On the Detection, Localization and Repair of Client-Side JavaScript Faults

by

Frolin S. Ocariza, Jr.

B.A.Sc. (Hons.), The University of Toronto, Canada, 2010

M.A.Sc., The University of British Columbia, Canada, 2012

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)

October 2016

Abstract

With web application usage becoming ubiquitous, there is greater demand for making such applications more reliable. This is especially true as more users rely on web applications to conduct day-to-day tasks, and more companies rely on these applications to drive their business. Since the advent of Web 2.0, developers often implement much of the web application's functionality at the client-side, using client-side JavaScript. Unfortunately, despite repeated complaints from developers about confusing aspects of the JavaScript language, little work has been done analyzing the language's reliability characteristics. With this problem in mind, we conducted an empirical study of real-world JavaScript bugs, with the goal of understanding their root cause and impact. We found that most of these bugs are *DOM-related*, which means they occur as a result of the JavaScript code's interaction with the Document Object Model (DOM).

Having gained a thorough understanding of JavaScript bugs, we designed techniques for automatically detecting, localizing and repairing these bugs. Our localization and repair techniques are implemented as the AUTOFLOX and VEJOVIS tools, respectively, and they target bugs that are DOM-related. In addition, our detection techniques – AUREBESH and HOLOCRON – attempt to find inconsistencies that occur in web applications written using JavaScript Model-View-Controller (MVC) frameworks. Based on our experimental evaluations, we found that these tools are highly accurate, and are capable of finding and fixing bugs in real-world web applications.

Preface

This dissertation is comprised of work conducted by myself, in collaboration with my advisors (Karthik Pattabiraman and Ali Mesbah) and two other colleagues. In particular, Chapters 2 to 5 are based on work published in various conferences and journals, and Chapter 6 is based on a paper to be submitted at a software engineering conference. I was the main author in all of these papers, and I was responsible for writing the manuscript, designing the approach (where applicable), and running the experiments. My collaborators were responsible for editing and writing portions of the manuscripts, manually analyzing bug reports, and extending my fault localization tool.

The publications associated with each chapter are listed below.

- Chapter 2:
  - "An Empirical Study of Client-Side JavaScript Bugs" [130]. F. Ocariza, K. Bajaj, K. Pattabiraman and A. Mesbah. In Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM 2013). ACM/IEEE. 55–64.
  - "A Study of Causes and Consequences of Client-Side JavaScript Bugs" [133] F. Ocariza, K. Bajaj, K. Pattabiraman, A. Mesbah. Transactions on Software Engineering (TSE). IEEE Computer Society.
- Chapter 3:
  - "AutoFLox: An Automatic Fault-Localizer for Client-Side JavaScript" [129]. F. Ocariza, K. Pattabiraman and A. Mesbah. In Proceedings of the International Conference on Software Testing, Verification and Validation (ICST 2012). IEEE Computer Society. 31–40.
  - "Automatic Fault Localization for Client-Side JavaScript" [134]. F. Ocariza, G. Li, K. Pattabiraman, A. Mesbah. Journal of Software Testing, Verification and Reliability (STVR). Vol 26, Issue 1. 2016. Wiley. 69–88
- Chapter 4:

- "Vejovis: Suggesting Fixes for JavaScript Faults" [131]. F. Ocariza, K. Pattabiraman and A. Mesbah. In Proceedings of the International Conference on Software Engineering (ICSE 2014). ACM. 837–847

- Chapter 5:
  - "Detecting Inconsistencies in JavaScript MVC Applications" [132]. F. Ocariza, K. Pattabiraman and A. Mesbah. In Proceedings of the International Conference on Software Engineering (ICSE 2015). IEEE Computer Society. 177–186.

- Chapter 6:
  - The author wrote a paper for this work which is in preparation for submission to a software engineering conference.

Table of Contents

Abstract . . . . . **ii**

Preface . . . . . **iii**

Table of Contents . . . . . **v**

List of Tables . . . . . **ix**

List of Figures . . . . . **xi**

Acknowledgements . . . . . **xiii**

Dedication . . . . . **xv**

1 Introduction **1**
- 1.1 Research Questions 5
- 1.2 Publications 7

2 Characteristics of JavaScript Faults **9**
- 2.1 Introduction 9
- 2.2 Background and Motivation 11
  - 2.2.1 Web Applications 12
  - 2.2.2 JavaScript Bugs 12
  - 2.2.3 Goal and Motivation 15
- 2.3 Experimental Methodology 16
  - 2.3.1 Research Questions 16
  - 2.3.2 Experimental Objects 18
  - 2.3.3 Collecting the Bug Reports 18
  - 2.3.4 Analyzing the Collected Bug Reports 20
- 2.4 Results 24
  - 2.4.1 Fault Categories 26
  - 2.4.2 Consequences of JavaScript Faults 29
  - 2.4.3 Causes of JavaScript Faults 32

2.4.4 Browser Specificity 35
2.4.5 Triage and Fix Time for JavaScript Faults 37
2.4.6 Prevalence of Type Faults 39
2.4.7 Threats to Validity 40
2.5 Discussion 42
2.6 Related Work 46
2.7 Conclusions 47

3 Automatic JavaScript Fault Localization 49
3.1 Introduction 49
3.2 Challenges and Motivation 51
3.2.1 Running Example 51
3.2.2 JavaScript Fault Localization 53
3.2.3 Challenges in Analyzing JavaScript Code 55
3.3 Scope of this Chapter 56
3.4 Approach 57
3.4.1 Usage Model 57
3.4.2 Trace Collection 58
3.4.3 Trace Analysis 60
3.4.4 Support for Challenging Cases 63
3.4.5 Assumptions 66
3.5 Tool Implementation 67
3.6 Empirical Evaluation 69
3.6.1 Goals and Research Questions 69
3.6.2 Methodology 69
3.6.3 Accuracy of AUTOFLOX 70
3.6.4 Real Bugs 73
3.6.5 Performance 76
3.7 Discussion 77
3.7.1 Limitations 77
3.7.2 Threats to Validity 78
3.8 Related Work 78
3.8.1 Web Application Reliability 78
3.8.2 Fault Localization 79
3.9 Conclusions and Future Work 81

4 Automatic JavaScript Fault Repair 82
4.1 Introduction 82
4.2 Background and Challenges 84
4.3 Common Developer Fixes 88
4.3.1 Methodology 88
4.3.2 Results 89

4.4 Fault Model 92
4.5 Approach 92
4.5.1 Data Collector 94
4.5.2 Analyzing Symptoms 95
4.5.3 Suggesting Treatments 101
4.5.4 Implementation: Vejovis 104
4.6 Evaluation 104
4.6.1 Subject Systems 104
4.6.2 Methodology 105
4.6.3 Results 106
4.7 Discussion 110
4.8 Related Work 111
4.9 Conclusion 112

5 Detecting Inconsistencies in JavaScript MVC Applications 113
5.1 Introduction 113
5.2 Running Example 115
5.3 Consistency Issues 119
5.4 Formal Model of MVC Applications 121
5.5 Approach 122
5.5.1 Overview 123
5.5.2 Finding the Models, Views and Controllers 125
5.5.3 Inferring Identifiers 126
5.5.4 Discovering MVC Groupings 129
5.5.5 Detecting Inconsistencies 129
5.6 Implementation 131
5.7 Evaluation 131
5.7.1 Subject Systems 131
5.7.2 Methodology 132
5.7.3 Results 136
5.8 Discussion 138
5.9 Related Work 140
5.10 Conclusion and Future Work 141

6 Cross-Language Inconsistency Detection 142
6.1 Introduction 142
6.2 Background and Motivation 145
6.2.1 Definitions 145
6.2.2 Motivating Examples 146
6.2.3 Challenges 147
6.3 Approach 148
6.3.1 Transforming Code into Trees 149

6.3.2 Finding Common Patterns 150
6.3.3 Establishing Rules from Patterns 151
6.3.4 Detecting Violations 155
6.4 Implementation 157
6.5 Evaluation 158
6.5.1 Subject Systems 158
6.5.2 Experimental Methodology 158
6.5.3 Prevalence of Inconsistencies (RQ1) 160
6.5.4 Real Bugs (RQ2) 162
6.5.5 Performance (RQ3) 165
6.5.6 Threats to Validity 165
6.6 Discussion 166
6.7 Related Work 167
6.8 Conclusions 168

7 Conclusions and Future Work 169
7.1 Expected Impact 170
7.2 Future Work 173

Bibliography 177

List of Tables

Table 2.1 Experimental subjects from which bug reports were collected. 17
Table 2.2 Impact Categories. 21
Table 2.3 Fault categories of the bug reports analyzed. Library data are shown in italics. 25
Table 2.4 Number of code-terminating failures compared to output-related failures. Library data are shown in italics. Data for DOM-related faults only are shown in parentheses. 30
Table 2.5 Impact categories of the bug reports analyzed. Library data are shown in italics. Impact categories data for DOM-related faults only are shown in parentheses. 31
Table 2.6 Error locations of the bug reports analyzed. Library data are shown in italics. 33
Table 2.7 Browser specificity of the bug reports analyzed. Library data are shown in italics. 36
Table 2.8 Average triage times (T) and fix times (F) for each experimental object, rounded to the nearest whole number. Library data are shown in italics. 38

Table 3.1 Results of the experiment on open-source web applications, assessing the accuracy of AUTOFLOX. 71
Table 3.2 Results of the experiment on production websites, assessing the robustness of AUTOFLOX (in particular, how well it works in production settings). 73
Table 3.3 Web applications and libraries in which the real bugs to which AUTOFLOX is subjected appear. 75
Table 3.4 Performance results 76

Table 4.1 List of commonly used CSS selector components. 85
Table 4.2 List of applications used in our study of common fixes. 89
Table 4.3 List of output messages. 103

Table 4.4 Accuracy results, with edit distance bound set to infinity i.e., no bound assigned. BR1 refers to the first bug report, and BR2, the second bug report (from each application). Data in parentheses are the results for when the edit distance bound is set to 5. . . 107
Table 4.5 Rank of the correct fix when suggestions are sorted by edit distance. The denominator refers to the total number of suggestions. Top ranked suggestions are in bold. 108
Table 4.6 Performance results. 109

Table 5.1 Real bugs found. The "Fault Type" column refers to the fault type number, as per Table 5.2. 132
Table 5.2 Types of faults injected. MV refers to "model variable", and CF refers to "controller function". 134
Table 5.3 Fault injection results. The size pertains to the combined lines of HTML and JavaScript code, not including libraries. 135
Table 5.4 Fault injection results per property. 138

Table 6.1 Number of real bugs found per application. The size is shown in KB, with lines of code (LOC) in parentheses. The size pertains to both the HTML and JavaScript code in each application, not including libraries. 162

List of Figures

Figure 2.1 Example that describes the error, fault, and failure of a JavaScript bug reported in Moodle. 14
Figure 2.2 Bug reports per calendar year. Note that there was one additional bug report from 2003, and two additional bug reports from 2015. 23
Figure 2.3 Pie chart of the distribution of fault categories. 24
Figure 2.4 Temporal graphs showing (a) the percent of DOM-related faults per year; (b) the percent of code-terminating failures per year. The red regression line represents DOM-related faults, while the blue regression line represents non-DOM-related faults; (c) the percent of bug reports whose error is located in the JavaScript code, per year; (d) the percent of browser-specific bugs per year; and (e) the percent of type faults per year . . . 28
Figure 2.5 Box plot for (a) triage time, and (b) fix time 37
Figure 2.6 Bar graphs showing (a) the percentage of bug reports that are type faults versus the percentage of bug reports that are not type faults and (b) the distribution of type fault categories . . . 39

Figure 3.1 Example JavaScript code fragment based on *tumblr.com*. . . . 52
Figure 3.2 Block diagram illustrating the proposed fault localization approach. 57
Figure 3.3 Example trace record for Line 5 of the running example from Figure 3.1. 59
Figure 3.4 Abridged execution trace for the running example showing the two sequences and the relevant sequence. Each trace record is appended with either a marker or the line number relative to the function. Numbers in parentheses refer to the line numbers relative to the entire JavaScript file. `root` refers to code outside a function. The line marked with a (**) is the direct DOM access, and the goal of this design is to correctly identify this line as the direct DOM access. 61
Figure 3.5 Example illustrating the approach for supporting `eval`. . . . 64

Figure 3.6 Example illustrating the mapping from the beautified version to the minified version, in the approach for supporting minified code. 66

Figure 4.1 JavaScript code of the running example. 86
Figure 4.2 The DOM state during execution of the JavaScript code in the running example. For simplicity, only the elements under `body` are shown. 87
Figure 4.3 High-level block diagram of our design. 93

Figure 5.1 HTML code of the "Search" view (search.html) 116
Figure 5.2 HTML code of the "Results" view (results.html) 117
Figure 5.3 JavaScript code of the models and controllers 118
Figure 5.4 JavaScript code of the routes 118
Figure 5.5 Block diagram of the def-use and grouping model for MVC framework identifiers. Solid arrows indicate a "defines" relation, while dashed arrows indicate a "uses" relation. Models, views, and controllers connected with the same line types form a grouping. 119
Figure 5.6 Block diagram of our approach. 124
Figure 5.7 Portion of *findIdentifiers* that updates $\omega_{\mathscr{M}}$ for every model. The other "identifier inclusion functions" are updated in a similar way. 126
Figure 5.8 Tree representing the model variables in the `Results` model of *MovieSearch*, including all the nested objects. The identifiers are shown at the top of each node, while the types are shown at the bottom. 128

Figure 6.1 Overview of our fault detection approach 148
Figure 6.2 Example of an intra-pattern consistency rule violation 153
Figure 6.3 Example of an unconditional link rule violation. The subtrees are slightly altered for simplicity. 156
Figure 6.4 Percentage of bug reports classified as an inconsistency for each MVC framework 161
Figure 6.5 Number of inconsistency categories with a particular frequency. Most inconsistency categories have just 1-2 inconsistencies in them. 161
Figure 6.6 Categories of code smells found by HOLOCRON. HC stands for Hardcoded Constants, UVU for Unsafe Value Usage, and MPI stands for Multi-Purpose Identifiers 164

Acknowledgements

First of all, I would like to extend my heartfelt gratitude to my doctoral advisors, Karthik Pattabiraman and Ali Mesbah; their guidance throughout the many years I spent in grad school has undoubtedly been one of the most crucial aspects of not only my eventual attainment of my degree, but also, my professional and personal growth. Karthik, you have been there since the beginning, having also been my MASc advisor, and I want to thank you for giving me the opportunity to work in your lab and constantly pushing me to strive harder and be better, whether it be in presenting, writing papers, and, in general, conducting research. Ali, you have been there pretty much since you joined UBC, and I would like to thank you for supporting me throughout those years, and for providing me with very valuable input in my work, especially in terms of learning more about web applications and submitting to reputable software engineering conferences/journals.

I would also like to thank my thesis committee members and examiners, including Sathish Gopalakrishnan and Matei Ripeanu, as well as my internship mentors and my CSRG colleagues for all the feedback that they have provided over the years, and for sitting through the many practice presentations I have given. I would like to say a special thanks to all my lab mates – past and present – from both the dependable systems lab and SALT lab. Even though I am somewhat notorious for not physically being in the lab as often as everyone else, the days that I did get to spend there have been thoroughly enjoyable; I appreciate all the times that you have helped me with my research, and it was a pleasure being able to provide my own input regarding your research. I am sure that the friendships we have formed will extend beyond the lab.

Of course, this section would not be complete if I neglected to acknowledge my family and my closest friends. Thank you first and foremost to my parents,

Frolin and Jeannette, who, needless to say, have provided me with immense emotional support, and laid out the foundations that helped me become who I am today. Thank you to my sister Linnette and my brother Marco, who were always there whenever I needed some time off from doing research. Thank you to all my relatives – most of whom are in the Philippines, some of whom are in Canada, and some of whom are in other parts of the world – for being a constant source of inspiration. I would also like to thank the CMPC members at Canadian Martyrs parish, especially the youth, for all their prayers and encouragement; special mention goes out to Kylo, Elijah, Mateo, Andrew, Erwin, and Paul, as well as fellow leaders Rvie and Jennifer; I thank God not only for this thesis, but also for the many abundant graces – including all these friendships and all the trials – that came along with it.

Dedication

To my friends and family.

Chapter 1

Introduction

The popularity of web applications in today's world is undeniable, with almost 50% of the entire world population visiting such applications, according to the most recent Internet World Stats survey [76]. In addition, web applications have become very important in conducting day-to-day tasks, such as search and social media, as well as large-scale corporate endeavours. As a result of this popularity, there is greater demand in ensuring that these web applications are *reliable*; in other words, the web application must contain very few functional bugs, which lead to either unexpected code termination or incorrect output. Indeed, web applications riddled with functional bugs can have potentially huge impact, especially since these applications have become the main product of some of the largest companies in the world (e.g., Google, Facebook). Further, conducting various security-critical tasks (e.g., credit card transactions, online banking) using web applications has become commonplace, which underscores the need for reliability.

Modern web applications are composed of two main components: the *server-side* and the *client-side*. In this setup, the server-side is responsible for backend operations such as responding to webpage requests and updating the database, while the client-side is responsible for providing an interface with which the user (i.e., the client) can interact with the application. In the past, the server-side provided most of the web application's functionality, with the client-side relegated to simply displaying the webpages sent by the server in response to a page request. As a result, researchers exclusively focused on analyzing and improving the web application's reliability at the server-side [61, 85, 163]. However, with the rise in popularity of Asynchronous JavaScript and XML (AJAX) development, much of

the web application's functionality is now being offloaded to the client-side [159].

Therefore, in order to ensure the reliability of the web application as a whole, the web developer also needs to ensure the reliability of the *JavaScript*[1] code at the client-side. This is because the JavaScript code is used to implement the functionality of the web application at the client-side, dictating how its various components work and interact with each other, and how the application responds to actions performed by the user. While other scripting languages exist for the web, JavaScript has become the *de facto* scripting language in this context, with over 93% of websites in the Internet using it as of May 2016 [169], and with the language topping the "Most Popular Technologies" category of the two most recent StackOverflow Developer Surveys [157]. Its popularity among web developers stems from the many advantages it presents, particularly from the standpoint of interactivity. First and foremost, the use of JavaScript is precisely what allows the developer to offload key functionalities to the client-side, which reduces the number of costly requests made to the server each time the user interacts with the webpage. Second, JavaScript's loose semantics make it very flexible, which gives developers freedom not only in terms of the types of programs they can create, but also in the way that they organize the code for these programs. Finally, JavaScript is very easy to deploy, as it is an interpreted language whose effects can immediately be seen on the browser, without undergoing complicated compilation or installation.

Having established the importance of JavaScript, we now ask the question: is the JavaScript code in most of these web applications already reliable? Frequent questions and complaints from developers about confusing aspects of the language seem to suggest otherwise. For example, there are a large number of JavaScript-related questions asked by frustrated programmers in the Q&A website StackOverflow [20]; in fact, "JavaScript" recently overtook "Java" as the most popular tag in StackOverflow, according to the latest RedMonk Programming Language Rankings [135]. Also, several books have been written specifically to help programmers delineate between the "good" and "bad" parts of JavaScript [40, 182]. Claims pertaining to reliability issues in JavaScript, however, have primarily been anecdotal in nature. In order to identify the reliability issues in JavaScript as a real problem,

[1]For simplicity, we use the term "JavaScript" to pertain to "client-side JavaScript", which is the superset of the core JavaScript language used for web application development

we need to demonstrate that JavaScript faults are both *prevalent* and *impactful*.

Unfortunately, recent research has placed studies regarding JavaScript reliability on the sidelines, mostly in favour of studies pertaining to JavaScript's security [64, 145, 175, 180], privacy [77, 124], and performance [53, 142, 144]. Further, as mentioned earlier, researchers in the past have also extensively studied the causes of web application faults at the server-side [29, 85, 136], including those that analyzed session-based workloads [61], server logs [163], and website outage incidents [138]. While useful, these server-side studies do not provide a clear view of client-side reliability, as programming practices for the latter differ significantly from those of the former. The use of the Node.js environment at the server-side bridges the gap between the server and the client to a limited extent, in that JavaScript is now also being used to program the server; however, these prior server-side studies either predate or do not consider Node.js, and client-side JavaScript code contains frequently-used APIs and features that are not available in server-side JavaScript code.

