous section.

A common heuristic-based approach to detect code smells is the use of source code metrics and thresholds [76, 108, 133, 135, 162]. In this work, we adopt a similar metric-based approach to identify smelly sections of JavaScript code.

In order to calculate the metrics, we first need to extract objects, functions, and their relationships from the source code. Due to the dynamic nature of JavaScript, static code analysis alone will not suffice, as discussed in Section 6.2. Therefore, in addition to static code analysis, we also employ dynamic analysis to monitor and infer information about objects and their relations at runtime.

Figure 6.1 depicts an overview of our approach. At a high level, (1) the configuration, containing the defined metrics and thresholds, is fed into the code smell detector. We automatically (2) intercept the JavaScript code of a given web application, by setting up a proxy between the server and the browser, (3) extract JavaScript code from all .js and HTML files, (4) parse the source code into an

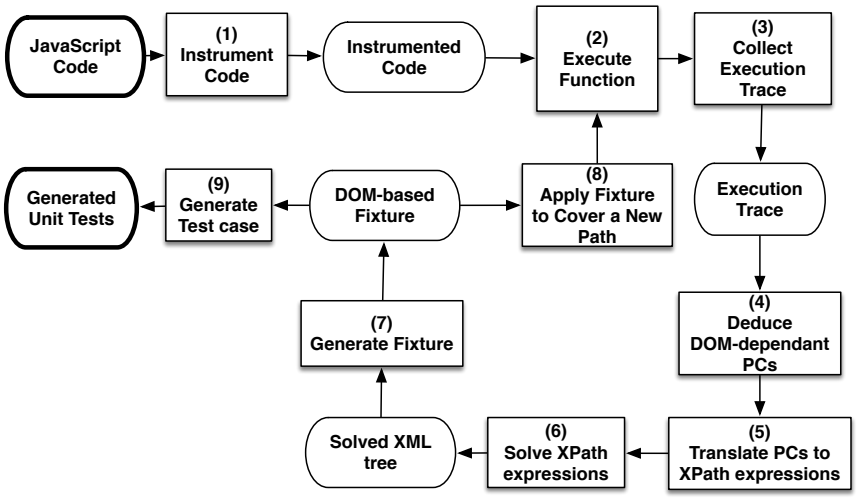


Figure 6.1: Processing view of JSNOSE, our JavaScript code smell detector.

Abstract Syntax Tree (AST) and analyze it by traversing the tree. During the AST traversal, the analyzer visits all program entities, objects, properties, functions, and code blocks, and stores their structure and relations. At the same time, we (2) instrument the code to monitor statement coverage, which is used for unused/dead code smell detection. Next, we (7) navigate the instrumented application in the browser to produce an execution trace, through an automated dynamic crawler, and (8) collect and use execution traces to calculate code coverage. We (5) extract patterns from the AST such as names of objects and functions, and (6) infer JavaScript objects, their types, and properties dynamically by querying the browser at runtime. Finally, (9) based on all the static and dynamic data collected, we detect code smells (10) using the metrics.

Table 6.1: Metric-based criteria for JavaScript code smell detection.

Code smell	Level	Detection method	Detection criteria	Metric
Closure smell	Function	Static & Dynamic	$LSC > 3$	LSC: Length of scope chain
Coupling JS/HTML/CSS	File	Static & Dynamic	$JSC > 1$	JSC: JavaScript coupling instance
Empty catch	Code block	Static	$LOC(catchBlock) = 0$	LOC: Lines of code
Excessive global variables	Code block	Static & Dynamic	$GLB > 10$	GLB: Number of global variables
Large object	Object	Static & Dynamic	[185]: $LOC(obj) > 750$ or $NOP > 20$	NOP: Number of properties
Lazy object	Object	Static & Dynamic	$NOP < 3$	NOP: Number of properties
Long message chain	Code block	Static	$LMC > 3$	LMC: Length of message chain
Long method/function	Function	Static & Dynamic	[108, 185]: $MLOC > 50$	MLOC: Method lines of code
Long parameter list	Function	Static & Dynamic	[185]: $PAR > 5$	PAR: Number of parameters
Nested callback	Function	Static & Dynamic	$CBD > 3$	CBD: Callback depth
Refused bequest	Object	Static & Dynamic	[108]: $BUR < \frac{1}{3}$ and $NOP > 2$	BUR: Base-object usage ratio
Switch statement	Code block	Static	$NOC > 3$	NOC: Number of cases
Unused/dead code	Code block	Static & Dynamic	$EXEC = 0$ or $RCH = 0$	EXEC: Execution count RCH: Reachability of code

6.5.1 Metrics and Criteria Used for Smell Detection

Table 6.1 presents the metrics and criteria we use in our approach to detect code smells in JavaScript applications. Some of these metrics and their corresponding thresholds have been proposed and used for detecting code smells in object-oriented languages [76, 108, 133, 135, 162]. In addition, we propose new metrics and criteria to capture the characteristics of JavaScript code smells discussed in Section 6.4.

Closure smell. We identify long scope chaining and accessing `this` in closures. If the length of scope chain (LSC) is greater than 3, or if `this` is used in an inner function closure, we report it as a closure smell instance.

Coupling JS/HTML/CSS. We count the number of occurrences of JavaScript within HTML tags, and CSS in JavaScript as described in Section 6.4.2. Our tool reports all such JavaScript coupling instances as code smell.

Empty catch. Detecting empty catches is straightforward in that the number of lines of code (LOC) in the catch block should be zero.

Excessive global variables. We extract global variables in JavaScript, which can be defined in three ways: (1) using a `var` statement outside of any function, such as `var x = value;`, (2) adding a property to the `window` global object, i.e., the container of all global variables, such as `window.foo = value;`, and (3) using a variable without declaring it by `var`. If the number of global variables (GLB) exceeds 10, we consider it as a code smell.

Large/Lazy object. An object that is doing too much or not doing enough work should be refactored. Large objects may be restructured or broken into smaller objects, and lazy objects maybe collapsed or combined into other classes. If an object's lines of code is greater than 750 or the number of its methods is greater than 20, it is identified as a large object [185]. We consider an object lazy, if the number of its properties (NOP) is less than 3.

Long message chain. If the length of a message chain (LMC), i.e., the number of items being chained by dots as explained in Section 6.4.4, in a statement is greater than 3, we consider it a long message chain and report it as a smell.

Long method/function. A method with more than 50 lines of code (MLOC) is identified as a long method smell [108, 185].

Long parameter list. We consider a parameter list long when the number of parameters (PAR) exceeds 5 [185].

Nested callback. We identify nested functions that pass a function type as an argument. If the callback depth (CBD) exceeds 3, we report it as a smell.

Refused bequest. If an object uses or specializes less than a third of its parent prototype, i.e., base-object usage ratio (BUR) is less than $\frac{1}{3}$, it is considered as refused parent bequest [108]. Further, the number of methods and the cyclomatic complexity of the child object should be above average since simple and small objects may unintentionally refuse a bequest. In our work, we slightly change this criteria to the constraint of NOP$>$2, i.e., not to be a lazy small object.

Switch statement. The problem with switch statements is duplicated code. Typically, similar switch statements are scattered throughout a program. If one adds or removes a clause in one switch, often has to find and repair the others too [85, 104]. When the number of switch cases (NOC) is more than three, it is considered as a code smell. This can also be applied to `if-then-else` statements with more than three branches.

Unused/dead code. Unused/dead code has negative effects on maintainability as it makes the code unnecessarily more difficult to understand [112, 145]. Unlike languages such as Java, due to the dynamic nature of JavaScript it is quite challenging to reason about dead JavaScript code statically. Hence, if the execution count (EXEC) of an statement remains 0 after executing the web application, we report it as a candidate unused/dead code. Reachability of code (RCH) is another metric we use to identify unreachable code.

6.5.2 Combining Static and Dynamic Analysis

Algorithm 5 presents our smell detection method. The algorithm is generic in the sense that the metric-based static and dynamic smell detection procedures can be defined and used according to any smell detection criteria. Given a JavaScript application A, a maximum crawling time t, and a set of code smell criteria τ, the algorithm generates a set of code smells CS.

The algorithm starts by looking for inline JavaScript code embedded in HTML (line 3). All JavaScript code is then extracted from JavaScript files and HTML

<script> tags (line 4). An AST of the extracted code is then generated using a parser (line 5). This AST is traversed recursively (lines 6, 19-21) to detect code smells using a static analyzer. Next the AST is instrumented (line 7) and transformed back to the corresponding JavaScript source code and passed to the browser (lines 8). The crawler then navigates (line 9-16) the application, and potential code smells are explored dynamically (line 9 14). After the termination of the exploration process, unused code is identified based on the execution trace and added to the list of code smells (line 17), and the resulting list of smells is returned (line 18).

Next, we present the relevant static and dynamic smell detection processes in detail.

Static Analysis. The static code analysis (Line 19) involves analyzing the AST by traversing the tree. During this step, we extract CSS style usage, objects, properties, inheritance relations, functions, and code blocks to calculate the smell metrics. If the calculated metrics violate the given criteria (τ), the smell is returned.

There are different ways to create objects in JavaScript. In this work, we only consider two main standard forms of using object literals, namely, through (1) the `new` keyword, and (2) `Object.create()`. To detect the prototype of an object, we consider both the non-standard form of using the `__proto__` property assignment, and the more general constructor functions through `Object.create()`.

In order to detect unreachable code, we search the AST nodes for `return`, `break`, `continue`, and `throw` statements. Whatever a statement is found right after these statements that is on the same node level in the AST, we mark it as potential unreachable code.

Dynamic Analysis. Dynamic analysis (Line 14) is performed for two reasons:

1. To calculate many of the metrics in Table 6.1, we need to monitor the creation/update of functions, objects, and their properties at runtime. To that end, a combination of static and dynamic analysis should be applied. The dynamic analysis is performed by executing a piece of JavaScript code in the browser, which enables retrieving a list of all global variables, objects, and functions (own properties of the `window` object) and dynamically de-

Algorithm 5: JavaScript Code Smell Detection

```
input : A JavaScript application A, the maximum exploration time t, the set of smell
        metric criteria τ
output: The list of JavaScript code smells CS
1  CS ← ∅
   Procedure EXPLORE() begin
2      while TIMELEFT(t) do
3          CS ← CS ∪ DETECTINLINEJSINHTML(τ)
4          code ← EXTRACTJAVASCRIPT(A)
5          AST ← PARSTOAST(code)
6          VISITNODE(AST.root)
7          ASTinst ← INSTRUMENT(AST)
8          INJECTJAVASCRIPTCODE(A,ASTinst)
9          C ← EXTRACTCLICKABLES(A)
10         for c ∈ C do
11             dom ← browser.GETDOM()
12             robot.FIREEVENT(c)
13             new_dom ← browser.GETDOM()
14             CS ← CS ∪ DETECTDYNAMICALLY(τ)
15             if dom.HASCHANGED(new_dom) then
16                 EXPLORE(A)
               end
           end
       end
17     CS ← CS ∪ DETECTUNUSEDCODE()
18     return CS
   end
   Procedure VISITNODE(ASTNode) begin
19     CS ← CS ∪ DETECTSTATICALLY(node, τ)
20     for node ∈ ASTNode.getChildren() do
21         VISITNODE(node)
       end
   end
```

tecting prototypes of objects (using `getPrototypeOf()` on each object). However, local objects in functions are not accessible via JavaScript code execution in the global scope. Therefore, we use static analysis and extract the required information from the parsed AST. The objects, functions, and properties information gathered this way is then fed to the smell detector process.

2. To detect unused/dead code we need to collect execution traces for measuring code coverage. Therefore, we instrument the code and record which parts

of it are invoked by exploring the application through automated crawling. However, this dynamic analysis can give false positives for non-executed, but reachable code. This is a limitation of any dynamic analysis approach since there is no guarantee of completeness (such as code coverage).

Note that our approach merely reports candidate code smells and the decision will always be upon developers whether or not to refactor the code smells.

6.5.3 Implementation

We have implemented our approach in a tool called JSNOSE, which is publicly available [34]. JSNOSE operates automatically, does not modify the web browser, is independent of the server technology, and requires no extra effort from the user. We use the WebScarab proxy to intercept the JavaScript/HTML code. To parse the JavaScript code to an AST and instrument the code, we use Mozilla Rhino [41]. To automatically explore and dynamically crawl the web application, we use CRAWLJAX [120]. The output of JSNOSE is a text file that lists all detected JavaScript code smells with their corresponding line numbers in a JavaScript file or an HTML page.

6.6 Empirical Evaluation

We have conducted an empirical study to evaluate the effectiveness and real-world relevance of JSNOSE. Our study is designed to address the following research questions:

RQ1: How effective is JSNOSE in detecting JavaScript code smells?

RQ2: Which code smells are more prevalent in web applications?

RQ3: Is there a correlation between JavaScript code smells and source code metrics?

Our experimental data along with the implementation of JSNOSE are available for download [34].

Table 6.2: Experimental JavaScript-based objects.

ID	Name	#JS files	JS LOC	#Functions	Average CC	Average MI	Description
1	PeriodicTable [13]	1	71	9	12	116	A periodic table of the elements
2	CollegeVis [4]	1	177	30	11	119	A visualization tool
3	ChessGame [20]	2	198	15	102	105	A simple chess game
4	Symbolistic [15]	1	203	20	28	109	A simple game
5	Tunnel [16]	0	234	32	29	116	A simple game
6	GhostBusters [7]	0	278	26	45	97	A simple game
7	TuduList [51]	4	782	89	106	94	A task manager (J2EE and MySQL)
8	FractalViewer [26]	8	1245	125	35	116	A fractal zoomer
9	PhotoGallery [39]	5	1535	102	53	102	A photo gallery (PHP without MySQL)
10	TinySiteCMS [49]	13	2496	462	54	115	A CMS (PHP without MySQL)
11	TinyMCE [50]	174	26908	4455	67	101	A WYSIWYG editor

6.6.1 Experimental Objects

We selected 11 web applications that make extensive use of client-side JavaScript, and fall under different application domains. The experimental objects along with their source code metrics are shown in Table 6.2. In the calculation of these source code metrics, we included inline HTML JavaScript code, and excluded blank lines, comments, and common JavaScript libraries such as jQuery, DWR, Scriptaculous, Prototype, and google-analytics. Note that we also exclude these libraries in the instrumentation step. We use CLOC [22] to count the JavaScript lines of code (JS LOC). Number of functions (including anonymous functions), ad cyclomatic complexity (CC) are all calculated using complexityReport.js [23]. The reported CC is across all JavaScript functions in each application.

6.6.2 Experimental Setup

We confine the dynamic crawling time for each application to 10 minutes, which is acceptable in a maintenance environment. Of course, the more time we designate for exploring the application, the higher statement coverage we may get and thus more accurate the detection of unused/dead code. For the crawling configuration, we set no limits on the crawling depth nor the maximum number of DOM states to be discovered. The criteria for code smell metrics are configured according to

those presented in Table 6.1.

To evaluate the effectiveness of JSNOSE (RQ1), we validate the produced results by JSNOSE against manual code inspection. Similar to [133], we measure precision and recall as follows:

Precision is the rate of true smells identified among the detected smells: $\frac{TP}{TP+FP}$

Recall is the rate of true smells identified among the existing smells: $\frac{TP}{TP+FN}$

where TP (true positives), FP (false positives), and FN (false negatives) respectively represent the number of correctly detected smells, falsely detected smells, and missed smells. To count TP, FP, and FN in a timely fashion while preserving accuracy, we only consider the first 9 applications since the last 2 applications have relatively larger code bases. In our manual validation process, we also consider runtime created/modified objects and functions that are inferred during JSNOSE dynamic analysis. It is worth mentioning that this manual process is a labour intensive task, which took approximately 6.5 hours for the 9 applications.

Note that the precision-recall values for detecting unused/dead code smell is calculated considering only "unreachable" code, which is code after an unconditional `return` statement. This is due to the fact that the accuracy of dead code detection depends on the running time and dynamic exploration strategy.

To measure the prevalence of JavaScript code smells (RQ2), we ran JSNOSE on all the 11 web applications and counted each smell instance.

To evaluate the correlation between the number of smells and application source code metrics (RQ3), we use R[14] to calculate the non-parametric Spearman correlation coefficients as well as the *p*-values. The Spearman correlation coefficient does not require the data to be normally distributed [103].