In our prior work, to answer the question posed above regarding the reliability of JavaScript, we conducted an empirical study of unhandled JavaScript exceptions logged as console messages [128]. In this study, we found that the most popular websites throw four JavaScript console messages on average – where these console messages are displayed as a result of an unhandled exception during the execution of the JavaScript code – even if the user is simply interacting with these websites normally (i.e., without attempting to break the website). However, this preliminary work only establishes the prevalence of JavaScript bugs. Ultimately, we want to know *how* these bugs appear, so that we can take steps towards preventing them, or eliminating them altogether. For this reason, the first goal of the dissertation work is *to find a way to understand both the* ***root causes*** *and the* ***impact*** *of these JavaScript bugs*.

Once we have sufficient understanding of JavaScript bugs, we can then leverage this understanding to find the best way to deal with these bugs. There are generally two approaches taken by researchers: (1) *error prevention* and (2) *fault detection*.[2] Error prevention involves finding ways to minimize the number of mis-

[2]In this dissertation, we use the term *error* to pertain to the mistake committed by the programmer when writing web application code; *fault*, to pertain to the propagation of the error into a JavaScript

takes made by a developer in the process of writing JavaScript code. Much of the work on error prevention for web applications has focused on code completion methods [2, 100, 149]. For example, Bajaj et al. introduced a tool called Dompletion [19], which helps programmers set up interactions of the JavaScript code with the Document Object Model (DOM) by autocompleting CSS selectors. In addition, most JavaScript IDEs provide some form of autocomplete functionality, including Eclipse [44], Aptana [12], and Brackets [3]. This error prevention approach is useful, but at the same time, it is limited, because errors are an inevitable part of programming. Further, not all errors are committed in the main web application code written by the developers; for instance, some errors are found in third party libraries. Hence, while error prevention helps the developer avoid certain errors, they do not give developers full assurance of the reliability of the code.

The second approach – fault detection – considers the possibility that the JavaScript code is faulty, and tries to find these faults *post facto*. There has been considerable work done on detecting syntax errors in JavaScript code, including JSLint [43] and Closure Compiler [57]; however, these tools do not detect semantic errors. There has also been considerable work on detecting faults through software testing [15, 109, 113–115, 137], as well as several record-replay tools developed for creating test cases [10, 30, 111, 151, 178]. Unfortunately, testing is limited, because it is a purely dynamic approach, which makes it susceptible to miss faults due to the large number of DOM states present in web applications. More importantly, testing alone does not suffice in improving the reliability of the code, because the developer would still have to debug any problems detected by the test cases, which is a very tedious task for developers [84, 165]. Prior research has explored ways to automate this debugging process, but most of these techniques look only for HTML validation issues, and primarily focus on the server-side [14, 32, 148]. With these issues in mind, our second goal is *to find an efficient way to automatically detect, localize, and repair JavaScript faults on the client-side*. Accomplishing this goal is challenging for various reasons:

- JavaScript is a dynamic language whose runtime behaviour is often contin-

variable, method call, or return value during execution; and *failure*, to pertain to the visible manifestation of the error in the form of an exception or output corruption. We introduce these concepts more formally in Chapter 2.

gent upon the Document Object Model (DOM) state; the DOM state pertains to the hierarchy and contents of the DOM data structure at a *certain* point during execution of the JavaScript code. Therefore, the DOM also needs to be analyzed in addition to the JavaScript code, which makes the analysis more challenging due to the large number of DOM states present in many web applications. As a result, static analysis often does not suffice, and hence, some dynamic analysis is required that keeps track of the JavaScript code's interaction with the DOM during execution;

- JavaScript executes asynchronously, and is triggered by the occurrence of events, including user-triggered events (i.e., clicks, hovers), page-triggered events (e.g., loads), and asynchronous function calls. These events may occur in different orders; although JavaScript follows a sequential execution model, it does not provide deterministic ordering, thereby complicating analysis;

- Many JavaScript frameworks have been created to facilitate the process of writing JavaScript code. These frameworks are widely used; for example, over 70% of all websites use jQuery as of 2016 [168]. Unfortunately, they also tend to complicate analysis of JavaScript code [100]. This is because different frameworks have been developed with different – and sometimes mutually exclusive – goals in mind [9]. For example, current frameworks adhere to varying programming patterns, such as Object-Oriented Programming (e.g., jQuery), Aspect-Oriented Programming (e.g., AspectJS), and Model-View-Controller (e.g., AngularJS), among others.

1.1 Research Questions

In light of what has been discussed, the overarching goal of this dissertation is *to understand the nature of JavaScript faults, and to use this understanding to develop automated techniques that improve the reliability of JavaScript-based web applications at the client-side*. To achieve this goal, we answer the research questions listed below. More specifically, the first research question concerns the understanding of JavaScript faults, while the second and third research questions concern the

localization, repair, and detection of these faults in an automated manner.

RQ1A : What are the characteristics of JavaScript faults in web applications?

In addition to establishing the prevalence of JavaScript faults, we also need to understand what caused them (i.e., the root cause) and what consequences they have (i.e., failure and impact). We addressed this research question by conducting a large-scale empirical study of 502 JavaScript bug reports, coming from 19 open-source web applications. One of the main findings of our study is that the majority of JavaScript faults relate to the interaction between the JavaScript code and the DOM; we call these *DOM-related faults*. We formally define DOM-related faults and describe our empirical study in further detail in Chapter 2.

RQ1B : How can we accurately and efficiently localize and repair JavaScript faults that appear in web applications?

Fault localization and repair are often the most time-consuming tasks in debugging. Using information gained from our bug report study, we developed two approaches that perform these tasks automatically, specifically targeting DOM-related faults. The first approach, which we describe in Chapter 3, performs automatic fault localization using a backward-slicing approach, and has been implemented in a tool called AUTOFLOX. The second approach, which we describe in Chapter 4, performs dynamic analysis of the JavaScript code to provide repair suggestions, and has been implemented in a tool called VEJOVIS.

RQ1C : Can we create a general technique that detects JavaScript faults in the presence of JavaScript frameworks? If so, how?

Before JavaScript faults can be localized and repaired, they must first be detected. In our initial attempt to address this question, we identified four types of inconsistencies that occur in JavaScript code that use the AngularJS framework [58], and we developed a static analysis approach for automatically detecting these inconsistencies. We implemented this technique in a tool called AUREBESH, the details of which we describe in Chapter 5.

AUREBESH, however, has significant limitations because it is specialized for (1) the framework that the web application uses (i.e., AngularJS), and (2) the types of inconsistencies that we identified above. Recognizing these limitations, we designed a static analysis technique that works not just for AngularJS applications, but, more generally, for web applications written using JavaScript Model-View-Controller (MVC) frameworks. This technique performs subtree pattern matching in the AST to infer the consistency rules automatically rather than require us to specify them, and is able to find inconsistencies that occur between two different programming languages. We implemented this technique in a tool called HOLOCRON, which is described in Chapter 6.

1.2 Publications

In response to the research questions enumerated in Chapter 1.1, we have published the following conference and journal papers:

- **F. Ocariza**, K. Pattabiraman, and A. Mesbah, "AutoFLox: an automatic fault localizer for client-side JavaScript," *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2012, 31-40. Best Paper Award Candidate. (Acceptance Rate: 27%) [129]

- **F. Ocariza**, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An empirical study of client-side JavaScript bugs," *IEEE/ACM International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2013, 55-64. (Acceptance Rate: 28%) [130]

- **F. Ocariza**, K. Pattabiraman, A. Mesbah, "Vejovis: suggesting fixes for JavaScript faults," *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2014, 837-847. (Acceptance Rate: 20%) [131]

- **F. Ocariza**, K. Pattabiraman, A. Mesbah, "Detecting inconsistencies in JavaScript MVC applications," *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2015, 11 pages. (Acceptance Rate: 18.5%) [132]

- **F. Ocariza**, G. Li, K. Pattabiraman, A. Mesbah, "Automatic fault localization for client-side JavaScript," *Journal of Software Testing, Verification & Reliability (STVR)*, 2016, vol. 26, issue 1, 69-88. [134]

- **F. Ocariza**, K. Bajaj, K. Pattabiraman, A. Mesbah, "A study of causes and consequences of client-side JavaScript bugs," *IEEE Transactions on Software Engineering* (in press). 17 pages. [133]

Chapter 2

Characteristics of JavaScript Faults

This chapter describes the empirical study that we conducted on JavaScript bug reports.[3] The goal of the work described by this chapter is to answer RQ1A from Chapter 1.1: *What are the characteristics of JavaScript faults in web applications?* Therefore, this empirical study allows us to understand the root causes, propagation characteristics, and impact of JavaScript bugs, which will enable us to mitigate these bugs.

We first introduce the problem and define some important terms in the next two sections, after which we will describe our experimental methodology and results.

2.1 Introduction

JavaScript contains several features that set it apart from traditional languages. First of all, JavaScript code executes under an asynchronous model. This allows event handlers to execute on demand, as the user interacts with the web application components. Secondly, much of JavaScript is designed to interact with the DOM, which, as described in Chapter 1, is a dynamic tree-like structure that includes the components in the web application and how they are organized. Using DOM API calls, JavaScript can be used to access or manipulate the components stored in the DOM, thereby allowing the web page to change without requiring a page reload.

While the above features allow web applications to be highly interactive, they also introduce additional avenues for faults in the JavaScript code. In a previous

[3]The main study in this chapter will appear at the Transactions on Software Engineering (TSE) [133]. The initial conference version appeared at the International Symposium on Empirical Software Engineering and Measurement (ESEM 2013) [130].

study [128], we collected JavaScript console messages (i.e., unhandled JavaScript exceptions) from fifty popular web applications to understand how prone web applications are to JavaScript faults and what kinds of JavaScript faults appear in these applications. While the study pointed to the prevalence of JavaScript faults, it did not explore their impact or root cause, nor did it analyze the kinds of failures they caused. Understanding the root cause and impact of the faults is vital for developers, testers, as well as tool builders to increase the reliability of web applications.

In this chapter, our goal is to discover the causes of JavaScript faults (the *error*) in web applications, and analyze their consequences (the *failure* and *impact*). Towards this goal, we conduct an empirical study of over 500 publicly available JavaScript bug reports. We choose bug reports as they typically have detailed information about a JavaScript fault and also reveal how a web application is expected to behave; this is information that would be difficult to extract from JavaScript console messages or static analysis. Further, we confine our search to bug reports that are marked "fixed", which eliminates spurious or superfluous bug reports.

A major challenge with studying bug reports, however, is that few web applications make their bug repositories publicly available. Even those that do, often classify the reports in ad-hoc ways, which makes it challenging to extract the relevant details from the report [27]. Therefore, we systematically gather bug reports and standardize their format in order to study them.

Our work makes the following main contributions:

- We collect and systematically categorize a total of **502 bug reports**, from 15 web applications and four JavaScript libraries, and put the reports in a standard format;

- We classify the JavaScript faults into multiple categories. We find that one category dominates the others, namely DOM-related JavaScript faults (more details below);

- We analyze how many of the bugs can be classified as type faults, which helps us assess the usefulness of programming languages that add strict typing systems to JavaScript, such as TypeScript [112] and Dart [22];

- We quantitatively analyze the nature (i.e., cause and consequences) and the impact of JavaScript faults;
- Where appropriate, we perform a temporal analysis of each bug report characteristic that we analyze. The results of this analysis will indicate how technological changes over the years have set the trend for these characteristics, enabling us to see if we are moving towards the right direction in improving the reliability of client-side JavaScript;
- We analyze the implications of the results on developers, testers, and tool developers for JavaScript code.

Our results show that around 68% of JavaScript faults are *DOM-related* faults, which occur as a result of a faulty interaction between the JavaScript code and the DOM. A simple example is the retrieval of a DOM element using an incorrect ID, which can lead to a null exception. Further, we find that DOM-related faults account for about 80% of the highest impact faults in the web application. Finally, we find that the majority of faults arise due to the JavaScript code rather than server side code/HTML, and that there are a few recurring programming patterns that lead to these bugs.

In addition to the above results, we also find that a small – but non-negligible – percentage (33%) of the bug reports are *type* faults, which we describe in Section 2.2. Furthermore, in our temporal analysis, we observed both downward trends in certain metrics (e.g., browser specificity) and upward trends in others (e.g., number of errors committed in the JavaScript code).

2.2 Background and Motivation

This section provides background information on the structure of modern web applications, and how JavaScript is used in such applications. We also define terms used throughout this dissertation such as JavaScript *error*, *fault*, *failure*, and *impact*. Finally, we describe the goal and motivation of our study.

2.2.1 Web Applications

Modern web applications contain three client-side components: (1) HTML code, which defines the webpage's initial elements and its structure; (2) CSS code, which defines these elements' initial styles; and (3) JavaScript code, which defines client-side functionality in the web application. These client-side components can either be written manually by the programmer, or generated automatically by the server-side (e.g., PHP) code.

The *Document Object Model* (DOM) is a dynamic tree data structure that defines the elements in the web application, their properties including their styling information, and how the elements are structured. Initially, the DOM contains the elements defined in the HTML code, and these elements are assigned the styling information defined in the CSS code. However, JavaScript can be used to manipulate this initial state of the DOM through the use of DOM API calls. For example, an element in the DOM can be accessed through its ID by calling the `getElementById()` method. The attributes of this retrieved DOM element can then be modified using the `setAttribute()` method. In addition, elements can be added to or removed from the DOM by the JavaScript code.

In general, a JavaScript method or property that retrieves elements or attributes from the DOM is called a ***DOM access method/property***. Examples of these methods/properties include `getElementById()`, `getElementsByTagName()`, and `parentNode`. Similarly, a JavaScript method or property that is used to update values in the DOM (e.g., its structure, its elements' properties, etc.) is called a ***DOM update method/property***. Examples include `setAttribute()`, `innerHTML`, and `replaceChild()`. Together, the access and update methods/properties constitute the DOM API.

2.2.2 JavaScript Bugs

JavaScript is particularly prone to faults, as it is a weakly typed language, which makes the language flexible but also opens the possibility for untyped variables to be (mis)used in important operations. In addition, JavaScript code can be dynamically created during execution (e.g., by using `eval`), which can lead to faults that are only detected at runtime. Further, JavaScript code interacts extensively with

the DOM, which makes it challenging to test/debug, and this leads to many faults as we find in our study.

JavaScript Bug Sequence. In this dissertation, we use the term *bug* as a catch-all term that pertains to an undesired behaviour of the web application's functionality. The following sequence describes the progression of a JavaScript bug, and the terms we use to describe this sequence:

1. The programmer makes a mistake at some point in the code being written or generated. These ***errors*** can range from simple mistakes such as typographical errors or syntax errors, to more complicated mistakes such as errors in logic or semantics. The error can be committed in the JavaScript code, or in other locations such as the HTML code or server-side code (e.g., PHP).

2. The error can propagate, for instance, into a JavaScript variable, the parameter or assignment value of a JavaScript method or property, or the return value of a JavaScript method during JavaScript code execution. Hence, by this point, the error has propagated into a ***fault***.

3. The fault either directly causes a JavaScript exception (***code-terminating failure***) or a corruption in the output (***output-related failure***). This is called the ***failure***.

Figure 2.1 shows a real-world example of the error, fault, and failure associated with a JavaScript bug report from the Moodle web application. Note that for output-related failures, the pertinent output can be one or a combination of many things, including the DOM, server data, or important JavaScript variables. We will be using the above error-fault-failure model to classify JavaScript bugs, as described in Section 2.3.

DOM-Related Faults. We define a DOM-related fault as follows:

Definition 1 (DOM-Related Fault) *A JavaScript bug $\mathscr{B}$ is considered to have propagated into a* DOM-related fault *if the corresponding error causes a DOM API method $\mathscr{DA}_m$ to be called (or causes an assignment to a DOM API property $\mathscr{DA}_p$ to be made), such that a parameter $\mathscr{P}$ passed to $\mathscr{DA}_m$ (or a value $\mathscr{A}$ assigned to $\mathscr{DA}_p$) is incorrect.*

Error: The programmer forgets to initialize the value of the `cmi.evaluation.comments` variable.

Fault: The `cmi.evaluation.comments` variable – which is uninitialized and hence has the value `null` – is used to access a property `X` (i.e., `cmi.evaluation.comments.X`) during JavaScript execution.

Failure: Since `cmi.evaluation.comments` is `null`, the code attempting to access a property through it leads to a null exception, which terminates JavaScript execution.

Figure 2.1: Example that describes the error, fault, and failure of a JavaScript bug reported in Moodle.

In other words, if a JavaScript error propagates into the parameter of a DOM access/update method or to the assignment value for a DOM access/update property – thereby causing an incorrect retrieval or an incorrect update of a DOM element – then the error is said to have propagated into a ***DOM-related fault***. For example, if an error eventually causes the parameter of the DOM access method `getElementById()` to represent a nonexistent ID, and this method is called during execution with the erroneous parameter, then the error has propagated into a DOM-related fault. However, if the error does not propagate into a DOM access/update method/property, the error is considered to have propagated into a ***non-DOM-related fault***. Note that based on this definition, the presence of a large number of DOM interactions in the JavaScript code does not necessarily imply the presence of a large percentage of DOM-related faults, as not all errors would necessarily propagate to any DOM API method calls in the code.

Type Faults. We are also interested in determining the prevalence of *type faults*, which we define as follows:

Definition 2 (Type Fault) *A JavaScript bug $\mathscr{B}$ is considered to have propagated into a* type fault *if there exists a statement $\mathscr{L}$ in the JavaScript code such that, during the execution of the code that reproduces $\mathscr{B}$, the statement $\mathscr{L}$ references an*

expression or variable $\mathcal{E}$ that it inherently assumes to be of type t_1, but $\mathcal{E}$'s actual type at runtime is t_2 (with $t_2 \neq t_1$).

In other words, a type fault occurs if the JavaScript code erroneously assumes during execution that a certain value is of a certain type (e.g., using the string API on a variable that is a number at runtime). Note that our comparison of types bears some similarities to Pradel et al.'s definition of consistent types [141]. In particular, we both make a distinction between different "type categories", such as primitive types and custom types; we describe this in further detail in Section 2.3.4.

Severity. While the appearance of a failure is clear-cut and mostly objective (i.e., either an exception is thrown or not; either an output contains a correct value or not), the severity of the failure is subjective, and depends on the context in which the web application is being used. For example, an exception may be classified as non-severe if it happens in a "news ticker" web application widget; but if the news ticker is used for something important – say, stocks data – the same exception may now be classified as severe. In this dissertation, we will refer to the severity as the ***impact*** of the failure. We determine impact based on a qualitative analysis of the web application's content and expected functionality.

2.2.3 Goal and Motivation

Our overall goal in this chapter is *to understand the sources and the impact of client-side JavaScript faults in web applications*. To this end, we conduct an empirical study of JavaScript bug reports in deployed web applications. There are several factors that motivated us to pursue this goal. First, understanding the root cause of JavaScript faults could help make developers aware of programming pitfalls to be avoided, and the results could pave the way for better JavaScript debugging techniques. Second, analyzing the impact could steer developers' and testers' attention towards the highest impact faults, thereby allowing these faults to be detected early. Finally, we have reason to believe that JavaScript faults' root causes and impacts differ from those of traditional languages because of JavaScript's permissive nature and its many distinctive features (e.g., event-driven model; interaction with the DOM; dynamic code creation; etc.)

Other work has studied JavaScript faults through console messages or through

static analysis [64, 65, 79, 186]. However, bug reports contain detailed information about the root cause of the faults and the intended behaviour of the application, which is missing in these techniques. Further, they typically contain the fix associated with the fault, which is useful in further understanding it, for example, to determine fix times.

2.3 Experimental Methodology

We describe our methodology for the empirical study on JavaScript faults. First, we enumerate the research questions that we want to answer. Then we describe the web applications we studied and how we collected their bug reports. All our collected empirical data is available for download.[4]

2.3.1 Research Questions

To achieve our goal, we address the following research questions through our bug report study:

RQ1: What categories of faults exist among reported JavaScript faults, and how prevalent are these fault categories?

RQ2: What is the nature of failures stemming from JavaScript faults? What is the impact of the failures on the web applications?

RQ3: What is the root-cause of JavaScript faults? Are there specific programming practices that lead to JavaScript faults?

RQ4: Do JavaScript faults exhibit browser-specific behaviour?

RQ5: How long does it take to triage a JavaScript fault reported in a bug report and assign it to a developer? How long does it take programmers to fix these JavaScript faults?