[14] http://www.r-project.org

Table 6.3: Precision-recall analysis (based on the first 9 applications), and detected code smell statistics (for all 11 applications).

	S1. Closure smells	**S2. Coupling JS/HTML/CSS**	**S3. Empty catch**	**Number of global variables**	**S4. Excessive global variables**	**S5. Large object**	**S6. Lazy object**	**S7. Long message chain**	**S8. Long method/function**	**S9. Long parameter list**	**S10. Nested callback**	**S11. Refused bequest**	**S12. Switch statement**	**S13. Unreachable code**	**S13. Unused/dead code**	**Number of smell instances**	**Number of types of smells**
TP_{total}	19	171	16	200	-	14	391	87	25	12	1	13	10	0	n/a	959	n/a
FP_{total}	0	0	0	0	-	4	73	0	0	0	0	6	0	0	n/a	83	n/a
FN_{total}	6	0	0	0	-	1	2	8	0	0	0	0	0	0	n/a	17	n/a
$Precision_{total}$	100%	100%	100%	100%	-	78%	85%	100%	100%	100%	100%	68%	100%	n/a	n/a	92%	n/a
$Recall_{total}$	76%	100%	100%	100%	-	94%	99%	92%	100%	100%	100%	100%	100%	n/a	n/a	98%	n/a
PeriodicTable	1	2	0	6	-	4	10	0	0	0	0	0	0	0	28%	23	4
CollegeVis	1	0	0	17	+	0	32	0	2	0	1	0	0	0	22%	53	5
ChessGame	0	7	0	**39**	+	3	9	4	0	2	0	0	0	0	36%	64	6
Symbolistic	0	0	0	4	-	0	17	0	1	0	0	1	0	0	20%	23	3
Tunnel	**9**	0	0	15	+	0	28	0	2	0	0	0	0	0	44%	54	4
GhostBusters	2	0	0	4	-	0	38	0	2	3	0	0	0	0	45%	49	4
TuduList	**6**	**47**	0	**45**	+	**6**	**138**	**78**	**12**	2	0	2	**7**	0	**65%**	343	10
FractalViewer	0	16	0	**40**	+	**7**	**117**	4	5	5	0	**16**	2	0	36%	212	9
PhotoGallery	0	**99**	**16**	30	+	0	73	1	1	0	0	0	1	0	**64%**	221	7
TinySiteCMS	2	7	0	**82**	+	3	13	4	3	**58**	0	0	0	0	22%	172	8
TinyMCE	3	3	1	4	-	5	23	4	0	2	1	3	3	0	**63%**	52	10
Average	2.2	16.5	1.5	26	+	2.5	45.2	8.6	2.6	6.5	0.2	2	1.2	0	40%	115	6.4
#Smelly apps	7	7	2	n/a	7	6	11	6	8	6	2	4	4	0	n/a	n/a	n/a
%Smelly apps	**64%**	**64%**	18%	n/a	**64%**	55%	**100%**	55%	**73%**	55%	18%	36%	36%	0%	n/a	n/a	n/a

6.6.3 Results

Effectiveness (RQ1). We report the precision and recall in the first 5 rows of Table 6.3. The reported TP_{total}, FP_{total}, and FN_{total}, are the sum of TP, FP, and FN values for the first 9 applications. Our results show that JSNOSE has an overall precision of 93% and an average recall of 98% in detecting the JavaScript code smells, which points to its effectiveness.

We observed that most false positives detected are related to large/lazy objects and refused bequest, which are primitive variables, object properties, and methods in jQuery. This is due to the diverse coding styles and different techniques in object manipulations in JavaScript, such as creating and initializing arrays of objects. There were a few false negatives in closure smells and long message chain, which are due to the permissive nature of jQuery syntax, complex chains of methods, array elements, as well as jQuery objects created via `$()` function.

Code smell prevalence (RQ2). Table 6.3 shows the frequency of code smells in each of the experimental objects. The results show that among the JavaScript code smells detected by JSNOSE, lazy object, long method/function, closure smells, coupling JS/HTML/CSS, and excessive global variables, are the most prevalent smells (appeared in 100%-64% of the experimental objects).

Tunnel and TuduList use many instances of `this` in closures. Major coupling smells in TuduList and PhotoGallary are with the use of CSS in JavaScript. Refused bequest are most observed in FractalViewer in objects inheriting from geometry objects. The high percentage of unused/dead code reported for TuduList, PhotoGallary, and TinyMCE is in fact not due to dead code per se, but is mainly related to the existence of admin pages and parts of the code which require precise data inputs that were not provided during the crawling process. On the other hand, TinyMCE has a huge number of possible actions and features that could not be exercised in the designated time of 10 minutes.

Correlations (RQ3). Table 6.4 shows the Spearman correlation coefficients between the source code metrics and the total number of smell instances/types. The results show that there exists a strong and significant positive correlation between the *types of smells* and LOC, number of functions, number of JavaScript files, and cyclomatic complexity. A weak correlation is also observed between the *number*

Table 6.4: Spearman correlation coefficients between number of code smells and code quality metrics.

Metric	Total number of smell instances	Total number of types of smells
Lines of code	$(r = 0.53, p = 0.05)$	$(r = 0.70, p = 0.01)$
# Functions	$(r = 0.57, p = 0.03)$	$(r = 0.76, p = 0.00)$
# JavaScript files	$(r = 0.53, p = 0.05)$	$(r = 0.85, p = 0.00)$
Cyclomatic complexity	$(r = 0.63, p = 0.02)$	$(r = 0.70, p = 0.01)$

of smell instances and the aforementioned source code metrics.

6.6.4 Discussion

Here, we discuss some of the limitations and threats to validity of our results.

Implementation Limitations. The current implementation of JSNOSE is not able to detect all various ways of object creation in JavaScript. Also it does not deal with various syntax styles of frameworks such as jQuery. For the dynamic analysis part, JSNOSE is dependent on the crawling strategy and execution time, which may affect the accuracy if certain JavaScript files are never loaded in the browser during the execution since the state space of web applications is typically huge. Since JSNOSE is using Rhino to parse the JavaScript code and generate the AST, if there exists a syntax error in a JavaScript file, the code in that file will not be parsed to an AST and thus any potential code smells within that file will be missed. Note that these are all implementation issues and not related to the design of our approach.

Threats to Validity. A threat to the external validity of our evaluation is with regard to the generalization of the results to other web applications. We acknowledge that more web applications should be evaluated to support the conclusions. To mitigate this threat we selected our experimental objects from different application domains, which exhibit variations in design, size, and functionality.

One threat to the internal validity of our study is related to the metrics and criteria we proposed in Table 6.1. However, we believe these metrics can effectively identify code smells described in Section 6.4. The designated 10 minutes time for crawling could also be increased to get more accurate results, however, we believe that in most maintenance environments this is acceptable considering frequent code releases. The validation and accuracy analysis performed by manual

inspection can be incomplete and inaccurate. We mitigated this threat by focusing on the applications with smaller sets of code smells so that manual comparison could be conducted accurately.

With respect to reliability of our evaluation, JSNOSE and all the web-based systems are publicly available, making the results reproducible.

6.7 Conclusions

In this work, we discussed a set of 13 JavaScript code smells and presented a metric-based smell detection technique, which combines static and dynamic analysis of the client-side code. Our approach, implemented in a tool called JSNOSE, can be used by web developers during development and maintenance cycles to spot potential code smells in JavaScript-based applications. The detected smells can be refactored to improve their code quality.

Our empirical evaluation shows that JSNOSE is effective in detecting JavaScript code smells; Our results indicate that lazy object, long method/function, closure smells, coupling between JavaScript, HTML, and CSS, and excessive global variables are the most prevalent code smells. Further, our study indicates that there exists a strong and significant positive correlation between the types of smells and LOC, cyclomatic complexity, and the number of functions and JavaScript files.

Chapter 7

Conclusions

Web applications are often written in multiple languages such as JavaScript, HTML, and CSS. JavaScript is extensively used to build responsive modern web applications. The event-driven and dynamic nature of JavaScript, and its interaction with the DOM, make it challenging to understand and test effectively. The work presented in this dissertation has focused on proposing new techniques to improve the quality of web and in particular JavaScript-based applications through automated testing and maintenance.