RQ6: How prevalent are type faults among the reported bugs?

RQ7: How have the characteristics of JavaScript faults – particularly those analyzed in the above research questions – varied over time?

[4] http://ece.ubc.ca/~frolino/projects/js-bugs-study/

Table 2.1: Experimental subjects from which bug reports were collected.

Application ID	Application Name	Version Range	Type	Description	Size of JS Code (KB)	Bug Report Search Filter	# of Reports Collected
1	Moodle	1.9-2.3.3	Web Application	Learning Management	352	(**Text** contains *javascript* OR *js* OR *jquery*) AND (**Issue type** is *bug*) AND (**Status** is *closed*) - Number of Results: 1209	30
2	Joomla	3.x	Web Application	Content Management	434	(**Category** is *JavaScript*) AND (**Status** is *Fixed*) - Number of Results: 62	11
3	WordPress	2.0.6-3.6	Web Application	Blogging	197	((**Description** contains *javascript* OR *js*) OR (**Keywords** contains *javascript* OR *js*)) AND (**status** is *closed*) - Number of Results: 875	30
4	Drupal	6.x-7.x	Web Application	Content Management	213	(**Text** contains *javascript* OR *js* OR *jQuery*) AND (**Category** is *bug report*) AND (**Status** is *closed(fixed)*) - Number of Results: 608	30
5	Roundcube	0.1-0.9	Web Application	Webmail	729	((**Description** contains *javascript* OR *js*) OR (**Keywords** contains *javascript* OR *js*)) AND (**status** is *closed*) - Number of Results: 234	30
6	WikiMedia	1.16-1.20	Web Application	Wiki Software	160	(**Summary** contains *javascript*) AND (**Status** is *resolved*) AND (**Resolution** is *fixed*) - Number of Results: 49	30
7	TYPO3	1.0-6.0	Web Application	Content Management	2252	(**Status** is *resolved*) AND (**Tracker** is *bug*) AND (**Subject** contains *javascript*) (Only one keyword allowed) - Number of Results: 81	30
8	TaskFreak	0.6.x	Web Application	Task Organizer	74	(**Search keywords** contain *javascript* OR *js*) AND (**User** is *any user*) - Number of Results: 57	6
9	Horde	1.1.2-2.0.3	Web Application	Webmail	238	(**Type** is *bug*) AND (**State** is *resolved (bug)*) AND ((**Summary** contains javascript) OR (**Comments** contain *javascript*)) - Number of Results: 300	30
10	FluxBB	1.4.3-1.4.6	Web Application	Forum System	8	(**Type** is *bug*) AND (**Status** is *fixed*) AND (**Search keywords** contain *javascript*) - Number of Results: 8	5
11	LimeSurvey	1.9.x-2.0.x	Web Application	Survey Maker	442	(**Status** is *closed*) AND (**Resolution** is *fixed*) AND (**Text** contains *javascript*) - Number of Results: 252	30
12	DokuWiki	2009-2014	Web Application	Wiki Software	446	(**Status** is *all closed tasks*) AND (**Task type** is *bug report*) AND (**Text** contains *javascript* OR *js*) - Number of Results: 159	30
13	phpBB	3.0.x	Web Application	Forum System	176	(**Status** is *closed*) AND (**Resolution** is *fixed*) AND (**Text** contains *javascript* OR *js*) - Number of Results: 112	30
14	MODx	1.0.3-2.3.0	Web Application	Content Management	1229	(**Type** is *issue*) AND (**Status** is *closed*) AND (**Text** contains *javascript* OR *js*) - Number of Results: 219	30
15	EZ Systems	3.5-5.3	Web Application	Content Management	180	(**Issue type** is *bug*) AND (**Status** is *closed*) AND (**Resolution** is *fixed*) AND (**Text** contains *javascript* OR *js*) - Number of Results: 229	30
16	jQuery	1.0-1.9	Library	—	94	(**Type** is *bug*) AND (**Resolution** is *fixed*) - Number of Results: 2421	30
17	Prototype.js	1.6.0-1.7.0	Library	—	164	(**State** is *resolved*) - Number of Results: 142	30
18	MooTools	1.1-1.4	Library	—	101	(**Label** is *bug*) AND (**State** is *closed*) - Number of Results: 52	30
19	Ember.js	1.0-1.1	Library	—	745	(**Label** is *bug*) AND (**State** is *closed*) - Number of Results: 347	30

2.3.2 Experimental Objects

To ensure representativeness, we collect and categorize bug reports from a wide variety of web applications and libraries. Each object is classified as either a web application or a JavaScript library. In preliminary work [130], we initially made this distinction to see if there are any differences between JavaScript faults in web applications and those in libraries. We did not, however, find any substantial differences after performing our analysis; therefore, our selection of additional experimental objects compared to this preliminary work was not influenced by this distinction, and we do not report them separately. In total, we collected and analyzed 502 bug reports from 15 web applications and four libraries.

Table 2.1 lists the range of the software versions considered for each experimental object. The web applications and libraries were chosen based on several factors, including their popularity, their prominent use of client-side JavaScript, and the descriptiveness of their bug reports (i.e., the more information its bug reports convey, the better). Another contributing factor is the availability of a bug repository for the web application or library, as such repositories were not always made public. In fact, finding web applications and libraries that satisfied these criteria was a major challenge in this study.

2.3.3 Collecting the Bug Reports

For each web application bug repository, we collect a total of **min**{30, *NumJSReports*} JavaScript bug reports, where *NumJSReports* is the total number of JavaScript bug reports in the repository. We chose 30 as the maximum threshold for each repository to balance analysis time with representativeness. To collect the bug reports for each repository, we perform the following steps:

Step 1 Use the filter/search tool available in the bug repository to narrow down the list of bug reports. The filters and search keywords used in each bug repository are listed in Table 2.1. In general, where appropriate, we used "javascript" and "js" as keywords to narrow down the list of bug reports (in some bug repositories, the keyword "jQuery" was also used to narrow down the list even further). Further, to reduce spurious or superfluous reports, we only considered bug reports with resolution "fixed", and type "bug" or "de-

fect" (i.e., bug reports marked as "enhancements" were neglected). Table 2.1 also lists the number of search results after applying the filters in each bug repository. The bug report repositories were examined between January 30, 2013 and March 13, 2013 (applications #1-8, 16-19), and between February 30, 2015 and March 25, 2015 (applications #9-15).

Step 2 Once we have the narrowed-down list of bug reports from Step 1, we manually examine each report in the order in which it was retrieved. Since the filter/search features of some bug tracking systems were not as descriptive (e.g., the TYPO3 bug repository only allowed the user to search for bug reports marked "resolved", but not "fixed"), we also had to manually check whether the bug report satisfied the conditions described in Step 1. If the conditions are satisfied, we analyzed the bug report; otherwise, we discarded it. We also discarded a bug report if its fault is found to *not* be JavaScript-related – that is, the error does not propagate into any JavaScript code in the web application. This step is repeated until $\mathbf{min}\{30, \mathit{NumJSReports}\}$ reports have been collected in the repository. The number of bug reports we collected for each bug repository is shown in Table 2.1. Note that three applications had fewer than 30 reports that satisfied the above criteria, namely Joomla, TaskFreak, and FluxBB. For all remaining applications, we collected 30 bug reports each.

Step 3 For each report, we created an XML file that describes and classifies the error, fault, failure, and impact of the JavaScript bug reported. The XML file also describes the fix applied for the bug. Typically this data is presented in raw form in the original bug report, based on the bug descriptions, developer discussions, patches, and supplementary data; hence, we needed to read through, understand, and interpret each bug report in order to extract all the information included in the corresponding XML file. We also include data regarding the date and time of each bug being assigned and fixed in the XML file. We have made these bug report XML files publicly available for reproducibility.[4]

2.3.4 Analyzing the Collected Bug Reports

The collected bug report data, captured in the XML files, enable us to qualitatively and quantitatively analyze the nature of JavaScript bugs.

Fault Categories. To address RQ1, we classify the bug reports according to the following fault categories that were identified through an initial pilot study:

- **Undefined/Null Variable Usage**: A JavaScript variable that has a `null` or `undefined` value – either because the variable has not been defined or has not been assigned a value – is used to access an object property or method. *Example*: The variable `x`, which has not been defined in the JavaScript code, is used to access the property `bar` via `x.bar`.

- **Undefined Method**: A call is made in the JavaScript code to a method that has not been defined. *Example*: The undefined function `foo()` is called in the JavaScript code.

- **Incorrect Method Parameter**: An unexpected or invalid value is passed to a native JavaScript method, or assigned to a native JavaScript property. *Example*: A string value is passed to the JavaScript Date object's `setDate()` method, which expects an integer. Another example is passing an ID string to the DOM method `getElementById()` that does not correspond to any IDs in the DOM. Note that this latter example is a type of *DOM-related fault*, which is a subcategory of Incorrect Method Parameter faults where the method/property is a DOM API method/property (as defined in Section 2.2.2).

- **Incorrect Return Value**: A user-defined method is returning an incorrect return value even though the parameter(s) is/are valid. *Example*: The user-defined method `factorial(3)` returns 2 instead of 6.

- **Syntax-Based Fault**: There is a syntax error in the JavaScript code. *Example*: There is an unescaped apostrophe character in a string literal that is defined using single quotes.

- **Other**: Errors that do not fall into the above categories. *Example*: There is a naming conflict between methods or variables in the JavaScript code.

Table 2.2: Impact Categories.

Type	Description	Examples
1	Cosmetic	Table is not centred; header is too small
2	Minor functionality loss	Cannot create e-mail addresses containing apostrophe characters, which are often only used by spammers
3	Some functionality loss	Cannot use delete button to delete e-mails, but delete key works fine
4	Major functionality loss	Cannot delete e-mails at all; cannot create new posts
5	Data loss, crash, or security issue	Browser crashes/hangs; entire application unusable; save button does not work and prevents user from saving considerable amount of data; information leakage

Note that we do not find instances where a bug report belongs to multiple fault categories, and hence, we disregard the "Multiple" category in our description of the results.

Failure Categories. The failure category refers to the observable consequence of the fault. For each bug report, we marked the failure category as either *Code-terminating* or *Output-related*, as defined in Section 2.2.2. This categorization helps us answer RQ2.

Impact Categories. We manually classify the impact of a JavaScript bug according to the classification scheme used by Bugzilla.[5] This scheme is applicable to any software application, and has also been used in other studies [28, 114]. Table 2.2 shows the categories. This categorization helps us answer RQ2.

Error Locations. The error location refers to the code unit or file where the error was made (either by the programmer or the server-side program generating the JavaScript code). For each bug report, we marked the error location as one of the following: (1) *JavaScript code (JS)*; (2) *HTML Code (HTML)*; (3) *Server-side code (SSC)*; (4) *Server configuration file (SCF)*; (5) *Other (OTH)*; and (6) *Multiple error locations (MEL)*. In cases where the error location is marked as either OTH or MEL, the location(s) is/are specified in the error description. The error locations

[5] http://www.bugzilla.org/

were determined based on information provided in the bug report description and comments. This categorization helps us answer RQ3.

Browser Specificity. In addition, we also noted whether a certain bug report is browser-specific – that is, the fault described in the report only occurs in one or two web browsers, but not in others – to help us answer RQ4.

Time for Fixing. To answer RQ5, we define the ***triage time*** as the time it took a bug to get assigned to a developer, from the time it was reported (or, if there is no "assigned" marking, the time until the first comment is posted in the report). We also define ***fix time*** as the time it took the corresponding JavaScript fault to get marked as "fixed", from the time it was triaged. We recorded the time taken for each JavaScript bug report to be triaged, and for the report to be fixed. Other studies have classified bugs on a similar basis [11, 104]. Further, we calculate times based on the calendar date; hence, if a bug report was triaged on the same date as it was reported, the triage time is recorded as 0.

Type Faults. Programming languages such as TypeScript [112] and Dart [22] aim to minimize JavaScript faults through a stricter typing system for JavaScript; hence, the main fault model targeted by these languages are type faults. In our work, we assess the usefulness of strong typing in such languages by examining the prevalence of type faults among the bug reports, which addresses RQ6.

In our analysis, we consider four different "type categories", listed below.

- **Primitive types [P]**: Values of type `string`, `boolean`, or `number`
- **Null/undefined types [Nu]**: `null` or `undefined`
- **Native "class" types [Nc]**: Objects native to client-side JavaScript (e.g., `Function`, `Element`, etc.)
- **Custom "class" types [C]**: User-defined objects

We first categorize a bug report as either a *type fault* or *not a type fault*, based on the definition provided in Section 2.2.2. For each bug report categorized as a type fault, we further categorize it as belonging to one of 16 subcategories, each of the form, "*<type category> expected, but <type category> actual*," which, for

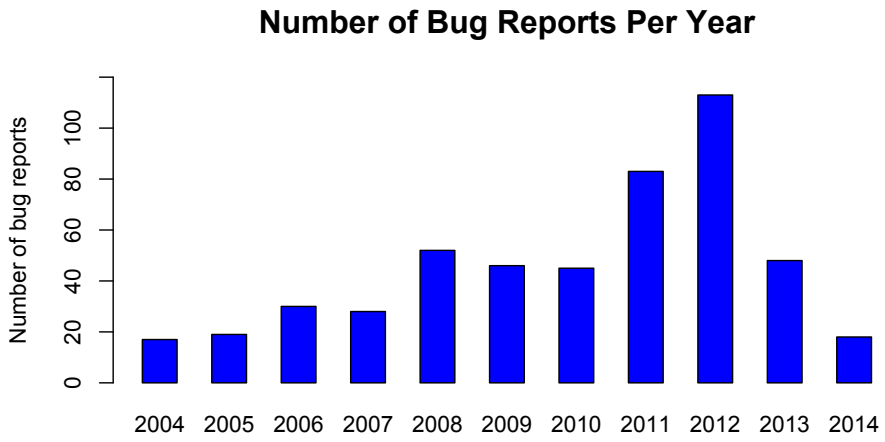


Figure 2.2: Bug reports per calendar year. Note that there was one additional bug report from 2003, and two additional bug reports from 2015.

simplicity, we abbreviate as <*type category*>*E*<*type category*>*A*. For example, a type fault belongs to the *PEPA* category if a value is expected to be of a certain primitive type in the JavaScript code – say `boolean` – but its actual type at runtime is a different primitive type – say `string`. Similarly, an *NcENuA* type fault occurs if a value is expected to be of a native "class" type, but its actual type at runtime is `null` or `undefined`.

Note that type comparisons are made in a way similar to Pradel et al.'s method for detecting inconsistent types [141]. Finally, note that we classify a bug report as "not a type fault" if we do not find a statement $\mathscr{L}$ that satisfies the definition given in Section 2.2.2.

Temporal Analysis. Where appropriate, for every data point we collect to answer the first six research questions, we analyze how these data have changed over time. The purpose of this analysis – which answers RQ7 – is to help us understand and speculate how historical factors such as browser improvements, the appearance of new JavaScript frameworks, and the rise in popularity of "Q&A" websites (e.g., StackOverflow), among others, have helped in the improvement of the reliability of client-side JavaScript, or degraded it.

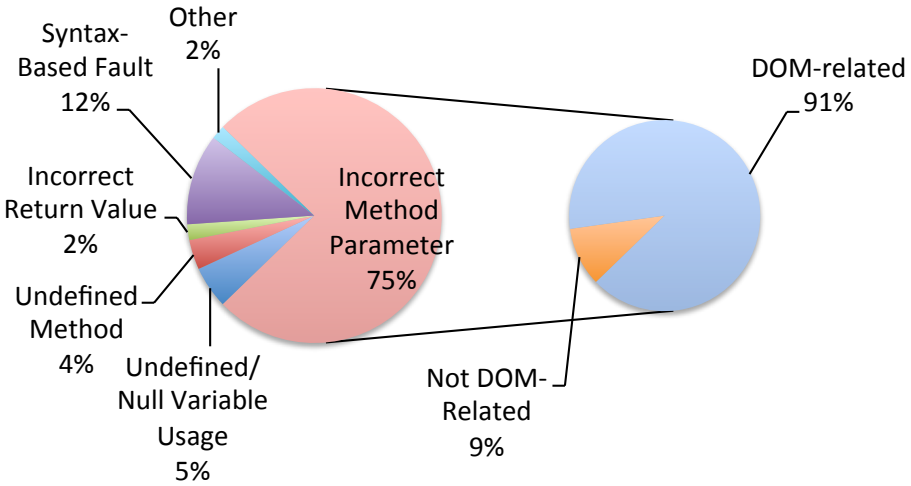


Figure 2.3: Pie chart of the distribution of fault categories.

To perform this temporal analysis, we mark each bug report with the year in which it was reported. In our specific case, the bug reports we collected were reported over a period of 13 calendar years, from 2003 to 2015. However, we neglect the years 2003 and 2015 in our analysis, since fewer than 3 bug reports were marked with each of these years; hence, we only consider 11 calendar years in our analysis (i.e., 2004 to 2014), each of which corresponds to at least 17 bug reports. The number of bug reports in each of these calendar years is shown in Figure 2.2.

2.4 Results

In this section, we present the results of our empirical study on JavaScript bug reports. The subsections are organized according to the research questions in Section 2.3.1.

Table 2.3: Fault categories of the bug reports analyzed. Library data are shown in italics.

Application	Undefined/Null Variable Usage	Undefined Method	Incorrect Return Value	Syntax-Based Fault	Other	Incorrect Method Parameter			Percent DOM-related
						DOM-related	Not DOM-related	Total	
Moodle	3	3	0	7	0	15	2	17	50%
Joomla	1	0	0	3	0	6	1	7	55%
WordPress	1	2	0	3	1	21	2	23	70%
Drupal	0	1	0	5	0	23	1	24	77%
Roundcube	3	0	0	4	0	22	1	23	73%
WikiMedia	2	4	0	5	0	15	4	19	50%
TYPO3	2	2	0	7	1	18	0	18	60%
TaskFreak	1	0	0	0	0	4	1	5	67%
Horde	1	0	0	4	0	24	1	25	80%
FluxBB	0	0	0	0	0	5	0	5	100%
LimeSurvey	0	0	0	4	0	24	2	26	80%
DokuWiki	2	0	0	3	2	20	3	23	67%
phpBB	3	1	0	3	0	23	0	23	77%
MODx	1	1	0	3	1	21	3	24	70%
EZ Systems	2	2	0	7	1	17	1	18	57%
jQuery	*0*	*0*	*1*	*0*	*0*	*26*	*3*	*29*	*87%*
Prototype.js	*0*	*1*	*2*	*0*	*0*	*22*	*5*	*27*	*73%*
MooTools	*3*	*1*	*3*	*0*	*1*	*19*	*3*	*22*	*63%*
Ember.js	*2*	*1*	*4*	*0*	*2*	*16*	*5*	*21*	*53%*
Overall	27	19	10	58	9	341	38	379	68%

2.4.1 Fault Categories

Table 2.3 shows the breakdown of the fault categories in our experimental objects. The pie chart in Figure 2.3 shows the overall percentages. As seen from the table and the figure, approximately 75% of JavaScript faults belong to the "Incorrect Method Parameter" category. This suggests that most JavaScript faults result from errors related to setting up the parameters of native JavaScript methods, or the values assigned to native JavaScript properties.

***Finding 1**: "Incorrect Method Parameter" faults account for around 75% of JavaScript faults.*

In our earlier study of unhandled JavaScript exceptions [128] and a preliminary version of our study on the fault localization of JavaScript bugs [129], we also noticed many "Incorrect Method Parameter" faults, but their prevalence was not quantified. Interestingly, we also observed in these earlier studies that many of the methods and properties affected by these faults are DOM methods/properties – in other words, DOM-related faults, as defined in Section 2.2. Based on these prior observations, we became curious as to how many of these "Incorrect Method Parameter" faults are DOM-related.

We further classified the "Incorrect Method Parameter" faults based on the methods/properties in which the incorrect values propagated, and found that 91% of these faults are DOM-related faults. This indicates that among *all* JavaScript faults, approximately 68% are DOM-related faults (see right-most pie chart in Figure 2.3). We find that DOM-related faults range from 50 to 100% of the total JavaScript faults across applications, as seen on the last column of Table 2.3.