7.1 Revisiting Research Questions and Future Directions

In the beginning of this thesis, we designed five research questions towards the goal of improving the quality of web applications. We believe that we have addressed the research questions with our contributions.

RQ1.4.1. *How can we effectively derive test models for web applications?*

Chapter 2. We proposed four aspects of a test model that can be derived by crawling, including *functionality coverage*, *navigational coverage*, *page structural coverage*, and *size of the test model*. We also presented a feedback-directed exploration approach, called FEEDEX, to guide the exploration towards achieving higher functionality, navigational, and page structural coverage while reducing the test model size [123]. Results of our experiments show that FEEDEX is capable of generating test models that are enhanced in all aspects compared to traditional exhaustive methods (DFS, BFS, and random).

Future directions. There are number of possible future work based on the pre-

sented technique in Chapter 2. In this work we only measured the client-side coverage. It would be interesting to include the server-side code coverage in the feedback loop as well to guide the exploration. Also our state expansion method is currently based on a memory-less greedy algorithm, which could leverage machine learning techniques to improve its effectiveness. We do not claim that the proposed state expansion heuristic and the scoring function is the best solution. Other heuristics, such as selecting states with the least coverage improvement in state expansion process, and other combinations for scoring function can be evaluated.

RQ1.4.2. *Can we utilize the knowledge in existing UI tests to generate new tests?*

Chapter 3. We proposed a technique [126] and a tool, called TESTILIZER [47], to generate DOM-based UI test cases using existing tests. This work is motivated by the fact that a human-written test suite is a valuable source of domain knowledge, which can be used to tackle some of the challenges in automated web application test generation. TESTILIZER mines the existing UI test suite (such as SELENIUM) to infer a model of the covered DOM states and event-based transitions including input values and assertions. It then expands the inferred model by exploring alternative paths and generates assertions for the new states. Finally it generates a new test suite from the extended model.

Our results supports that leveraging input values and assertions from human-written test suites can be helpful in generating more effective test cases. TESTILIZER easily outperforms a random test generation technique, provides substantial improvements in the fault detection rate compared with the original test suite, while slightly increasing code coverage too.

Future directions. TESTILIZER is limited to applications that already have human-written tests, which may not be so prevalent in practice. On the other hand, many web applications are similar to each other in terms of design and code base, such as being built on top of the same content management system. We propose an open research problem whether human-written tests can be leveraged to generate effective tests for applications without existing tests. This is, however, challenging particularly for assertion generation based on learned patterns. DOM-based assertions on abstract DOM states of an application may require some changes to be

applied on similar abstract DOM state of another application. This also requires defining similarity measures between applications for testing purposes.

Another possible extension of this work is the test generation approach given the extended SFG with newly generated assertions. The current graph traversal method in TESTILIZER may produce test cases that share common paths, which do not contribute much to fault detection or code coverage. An optimization could be realized by guiding the test generation algorithm towards sates that have more constrained DOM-based assertions.

RQ1.4.3. *What is the quality of JavaScript tests in practice and which part of the code are hard to cover and test?*

Chapter 4. While some JavaScript features are known to be hard to test, no empirical study was done earlier towards measuring the quality and coverage of JavaScript tests. We present the first empirical study of JavaScript tests to characterize their prevalence, quality metrics (code coverage, test code ratio, test commit ratio, and average number of assertions per test), and shortcomings [125].

We found that a considerable percentage of JavaScript projects do not have any test and this is in particular for projects with JavaScript at client-side. On the other hand almost all purely server-side JavaScript projects have tests and the quality of those tests are higher compared to tests for client-side. On average JavaScript tests lack proper coverage for event-dependent callbacks, asynchronous callbacks, and DOM-related code.

Future directions. The result of this study can be used to improve JavaScript test generation tools in producing more effective test cases that target hard-to-test code. Also evaluating the effectiveness of JavaScript tests by measuring their mutation score, may reveal shortcomings regarding the quality of written assertions. Another possible direction could be designing automated JavaScript code refactoring techniques towards making the code more testable and maintainable.

RQ1.4.4. *How can we automate fixture generation for JavaScript unit testing?*

Chapter 5. Proper test fixtures are required to cover DOM-dependent statements and conditions in unit testing JavaScript code. However, generating such fixtures is not an easy task. We proposed a DOM-based test fixture generation technique [127] and a tool, called CONFIX [24], which is based on concolic (concrete and

symbolic) execution. Our approach guides program executing through different branches of a function. CONFIX automatically generate a set of JavaScript unit tests with DOM fixtures and DOM function arguments. Our empirical results show that the generated fixtures substantially improve code coverage compared to test suites without these fixtures, and the overhead is negligible .

Future directions. Our DOM-based fixture generation technique could be enhanced by adding support for statements that require event-triggering, and handling more complex constraints by improving the integer/string constraint solver to generate XPath expressions with proper structure and attribute values. Also existing JavaScript test input generator tools, such as JALANGI, could be combined with CONFIX in such a way that the existing tool uses CONFIX in the case of DOM-related constraints.

RQ1.4.5. *Which JavaScript code smells are prevalent in practice and what maintenance issues they cause?*

Chapter 6. We proposed a tool and technique, called JSNOSE [34], to detect JavaScript code smells [124]. It uses static and dynamic analysis of the client-side code to detect objects, functions, variables, and code blocks. We collected 13 JavaScript code smells by studying various online development resources and books that discuss bad JavaScript coding patterns and what maintenance issues they make. Our empirical evaluation shows that (1) JSNOSE can accurately detect these code smells, and (2) lazy object, long method/function, closure smells, coupling between JavaScript, HTML, and CSS, and excessive global variables are the most prevalent code smells.

Future directions. JSNOSE can be used during development and maintenance cycles to spot potential JavaScript code smells, which can be refactored to improve the code quality. However, manual code refactoring may take a significant amount of time and may also introduce new bugs if changes are not done properly. Current JavaScript refactoring techniques [8, 11, 83, 84] support basic operations such as renaming of variables or object properties, or encapsulation of properties and extraction of modules. One possible future direction is designing an automated tool for JavaScript code refactoring for the detected code smell instances. Examples of the required code refactoring is explained in our work, however, a major chal-

lenge towards building an effective JavaScript code refactoring is to guarantee code changes do not affect the application behaviour.

7.2 Concluding Remarks

The work presented in this dissertation has focused on advancing the state-of-the-art in automated testing and maintenance of web applications. Although, the proposed approaches have been tailored to web and in particular JavaScript-based applications, a number of contributions are applicable to other types of applications, programming languages, and software analysis domains as following:

- Our feedback-directed exploration technique [123] decides about next state to expand and next event to exercise based on scoring functions. Such scoring functions are generic and can be changed to adapt the model inference for other purposes such as program comprehension.
- Our proposed UI testing techniques for test model generation [123], and using existing tests to generate new ones [126], can be adapted to other types of application that have rich user interface interactions such as mobile applications and desktop GUI-based applications. The major difference is the DOM structure and its elements compared to a specific designed user interface.
- The idea of leveraging knowledge in existing tests for UI test generation [126] can also be applied for unit test generation. For instance some input values and program paths can be reused.
- Our technique to generate JavaScript unit tests with DOM-based fixture [127] can be adapted for building test fixtures in the form of partial UI that is required for proper unit testing of mobile and other GUI-based applications.
- Our empirical study to analyze prevalence, quality, and shortcomings of existing JavaScript tests, can be similarly conducted to study unit tests written in other languages. In particular, less studied popular languages such as Ruby, Objective-C, Python, and Go can be interesting to analyse.

- Our JavaScript code smell detection technique [124] can be applied on other implementations of ECMAScript (e.g. JScript .NET, and ActionScript), supersets of JavaScript (e.g. CoffeeScript and TypeScript), prototype-based programming languages (e.g. Self, Lua, Common Lisp), and languages that support first-class functions (e.g. ML, Haskell, Scala, Perl, Python, PHP).

Bibliography

[1] AddressBook. https://sourceforge.net/projects/php-addressbook/. Accessed: 2016-01-30. → pages 59

[2] Brotherhood social network. https://github.com/HSJared/Social-Network/. Accessed: 2016-01-30. → pages 59

[3] Checkstyle. http://checkstyle.sourceforge.net/. Accessed: 2015-04-30. → pages 133

[4] CollegeVis. https://github.com/nerdyworm/collegesvis. Accessed: 2013-09-30. → pages 149

[5] CookeryBook. https://github.com/achudars/adaptable-cookery-book. Accessed: 2016-01-30. → pages 59

[6] WSO2 EnterpriseStore. https://github.com/wso2/enterprise-store. Accessed: 2014-11-30. → pages 59

[7] GhostBusters. http://10k.aneventapart.com/2/Uploads/657. Accessed: 2013-09-30. → pages 149

[8] Eclipse JavaScript development tools. http://wiki.eclipse.org/JSDT. Accessed: 2015-04-30. → pages 158

[9] Jslint: The JavaScript code quality tool. http://www.jslint.com/. Accessed: 2015-04-30. → pages 8, 130, 133

[10] Mozilla developer network's JavaScript reference. https://developer.mozilla.org/en-US/docs/JavaScript/Reference. Accessed: 2013-09-30. → pages 134, 136

[11] JetBrains JavaScript editor. http://www.jetbrains.com/editors/javascript_editor.jsp. Accessed: 2015-04-30. → pages 158

[12] PMD code analyzer. http://pmd.sourceforge.net/. Accessed: 2015-04-30. → pages 8

[13] PeriodicTable. http://code.jalenack.com/periodic/. Accessed: 2013-09-30. → pages 149

[14] Environment simulated study room. https://github.com/NhatHo/Environment-Simulated-Study-Room/. Accessed: 2016-01-30. → pages 59

[15] Symbolistic. http://10k.aneventapart.com/2/Uploads/652. Accessed: 2013-09-30. → pages 149

[16] Tunnel. http://arcade.christianmontoya.com/tunnel. Accessed: 2013-09-30. → pages 149

[17] WARI: Web application resource inspector. http://wari.konem.net. Accessed: 2015-04-30. → pages 8, 130, 133

[18] WolfCMS. https://github.com/wolfcms/wolfcms. Accessed: 2014-11-30. → pages 59

[19] Callback hell: A guide to writing elegant asynchronous JavaScript programs. http://callbackhell.com/. Accessed: 2013-09-30. → pages 140

[20] SimpleChessGame. p4wn.sourceforge.net. Accessed: 2013-09-30. → pages 27, 149

[21] Claroline. https://github.com/claroline/Claroline. Accessed: 2014-11-30. → pages 59

[22] CLOC. http://cloc.sourceforge.net. Accessed: 2013-09-30. → pages 28, 149

[23] complexityReport.js. https://npmjs.org/package/complexity-report/. Accessed: 2013-09-30. → pages 149

[24] ConFix. http://salt.ece.ubc.ca/software/confix/. Accessed: 2015-11-30. → pages 9, 102, 121, 127, 157

[25] FeedEx. https://github.com/saltlab/FeedEx. Accessed: 2013-09-30. → pages 8, 26, 27

[26] FractalViewer. http://onecm.com/projects/canopy. Accessed: 2013-09-30. → pages 27, 149

[27] Google closure compiler. https://developers.google.com/closure/. Accessed: 2013-09-30. → pages 8, 130, 133

[28] Examples of hard to test JavaScript. https://www.pluralsight.com/blog/software-development/6-examples-of-hard-to-test-javascript. Accessed: 2016-08-30. → pages 5, 75, 82

[29] Hotel Reservation. https://github.com/andyfeds/HotelReservationSystem. Accessed: 2015-11-30. → pages 122

[30] Istanbul - a JS code coverage tool written in JS. https://github.com/gotwarlost/istanbul. Accessed: 2016-08-30. → pages 79

[31] Jasmine. https://github.com/pivotal/jasmine. Accessed: 2016-08-30. → pages 4, 75, 104

[32] JavaParser. https://code.google.com/p/javaparser/. Accessed: 2014-11-30. → pages 57

[33] Jscover. http://tntim96.github.io/JSCover/. Accessed: 2014-05-30. → pages 63, 79, 124

[34] JSNose. https://github.com/saltlab/JSNose. Accessed: 2013-09-30. → pages 9, 148, 158

[35] JsUnit. http://jsunit.net/. Accessed: 2016-08-30. → pages 104

[36] Mocha. https://mochajs.org/. Accessed: 2016-08-30. → pages 4, 75

[37] Nodeunit. https://github.com/caolan/nodeunit. Accessed: 2016-08-30. → pages 4, 75

[38] Organizer. http://www.apress.com/9781590596951. Accessed: 2014-11-30. → pages 41

[39] Phormer Photogallery. http://sourceforge.net/projects/rephormer/. Accessed: 2013-09-30. → pages 27, 59, 122, 149

[40] QUnit. http://qunitjs.com/. Accessed: 2016-08-30. → pages 1, 4, 75, 104

[41] Mozilla Rhino. https://github.com/mozilla/rhino. Accessed: 2013-08-30. → pages 26, 79, 117, 148

[42] Selenium HQ. http://seleniumhq.org/. Accessed: 2016-08-30. → pages 1, 30, 38, 109, 117

[43] SimpleToDo. https://github.com/heyamykate/vanillaJS. Accessed: 2015-11-30. → pages 122

[44] Sudoku game. http://www.dhtmlgoodies.com/scripts/game_sudoku/game_sudoku.html. Accessed: 2015-11-30. → pages 122

[45] TacirFormBuilder. https://github.com/ekinertac/TacirFormBuilder. Accessed: 2013-09-30. → pages 27

[46] Writing testable JavaScript. http://www.adequatelygood.com/Writing-Testable-JavaScript.html, . Accessed: 2016-08-30. → pages 5, 75, 82

[47] Testilizer. https://github.com/saltlab/Testilizer, . Accessed: 2014-11-30. → pages 8, 39, 57, 58, 156

[48] How to unit test private functions in JavaScript. https://philipwalton.com/articles/how-to-unit-test-private-functions-in-javascript/, . Accessed: 2016-08-30. → pages 5, 75, 82, 85

[49] TinySiteCMS. tinysitecms.com, . Accessed: 2013-09-30. → pages 149

[50] TinyMCE. tinymce.com, . Accessed: 2013-09-30. → pages 27, 149

[51] TuduListManager. julien-dubois.com/tudu-lists/. Accessed: 2013-09-30. → pages 27, 149

[52] Which JavaScript test library should you use? http://www.techtalkdc.com/which-javascript-test-library-should-you-use-qunit-vs-jasmine-vs-mocha/. Accessed: 2016-08-30. → pages 4, 75

[53] Writing testable code in JavaScript: A brief overview. https://www.toptal.com/javascript/writing-testable-code-in-javascript. Accessed: 2016-08-30. → pages 5, 75, 82

[54] JSter JavaScript Libraries Catalog. http://jster.net/catalog, 2014. Accessed: 2016-08-30. → pages 76, 97

[55] Most depended-upon NMP packages. https://www.npmjs.com/browse/depended, 2014. Accessed: 2016-08-30. → pages 76

[56] Github Showcases. https://github.com/showcases, 2014. Accessed: 2016-08-30. → pages 76, 97

[57] Testing and deploying with ordered npm run scripts. http://blog.npmjs.org/post/127671403050/testing-and-deploying-with-ordered-npm-run-scripts, 2015. Accessed: 2016-08-30. → pages 87, 95

[58] SLOC (source lines of code) counter. https://github.com/flosse/sloc/, 2016. Accessed: 2016-08-30. → pages 77

[59] TestScanner. https://github.com/saltlab/testscanner, 2016. Accessed: 2016-08-30. → pages 76, 79, 97

[60] C. Q. Adamsen, A. Møller, and G. Mezzetti. Systematic execution of android test suites in adverse conditions. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 83–93. ACM, 2015. → pages 72

[61] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong. Diversifying search results. In *Proc. of the International Conference on Web Search and Data Mining*, pages 5–14. ACM, 2009. → pages 16, 34

[62] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding JavaScript event-based interactions. In *Proc. of the ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 367–377. ACM, 2014. → pages 1, 6, 101, 106

[63] N. Alshahwan and M. Harman. State aware test case regeneration for improving web application test suite coverage and fault detection. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 45–55, 2012. → pages 72