Lastly, in order to assess how many of the DOM-related faults result from developers' erroneous understanding of the DOM, we make a distinction between *strong DOM-related faults* and *weak DOM-related faults*. A DOM-related fault is classified as *strong* if the incorrect parameter passed to the DOM method/property represents an inconsistency with the DOM; this includes, for example, cases where an incorrect or non-existent selector/ID is passed to a DOM method. Otherwise, the DOM-related fault is classified as weak; this includes, for example, cases where

the wrong text value is assigned to `innerHTML`, or the wrong attribute value is assigned to an attribute. Overall, we find that strong DOM-related faults make up 88% of all DOM-related faults. This result therefore strongly indicates that *most DOM-related faults occur as a result of an inconsistency between what the developers think is the DOM contents, and what actually is the DOM's contents.*

> ***Finding 2****: DOM-related faults account for 91% of "Incorrect Method Parameter" faults. Hence, the majority – around 68% – of JavaScript faults are DOM-related, and the majority – around 88% – of these DOM-related faults are of the "strong" variety.*

Temporal Analysis. Figure 2.4a shows a scatter plot of the percentage of bug reports marked as DOM-related per calendar year. The linear regression line has a downward slope, indicating an overall decrease in the percentage of DOM-related faults over the years. However, this decrease is very small, i.e., a decrease of 7 percentage points, which corresponds to a 10% decrease. Hence, the percentage of DOM-related faults reported in the repositories we analyzed has remained rather consistent over the years.

> ***Finding 3****: The percentage of DOM-related faults among all JavaScript faults has experienced only a very small decrease (10%) over the past ten years.*

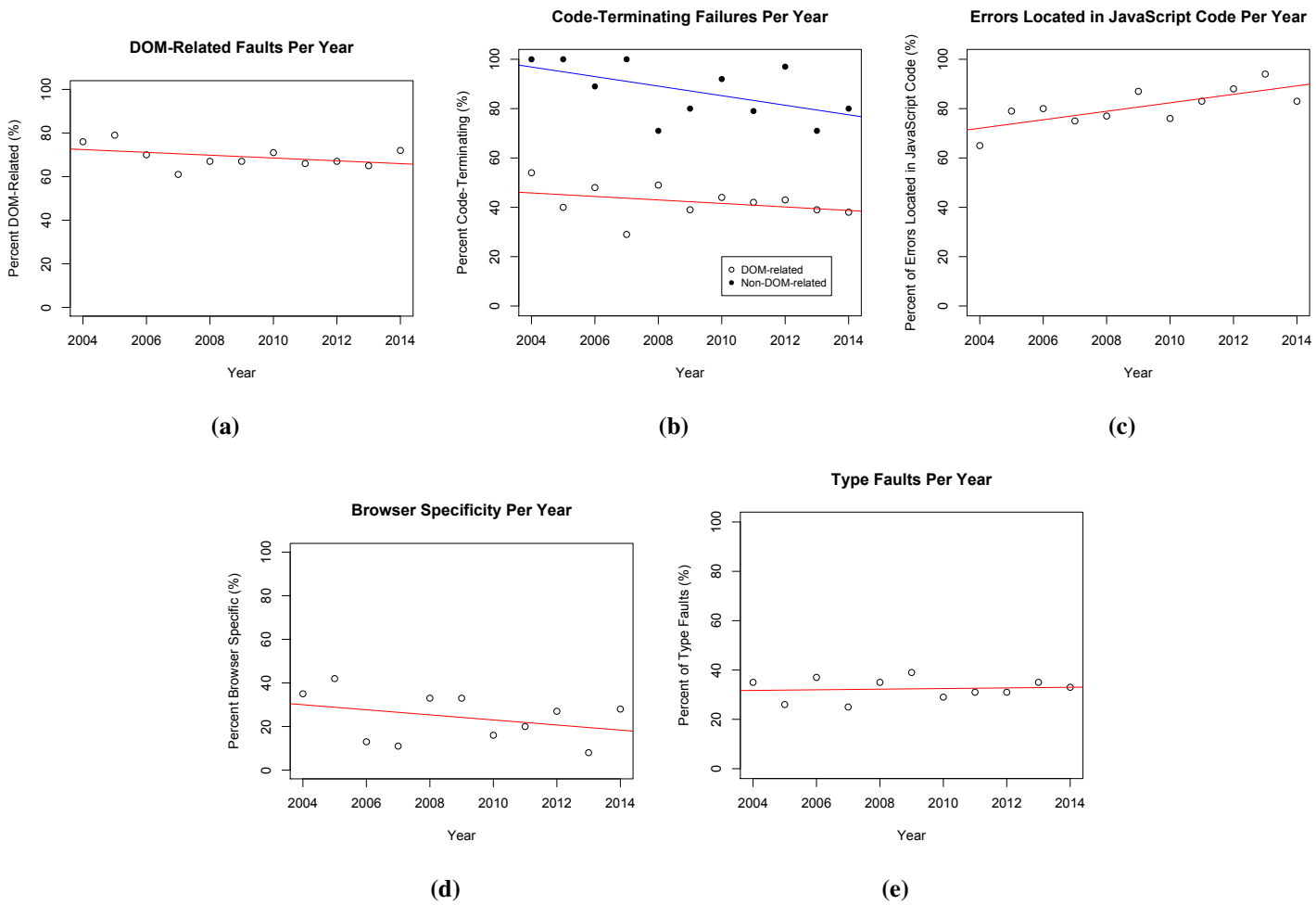


Figure 2.4: Temporal graphs showing (a) the percent of DOM-related faults per year; (b) the percent of code-terminating failures per year. The red regression line represents DOM-related faults, while the blue regression line represents non-DOM-related faults; (c) the percent of bug reports whose error is located in the JavaScript code, per year; (d) the percent of browser-specific bugs per year; and (e) the percent of type faults per year

2.4.2 Consequences of JavaScript Faults

We now show the failure categories of the bug reports we collected, as well as the impact of the JavaScript faults that correspond to the reports.

Failure Categories. Table 2.4 shows the distribution of failure categories amongst the collected reports; all faults are classified as either leading to a code-terminating failure or an output-related failure (these terms are defined in Section 2.3.4). Note that the goal of this classification is *not* to assess the severity of the bugs, but rather, to determine the nature of the failure (i.e., whether there is a corresponding error message or not). Making this distinction will be helpful for developers of fault localization tools, for example, as these error messages can naturally act as a starting point for analysis of the bug.

As the table shows, around 57% of JavaScript faults are code-terminating, which means that in these cases, an exception is thrown. Faults that lead to code-termination are generally easier to detect, since the exceptions have one or more corresponding JavaScript error message(s) (provided the error can be reproduced during testing). On the other hand, output-related failures do not have such messages; they are typically only detected if the user observes an abnormality in the behaviour of the application.

Since the majority of JavaScript faults are DOM-related, we explored how these failure categories apply to DOM-related faults. Interestingly, we found that for DOM-related faults, most failures are output-related (at 57%), while for non-DOM-related faults, most failures are code-terminating (at 86%). This result suggests that DOM-related faults may be more difficult to detect than non-DOM-related faults, as most of them do not lead to exceptions or error messages.

***Finding 4**: While most non-DOM-related JavaScript faults lead to exceptions (around 86%), only a relatively small percentage (43%) of DOM-related faults lead to such exceptions.*

Temporal Analysis. A scatter plot of the percentage of code-terminating failures per year is shown in Figure 2.4b. This figure shows the percentage of code-terminating failures, over time, for each fault category (i.e., DOM-related vs non-

Table 2.4: Number of code-terminating failures compared to output-related failures. Library data are shown in italics. Data for DOM-related faults only are shown in parentheses.

Application	Code-terminating	Output-related
Moodle	21 (8)	9 (7)
Joomla	8 (3)	3 (3)
WordPress	11 (3)	19 (18)
Drupal	12 (5)	18 (18)
Roundcube	18 (11)	12 (11)
WikiMedia	19 (4)	11 (11)
TYPO3	21 (9)	9 (9)
TaskFreak	3 (2)	3 (2)
Horde	17 (11)	13 (13)
FluxBB	3 (3)	2 (2)
LimeSurvey	11 (7)	19 (17)
DokuWiki	9 (4)	21 (16)
phpBB	19 (12)	11 (11)
MODx	21 (14)	9 (7)
EZ Systems	27 (15)	3 (2)
jQuery	*17 (13)*	*13 (13)*
Prototype.js	*10 (7)*	*20 (15)*
MooTools	*21 (10)*	*9 (9)*
Ember.js	*16 (5)*	*14 (11)*
Overall	284 (146)	218 (195)

DOM-related). In both cases, there is a net decrease in the number of code-terminating failures, with a slightly larger decrease for non-DOM-related faults. The overall decrease in code-terminating failures may be explained in part by the improvements in error consoles as well as the introduction of tools such as Firebug,[6] both of which facilitate the process of debugging code-terminating failures within the web browser.

***Finding 5**: The percentage of code-terminating failures experienced a net decrease for both DOM-related faults (17% decrease) and non-DOM-related faults (21% decrease) over the past ten years.*

Impact Categories. The impact indicates the severity of the failure. Hence, we also classify bug reports based on impact categories as defined in Section 2.3.4 (i.e., Type 1 has lowest severity, and Type 5 has highest severity).

The impact category distribution for each web application and library is shown

[6] http://getfirebug.com/

Table 2.5: Impact categories of the bug reports analyzed. Library data are shown in italics. Impact categories data for DOM-related faults only are shown in parentheses.

Application	Type 1	Type 2	Type 3	Type 4	Type 5
Moodle	10 (5)	12 (5)	0 (0)	6 (3)	2 (2)
Joomla	2 (2)	2 (0)	4 (2)	2 (1)	1 (1)
WordPress	4 (4)	7 (3)	12 (9)	3 (2)	4 (3)
Drupal	3 (3)	2 (1)	17 (12)	1 (1)	7 (6)
Roundcube	2 (2)	5 (4)	14 (9)	5 (3)	4 (4)
WikiMedia	2 (1)	8 (6)	15 (6)	1 (0)	4 (2)
TYPO3	0 (0)	4 (2)	20 (13)	5 (2)	1 (1)
TaskFreak	2 (1)	1 (1)	1 (0)	2 (2)	0 (0)
Horde	6 (3)	7 (6)	13 (11)	2 (2)	2 (2)
FluxBB	1 (1)	1 (1)	2 (2)	1 (1)	0
LimeSurvey	5 (4)	4 (2)	19 (16)	1 (1)	1 (1)
DokuWiki	5 (3)	7 (3)	16 (13)	2 (1)	0 (0)
phpBB	3 (0)	5 (4)	16 (14)	6 (5)	0 (0)
MODx	2 (1)	5 (3)	18 (14)	3 (2)	2 (1)
EZ Systems	1 (1)	2 (0)	25 (15)	2 (1)	0 (0)
jQuery	*3 (3)*	*13 (13)*	*1 (1)*	*11 (8)*	*2 (1)*
Prototype.js	*0 (0)*	*7 (6)*	*19 (12)*	*2 (2)*	*2 (2)*
MooTools	*0 (0)*	*16 (8)*	*10 (8)*	*3 (3)*	*1 (0)*
Ember.js	*2 (0)*	*15 (10)*	*10 (4)*	*2 (1)*	*1 (1)*
Overall	53 (34)	123 (78)	232 (161)	60 (41)	34 (27)

in Table 2.5. Most of the bug reports were classified as having Type 3 impact (i.e., some functionality loss). Type 1 and Type 5 impact faults are the fewest, with 53 and 34 bug reports, respectively. Finally, Type 2 and Type 4 impact faults are represented by 123 and 60 bug reports, respectively. The average impact of the collected JavaScript bug reports is close to the middle, at 2.80, which is in line with other studies [34].

Table 2.5 also shows the impact distribution for DOM-related faults in parentheses. As seen in the table, each impact category is comprised primarily of DOM-related faults. Further, almost 80% (27 out of 34) of the highest severity faults (i.e., Type 5 faults) are DOM-related. Additionally, 13 of the 19 experimental subjects contain at least one DOM-related fault with Type 5 impact. This result suggests that *high severity failures often result from DOM-related faults*. We find that these high-impact faults broadly fall into three categories.

1. **Application/library becomes unusable**. This occurs because an erroneous feature is preventing the user from using the rest of the application, particularly in DOM-related faults, which make up 11 of the 15 faults in this cate-

gory. For example, one of the faults in Drupal prevented users from logging in (due to incorrect attribute values assigned to the username and password elements), so the application could not even be accessed.

2. **Data loss**. Once again, this is particularly true for DOM-related faults, which account for 13 out of the 14 data-loss-causing faults that we encountered. One example comes from Roundcube; in one of the bug reports, the fault causes an empty e-mail to be sent, which causes the e-mail written by the user to be lost. As another example, a fault in WordPress causes server data (containing posts) to be deleted automatically without confirmation.

3. **Browser hangs and information leakage**. Hangs often occur as a result of a bug in the browser; the Type 5 faults leading to browser hangs that we encountered are all browser-specific. Information leakage occurred twice – in TYPO3 and MODx – as a result of JavaScript faults that caused potentially security-sensitive code from the server to be displayed on the page; one of these bugs leading to information leakage is DOM-related.

***Finding 6**: About 80% of the highest severity JavaScript faults are DOM-related.*

2.4.3 Causes of JavaScript Faults

Locations. Before we can determine the causes, we first need to know *where* the programmers committed the programming errors. To this end, we marked the error locations of each bug report; the error location categories are listed in Section 2.3.4. The results are shown in Table 2.6. As the results show, the vast majority (83%) of the JavaScript faults occur as a result of programming errors in the JavaScript code itself. When only DOM-related faults were considered, a similar distribution of fault locations was observed; in fact, the majority is even larger for DOM-related faults that originated from the JavaScript code, at 89%. Although JavaScript code could be automatically written by external tools, we observed that the fix for these bugs involved the *manual* modification of the JavaScript file(s) where the error is

Table 2.6: Error locations of the bug reports analyzed. Library data are shown in italics.

Legend: JS = JavaScript code, HTML = HTML code, SSC = Server-side code, SCF = Server configuration file, OTH = Other, MEL = Multiple error locations

Application	JS	HTML	SSC	SCF	OTH	MEL
Moodle	22	2	6	0	0	0
Joomla	9	0	1	0	0	1
WordPress	24	0	6	0	0	0
Drupal	29	0	1	0	0	0
Roundcube	26	0	4	0	0	0
WikiMedia	25	0	5	0	0	0
TYPO3	18	1	9	2	0	0
TaskFreak	6	0	0	0	0	0
Horde	22	1	4	2	1	0
FluxBB	5	0	0	0	0	0
LimeSurvey	25	1	3	0	1	0
DokuWiki	26	1	2	0	1	0
phpBB	23	5	0	1	1	0
MODx	23	1	6	0	0	0
EZ Systems	19	5	6	0	0	0
jQuery	*30*	–	–	–	*0*	*0*
Prototype.js	*25*	–	–	–	*4*	*1*
MooTools	*30*	–	–	–	*0*	*0*
Ember.js	*30*	–	–	–	*0*	*0*
Overall	417	17	53	5	8	2

located. This observation provides a good indication that JavaScript faults typically occur because the programmer herself writes erroneous code, as opposed to server-side code automatically generating erroneous JavaScript code, or HTML.

***Finding 7**: Most JavaScript faults (83%) originate from manually-written JavaScript code as opposed to code automatically generated by the server.*

Patterns. To understand the programmer mistakes associated with JavaScript errors, we manually examined the bug reports for errors committed in JavaScript code (which were the dominant category). We found that 55% of the errors fell into the following common patterns (the remaining 45% of the errors followed miscellaneous patterns):

1. **Erroneous input validation**. Around 16% of the bugs occurred because inputs passed to the JavaScript code (i.e., user input from the DOM or inputs to JavaScript functions) are not being validated or sanitized. The most

common mistake made by programmers in this case is neglecting valid input cases. For example, in the jQuery library, the `replaceWith()` method is allowed to take an empty string as input; however, the implementation of this method does not take this possibility into account, thereby causing the call to be ignored.

2. **Error in writing a string literal**. Approximately 13% of the bugs were caused by a mistake in writing a string literal in the JavaScript code. These include forgetting prefixes and/or suffixes, typographical errors, and including wrong character encodings. About half of these errors relate to writing a syntactically valid but incorrect CSS selector (which is used to retrieve DOM elements) or regular expression.

3. **Forgetting null/undefined check**. Around 10% of the bugs resulted from missing null/undefined checks for a particular variable, assuming that the variable is *allowed* to have a value of `null` or `undefined`.

4. **Neglecting differences in browser behaviour**. Around 9% of the bugs were caused by differences in how browsers treat certain methods, properties or operators in JavaScript. Of these, around 60% pertain to differences in how browsers implement native JavaScript methods. For example, a fault occurred in WikiMedia in Internet Explorer 7 and 8 because of the different way those browsers expect the `history.go()` method to be used.

5. **Error in syntax**. Interestingly, around 7% of bugs resulted from syntax errors in the JavaScript code that were made by the programmer. Note, also, that we found instances where server-side code generated syntactically incorrect JavaScript code, though this is not accounted for here.

***Finding 8**: There are several recurring error patterns – causing JavaScript faults – that arise from JavaScript code.*

Temporal Analysis. When plotted per year, the percentage of bug reports whose corresponding error is located in the JavaScript code results in a regression line that

has a positive slope, as seen in Figure 2.4c. In other words, the percentage of bug reports whose errors are located in the JavaScript code generally increased from the year 2004 to 2014; in this case, based on the endpoints of the regression line, there was an increase of around 25%. We believe this trend is a product of client-side scripting gaining more prominence in web application development as the years went by. In particular, during this time period, new and richer web standards for ECMAScript [45] and XMLHttpRequest [159] were being introduced, giving way for the rise in popularity of AJAX, which developers could use to offload server-side functionality to the client-side. As a result, since JavaScript is being used more frequently, more errors are being committed in JavaScript code. In addition, the overall increase in the trend may also be attributable to JavaScript code becoming more complex as JavaScript gradually rose in popularity over time.

***Finding 9**: Among JavaScript bugs, the percentage of errors committed in the JavaScript code has experienced a 25% increase over the past ten years.*

2.4.4 Browser Specificity

We analyzed the browser specificity of the bug reports we collected. A bug is browser specific if it occurs in at most two web browsers. As Table 2.7 shows, most JavaScript faults (77%) are non-browser specific (the same percentage is acquired when only DOM-related faults are considered). However, among the browser-specific faults, about 64% are specific to Internet Explorer (IE).

After analyzing the IE-specific faults, we found that most of them (56%) were due to the use of methods and properties that were not supported in that browser (particularly in earlier versions, pre-Internet Explorer 8). This is likely because the use of browser-specific method and property names (which may not be standards-compliant) is more prevalent in IE than in other browsers. In addition, IE has low tolerance of small errors in the JavaScript code. For example, 21% of the IE-specific faults occurred because IE could not handle trailing commas in object-creation code; while these trailing commas are syntax errors as per the ECMAScript standard, other browsers can detect their presence and remove them.

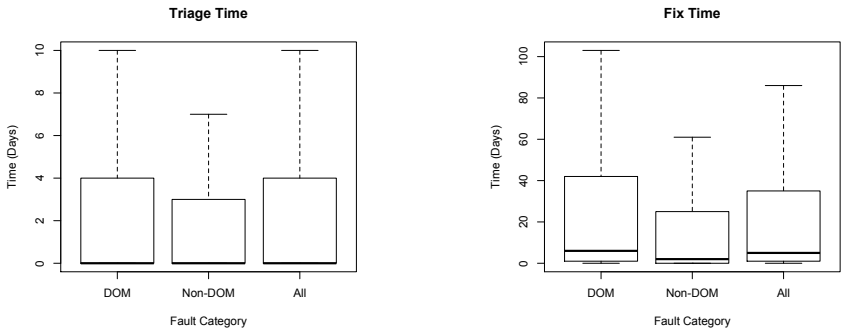
Table 2.7: Browser specificity of the bug reports analyzed. Library data are shown in italics.

Legend: IE = Internet Explorer, FF = Firefox, CHR = Chrome, SAF = Safari, OPE = Opera, OTH = Other, NBS = Not browser-specific, MUL = Multiple

Application	**IE**	**FF**	**CHR**	**SAF**	**OPE**	**OTH**	**NBS**	**MUL**
Moodle	4	0	0	0	0	0	25	1
Joomla	1	0	0	0	0	0	10	0
WordPress	1	0	0	0	0	1	28	0
Drupal	2	0	1	1	0	0	26	0
Roundcube	5	0	1	0	1	1	22	0
WikiMedia	6	0	0	0	0	0	24	0
TYPO3	7	1	0	0	1	0	20	1
TaskFreak	1	0	0	1	0	0	4	0
Horde	8	1	0	3	0	0	18	0
FluxBB	0	0	0	0	0	0	5	0
LimeSurvey	5	1	0	0	0	0	24	0
DokuWiki	2	1	2	0	1	0	23	1
phpBB	2	2	0	1	2	1	22	0
MODx	3	0	0	1	0	0	26	0
EZ Systems	1	0	1	0	0	0	27	1
jQuery	*7*	*0*	*0*	*0*	*0*	*0*	*22*	*1*
Prototype.js	*8*	*1*	*1*	*2*	*1*	*0*	*14*	*3*
MooTools	*10*	*2*	*0*	*0*	*1*	*0*	*17*	*0*
Ember.js	*2*	*0*	*0*	*0*	*0*	*0*	*28*	*0*
Overall	75	9	6	9	7	3	385	8

***Finding 10**: Most JavaScript faults (77%) are not browser-specific.*

Temporal Analysis. Figure 2.4d shows the scatter plot for the percentage of browser specific bug reports per year. Here, the regression line has a negative slope; more specifically, the regression line shows the browser specificity decreasing by around 50% (i.e., from 32% in 2004 to 17% in 2014). This decrease in browser specificity is consistent with results found in a recent study [20], which show an overall decrease in cross-browser-related questions in StackOverflow from 2009 to 2012; this work posits that the decrease may have been caused by the maturation of browser support for HTML5, which was the focus of the study. Other factors that may have contributed to this decline include the introduction of JavaScript libraries designed to eliminate cross-browser incompatibilities, some of which are used in the web applications we studied, as well as extensive research done recently on ways to mitigate cross-browser compatibility issues [36, 37, 108, 150].



(a) **(b)**

Figure 2.5: Box plot for (a) triage time, and (b) fix time

***Finding 11**: The percentage of browser-specific faults among all JavaScript faults has experienced a 50% decrease over the past ten years.*

2.4.5 Triage and Fix Time for JavaScript Faults

We calculated the triage time and fix time for each bug report. The results are shown in Figure 2.5a (triage time) and Figure 2.5b (fix time) as box plots (the outliers are not shown). Most of the bug reports were triaged on the same day they were reported, which explains why the median of the triage time for all bug reports, as seen in Figure 2.5a, is 0. Also, as seen in Figure 2.5b, the median of the fix time for all bug reports is 5 days. Finally, in addition to the box plots, we calculated the mean triage and fix times for each bug report. We found that, on average, the triage time for JavaScript faults is approximately 29 days, while the average fix time is approximately 65 days (see Table 2.8).

As before, we made the same calculations for DOM-related faults and non-DOM-related faults. The comparisons are shown in Figures 2.5a and 2.5b, as well as in Table 2.8. From the box plots, we find that both DOM-related faults and non-DOM-related faults have a median triage time of 0 days, which indicates that the majority of either of these faults gets triaged on the same day as they are reported. For the fix times, DOM-related faults have a median fix time of 6 days, compared

Table 2.8: Average triage times (T) and fix times (F) for each experimental object, rounded to the nearest whole number. Library data are shown in italics.

Application	All Faults		DOM-Related Faults Only		Non-DOM-Related Faults Only	
	T	**F**	**T**	**F**	**T**	**F**
Moodle	248	10	205	12	292	8
Joomla	4	57	1	67	8	46
WordPress	1	138	2	150	0	108
Drupal	7	66	3	47	22	130
Roundcube	18	118	25	160	0	9
WikiMedia	18	26	36	44	1	8
TYPO3	7	55	7	67	6	36
TaskFreak	23	17	32	23	6	5
Horde	4	7	5	8	0	1
FluxBB	1	5	1	5	–	–
LimeSurvey	10	47	5	58	28	5
DokuWiki	11	13	10	15	13	9
phpBB	3	60	3	64	4	46
MODx	86	46	116	64	16	4
EZ Systems	35	41	22	41	52	42
jQuery	*1*	*33*	*1*	*36*	*1*	*10*
Prototype.js	*28*	*343*	*33*	*294*	*14*	*478*
MooTools	*10*	*48*	*10*	*49*	*9*	*47*
Ember.js	*0*	*11*	*1*	*14*	*0*	*8*
Overall	29	65	26	71	37	52

to 2 days for non-DOM-related faults. With respect to the means (Table 2.8), we found that DOM-related faults have an average triage time of 26 days, compared to 37 days for non-DOM-related faults. On the other hand, DOM-related faults have an average fix time of 71 days, compared to 52 days for non-DOM-related faults.

Taking the above results into account, it appears that DOM-related faults generally have lower triage times than non-DOM-related faults, while DOM-related faults have higher fix times than non-DOM-related faults. This suggests that developers find DOM-related faults important enough to be triaged more promptly than non-DOM-related faults. However, DOM-related faults take longer to fix, perhaps because of their inherent complexity.

***Finding 12**: On average, DOM-related faults get triaged more promptly than non-DOM-related faults (26 days vs. 37 days); however, DOM-related faults take longer to fix than non-DOM-related faults (71 days vs. 52 days).*

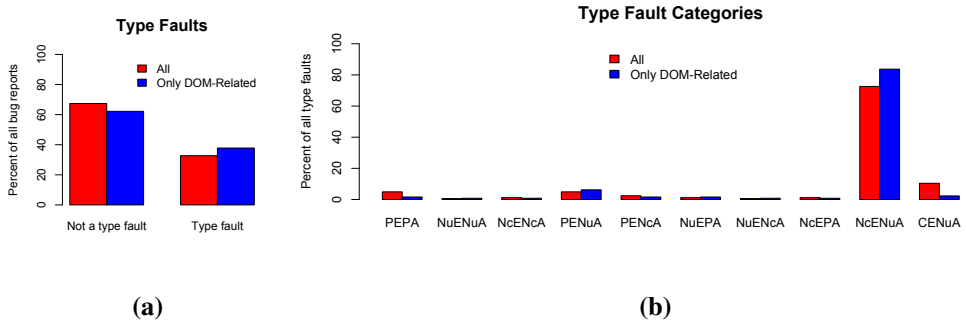


Figure 2.6: Bar graphs showing (a) the percentage of bug reports that are type faults versus the percentage of bug reports that are not type faults and (b) the distribution of type fault categories

2.4.6 Prevalence of Type Faults

We now discuss our findings regarding the prevalence of type faults in the bug reports that we analyzed. As discussed in Section 2.3.4, each type fault category is identified as "__E__A", where the first blank is represented by the abbreviation of the expected type category, and the second blank is represented by the abbreviation of the actual type category.

The results are shown in Figure 2.6. Overall, as seen in Figure 2.6a, only about 33% of the bug reports in our study were classified as type faults. Further, of all the type faults, 72% belong to the *NcENuA* category, in which a native "class" type is expected, but the actual type at runtime is `null` or `undefined` (see Figure 2.6b). These results show that in the subject systems, the vast majority of the bugs are *not* type faults; thus, programming languages introducing stronger typing systems to JavaScript, such as TypeScript and Dart, as well as type checkers, may not eliminate most JavaScript faults. It is worth noting, however, that these languages have other advantages apart from strong typing, including program comprehension and support for class-based object-oriented programming.

We also studied the DOM-related faults, to determine how many of them are type faults. The results for DOM-related faults are also shown in Figure 2.6. Overall, we found that 38% of DOM-related faults are also type faults. Most of these type faults are also of the *NcENuA* variety; in this case, the expected native "class" type is, for the most part, `Element`. This finding suggests that stronger typing systems and type checkers may also not eliminate most DOM-related faults.

We also analyzed the severity of type faults, based on the impact types assigned to each bug report. We found that out of all the Type 4 and Type 5 impact bug reports, which are the highest severity bugs, about 30% are type faults; considering Type 5 impact bug reports alone, about 18% are type faults. Therefore, based on our results, these languages have limited ability in eliminating the majority of the *highest impact* JavaScript bugs in web applications.

***Finding 13**: The majority (67%) of JavaScript faults are not type faults, and the majority (62%) of DOM-related faults are also not type faults. In addition, the majority of Type 4 and Type 5 impact bugs are not type faults, at 70%.*

Temporal Analysis. As seen in Figure 2.4e, the regression line for the percentage of type faults is relatively flat (with a slight 3% increase from 2004 to 2014). Therefore, despite all the technology that has been developed to eliminate these type faults, the percentage of type faults has remained more or less constant over the years. This may be caused by the fact that most of the type faults we found in our study belong to the *NcENuA* category, where a native "class" type is expected, but the actual type at runtime is `null` or `undefined`. In particular, current type checkers normally cannot predict if the return value of a native JavaScript method is `null` or `undefined`; for example, they cannot predict if the return value of `getElementById()` is `null` without looking at the DOM – which almost none of them do – so that particular type fault will be missed.

***Finding 14**: The percentage of type faults among all JavaScript faults has remained constant (with only a 3% increase) over the past ten years.*

2.4.7 Threats to Validity

An internal validity threat is that the bug classifications were made by two individuals, which may introduce inconsistencies and bias, particularly in the classification of the impacts. In order to mitigate any possibilities of bias, we conducted a review

process in which each person reviews the classifications assigned by the other person. Any disagreements were discussed until a consensus on the classification was reached.

Another internal threat is in our analysis of type faults, in which we assumed that a bug report does not correspond to a type fault if the existence of a statement $\mathscr{L}$ from the definition given in Section 2.2.2 could not be established, based on our qualitative reading of the bug report. Hence, the percentage of type faults we presented in Section 2.4.6 is technically a lower bound on the actual number of type faults. Nonetheless, the lack of any indication in a bug report that a statement $\mathscr{L}$ exists strongly suggests that inconsistencies in types are not an issue with the bug.

In terms of external threats, our results are based on bug reports from a limited number of experimental subjects, from a limited time duration of ten years, which calls into question the representativeness; unfortunately, public bug repositories for web applications are not abundant, as previously mentioned. We mitigated this by choosing web applications that are used for different purposes, including content management, webmail, and wiki. Further, prior to 2004, few websites used JavaScript, since AJAX programming – which is often credited for popularizing JavaScript – did not become widespread until around 2005 when Google Maps was introduced [159].

In addition, our choice of analyzing a maximum of 30 bug reports per application is an external threat, as they may not represent all types of bugs present in each of our experimental subjects. To mitigate this, we reported some application-specific data, such as the percentage of DOM-related faults per application (Table 2.3); the similarity in these percentages suggests that the bugs that occur in each of our experimental subjects do in fact have common traits (e.g., the prevalence of DOM-related faults is true not only in aggregate, but also per application).

A construct validity threat is that the bug reports may not be fully representative of the JavaScript faults that occur in web applications. This is because certain kinds of faults – such as non-deterministic faults and faults with low visual impact – may go unreported. In addition, we focus exclusively on bug reports that were fixed. This decision was made since the root cause would be difficult to determine from open reports, which have no corresponding fix. Further, open reports may not be

representative of real bugs, as they are not deemed important enough to fix.

For the triage and fix times, we did not account for possible delays in marking a bug report as "assigned" or "fixed", which may skew the results. In addition, the triage time is computed as the time until the first developer comment, when there is no "assigned" marking; although we find this approximation reasonable, the developer may not have started fixing until some days after the first comment was posted. Finally, the triage and fix times may be influenced by external factors apart from the complexity of the bugs (e.g., bug replication, vacations, etc.). These are likewise construct validity threats. Nonetheless, note that other studies have similarly used time to estimate the difficulty of a fix, including Weiss et al. [176] and Kim and Whitehead [89]. In the former, the authors point out that the time it takes to fix a bug is indicative of the effort required to fix it; the difference between our estimation and theirs is that they use the fix time *reported* by developers assigned to the bug, which is unavailable in the bug repositories we studied.

2.5 Discussion

In this section, we discuss the implications of our findings on web application developers, testers, developers of web analysis tools, and designers of web application development frameworks.

Findings 1 and 2 reveal the difficulties that web application developers have in setting up values passed or assigned to native JavaScript methods and properties – particularly DOM methods and properties. Finding 2, in particular, also shows that most of the DOM-related faults that occur in web applications are *strong* DOM-related faults, indicating a mismatch between the programmer's *expectation* of the DOM and the *actual* DOM. Many of these difficulties arise because the asynchronous, event-driven JavaScript code must deal with the highly dynamic nature of the DOM. This requirement forces the programmer to have to think about how the DOM is structured and what properties its elements possess at certain DOM interaction points in the JavaScript code; doing so can be difficult because (1) the DOM frequently changes at runtime and can have many states, and (2) there are many different ways a user can interact with the web application, which means there are many different orders in which JavaScript event handlers can ex-

ecute. This suggests the need to equip these programmers with appropriate tools that would help them reason about the DOM, thereby simplifying these DOM-JavaScript interactions.

These first two findings also impact web application testers, as they reveal certain categories of JavaScript faults that users consider important enough to report, and hence, that testers should focus on. Currently, one of the most popular ways to test JavaScript code is through unit testing, in which modules are tested individually; when creating these unit tests, a mock DOM object is usually needed, in order to allow the DOM API method calls present in the module to function properly. While useful, this approach often does not take into account the changing states of the DOM when users are interacting with the web application in real settings, because testers often create these mock DOM objects simply to prevent the calls from failing. Finding 2, in contrast, suggests that testers need to be more "DOM-aware", in that they need to take extra care in ensuring that these mock objects emulate the actual DOM as closely as possible.

In addition to unit testing, web application testers also perform end-to-end (E2E) testing; here, user actions (e.g., clicks, hovers, etc.) are automatically applied to individual webpages to assert that certain conditions about the DOM are satisfied after carrying out these user actions. E2E testing can help detect DOM-related faults; however, the problem is that these tests often require the tester to know certain properties about the DOM in order to set up the user actions in the tests. As mentioned above, keeping track of these properties of the DOM is difficult to do, judging by the large percentage of strong DOM-related faults we observed in our study. This makes E2E tests themselves susceptible to DOM-related faults if written manually, which motivates the potential usefulness of automated tools for writing these tests, including record-replay and webpage crawling techniques.

With regards to Findings 4, 6, and 12, these results suggest that web application testers should also prioritize *emulating* DOM-related faults, as most high-impact faults belong to this category. One possible way to do this is to prioritize the creation of tests that cover DOM interaction points in the JavaScript code. By doing so, testers can immediately find most of the high-impact faults. This early detection is useful because, as Finding 4 suggests, DOM-related faults often have no accompanying error messages and can be more difficult to detect. Further, as

Finding 12 suggests, DOM-related faults take longer to fix on average compared to non-DOM-related faults.

As mentioned previously, the presence of error messages in JavaScript bugs can be useful, as these messages can provide a natural starting point for analysis of these bugs. Indeed, our fault localization tool AUTOFLOX [134] – which we describe in detail in Chapter 3 – uses error messages to automatically infer the line of code where the failure takes place – which, in this case, is the same as the exception point – as well as to determine the backward slice of the `null` or `undefined` values that led to the exception. However, as Finding 4 suggests, the majority of DOM-related faults do *not* lead to exceptions and hence, do not have accompanying error messages. This points to the need to devise alternative ways to automatically determine the line of code where the failure takes place when performing fault localization. One possibility is to give developers the ability to select, on the webpage itself, any DOM element that is observed to be incorrect. A static analyzer can then try to guess which lines of JavaScript code were the latest ones to update the element; these lines will therefore be the starting point for localization.

As for Findings 7 and 8, these results can be useful for developers of static analysis tools for JavaScript. Many of the current static analysis tools only address syntactic issues with the JavaScript code (e.g., JSLint,[7] Closure Compiler,[8] JSure[9]), which is useful since a few JavaScript faults occur as a result of syntax errors, as described in Section 2.4.3. However, the majority of JavaScript faults occur because of errors in semantics or logic. Some developers have already started looking into building static semantics checkers for JavaScript, including TAJS [78], which is a JavaScript type analyzer. However, the programming mistakes we encountered in the bug reports (e.g., erroneous input validations, erroneous CSS selectors, etc.) call for more powerful tools to improve JavaScript reliability.

The recurring patterns to which Finding 8 refers can be a starting point for devising a taxonomy for common JavaScript errors. This taxonomy can be helpful in two ways. First, it can facilitate the code review process, as the taxonomy helps

[7] http://www.jslint.com
[8] http://code.google.com/closure/compiler/
[9] https://github.com/berke/jsure

web developers identify certain "hot spots" in the code that have historically been susceptible to error. In addition, tools such as FindBugs[10] use a taxonomy of common coding patterns and smells to automatically detect errors in Java code using static analysis; in the same vein, a taxonomy for JavaScript errors can help achieve the same automatic error detection task for JavaScript code.

While Finding 10 suggests that most JavaScript faults are non-browser specific, we did find a few (mostly IE-specific) faults that are browser-specific. Hence, it is useful to design JavaScript development tools that recognize cross-browser differences and alerts the programmer whenever she forgets to account for these. Some Integrated Development Environments (IDEs) for JavaScript have already implemented this feature, including NetBeans[11] and Aptana.[12]

Finding 13 shows that there is a significant number, though a minority, of type faults encountered in the subject systems, some of which have high impact. This provides motivation for the development of the strongly-typed languages mentioned earlier, as well as type checkers [51, 66, 78]. However, such tools and languages are far from being a panacea, as the vast majority of JavaScript faults are not type faults, according to our study. Therefore, tool developers should not focus exclusively on type checking when looking for ways to improve the reliability of JavaScript code because type checking, while useful, does not suffice in detecting or eliminating most JavaScript faults.

According to Findings 5 and 11, there was a significant decrease in the number of code-terminating failures and browser specific faults; this suggests that developers of browsers and browser-based tools are heading towards the right direction in terms of facilitating the debugging process for JavaScript faults and ensuring cross-browser compliance of JavaScript code. However, from Findings 3 and 14, we observed that the number of DOM-related faults and type faults has remained relatively constant from 2004 to 2014; hence, developers, testers, and tool designers must pay more careful attention towards these two kinds of faults, especially DOM-related faults, as they constitute almost 70% of all JavaScript faults.

Finding 9 shows that the percentage of errors located in the JavaScript code

[10] http://findbugs.sourceforge.net/
[11] http://netbeans.org/
[12] http://www.aptana.com/

has increased from 2004 to 2014. This suggests that tools for improving the client-side reliability should consider performing an analysis of the client-side code itself – which is where the majority of JavaScript bugs arise as per Findings 7 and 8 – instead of simply looking at how server-side code generates malformed client-side code, as other tools have done [33, 148].

Finally, recall that this study focuses on *client-side* JavaScript; hence, the results may not directly be applicable to JavaScript developers at the server-side who use Node.js. For example, there is no DOM present at the server-side, which indicates that DOM-related faults will not be present there. Nonetheless, many of the error and fault patterns we found are not client-side-specific. A recent paper by Hanam et al. [68], for instance, sheds light on some of the pervasive JavaScript bug patterns that appear at the server-side. One of the bug patterns they found is "Dereferenced Non-Values", which corresponds to both the "Undefined/Null Variable Usage" fault category and the "Forgetting null/undefined check" error pattern. Hence, JavaScript developers at the server-side can still gather important takeaways from our results, even if our study was not specifically targeted towards server-side JavaScript.

2.6 Related Work

There has been a large number of empirical studies conducted on faults that occur in various types of software applications [34, 38, 50, 97, 179, 189]. Here, we focus on only those studies that pertain to web applications.

Server-Side Studies. In the past, researchers have studied the causes of web application faults at the server-side using session-based workloads [61], server logs [163], and website outage incidents [138]. Further, there have been studies on the control-flow integrity [29] and end-to-end availability [85, 136] of web applications. Finally, studies have been conducted which propose web application fault models and taxonomies [46, 102, 156]. Our current study differs from these papers in that we focus on web application faults that occur at the client-side, particularly ones that propagate into the JavaScript code.

Client-Side Studies. Several empirical studies on the characteristics of client-side JavaScript have been made. For instance, Ratanaworabhan et al. [142] used their

JSMeter tool to analyze the dynamic behaviour of JavaScript in web applications. Similar work was conducted by Richards et al. [144] and Martinsen et al. [105]. A study of parallelism in JavaScript code was also undertaken by Fortuna et al. [53]. Finally, there have been empirical studies on the security of JavaScript. These include empirical studies on cross-site scripting (XSS) sanitization [175], privacy-violating information flows [77], and remote JavaScript inclusions [124, 180]. Unlike our work which studies functional JavaScript faults, these related papers address non-functional properties such as security and performance.