[64] N. Alshahwan, M. Harman, A. Marchetto, R. Tiella, and P. Tonella. Crawlability metrics for web applications. In *Proc. International Conference on Software Testing, Verification and Validation (ICST)*, pages 151–160, Washington, DC, USA, 2012. IEEE Computer Society, IEEE Computer Society. ISBN 978-0-7695-4670-4. doi:10.1109/ICST.2012.95. URL http://dx.doi.org/10.1109/ICST.2012.95. → pages 12, 33, 34

[65] M. Alshraideh. A complete automation of unit testing for JavaScript programs. *Journal of Computer Science*, 4(12):1012, 2008. → pages 127

[66] S. Artzi, J. Dolby, S. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *Proc. International Conference on Software Engineering (ICSE)*, pages 571–580. ACM, 2011. → pages 1, 4, 5, 6, 35, 39, 71, 75, 96, 101, 127

[67] C. Atkinson, O. Hummel, and W. Janjic. Search-enhanced testing (nier track). In *Proceedings of the 33rd International Conference on Software Engineering (ICSE-NIER track)*, pages 880–883. ACM, 2011. → pages 72

[68] M. Benedikt, J. Freire, and P. Godefroid. VeriWeb: Automatically testing dynamic web sites. In *Proc. of the International World Wide Web Conference (WWW)*, pages 654–668, 2002. → pages 2, 12, 32, 33

[69] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *Proc. of the Symposium on Principles of Database Systems*, pages 25–36. ACM, 2005. doi:10.1145/1065167.1065172. → pages 112

[70] K. Benjamin, G. von Bochmann, M. E. Dincturk, G.-V. Jourdan, and I.-V. Onut. A strategy for efficient crawling of rich internet applications. In *Proc. of the International Conference on Web Engineering (ICWE)*, pages 74–89. Springer-Verlag, 2011. → pages 3, 33, 34

[71] C.-P. Bezemer, A. Mesbah, and A. van Deursen. Automated security testing of web widget interactions. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering (ESEC-FSE)*, pages 81–91. ACM, 2009. → pages 2

[72] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2: 27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm. → pages 58

[73] S. Choudhary, M. E. Dincturk, S. M. Mirtaheri, G.-V. Jourdan, G. von Bochmann, and I. V. Onut. Building rich internet applications models: Example of a better strategy. In *Proc. of the International Conference on Web Engineering (ICWE)*. Springer, 2013. → pages 3, 34

[74] S. R. Choudhary, M. Prasad, and A. Orso. Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *Proc. International Conference on Software Testing, Verification and Validation (ICST)*, pages 171–180. IEEE Computer Society, 2012. → pages 2, 3, 12, 32, 33, 38

[75] J. Clark and S. DeRose. Xml path language (xpath). http://www.w3.org/TR/1999/REC-xpath-19991116, November 1999. → pages 112

[76] Y. Crespo, C. López, R. Marticorena, and E. Manso. Language independent metrics support towards refactoring inference. In *9th ECOOP Workshop on QAOOSE*, volume 5, pages 18–29, 2005. → pages 133, 141, 144

[77] D. Crockford. *JavaScript: the good parts*. O'Reilly Media, Incorporated, 2008. → pages 8, 84, 130, 134, 135, 136, 139, 140

[78] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 85–96, 2010. → pages 71

[79] M. E. Dincturk, S. Choudhary, G. von Bochmann, G.-V. Jourdan, and I. V. Onut. A statistical approach for efficient crawling of rich internet applications. In *Proc. of the International Conference on Web Engineering (ICWE)*, pages 362–369. Springer, 2012. → pages 3, 34

[80] C. Duda, G. Frey, D. Kossmann, R. Matter, and C. Zhou. Ajax crawl: Making Amax applications searchable. In *Proc. of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 78–89. IEEE Computer Society, 2009. → pages 23, 33

[81] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher. Leveraging user-session data to support web application testing. *IEEE Trans. Softw. Eng.*, 31(3): 187–202, 2005. ISSN 0098-5589. → pages 4, 70

[82] M. Erfani, I. Keivanloo, and J. Rilling. Opportunities for clone detection in test case recommendation. In *IEEE Computer Software and Applications Conference Workshops (COMPSACW)*, pages 65–70. IEEE, 2013. → pages 72

[83] A. Feldthaus and A. Møller. Semi-automatic rename refactoring for JavaScript. In *Proc. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 323–338. ACM, 2013. → pages 8, 158

[84] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported refactoring for JavaScript. In *Proc. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011. → pages 8, 158

[85] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999. → pages 7, 130, 131, 132, 134, 141, 145

[86] F. Galassi. Refactoring to unobtrusive JavaScript. JavaScript Camp 2009. → pages 132, 134, 137, 138

[87] K. Gallaba, A. Mesbah, and I. Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in JavaScript. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 247–256. IEEE Computer Society, 2015. → pages 76, 77, 84, 98

[88] V. Garousi, A. Mesbah, A. Betin Can, and S. Mirshokraie. A systematic mapping study of web application testing. *Information and Software Technology*, 2013. → pages 12

[89] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *Proc. of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 342–351, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi:10.1145/1250734.1250773. URL http://doi.acm.org/10.1145/1250734.1250773. → pages 102, 115, 117

[90] M. Ghafari, C. Ghezzi, A. Mocci, and G. Tamburrelli. Mining unit tests for code recommendation. In *International Conference on Program Comprehension (ICPC)*, pages 142–145. ACM, 2014. → pages 72

[91] GitHut. A small place to discover languages in GitHub. http://githut.info, 2015. → pages 1, 75

[92] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi:10.1145/1065010.1065036. → pages 107

[93] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *Proc. of the International World Wide Web Conference (WWW)*, pages 561–570. ACM, 2009. ISBN 978-1-60558-487-4. → pages 34

[94] M. Harrold, R. Gupta, and M. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):270–285, 1993. → pages 17, 24

[95] P. Heidegger and P. Thiemann. Contract-driven testing of Javascript code. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, TOOLS'10, pages 154–172, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-13952-3, 978-3-642-13952-9. URL http://dl.acm.org/citation.cfm?id=1894386.1894395. → pages 5, 6, 75, 127

[96] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, first edition, 2010. ISBN 0321601912, 9780321601919. → pages 28

[97] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proc. of the International Conference on Software Engineering (ICSE)*, 2014. → pages 5, 63, 79, 98

[98] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in javascript web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 270–283. ACM, 2010. → pages 98

[99] W. Janjic and C. Atkinson. Utilizing software reuse experience for automated test recommendation. In *Proceedings of the 8th International Workshop on Automation of Software Test (AST)*, pages 100–106. IEEE Press, 2013. → pages 72

[100] C. S. Jensen, A. Møller, and Z. Su. Server interface descriptions for automated testing of JavaScript web applications. In *Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 510–520. ACM, 2013. → pages 118

[101] 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*, ESEC/FSE'11, pages 59–69. ACM, 2011. → pages 34, 101

[102] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, FSE 2014, pages 654–665, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi:10.1145/2635868.2635929. URL http://doi.acm.org/10.1145/2635868.2635929. → pages 97

[103] S. H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, 2002. → pages 63, 150

[104] J. Kerievsky. *Refactoring to patterns*. Pearson Deutschland GmbH, 2005. → pages 131, 134, 145

[105] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 105–116. ACM, 2009. doi:10.1145/1572272.1572286. → pages 118

[106] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. → pages 106

[107] M. Landhäußer and W. F. Tichy. Automated test-case generation by cloning. In *Proceedings of the 7th International Workshop on Automation of Software Test (AST)*, pages 83–88. IEEE Press, 2012. → pages 72

[108] M. Lanza and R. Marinescu. *Object-oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006. → pages 133, 141, 143, 144, 145

[109] G. Li, E. Andreasen, and I. Ghosh. SymJS: Automatic symbolic testing of JavaScript web applications. In *Proc. of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, page 11 pages. ACM, 2014. → pages 6, 96, 101, 107, 127

[110] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy. Google's deep web crawl. *Proc. VLDB Endow.*, 1(2):1241–1252, 2008. → pages 33

[111] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *ESEC/FSE*, 2013. → pages 101

[112] M. Mäntylä, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 381–384. IEEE Computer Society, 2003. → pages 145

[113] A. Marchetto and P. Tonella. Using search-based algorithms for Ajax event sequence generation during testing. *Empirical Software Engineering*, 16 (1):103–140, 2011. → pages 2