In recent work, Bajaj et al. [20] mined web application-related questions in StackOverflow to determine common difficulties that developers face when writing client-side code. This work is similar to the current one in that it attempts to infer reliability issues with JavaScript, using developers' questions as an indicator. However, unlike our current work, this study does not make any attempt to determine the characteristics of JavaScript faults.

Our earlier work [128] looked at the characteristics of failures caused by JavaScript faults, based on console logs. However, we did not study the causes or impact of JavaScript faults, nor did we examine bug reports as we do in this study. To the best of our knowledge, we are the first to perform an empirical study on the characteristics of these real-world JavaScript faults, particularly their causes and impacts.

Finally, in very recent work which followed the original paper on which this current work is based, Pradel et al. [141] and Bae et al. [18] proposed tools for detecting type inconsistencies and web API misuses in JavaScript code, respectively. These studies provide examples of common type faults and common web API misuse patterns. However, they do not establish the prevalence of their respective fault categories, nor do they identify DOM-related faults as an important subclass of JavaScript faults.

2.7 Conclusions

Client-side JavaScript contains many features that are attractive to web application developers and is the basis for modern web applications. However, it is prone to errors that can impact functionality and user experience. In this chapter, we per-

form an empirical study of over 500 bug reports from various web applications and JavaScript libraries to help us understand the nature of the errors that cause these faults, and the failures to which these faults lead. Our results show that (1) around 68% of JavaScript faults are DOM-related; (2) most (around 80%) high severity faults are DOM-related; (3) the vast majority (around 83%) of JavaScript faults are caused by errors manually introduced by JavaScript code programmers; (4) error patterns exist in JavaScript bug reports; (5) DOM-related faults take longer to fix than non-DOM-related faults; (6) only a small but non-negligible percentage of JavaScript faults are type faults; and (7) although the percentage of code-terminating failures and browser-specific faults has decreased over the past ten years, the percentage of DOM-related faults and type faults has remained relatively constant.

Chapter 3

Automatic JavaScript Fault Localization

This chapter describes the technique that we have designed for automatically localizing DOM-related JavaScript faults.[13] Therefore, our goal in this chapter is to answer RQ1B from Chapter 1.1 (*How can we accurately and efficiently localize and repair JavaScript faults that appear in web applications?*), focusing in particular on localization. We implemented our technique in a tool called AUTOFLOX,[14] the details of which we now describe.

3.1 Introduction

JavaScript-based applications at the client-side suffer from multiple dependability problems due to their distributed, dynamic nature, as well as the loosely typed semantics of JavaScript. A common way of gaining confidence in software dependability is through *testing*. Although testing of modern web applications has received increasing attention in the recent past [15, 101, 109, 137], there has been little work on what happens after a test reveals an error. Debugging of web applications is still an expensive and mostly manual task. Of all debugging activities, locating the faults, or *fault localization*, is known to be the most expensive [84, 165].

The fault localization process usually begins when the developers observe a failure in a web program either spotted manually or through automated testing

[13] The main study in this chapter appeared at the Journal of Software Testing, Verification and Reliability (STVR) [134]. The initial conference version appeared at the International Conference on Software Testing, Verification and Validation (ICST 2012) [129].

[14] AUTOFLOX stands for "**Auto**matic **F**ault **Lo**calization for Aja**x**"

techniques. The developers then try to understand the root cause of the failure by looking at the JavaScript code, examining the DOM tree, modifying the code (e.g., with alerts or tracing statements), running the application again, and manually going through the initial series of navigational actions that led to the faulty state or running the corresponding test case.

Manually isolating a JavaScript fault's root cause requires considerable time and effort on the part of the developer. This is partly due to the fact that the language is not type-safe, and has loose fault-detection semantics. Thus, a fault may propagate undetected in the application for a long time before finally triggering an exception. Additionally, faults may arise in third-party code (e.g., libraries, widgets, advertisements) [128], and may be outside the expertise of the web application's developer.

Further, faults may arise due to subtle asynchronous and dynamic interactions at runtime between the JavaScript code and the DOM tree, which make it challenging to understand their root causes. Indeed, from our large-scale study of over 500 JavaScript bug reports, as described in Chapter 2, we found that faulty interactions between the JavaScript code and the DOM – which are called DOM-related faults – comprise over 68% of all JavaScript bugs [130]. From this same study, we also found that these DOM-related faults take longer to fix, on average, compared to all other fault types; hence, these DOM-JavaScript interactions are of particularly great concern when localizing faults. For these reasons, this chapter focuses on the localization of DOM-related faults.

Although fault localization in general has been an active research topic [1, 4, 39, 84], automatically localizing web faults has received very limited attention from the research community. To the best of our knowledge, automated fault localization for JavaScript-based web applications has not been addressed in the literature yet.

To alleviate the difficulties with manual web fault localization, we propose an automated technique based on *dynamic backward slicing* of the web application to localize DOM-related JavaScript faults. The proposed fault localization approach is implemented in a tool called AUTOFLOX. In addition, AUTOFLOX has been empirically evaluated on six open-source web applications and three production web applications, along with seven web applications containing real bugs. The

main contributions in this chapter include:

- A discussion of the challenges surrounding JavaScript fault localization, highlighting the real-world relevance of the problem and identifying DOM-related JavaScript faults as an important sub-class of problems in this space;
- A fully automated technique for localizing DOM-related JavaScript faults, based on dynamic analysis and backward slicing of JavaScript code. Our technique can localize faults in the presence of the `eval` function, anonymous functions, and minified JavaScript code. In addition, our technique is capable of localizing multiple faults;
- An open-source tool, called AUTOFLOX, implementing the fault localization technique. AUTOFLOX has been implemented both as a stand-alone program that runs with the CRAWLJAX tool, as well as an Eclipse plugin;
- An empirical study to validate the proposed technique, demonstrating its efficacy and real-world relevance. The results of this study show that the proposed approach is capable of successfully localizing DOM-related faults with a high degree of accuracy (over 96%) and *no false positives*. In addition, AUTOFLOX is able to localize JavaScript faults in production websites, as well as 20 actual, reported bugs from seven real-world web applications.

3.2 Challenges and Motivation

This section describes how JavaScript differs from other traditional programming languages and discusses the challenges involved in localizing faults in JavaScript code. First, a JavaScript code fragment that is used as a running example throughout this chapter is presented.

3.2.1 Running Example

Figure 3.1 presents an example JavaScript code fragment to illustrate some of the challenges in JavaScript fault localization. This code fragment is based on a fault in a real-world web application.[15]

[15] https://www.tumblr.com

```
1 function changeBanner(bannerID) {
2   clearTimeout(changeTimer);
3   changeTimer = setTimeout(changeBanner, 5000);

5   prefix = "banner_";
6   currBannerElem = document.getElementById(prefix + currentBannerID);
7   bannerToChange = document.getElementById(prefix + bannerID);
8   currBannerElem.removeClassName("active");
9   bannerToChange.addClassName("active");
10  currentBannerID = bannerID;
11 }
12 currentBannerID = 1;
13 changeTimer = setTimeout(changeBanner, 5000);
```

Figure 3.1: Example JavaScript code fragment based on *tumblr.com*.

The web application pertaining to the code fragment in Figure 3.1 consists of a banner at the top of the page. The image shown on the banner cycles through four images periodically (every 5000 milliseconds). The four images are each wrapped in `div` elements with DOM IDs `banner_1` through `banner_4`. The `div` element wrapping the image being shown is identified as "active" via its `class` attribute.

In the above code, the `changeBanner` function (Lines 1 to 10) updates the banner image to the next one in the sequence by updating the DOM. Lines 12 and 13 which are outside the function are executed at load time. Line 12 sets the value of variable `currentBannerID` to 1, indicating that the current image being shown is `banner_1`. Line 13 sets a timer that will asynchronously call the `changeBanner` function after 5 seconds (i.e., 5000 milliseconds). After each execution of the `changeBanner` function, the timeout function is cleared and reset so that the image is changed again after 5 seconds.

The JavaScript code in Figure 3.1 will throw a `null` exception in Line 9 when executed. Specifically, in the `setTimeout` calls, `changeBanner` is invoked without being passed a parameter, even though the function is expecting an argument, referenced by `bannerID`. Omitting the argument will not lead to an interpretation-time exception; rather the `bannerID` will be set to `undefined` when `changeBanner` executes. As a result, the second `getElementById` call will look for the ID "`banner_undefined`" in the DOM; since this ID does not exist, a `null` will be returned. Hence, accessing the `addClassName` method

via `bannerToChange` in Line 9 will lead to a `null` exception.

Note that this error arises due to the loose typing and permissive error semantics of JavaScript. Further, to understand the root cause of the error, one needs to analyze the execution of both the JavaScript code and the DOM. However, once the fault has been identified, the fix is relatively straightforward, viz. modify the `setTimeout` call in Line 13 to pass a valid value to the `changeBanner` function.

3.2.2 JavaScript Fault Localization

Although JavaScript is syntactically similar to languages such as Java and C++, it differs from them in two important ways, which makes fault localization challenging.

Asynchronous Execution: JavaScript code is executed asynchronously, and is triggered by the occurrence of user-triggered events (e.g., click, mouseover), load events, or events resulting from asynchronous function calls. These events may occur in different orders; although JavaScript follows a sequential execution model, it does not provide deterministic ordering. In Figure 3.1, the execution of the lines outside the `changeBanner` function is triggered by the load event, while the execution of the `changeBanner` itself is triggered asynchronously by a timeout event via the `setTimeout` call. Thus, each of these events triggered the execution of two different sequences of JavaScript code. In particular, the execution sequence corresponding to the load event is Line 12 $\rightarrow$ Line 13, while the execution sequence corresponding to the asynchronous event is Line 2 $\rightarrow$ Line 3 $\rightarrow$ Line 5 $\rightarrow$ Line 6 $\rightarrow$ Line 7 $\rightarrow$ Line 8 $\rightarrow$ Line 9.

In traditional programming languages, the goal of fault localization is to find the *erroneous lines* of code. For JavaScript, its asynchronous characteristic presents an additional challenge. The programmer will not only need to find the erroneous lines, but she will also have to map each executed sequence to the event that triggered their execution in order to understand the root cause of the fault. In addition, event handlers may overlap, as a particular piece of JavaScript code may be used by multiple event handlers. Thus, manual fault localization in client-side JavaScript is a tedious process, especially when many events are triggered.

DOM Interactions: In a web application, JavaScript code frequently interacts with the DOM, which characterizes the dynamic HTML structure and elements present in the web page. As a result, the origin of a JavaScript fault is not limited to the JavaScript code; the JavaScript fault may also result from an error in the DOM. With regards to fault localization, the notion of an "erroneous line" of code may not apply to JavaScript because it is possible that the error is in the DOM rather than the code. This is particularly true for DOM-related faults, which lead to either exceptions or incorrect DOM element outputs as a result of a DOM access or update. As a result, for such faults, one needs to formulate the goal of fault localization to isolate the first line of JavaScript code containing a call to a DOM access function (e.g., `getAttribute()`, `getElementById()`) or a DOM update function/property (e.g., `setAttribute()`, `innerHTML`) that directly causes JavaScript code to throw an exception, or to update a DOM element incorrectly. This line is referred to as the *direct DOM interaction*.

For the example in Figure 3.1, the JavaScript exception occurs in Line 9, when the `addClassName` function is called on `bannerToChange`, which is `null`. The `null` value originated from Line 7, when the DOM access function `getElementById` returned `null`; thus, the direct DOM interaction is actually at Line 7. Note that even though this direct DOM interaction does not represent the actual "erroneous" lines which contain the missing parameter to the `changeBanner` function (Lines 3 and 13), knowing that `getElementById` in Line 7 returned `null` provides a hint that the value of either "`prefix`" or "`bannerID`" (or both) is incorrect. Using this knowledge, the programmer can isolate the erroneous line of code as she has to track the values of only these two variables. While in this simple example, the direct DOM interaction line is relatively easy to find, in more complex code the `null` value could propagate to many more locations and the number of DOM interactions to consider could be much higher, making it challenging to identify the direct DOM interaction. This is the challenge addressed in this chapter.

3.2.3 Challenges in Analyzing JavaScript Code

In addition to the challenges described in the previous subsection, JavaScript also contains several features that complicate the process of analyzing JavaScript code for fault localization. These are described below.

Eval: JavaScript allows programmers to dynamically create code through the use of the `eval` method. This method takes a string value as a parameter, where the string evaluates into JavaScript code. Although alternatives to certain uses of `eval` have been introduced in the language (e.g., the `JSON` API), studies show that `eval` use remains pervasive among web developers [145].

The presence of `eval` poses a challenge to both manual and automatic analysis of JavaScript code. The reason is twofold. First, the string parameter to `eval` is typically not just a simple string literal, but rather, a concatenation of multiple string values whose value cannot be determined based on a simple source-code level inspection; hence, it is difficult to infer the JavaScript code generated by `eval` at runtime. Second, the scope of variables introduced in `eval` code is directly linked to the scope in which the `eval` call is made; hence, `eval` code cannot be analyzed in isolation, but must be analyzed in relation to where the `eval` call is made. These, in turn, make fault localization more difficult, since the developer cannot easily keep track of the values that are created or modified through `eval`.

Anonymous Functions: Since JavaScript treats functions as first-class citizens in the form of `Function` literals, programmers can define functions without providing them with a name; these unnamed functions are known as *anonymous functions*. Hence, when tracking the propagation of a JavaScript fault, it does not suffice to identify the lines of code involved in the propagation solely based on the function name, particularly if a JavaScript fault originates from or propagates through an anonymous function.

Minified Code: Before deploying a web application, it is common practice for web developers to *minify* their JavaScript code, which compresses the code into one line. While this minification process reduces the size of JavaScript files, it also makes JavaScript code more difficult to read and analyze. This makes it very difficult to localize faults in minified code, as the developer will have a hard time keeping track of the relevant lines of code.

3.3 Scope of this Chapter

In the bug report study described in Chapter 2, we found that over 68% of JavaScript faults experienced by web applications are DOM-related faults. Recall that a fault is considered DOM-related if the corresponding error propagates into the parameter value of a DOM API method, such as `getElementById` and `querySelector`. In addition, these DOM-related faults comprise 80% of the highest impact JavaScript faults, according to the same study. Due to their prominence and severity, *we focus on the study of DOM-related faults in this chapter*.

Recall from Chapter 2 that DOM-related faults can be further divided into two classes, based on their failure characteristics, listed below:

1. **Code-terminating DOM-related JavaScript faults**: A DOM-related fault that leads to a code-terminating failure. In other words, a DOM access function returns a `null`, `undefined`, or incorrect value, which then propagates into several variables and eventually causes an exception.

2. **Output DOM-related JavaScript faults**: A DOM-related fault that leads to an output-related failure. In other words, a DOM update function sets the value of a DOM element property to an incorrect value without causing the code to halt.

The fault localization approach described in this chapter can localize code-terminating DOM-related JavaScript faults automatically, requiring only the URL of the web application and the DOM elements needed to reproduce the failure as input from the user. Hence, in the sections that follow, it is assumed that the fault being localized leads to a code-terminating failure. However, note that the proposed approach can also support output DOM-related JavaScript faults, but the approach would only be semi-automatic, as the user must also provide the location of the failing line of code to initiate the localization process.

For code-terminating DOM-related JavaScript faults, the direct DOM interaction is the DOM access function that returned the `null`, `undefined`, or incorrect value, and is referred to as the *direct DOM access*.

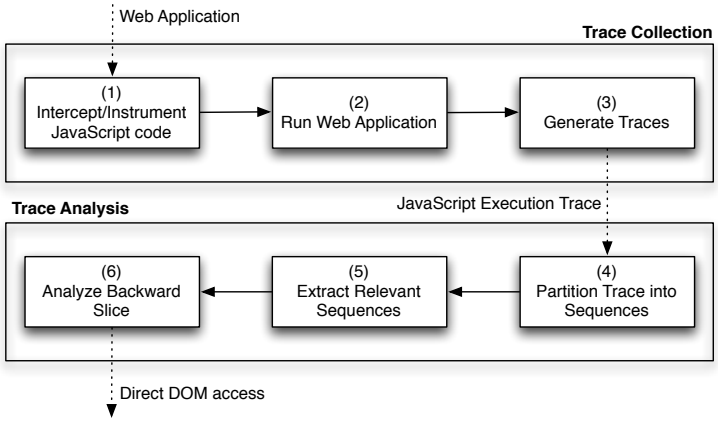


Figure 3.2: Block diagram illustrating the proposed fault localization approach.

3.4 Approach

Our proposed fault localization approach consists of two phases: (1) *trace collection*, and (2) *trace analysis*. The trace collection phase involves crawling the web application and gathering traces of executed JavaScript statements until the occurrence of the failure that halts the execution. After the traces are collected, they are parsed in the trace analysis phase to find the direct DOM access. The two phases are described in detail in this section. A block diagram of the approach is shown in Figure 3.2. We first describe the usage model of the proposed approach.

3.4.1 Usage Model

Because the focus is on fault localization, we assume that the failure whose corresponding fault needs to be localized has been detected before the deployment of the proposed technique. Further, we also assume that the user is able to replicate the failure during the localization process, either through a test case, or by knowing the sequence of user events that would trigger the failure.

The approach is designed to automate the fault localization process. The only manual intervention required from the user is at the very beginning, where the user would have to specify which elements in the web application to click (during the trace collection phase) in order for the failure to occur.

The output of the approach is the direct DOM access corresponding to the fault being localized and specifies, (1) the function containing the direct DOM access, (2) the line number of the direct DOM access relative to this function, and (3) the JavaScript file containing the direct DOM access.

3.4.2 Trace Collection

In the trace collection phase, the web application is crawled (by systematically emulating the user actions and page loads) to collect the trace of executed JavaScript statements that eventually lead to the failure. This trace is generated through on-the-fly instrumentation of *each line* of client-side JavaScript code before it is passed on to and loaded by the browser (box 1, Figure 3.2). Thus, for every line l of JavaScript code executed, the following information is written to the trace: (1) the function containing the line, (2) the line number relative to the function to which it belongs, (3) the names and scopes (global or local) of all the variables within the scope of the function, and (4) the values of these variables prior to the execution of the line. In the example in Figure 3.1, the order of the first execution is as follows: Line 12 $\rightarrow$ Line 13 $\rightarrow$ Line 2 $\rightarrow$ Line 3 $\rightarrow$ Line 5 $\rightarrow$ Line 6 $\rightarrow$ Line 7 $\rightarrow$ Line 8 $\rightarrow$ Line 9. Thus, each of these executed lines will have an entry in the trace corresponding to it. The trace record for Line 5 is shown in Figure 3.3. Note that in this figure, the trace record prefix contains the name of the function and the line number relative to this function; the variable names, scopes, and values are also shown, and other variables which have not been assigned values up to the current line are marked with "none". In the figure, `bannerID`'s value is recorded as "none" because this parameter is unspecified in the `setTimeout` call.

In addition to the trace entries corresponding to the executed lines of JavaScript code, three special markers, called ERROR, ASYNCCALL and ASYNC, are added to the trace. The ERROR marker is used in the trace analysis phase to determine at which line of JavaScript code the exception was thrown; if a line l is marked with the ERROR marker, then the value $l.failure$ is set to `true`. The ASYNCCALL and ASYNC markers address the asynchronous nature of JavaScript execution as described in Section 3.2. In particular, these two markers are used to determine the points in the program where asynchronous function calls have been made, thereby

```
1 Trace Record Prefix:
2   changeBanner:::4
3 Variables:
4   currentBannerID (global): 1
5   changeTimer (global): 2
6   bannerID (local): none
7   prefix (local): none
8   currBannerElem (local): none
9   bannerToChange (local): none
```

Figure 3.3: Example trace record for Line 5 of the running example from Figure 3.1.

simplifying the process of mapping each execution trace to its corresponding event. If a line l is marked with the ASYNC or ASYNCCALL marker, then the values $l.async$ or $l.asynccall$, respectively, are set to `true`.

The ERROR marker is added when a failure is detected (the mechanism to detect failures is discussed in Section 3.5). It contains information about the exception thrown and its characteristics. In the example in Figure 3.1, the ERROR marker is placed in the trace after the entry corresponding to Line 9, as the null exception is thrown at this line.

The second marker, ASYNCCALL, is placed after an asynchronous call to a function (e.g., via the `setTimeout` function). Each ASYNCCALL marker contains information about the caller function and a unique identifier that distinguishes it from other asynchronous calls. Every ASYNCCALL marker also has a corresponding ASYNC marker, which is placed at the beginning of the asynchronous function's execution, and contains the name of the function as well as the identifier of the asynchronous call. In the example in Figure 3.1, an ASYNCCALL marker is placed in the trace after the execution of Line 13, which has an asynchronous call to `changeBanner`. The corresponding ASYNC marker is placed before the execution of Line 2, at the beginning of the asynchronously called function `changeBanner`.

To insert the ASYNCCALL and ASYNC markers, the known asynchronous functions in JavaScript are overridden by a trampoline function that sets up and writes the ASYNCCALL marker to the trace. The trampoline function then calls the original function with an additional parameter indicating the identifier of the asynchronous call. This parameter is written to the trace within the called function

along with the ASYNC marker to uniquely identify the asynchronous call.

3.4.3 Trace Analysis

Once the trace of executed statements has been collected, the trace analysis phase begins. The goal of this phase is to analyze the trace entries and find the direct DOM access responsible for the JavaScript failure. First, the approach partitions the trace into *sequences*, where a sequence $(l_1, l_2, ..., l_n)$ represents the series of JavaScript statements $l_1, l_2, ..., l_n$ that were triggered by the same event (e.g., a page load). Each sequence corresponds to exactly one event. This step corresponds to box 4 in Figure 3.2. As mentioned in the previous section, the executed JavaScript program in the example in Figure 3.1 consists of two sequences: one corresponding to the load event, and the other corresponding to the timeout event.

After partitioning the trace into sequences, the algorithm looks for the sequence that contains the direct DOM access (box 5 in Figure 3.2). This is called the *relevant sequence*. The relevant sequence ρ is initially chosen to be the sequence that contains the ERROR marker,[16] that is, at the beginning of the algorithm, ρ is initialized as follows:

$$\rho \leftarrow (l_1, l_2, ..., l_n) \iff \exists l_i \in \{l_1, l_2, ..., l_n\},\ l_i.failure = \texttt{true} \tag{3.1}$$

This marker will always be the last element of the relevant sequence, since the execution of the sequence must have halted once the failure occurred; hence, it suffices to check if $l_n.failure = \texttt{true}$ in Expression (3.1). The direct DOM access will be found within the initial relevant sequence provided the sequence was not triggered by an asynchronous function call but rather by the page load or user-triggered event. However, if the relevant sequence was triggered asynchronously, i.e., it begins with an ASYNC marker, then the sequence containing the corresponding asynchronous call (i.e., with the ASYNCCALL marker) is prepended to the relevant sequence to create the new relevant sequence. This process is continued recursively until the top of the trace is reached or the sequence does not begin with an ASYNC marker.

[16]For output-related DOM-related JavaScript faults, the ERROR marker is replaced by an analogous marker that represents the failure line identified by the user.

```
1 Sequence 1:
2   root:::12 (Line 12)
3   root:::13 (Line 13)
4   root:::ASYNC_CALL - ID = 1
5 Sequence 2:
6   changeBanner:::ASYNC - ID = 1
7   changeBanner:::1 (Line 2)
8   changeBanner:::2 (Line 3)
9   changeBanner:::4 (Line 5)
10  changeBanner:::5 (Line 6)
11  changeBanner:::6 (Line 7)
12  changeBanner:::7 (Line 8)
13  changeBanner:::8 (Line 9) - FAILURE
14 Relevant Sequence:
15  root:::12 (Line 12)
16  root:::13 (Line 13)
17  changeBanner:::1 (Line 2)
18  changeBanner:::2 (Line 3)
19  changeBanner:::4 (Line 5)
20  changeBanner:::5 (Line 6)
21  changeBanner:::6 (Line 7) **
22  changeBanner:::7 (Line 8)
23  changeBanner:::8 (Line 9) - FAILURE
```

Figure 3.4: Abridged execution trace for the running example showing the two sequences and the relevant sequence. Each trace record is appended with either a marker or the line number relative to the function. Numbers in parentheses refer to the line numbers relative to the entire JavaScript file. `root` refers to code outside a function. The line marked with a (**) is the direct DOM access, and the goal of this design is to correctly identify this line as the direct DOM access.

In the running example, the relevant sequence is initially set to the one corresponding to the timeout event and consists of (Line 2, Line 3, Line 5, Line 6, Line 7, Line 8, Line 9) (see Sequence 2 in Figure 3.4). Because the relevant sequence begins with an ASYNC marker, the sequence containing the asynchronous call (see Sequence 1 in Figure 3.4) is prepended to it to create the new, final relevant sequence. However, there are no more sequences left in the trace and the process terminates. Although in this example, the relevant sequence consists of all executed statements, this will not always be the case, especially in complex web applications where many events are triggered.

Once the relevant sequence has been found, the algorithm starts locating the direct DOM access within that sequence (box 6 in Figure 3.2). To do so, it analyzes the backward slice of the variable in the line marked with the ERROR marker, i.e., the line l such that $l.failure = \texttt{true}$. If the line l itself contains the direct DOM

access, the process is halted and the line is identified as the direct DOM access. If not, a variable called `null_var` is introduced to keep track of the most recent variable to have held the `null` value.

The initial value of `null_var` is inferred from the error message contained in the ERROR marker. The message is typically of the form ***x** is null*, where **x** is the identifier of a variable; in this case, the initial value of `null_var` is set to the identifier **x**. The relevant sequence is traversed backward and `null_var` is updated based on the statement encountered:

1. If the statement is an assignment of the form `null_var = new_var`, `null_var` is set to the identifier of `new_var`.

2. If it is a return statement of the form `return ret_var;`, where the return value is assigned to the current `null_var` in the calling function, `null_var` is set to the identifier of `ret_var`.

3. If it is a function call of the form `foo(..., arg_var ,...)` where `foo()` is a function with `arg_var` as one of the values passed, and the current `null_var` is the parameter to which `arg_var` corresponds in the declaration of `foo()`, `null_var` is set to the identifier of `arg_var`.

If the line does not fall into any of the above three forms, it is ignored and the algorithm moves to the previous line. Note that although syntactically valid, an assignment of the form `null_var = new_var1 op new_var2 op ...`, where `op` is a binary operator, makes little semantic sense as these operations are not usually performed on DOM element nodes (for instance, it makes no sense to add two DOM element nodes together). Hence, it is assumed that such assignments will not appear in the JavaScript code. Therefore, at every statement in the code, `null_var` takes a unique value. In addition, this implies that there can only be one possible direct DOM access along the null propagation path.

The algorithm ends when `new_var`, `ret_var`, or `arg_var` is a call to a DOM access function. The line containing this DOM access is then identified as the direct DOM access.

In the example in Figure 3.1, the `null_var` is initialized to `bannerToChange`. The trace analyzer begins at Line 9 where the ERROR marker is placed; this is also

the last line in the relevant sequence, as seen in Figure 3.4. Because this line does not contain any DOM access functions, the algorithm moves to the previous line in the relevant sequence, which is Line 8. It then determines that Line 8 does not take on any of the above three forms and moves to Line 7. The algorithm then determines that Line 7 is of the first form listed above. It checks the `new_var` expression and finds that it is a DOM access function. Therefore, the algorithm terminates and identifies Line 7 as the direct DOM access.

3.4.4 Support for Challenging Cases

As explained in Section 3.2.3, programmers typically use features of the JavaScript language that complicate the process of analyzing JavaScript code. This subsection describes how the approach was extended to handle these features.

Eval

As described in Section 3.4.2, the approach instruments each line of JavaScript code to retrieve three pieces of information, namely the containing function, the line number, and an array of the names and values of all in-scope variables. The function that is responsible for adding this information to the execution trace is as follows, where the parameters correspond to the retrieved information.

```
send(functionName, lineNo, variableArray)
```
(3.2)

The `send()` function is included prior to every line of JavaScript code, which is useful for retrieving trace records for statically loaded code; however, the `send()` function does not collect trace records for JavaScript code generated through `eval`.

A naïve approach for extending the approach to handle `eval` would be to simply add a call to `send()` prior to every line in the string passed to the `eval` call. The problem with this approach is that the `eval` parameter is not necessarily a string literal; hence, its value may not be known until the `eval` call is made at runtime. To make the approach more general, every call to `eval` in the JavaScript code is replaced with a call to a wrapper function called `processEval()`. This function first evaluates the *string value* of the expression passed to `eval`. Thereafter, the function parses this string value and adds a call to the `send()` function

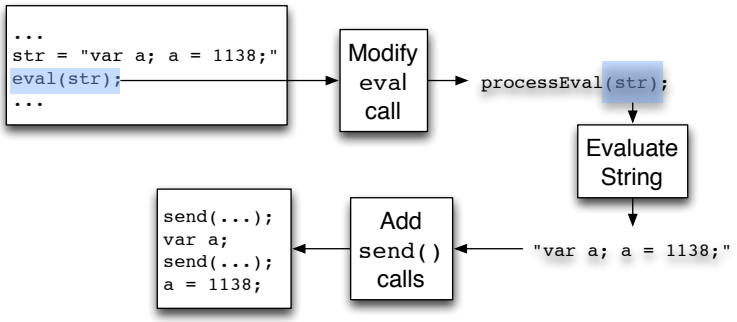


Figure 3.5: Example illustrating the approach for supporting `eval`.

prior to each expression statement in the parsed string; this generates a new string, which comprises of the original parameter to `eval`, but with a call to `send()` prior to each statement. Finally, this new string is passed to `eval` in order for the corresponding code to execute. The approach described is illustrated in Figure 3.5. Note that the string value passed to `eval` can always be resolved, since the string is processed dynamically at runtime.

New Variables. Note that it is possible for new variables to be declared in `eval` code. Hence, the approach needs a way to update the array of variables that is passed to the `send()` function. To do this, an array object is created at the beginning of every function in the JavaScript code. The array is initialized with the names and scopes of all the variables declared within the function, and is updated with new variables whenever a call to `processEval()` is made.

Since the string value passed to `eval` is parsed separately from the rest of the JavaScript code (i.e., the code outside `eval`), `processEval()` may inaccurately label the scope of a variable defined in the `eval` code as "global". In order to make sure the variables declared in `eval` are marked with the correct scope (i.e., local or global), a *scope marker* is passed to `processEval()` as a parameter. This scope marker contains the value `root` if the `eval` call is made outside a function; otherwise, the scope marker is assigned the name of the function. Thus, if the scope marker has value `root`, the "global" markings are retained for the new variables; otherwise, the "global" markings are changed to "local". This ensures that variable scopes are accurately recorded.

Anonymous Functions

The initial approach relied on the names of the functions containing the line numbers to determine the dynamic backward slice. More specifically, during the trace collection phase, the technique records the name of the function, which is then included in the corresponding trace record. During the trace analysis phase, the technique then fetches the name of the function from the trace record so that it knows where to find the line of JavaScript code that needs to be analyzed.

Unfortunately, this approach does not work for anonymous functions, since they are not given a name. To account for this limitation, the trace collection scheme was modified so that it assigns a unique name for every anonymous function encountered during the crawling. In particular, each anonymous function is assigned a name of the form *anonymous-file-script-line*, where *file* is the file name; *script* is the index of the script tag containing the function (i.e., the order that the script tag appears in the file); and *line* is the line number of the function relative to the script tag in which it is defined. Note that the script tag is only applicable to JavaScript code embedded in `.html` files. If the code is included in a `.js` file, *script* is simply assigned 0.

In the trace analysis phase, if a trace whose corresponding function name is of the form *anonymous-file-script-line* is encountered, then the trace must correspond to a line of code located in an anonymous function. The location of the line of code is determined by taking the *file*, *script*, and *line* portions of the function name, and the line of code is fetched once found.

Minified Code

In order to handle minified code, the approach first "beautifies" (i.e., unminifies) this code. The trace collection and trace analysis phases will then both proceed as before, but this time, operating on the beautified version of the code. The problem is that by operating on the beautified version, the approach will output the line number of the direct DOM access *in the beautified version*; since this beautified version is transparent to the developer, this line number will not be very meaningful.

Therefore, the challenge, in this case, is in mapping every line number in the beautified version with the *column* number in the minified version. In most cases,

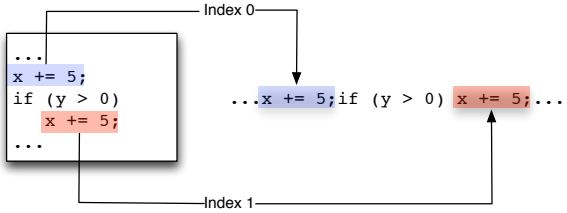


Figure 3.6: Example illustrating the mapping from the beautified version to the minified version, in the approach for supporting minified code.

this mapping is achieved by performing a string match with the minified version, and identifying the starting column of the matching string. However, this approach will not always work because of the possibility of identical lines; for instance, if the running example were originally minified, and Figure 3.1 is the beautified version, then lines 3 and 13 – which are identical – will lead to multiple matches with the minified version. To account for this possibility, a regular expression is used to identify all the identical lines in the beautified version. The identical lines in the beautified version are then sequentially assigned an index, and a regular expression is used to find *all* the matches in the minified version. In this case, the line with the *n*th index is mapped to the column where the *n*th match is located. This is illustrated in Figure 3.6.

3.4.5 Assumptions

The described approach makes a few simplifying assumptions, listed below. In the evaluation described in Section 3.6, the correctness of the approach will be assessed on various open-source web applications, thus evaluating the reasonableness of these assumptions in the real world.

1. The JavaScript error is manifested in a null exception, where the null value is originated from a call to a DOM access function.

2. There are no calls to recursive functions in the relevant sequence. More specifically, the approach relies on the (*name*, *file*, *script*) tuple – where *name* is either the function name or, in the case of anonymous functions, some uniquely assigned name – to distinguish functions from each other. Since traces that point to a recursive function map to the same tuple, the approach

cannot distinguish between calls to the same line from different recursion levels.

3. There are no object property accesses in the null propagation path. In other words, the approach assumes that `null_var` will only be a single identifier, and not a series of identifiers connected by the dot operator (e.g., `a.property`, `this.x`, etc.)

3.5 Tool Implementation

The approach described in Section 3.4 has been implemented in an automated tool called AUTOFLOX[17] using the Java programming language. In addition, a number of existing tools are used to assist in the trace collection phase, including RHINO [119] for parsing and instrumenting the JavaScript code, and jsbeautifier [99] for beautifying minified code.

AUTOFLOX has been implemented in two different interfaces.

CRAWLJAX Interface. In the first interface, AUTOFLOX prompts the user for the URL of the web application containing the fault, and crawls this application to perform trace collection. Here, the CRAWLJAX [110] tool is used to systematically crawl the web application and trigger the execution of JavaScript code corresponding to user events. Other tools such as WaRR [10], Mugshot [111], and Selenium [75] can aid in the reproduction phase. However, those tools require manually or programatically interacting with the web application at hand. Thus, CRAWLJAX was used because of the level of automation and flexibility it provides. Prior to crawling the web application, the AUTOFLOX user can specify which elements in the web application the crawler should examine during the crawling process (otherwise the default settings are used). These elements should be chosen so that the JavaScript error is highly likely to be reproduced.[18] In this mode, only one fault can be localized at a time.

Eclipse Interface. In the second interface, AUTOFLOX runs as an Eclipse IDE [44] plugin. Here, the programmer can develop her web application project on Eclipse;

[17] http://ece.ubc.ca/~frolino/projects/autoflox/

[18] While non-deterministic errors can be localized with AUTOFLOX, they may require multiple runs to reproduce the error (i.e., until the error appears)

with the project files open, she can subsequently click the "Run AUTOFLOX" button to run the tool. Doing so will open the Firefox web browser, which allows the user to replicate the fault by either interacting with the application, or running a test case (e.g., a Selenium test case). Where applicable, AUTOFLOX will then output the direct DOM access each time a null exception is thrown. Note that in this interface, AUTOFLOX is able to localize multiple faults by assigning a unique ID to each exception encountered.

The JavaScript code instrumentation and tracing technique used in the proposed approach is based on an extension of the INVARSCOPE [63] plugin to CRAWLJAX. The following modifications were made to INVARSCOPE in order to facilitate the trace collection process:

1. While the original INVARSCOPE tool only collects traces at the function entry and exit points, the modified version collects traces at every line of JavaScript code to ensure that the complete execution history can be analyzed in the trace analysis phase.

2. The original INVARSCOPE does not place information on the scope of each variable in the trace; thus, it has been modified to retrieve this information and include it in the trace.

3. The modifications allow asynchronous function calls to be overridden, and to place extra instrumentation at the beginning of each function to keep track of asynchronous calls (i.e., to write the ASYNCCALL and ASYNC markers in the trace).

4. Finally, `Try-Catch` handlers are placed around each function call in the JavaScript code in order to catch exceptions and write ERROR markers to the trace in the event of an exception.

Note that the tool allows the user to exclude specific JavaScript files from being instrumented. This can speed up the trace collection process, especially if the user is certain that the code in those files does not contain the direct DOM access.

Finally, the trace analysis phase has also been added as a part of the AUTOFLOX tool implementation, and requires no other external tools.

3.6 Empirical Evaluation

3.6.1 Goals and Research Questions

We conducted an empirical study to evaluate the accuracy and real-world relevance of the proposed fault localization approach.

The research questions that are answered in the evaluation are as follows:

RQ1: *What is the fault localization accuracy of* AUTOFLOX*? Are the implementation assumptions reasonable?*

RQ2: *Is* AUTOFLOX *capable of localizing bugs from real-world web applications?*

RQ3: *What is the performance overhead of* AUTOFLOX *on real-world web applications?*

3.6.2 Methodology

The subsections that follow address each of the above questions. An overview of the evaluation methodology used to answer each research question is shown below.

To answer **RQ1**, AUTOFLOX is run on six open-source web applications and three production websites. DOM-related JavaScript faults are injected into the applications and AUTOFLOX is run to localize the direct DOM accesses corresponding to the faults.

To address **RQ2**, AUTOFLOX is subjected to 20 bugs (which satisfy the fault model) that have previously been observed and reported for seven open-source web applications. Most of these bugs come from the bug report study in Chapter 2.

The performance (**RQ3**) is measured by calculating the overhead incurred by the instrumentation and the time it takes for the tool to find the direct DOM access.

Note that the experiments were performed on an Ubuntu 12.04 platform using the Firefox v. 33.0.2 web browser. The machine used was a 2.66 GHz Intel Core 2 Duo, with 4 GB of RAM.

3.6.3 Accuracy of AUTOFLOX

To answer RQ1, a fault injection experiment was performed on six open-source web applications, shown in Table 3.1. As seen in this table, the applications consist of thousands of lines of JavaScript code each. Fault injection was used to establish the ground truth for measurement of the accuracy of AUTOFLOX. However, the fault injection process was not automated. Rather, a search was first made for calls to DOM access functions – either from the DOM API or from popular JavaScript libraries – that return `null`, such as `getElementById()`, `getAttribute()` and `$()`. The faults were then manually injected by mutating the parameter of the DOM access function into some garbage value; this parameter mutation will ensure that the call to the function will return a `null` value, thereby leading to a null exception in a later usage, and emulating DOM-related faults. In order to demonstrate the effect of adding support for anonymous functions, `eval`, and minified code, the fault injection experiment was run twice – once with these new modules enabled, and once with the modules disabled.

Only one mutation is performed in each run of the application to ensure controllability. For each injection, the direct DOM access is the mutated line of JavaScript code. Thus, the goal is for AUTOFLOX to successfully identify this mutated line as the direct DOM access, based on the message printed due to the exception.

Furthermore, localization was performed on injected faults rather than actual faults because no known code-terminating DOM-related faults existed in these web applications at the time of the experiment. However, AUTOFLOX is also used to localize real faults that appear in seven other web applications, which is described in further detail in Section 3.6.4.

Table 3.1: Results of the experiment on open-source web applications, assessing the accuracy of AUTOFLOX.