[114] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of Ajax web applications. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 121–130. IEEE, IEEE Computer Society, 2008. ISBN 978-0-7695-3127-4. → pages 12, 32, 33, 35

[115] S. McAllister, E. Kirda, and C. Kruegel. Leveraging user interactions for in-depth testing of web applications. In *Recent Advances in Intrusion Detection*, volume 5230 of *LNCS*, pages 191–210. Springer, 2008. URL http://dx.doi.org/10.1007/978-3-540-87403-4_11. → pages 70

[116] A. M. Memon. An event-flow model of gui-based applications for testing. *Software testing, verification and reliability*, 17(3):137–157, 2007. ISSN 0960-0833. doi:http://dx.doi.org/10.1002/stvr.v17:3. → pages 68

[117] F. Menczer, G. Pant, and P. Srinivasan. Topical web crawlers: Evaluating adaptive algorithms. *ACM Transactions on Internet Technology (TOIT)*, 4 (4):378–419, 2004. → pages 33

[118] A. Mesbah and S. Mirshokraie. Automated analysis of CSS rules to support style maintenance. In *Proc. International Conference on Software Engineering (ICSE)*, pages 408–418. IEEE Computer Society, 2012. → pages 133

[119] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 561–570. ACM, 2011. → pages 2, 12, 32, 33

[120] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012. → pages 2, 8, 14, 22, 26, 33, 45, 57, 148

[121] A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering (TSE)*, 38(1):35–53, 2012. ISSN 0098-5589. doi:10.1109/TSE.2011.28. URL http://dx.doi.org/10.1109/TSE.2011.28. → pages 2, 3, 4, 8, 12, 15, 26, 32, 33, 35, 38, 39, 45, 47, 71, 127

[122] G. Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007. → pages 103

[123] A. Milani Fard and A. Mesbah. Feedback-directed exploration of web applications to derive test models. In *Proc. of the International Symposium*

on Software Reliability Engineering (ISSRE), pages 278–287. IEEE Computer Society, 2013. → pages iii, 8, 10, 11, 47, 71, 127, 155, 159

[124] A. Milani Fard and A. Mesbah. JSNose: Detecting JavaScript code smells. In *Proc. of the International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125. IEEE Computer Society, 2013. → pages iv, 9, 10, 32, 80, 98, 129, 158, 160

[125] A. Milani Fard and A. Mesbah. JavaScript: The (un)covered parts. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, page 11 pages. IEEE Computer Society, 2017. → pages iii, 9, 10, 74, 157

[126] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 67–78. ACM, 2014. → pages iii, 8, 10, 37, 98, 127, 156, 159

[127] A. Milani Fard, A. Mesbah, and E. Wohlstadter. Generating fixtures for JavaScript unit testing. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 190–200. IEEE Computer Society, 2015. → pages iii, 5, 9, 10, 72, 75, 82, 83, 95, 96, 100, 157, 159

[128] S. Mirshokraie and A. Mesbah. JSART: JavaScript assertion-based regression testing. In *Proc. of the Internatinoal Conference on Web Engineering (ICWE)*, pages 238–252. Springer, 2012. → pages 2, 12, 32, 33, 35, 71

[129] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Efficient JavaScript mutation testing. In *Proc. of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2013. → pages 2, 5, 12, 33, 62, 97, 99

[130] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. JSeft: Automated JavaScript unit test generation. In *Proc. of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE Computer Society, 2015. → pages 1, 4, 5, 6, 71, 72, 75, 95, 96, 101, 122, 126, 127

[131] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Atrina: Inferring unit oracles from GUI test cases. In *Proceedings of the International*

Conference on Software Testing, Verification, and Validation (ICST), page 11 pages. IEEE Computer Society, 2016. → pages 5, 75, 95

[132] M. Mirzaaghaei, F. Pastore, and M. Pezze. Supporting test suite evolution through test case adaptation. In *Proc. of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 231–240. IEEE Computer Society, 2012. → pages 72

[133] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010. → pages 133, 141, 144, 150

[134] A. Moosavi, S. Hooshmand, S. Baghbanzadeh, G.-V. Jourdan, G. V. Bochmann, and I. V. Onut. Indexing rich internet applications using components-based crawling. In *Proc. of the International Conference on Web Engineering (ICWE)*, pages 200–217. Springer-Verlag, 2014. → pages 3, 33, 34

[135] M. J. Munro. Product metrics for automatic identification of "bad smell" design problems in Java source-code. In *Proc. International Symposium Software Metrics*, pages 15–15. IEEE, 2005. → pages 133, 141, 144

[136] R. Murphey. JS minty fresh: Identifying and eliminating JavaScript code smells. http://fronteers.nl/congres/2012/sessions/js-minty-fresh-rebecca-murphey. → pages 132, 134

[137] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen. Detection of embedded code smells in dynamic web applications. In *Proc. of the International Conference on Automated Software Engineering (ASE)*, pages 282–285. ACM, 2012. → pages 7, 130, 132, 133, 137

[138] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Building call graphs for embedded client-side code in dynamic web applications. In *Proc. of the International Symposium on Foundations of Software Engineering (FSE)*, pages 518–529. ACM, 2014. doi:10.1145/2635868.2635928. → pages 127

[139] H. V. Nguyen, H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen. Mining interprocedural, data-oriented usage patterns in javascript web applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 791–802. ACM, 2014. → pages 98

[140] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 736–747. ACM, 2012. → pages 98

[141] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. An empirical study of client-side JavaScript bugs. In *Proc. of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 55–64. IEEE Computer Society, 2013. → pages 1, 6, 96, 101, 103, 106, 126

[142] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. A study of causes and consequences of client-side JavaScript bugs. *IEEE Transactions on Software Engineering (TSE)*, page 17 pages, 2017. → pages 5, 97, 98

[143] F. J. Ocariza, K. Pattabiraman, and A. Mesbah. AutoFLox: An automatic fault localizer for client-side JavaScript. In *Proc. of the International Conference on Software Testing, Verification and Validation (ICST'12)*, pages 31–40. IEEE Computer Society, 2012. → pages 34

[144] C. Olston and M. Najork. Web crawling. *Foundations and Trends in Information Retrieval*, 4(3):175–246, 2010. → pages 12, 16, 33

[145] V. Özçelik. o2.js JavaScript conventions & best practices. https://github.com/v0lkan/o2.js/blob/master/CONVENTIONS.md. → pages 132, 134, 139, 145

[146] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. 29th Int. Conf. on Sw. Engineering (ICSE'07)*, pages 75–84. IEEE Computer Society, IEEE Computer Society, 2007. doi:http://dx.doi.org/10.1109/ICSE.2007.37. → pages 71

[147] J. Padolsey. jQuery code smells. http://james.padolsey.com/javascript/jquery-code-smells/. → pages 132, 134, 138, 139

[148] K. Pattabiraman and B. Zorn. DoDOM: Leveraging DOM Invariants for Web 2.0 Application Robustness Testing. In *Proc. of the International Symposium on Sw. Reliability Eng. (ISSRE)*, pages 191–200. IEEE Computer Society, 2010. → pages 35, 55, 71

[149] M. Pawlik and N. Augsten. RTED: a robust algorithm for the tree edit distance. *Proc. VLDB Endow.*, 5(4):334–345, Dec. 2011. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=2095686.2095692. → pages 23, 27

[150] M. Pezze, K. Rubinov, and J. Wuttke. Generating effective integration test cases from unit ones. In *Proc. International Conference on Software Testing, Verification and Validation (ICST)*, pages 11–20. IEEE, 2013. → pages 4, 72

[151] S. Porto. A plain english guide to JavaScript prototypes. http://sporto.github.com/blog/2013/02/22/a-plain-english-guide-to-javascript-prototypes/. → pages 130

[152] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development*, WebApps'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1863166.1863169. → pages 98

[153] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM, 2010. ISBN 978-1-4503-0019-3. doi:http://doi.acm.org/10.1145/1806596.1806598. → pages 98, 106, 132

[154] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do. In *ECOOP 2011–Object-Oriented Programming*, pages 52–78. Springer, 2011. → pages 98

[155] K. Rubinov and J. Wuttke. Augmenting test suites automatically. In *Proc. of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1433–1434, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337438. → pages 72

[156] T. Sakai. Evaluation with informational and navigational intents. In *Proc. of the International Conference on World Wide Web (WWW)*, pages 499–508. ACM, 2012. → pages 34

[157] R. Santos, C. Macdonald, and I. Ounis. Selectively diversifying web search results. In *Proc. of the International Conference on Information and knowledge management*, pages 1179–1188. ACM, 2010. → pages 34

[158] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *Proc. Symp. on Security and Privacy (SP'10)*, SP '10, pages 513–528, Washington, DC, USA,

2010. IEEE Computer Society. ISBN 978-0-7695-4035-1. doi:http://dx.doi.org/10.1109/SP.2010.38. URL http://dx.doi.org/10.1109/SP.2010.38. → pages 6, 35, 96, 107, 108, 120, 127

[159] M. Schur, A. Roth, and A. Zeller. Mining behavior models from enterprise web applications. In *Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering*, Proc. of the Foundations of Software Engineering (ESEC/FSE), pages 422–432. ACM, 2013. → pages 3, 4, 38, 71

[160] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE, pages 263–272. ACM, 2005. ISBN 1-59593-014-0. doi:10.1145/1081706.1081750. URL http://doi.acm.org/10.1145/1081706.1081750. → pages 107

[161] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 488–498. ACM, 2013. ISBN 978-1-4503-2237-9. doi:10.1145/2491411.2491447. URL http://doi.acm.org/10.1145/2491411.2491447. → pages 6, 96, 101, 106, 107, 108, 120, 123, 126, 127

[162] F. Simon, F. Steinbruckner, and C. Lewerentz. Metrics based refactoring. In *Proc European Conference on Software Maintenance and Reengineering (CSMR)*, pages 30–38. IEEE, 2001. → pages 133, 141, 144

[163] K. Simpson. Native JavaScript: sync and async. http://blog.getify.com/native-javascript-sync-async. → pages 134, 140

[164] R. Song, H. Liu, J.-R. Wen, and W.-Y. Ma. Learning important models for web page blocks based on layout and content analysis. *ACM SIGKDD Explorations Newsletter*, 6(2):14–23, 2004. → pages 53, 54

[165] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *ASE'05: Proc. 20th IEEE/ACM Int. Conf. on Automated Sw. Eng.*, pages 253–262. ACM, 2005. ISBN 1-59593-993-4. doi:http://doi.acm.org/10.1145/1101908.1101947. URL http://doi.acm.org/10.1145/1101908.1101947. → pages 4, 70

[166] P. Srinivasan, F. Menczer, and G. Pant. A general evaluation framework for topical crawlers. *Information Retrieval*, 8(3):417–447, 2005. → pages 16

[167] Stack Overflow. 2016 developer survey. http://stackoverflow.com/research/developer-survey-2016, 2016. → pages 1, 75

[168] K.-C. Tai. The tree-to-tree correction problem. *Journal of the ACM (JACM)*, 26(3):422–433, 1979. ISSN 0004-5411. doi:10.1145/322139.322143. URL http://doi.acm.org/10.1145/322139.322143. → pages 23

[169] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra. Guided test generation for web applications. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 162–171. IEEE Computer Society, 2013. → pages 3, 33, 34, 38

[170] N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 134–153. Springer Verlag, April 2008. URL http://research.microsoft.com/apps/pubs/default.aspx?id=81193. → pages 107

[171] P. Tonella and F. Ricca. Statistical testing of web applications. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(1-2): 103–127, 2004. ISSN 1532-060X. doi:http://dx.doi.org/10.1002/smr.284. → pages 2, 12, 33

[172] P. Tonella, F. Ricca, A. Stocco, and M. Leotta. Automated generation of visual web tests from DOM-based web tests. In *Proceedings of the International Symposium on Applied Computing (SAC)*, pages 775–782. ACM, 2015. → pages 72

[173] M. E. Trostler. *Testable JavaScript*. O'Reilly Media, Incorporated, 2013. → pages 5, 75, 82

[174] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. JDeodorant: Identification and removal of type-checking bad smells. In *Proc. European Conference on Software Maintenance and Reengineering (CSMR)*, pages 329–331, 2008. → pages 133

[175] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5): 297–312, 2012. → pages 2

[176] A. Vahabzadeh, A. Milani Fard, and A. Mesbah. An empirical study of bugs in test code. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 101–110. IEEE Computer Society, 2015. → pages 10, 98

[177] A. Valmari. The state explosion problem. In *LNCS: Lectures on Petri Nets I, Basic Models, Advances in Petri Nets*, pages 429–528. Springer-Verlag, 1998. ISBN 3-540-65306-6. → pages 3, 12

[178] E. Van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proc. of the Working Conference on Reverse Engineering (WCRE)*, pages 97–106. IEEE Computer Society, 2002. → pages 130, 134

[179] V. Vapnik. *The nature of statistical learning theory*. springer, 2000. → pages 53

[180] W3C. Document Object Model (DOM) level 2 events specification. http://www.w3.org/TR/DOM-Level-2-Events/, 13 November 2000. → pages 1, 7, 130, 137

[181] Y. Wang, S. Person, S. Elbaum, and M. B. Dwyer. A framework to advise tests using tests. In *Proc. of ICSE NIER*. ACM, 2014. → pages 4, 72

[182] S. Wei and B. G. Ryder. Practical blended taint analysis for JavaScript. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 336–346. ACM, 2013. → pages 106

[183] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. An empirical analysis of XSS sanitization in web application frameworks. *Electrical Engineering and Computer Sciences University of California at Berkeley, Technical Report*, pages 1–17, 2011. → pages 98

[184] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982. doi:10.1093/comjnl/25.4.465. → pages 3

[185] L. Williams, D. Ho, and S. Heckman. Software metrics in eclipse. http://realsearchgroup.org/SEMaterials/tutorials/metrics/. → pages 143, 144, 145

[186] M. R. Woodward. Mutation testing?its origin and evolution. *Information and Software Technology*, 35(3):163–169, 1993. → pages 62

[187] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *Formal Approaches to Software Testing*, pages 60–69. Springer, 2004. → pages 71

[188] D. Xu, W. Xu, B. K. Bavikati, and W. E. Wong. Mining executable specifications of web applications from Selenium IDE tests. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pages 263–272. IEEE, 2012. → pages 4, 71

[189] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: techniques and tradeoffs. In *Proc. of the International Symposium on Foundations of Software Engineering (FSE)*, pages 257–266. ACM, 2010. → pages 4, 72

[190] S. Yoo and M. Harman. Test data regeneration: generating new test data from existing test data. *Software Testing, Verification and Reliability*, 22 (3):171–201, 2012. → pages 4, 71

[191] X. Yuan and A. M. Memon. Using gui run-time state as feedback to generate test cases. In *ICSE '07: Proc. of the 29th international conference on Software Engineering*, ICSE '07, pages 396–405, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi:http://dx.doi.org/10.1109/ICSE.2007.94. URL http://dx.doi.org/10.1109/ICSE.2007.94. → pages 71

[192] X. Yuan and A. M. Memon. Iterative execution-feedback model-directed GUI testing. *Information and Software Technology*, 52(5):559–575, 2010. → pages 4, 71

[193] C. Yue and H. Wang. Characterizing insecure JavaScript practices on the web. In *Proc. of the International World Wide Web Conference (WWW)*, pages 961–970. ACM, 2009. → pages 98

[194] A. Zaidman, B. van Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production and test code. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 220–229, 2008. → pages 98

[195] N. C. Zakas. Writing efficient JavaScript. In S. Souders, editor, *Even Faster Web Sites*. O'Reilly, 2009. → pages 134, 135

[196] N. C. Zakas. *Maintainable JavaScript - Writing Readable Code*. O'Reilly, 2012. → pages 134, 137, 138

[197] D. Zhang, W. Wang, D. Liu, Y. Lei, and D. Kung. Reusing existing test cases for security testing. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 323–324. IEEE Computer Society, 2008. → pages 72

[198] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA*, pages 385–396. ACM, 2014. → pages 69

[199] Y. Zhang and A. Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 214–224. ACM, 2015. → pages 5, 79, 98

[200] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proc. of the International World-Wide Web Conference (WWW)*, pages 805–814. ACM, 2011. → pages 34

[201] Z. Zhu, Y. Zou, B. Xie, Y. Jin, Z. Lin, and L. Zhang. Mining api usage examples from test code. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–310. IEEE Computer Society, 2014. → pages 72