JavaScript Web Applications	Lines of JS code	# of mutations	# direct DOM accesses identified	`eval` Support Increase	Anon. Support Increase	Percentage identified
TASKFREAK	3044	39	39 (38)	+1	–	100% (97.4%)
TUDU	11653	9	9 (0)	–	+9	100% (0%)
WORDPRESS	8366	17	14 (9)	–	+5	82.4% (52.9%)
CHATJAVASCRIPT	1372	10	10 (10)	–	–	100% (100%)
JSSCRAMBLE	131	6	6 (6)	–	–	100% (100%)
JS TODO	241	2	2 (1)	–	+1	100% (50%)
OVERALL		83	80 (64)	+1	+15	96.4% (77.1%)

Table 3.1 shows the results of the experiments; the results for the case where the new modules are disabled are shown in parentheses. As shown in the table, with the new modules enabled, AUTOFLOX was able to identify the direct DOM access for all mutations performed in five of the six applications, garnering *100% accuracy* in these applications; when all six applications are considered, the overall accuracy was 96.4%. In contrast, when the new modules are disabled, only two of the applications (CHATJAVASCRIPT and JSSCRAMBLE) had perfect accuracies, and the overall accuracy was significantly lower, at 77.1%.

Taking a closer look at the unsuccessful cases when the new modules are disabled, it was found that AUTOFLOX was not able to accurately pinpoint the direct DOM access because in these cases, the dynamic backward slice included lines from anonymous function code and `eval` code. In particular, 15 of the unsuccessful cases resulted from the presence of anonymous function code, while 4 of the unsuccessful cases resulted from the presence of `eval` code. This result demonstrates that these features are commonly used in JavaScript code, and it is therefore important to add support for them, as has been done.

For the case where the new modules are enabled, the only application for which AUTOFLOX had imperfect accuracy was WORDPRESS, where it failed to detect three direct DOM access lines; in all three cases, AUTOFLOX generated an error message stating that the direct DOM access could not be found. Further analysis of the JavaScript code in WORDPRESS revealed that in these three unsuccessful cases, the dynamic backward slice included calls to the `setTimeout()` method, where the first parameter passed to the method is a function literal; this is currently not supported by AUTOFLOX.[19] Note, however, that this is an implementation issue which does not fundamentally limit the design; one possible way to overcome this problem is by integrating the parameter of `setTimeout()` as part of the `eval`-handling module.

Overall, this experiment demonstrates that AUTOFLOX has an accuracy of 96.4% across the six open-source web applications. Note that AUTOFLOX had *n*o false positives, i.e., there were no cases where the tool incorrectly localized a fault or said that a fault had been localized when it had not.

[19] Note that in the running example, the `setTimeout()` call is passed a function *identifier*, which is different from a function *literal*.

Table 3.2: Results of the experiment on production websites, assessing the robustness of AUTOFLOX (in particular, how well it works in production settings).

Production Website	Total Number of faults	Number of direct DOM accesses identified	Anonymous Support Increase	Minified Support Increase
ubuntu.com	1	1 (0)	+1	–
Hacker News	2	2 (2)	–	–
W3C School	1	1 (0)	–	+1
OVERALL	4	4 (2)	+1	+1

Production website: AUTOFLOX was also used to localize faults on three production websites, listed in Table 3.2. For these websites, the faults are injected in the homepage, in a way similar to what was done in the fault injection experiment described earlier.[20] As before, the experiment was run twice, enabling the new modules in one case, and disabling them in the other.

As Table 3.2 shows, AUTOFLOX was able to identify all four of the direct DOM accesses, with the new modules enabled. In contrast, with the new modules disabled (whose results are shown in parentheses in Table 3.2), the tool only identified two of the four direct DOM accesses. One of the unsuccessful cases (ubuntu.com) resulted from the presence of an anonymous function, while the other unsuccessful case (W3C School) resulted from the presence of minified code (which is common practice in many production websites).

The overall implication of this study is that the assumptions made by AUTOFLOX *are reasonable as they were followed both in the open-source applications and in the production websites*. Further, *the new features added to* AUTOFLOX *significantly boosted its accuracy*. Later in Section 3.7, the broader implications of the assumptions made by AUTOFLOX are discussed.

3.6.4 Real Bugs

In order to assess how well AUTOFLOX works on actual bugs that have appeared in real-world web applications, we collected 19 bug reports from the applications

[20] Since we did not have write access to the JavaScript source code for these websites, the mutation was performed by manually modifying the code intercepted by the proxy.

in our bug report study (Chapter 2) and subjected AUTOFLOX to them to see if it can successfully identify the direct DOM access. In addition, we found a real bug in the production website 'tumblr', which was also included in the experiment; this is the running example described earlier. These 20 bugs come from seven open-source web applications and libraries, shown in Table 3.3, and have all been fixed by their respective developers. Here, ground truth is established by comparing the direct DOM access output by AUTOFLOX with the actual direct DOM access, which is identified by the developers in the corresponding bug report (or, in the case of tumblr, by analyzing the code to determine the fix).

In the end, AUTOFLOX was able to successfully identify the direct DOM access for *all 20 of the bugs*. Table 3.3 also shows the distance of the backward slice from the direct DOM access to the line where the `null` exception takes place (second-last column), and the number of functions spanned by the backward slice (last column). Note that finding the direct DOM access is non-trivial for some of these bugs. For example, one of the bugs in Moodle (MD2) had a dynamic backward slice that spanned multiple lines within the same function, and in fact the variable to which the DOM element is being assigned is constantly being reused in that function to refer to other elements. In addition, the dynamic backward slice for a bug in WordPress (WP1) spanned over 40 lines, spread out across multiple functions; manually tracing through these lines and functions can evidently be a time-consuming process. This demonstrates the usefulness of AUTOFLOX, as well as its robustness, as it is capable of localizing real bugs in real web applications.

Table 3.3: Web applications and libraries in which the real bugs to which AUTOFLOX is subjected appear.

Application/Library	Number of Bugs	Bug Identifier	Identified by AUTOFLOX?	Backward Slice Length (LOC)	Number of Functions in Backward Slice
Joomla	3	JM1	✓	2	1
		JM2	✓	2	1
		JM3	✓	8	2
Moodle	6	MD1	✓	2	1
		MD2	✓	5	1
		MD3	✓	7	2
		MD4	✓	8	2
		MD5	✓	2	1
		MD6	✓	1	1
MooTools	2	MT1	✓	1	1
		MT2	✓	1	1
Prototype	1	PT1	✓	1	1
Tumblr	1	TB1	✓	1	1
WikiMedia	4	WM1	✓	2	1
		WM2	✓	1	1
		WM3	✓	2	1
		WM4	✓	1	1
WordPress	3	WP1	✓	44	4
		WP2	✓	5	1
		WP3	✓	10	2

Table 3.4: Performance results

Production Website	Trace Collection Overhead	Total Time (seconds)
ubuntu.com	63.3%	50.3
Hacker News	30.9%	28.8
W3C School	119.3%	56.8
Tumblr	35.0%	33.3

3.6.5 Performance

The performance overhead of AUTOFLOX is reported in this section. The measurements are performed on the production websites because production code is more complex than development code (such as the ones in the open-source web applications tested above), and hence incurs higher performance overheads. The following metrics are measured: (1) performance overhead due to instrumentation in the trace collection phase, and (2) time taken by the trace analyzer to find the direct DOM access. To measure (1), the production websites are crawled using CRAWLJAX both with instrumentation and without instrumentation; the baseline is the case where the web application is run only with CRAWLJAX. For measuring (2), AUTOFLOX was run on the collected trace. Note that the Eclipse interface experiences similar overheads.

Table 3.4 shows the performance measurements. As the table shows, the overhead incurred by the trace collection phase (average of three runs) ranges from 30.9% in Hacker News to 119.3% in W3C School. Also, on average, the trace analysis phase ran for 0.1s in all four websites. Note that AUTOFLOX's trace collection module is only intended to be turned on when a fault needs to be localized – when interacting with a website as normal, the module will be off; hence, the high overheads in some websites (e.g., W3C School) are not expected to be problematic. Indeed, AUTOFLOX does not run for more than a minute in any of the websites, from trace collection to fault localization.

3.7 Discussion

Some issues relating to the limitations of AUTOFLOX and some threats to the validity of the evaluation are now discussed.

3.7.1 Limitations

Currently, AUTOFLOX requires the user to specify the elements that will be clicked during the web application run to replicate the failure. This process can be tedious for the programmer if she is not aware of all the DOM elements (and their corresponding IDs) present in the web application, and will often require the programmer to search for these elements in the source code of the web application. The Eclipse plugin version of AUTOFLOX mitigates this problem to a certain extent, by asking the user to replicate the failure by manually interacting with the web application; however, doing this for a large set of bugs may be tedious, and ways to automate this process without sacrificing accuracy are currently being explored.

One way to simplify the above task and effectively automate the process of identifying all the DOM IDs is to do a preliminary run of the web application that detects all the DOM elements — where all elements are considered clickable — and present this list of DOM elements to the user. However, this approach would have the disadvantage of having to run the web application multiple times, which would slow down the fault localization process. In addition, this approach may not be able to detect DOM elements created dynamically by the JavaScript code if only a subset of the web application is crawled.

As seen from the accuracy results in the evaluation, although AUTOFLOX is capable of handling calls to `eval` (or `eval`-like functions such as `setTimeout`) where a string (or a concatenation of strings) is passed as the parameter, it currently does not support the case where a function literal is passed as a parameter to this function. Based on the applications that were evaluated, passing function literals to `eval`-like functions does not seem to be common practice among developers. Most parameters to `setTimeout`, for instance, are in the form of function identifiers, similar to the running example.

3.7.2 Threats to Validity

An external threat to the validity of the evaluation is that only a limited number of web applications are considered to assess the correctness of AUTOFLOX. However, these applications have been chosen as they contain many lines of JavaScript code, thereby allowing multiple fault injections to be performed per application.

In terms of internal validity, a fault injection approach was used to emulate the DOM-related faults in the evaluation. The threat here is that the faults injected may not be completely representative of the types of faults that happen in the real world. Nonetheless, the bug report described in Chapter 2 provides supportive evidence that the bugs that were injected are prominent and must therefore be considered. Further, AUTOFLOX was also tested on real bugs in one of the experiments, demonstrating its applicability in more realistic settings.

Finally, while AUTOFLOX is openly available, and the fault injection experiment on the six open-source web applications is replicable, the experiment on production websites is not guaranteed to be replicable, as the source code of these websites may change over time, and we do not have access to prior versions of the website.

3.8 Related Work

Here, related work is classified into two broad categories: web application reliability and fault localization.

3.8.1 Web Application Reliability

Web applications have been an active area of research for the past decade. The work described in this chapter focuses on reliability techniques that pertain to JavaScript-based web applications, which are a more recent phenomenon.

Static analysis. There have been numerous studies to find errors and vulnerabilities in web applications through static analysis [18, 64, 65, 186]. Because JavaScript is a difficult language to analyze statically, these techniques typically restrict themselves to a safe subset of the language. In particular, they do not model the DOM, or they oversimplify the DOM, which can lead to both false positives and false negatives. Jensen et al. [79] 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. In contrast, the proposed technique is based on dynamic execution, and as a result, does not suffer from false positives.

Testing and replay. Automated testing of JavaScript-based web applications is an active area of research [15, 109, 115, 137]. ATUSA [109] is an automated technique for enumerating the state space of a JavaScript-based web application and finding errors or invariant violations specified by the programmer. JSart [113] and DODOM [137] dynamically derive invariants for the JavaScript code and the DOM respectively. Finally, MUTANDIS [114] determines the adequacy of JavaScript test cases using mutation testing. However, none of these techniques focus on fault localization. Alimadadi et al. recently introduced a program comprehension tool called CLEMATIS [8], which maps user events to JavaScript code; although this tool can help the developer narrow down the list of JavaScript lines to consider, it does not pinpoint a precise fault location, as AUTOFLOX does.

WaRR [10], Mugshot [111], and Jalangi [151] – among others [30, 178] – replay a web application's execution after a failure in order to reproduce the events that led to the failure. However, they do not provide any support for localizing the fault, and leave it to the programmer to do so. As shown in Section 3.2, this is often a challenging task.

Finally, tools such as Firefox's Firebug [70] plug-in exist to help JavaScript programmers debug their code. However, such tools are useful only for the bug identification phase of the debugging process, and not the fault localization phase.

3.8.2 Fault Localization

Fault localization techniques isolate the root cause of a fault based on the dynamic execution of the application. They can be classified into Spectrum-based and Slicing-based.

Spectrum-based fault localization techniques [24, 154, 188] include Pinpoint [35], Tarantula [84], Whither [143], and MLNDebugger [188]. Additionally, MUSE [117] and FIFL [183] also perform spectrum-based fault localization based on injected mutants in traditional programs, focusing primarily on regression bugs. These techniques execute the application with multiple inputs and gather the dy-

namic execution profile of the application for each input. They assume that the executions are classified as success or failure, and look for differences in the profile between successful and failing runs. Based on the differences, they isolate the parts of the application, which are likely responsible for the failure. However, spectrum-based techniques are difficult to adapt to web applications, as web applications are rarely deterministic, and hence they may incur false positives. Also, it is not straightforward to classify a web application's execution as success or failure, as the results depend on its usage [42].

Slicing-based fault localization techniques have been proposed by Agrarwal et al. [4] and Zhang et al. [185]. These techniques isolate the fault based on the dynamic backward slice of the faulty statement in the code. AUTOFLOX is similar to this body of work in that it also extracts the dynamic backward slice of the JavaScript statement that throws an exception. However, it differs in two ways. First, it focuses on errors in the DOM-JavaScript interaction. The DOM is unique to web applications and hence the other fault-localization techniques do not consider it. Second, JavaScript code is often executed asynchronously in response to events such as mouse clicks and timeouts, and does not follow a deterministic control-flow (see Section 3.2.2 for more details).

Web Fault localization. As a complementary tool to AUTOFLOX, we developed VEJOVIS [131], which is a JavaScript fault repair suggestion tool, and is described in more detail in Chapter 4. This tool is similar to AUTOFLOX in that it also targets DOM-related faults and uses a backward slicing approach. However, unlike AUTOFLOX, which starts with the line of code that leads to the failure and tries to find the direct DOM access by examining the dynamic backward slice, VEJOVIS starts with the direct DOM access, and examines its parameters to see how it can be fixed to match the DOM.

To the best of our knowledge, the only papers apart from the current work that has explored fault localization in the context of web applications are those by Artzi et al. [14] and Samimi et al. [148]. Like AUTOFLOX, the goal of the tools proposed in these papers is to automatically localize web application faults, achieving high accuracies. However, their work differs from the current one in various aspects: (1) they focus on the server-side code, i.e., PHP, while the current work focuses on the client-side and (2) they localize HTML validation errors, while the current

work's proposed approach localizes JavaScript faults. In addition, Artzi et al. have opted for a spectrum-based approach based on Tarantula, while AUTOFLOX is a dynamic slicing-based approach. To the best of our knowledge, automated fault localization for JavaScript-based web applications has not been addressed in the literature.

3.9 Conclusions and Future Work

In this chapter, we introduced a fault-localization approach for JavaScript-based web applications. The approach is based on dynamic slicing, and addresses the two main problems that inhibit JavaScript fault localization, namely asynchronous execution and DOM interactions. Here, the focus is on DOM-related JavaScript faults, which is the most prominent class of JavaScript faults. The proposed approach has been implemented as an automated tool, called AUTOFLOX, which is evaluated using six open-source web applications and four production websites. The results indicate that AUTOFLOX can successfully localize over 96% of the faults, with no false positives.

There are several ways in which the work outlined in this chapter will be extended. First, the current work focuses on code-terminating JavaScript faults, i.e., faults that lead to an exception thrown by the web application. However, not all DOM-related faults belong to this category. The design will therefore be extended to include a more automated technique for localizing non-code terminating JavaScript faults. In addition, the empirical evaluation will be extended to perform user studies of the AUTOFLOX tool, in order to measure its ease of use and efficacy in localizing faults. This is also an avenue for future work.

Chapter 4

Automatic JavaScript Fault Repair

This chapter describes our technique for automatically suggesting repairs for DOM-related faults.[21] Our goal in this chapter is once again to answer RQ1B from Chapter 1.1, but this time, the focus is on its latter half (i.e., fault repair). We implemented our repair technique in a tool called VEJOVIS,[22] which we describe below.

4.1 Introduction

JavaScript is used extensively in modern web applications to manipulate the contents of the webpage displayed on the browser and to retrieve information from the server by using HTTP requests. To alter the contents of the webpage, JavaScript code manipulates the DOM, as discussed in the previous chapters. In particular, through the use of DOM API methods, the JavaScript code is capable of retrieving elements from the DOM, as well as changing the DOM by modifying element properties, adding elements, and deleting elements.

Due to the dynamic nature of the JavaScript language and the interaction with the DOM, JavaScript-based web applications are prone to errors, and indeed, the results of our bug report study in Chapter 2 point to the prevalence, impact, and complexity of DOM-related JavaScript faults in web applications. Therefore, in this chapter, our goal is to *facilitate the process of fixing DOM-related JavaScript faults*, by providing suggestions to the programmer during web application testing

[21] The main study in this chapter appeared at the International Conference on Software Engineering (ICSE 2014) [131].

[22] VEJOVIS is named after the Roman god of healing

and debugging tasks. To this end, we first perform a study of real-world JavaScript faults to understand the common patterns in how programmers fix such faults. Then, based on these common fix patterns, we propose an automatic approach for suggesting repairs. In this chapter, we use the term *repair* to encompass both fixes and workarounds for the fault, similar to other related work [121, 148].

Our approach starts from the wrong DOM API method/property, and uses a combination of static and dynamic analysis to identify the lines of code in the backward slice of the parameters or assignment values of DOM methods/properties. Once these lines are localized, it uses a string solver to find candidate replacement DOM elements, and propagates the candidate values along the backward slice to find the fix.

We implement our approach in an open-source tool called VEJOVIS. VEJOVIS is deployed on a web application after the occurrence and subsequent localization of a JavaScript fault [129]. It requires neither specifications/annotations from the programmer nor any changes to the JavaScript/DOM interaction model, and can hence be deployed on unmodified web applications.

Prior work on suggesting repairs for web application faults has focused on server-side code (e.g., PHP) [121, 148], including workarounds for web API calls [33]. Other work [80] automatically transforms unsafe `eval` calls in JavaScript code to safe alternatives. None of these techniques, however, deal with DOM-related JavaScript faults. *To the best of our knowledge,* VEJOVIS *is the first technique to automatically suggest repairs for DOM-related JavaScript faults in web applications*.

We make the following contributions in this chapter:

- We categorize *common fixes* applied by web developers to DOM-related JavaScript faults, based on an analysis of 190 online bug reports. We find that fixes involving modifications of DOM method/property parameters or assignment values into valid replacement values (i.e., values consistent with the DOM) are the most common, followed by those involving validation of DOM elements before their use;
- Based on the above study, we present an algorithm for finding *valid replacement parameters* passed to DOM API methods, which can potentially be

used to replace the original (and possibly erroneous) parameter used to retrieve elements from the DOM. The replacements are found based on the CSS selector grammar, and using information on the DOM state at the time the JavaScript code executed. The aim is to suggest replacements that are *valid* in the current DOM, and to suggest as few replacements as possible;

- We present an algorithm for suggesting repairs in the form of *actionable messages* to the programmer based on code-context. The actionable messages contain detailed directions prompting the programmer to make modifications to the code so as to eliminate the "symptoms" observed during the program's failure run;
- We describe the implementation, called VEJOVIS, which integrates the previous two contributions. We evaluate our technique on 22 real-world JavaScript bugs from 11 web applications. In our case study, VEJOVIS was able to provide correct repair suggestions for 20 of the 22 bugs. Further, the correct fix was ranked first in the list of repairs for 13 of the 20 bugs. We also found that limiting the suggestions to those that are within an edit distance of 5 relative to the original selector can decrease the number of suggestions in the other 7 bugs to 3 per bug, while reducing the number of correct fixes to 16 (from 20).

4.2 Background and Challenges

Client-side JavaScript is used primarily to interact with i.e., to access, traverse, or manipulate the DOM. In most modern web applications, these interactions are used to incrementally update the browser display with client-side state changes without initiating a page load. Note that this is different from what happens during URL-based page transitions where the entire DOM is repopulated with a new HTML page from the server.

As we saw in Chapter 3, JavaScript provides DOM API methods and properties that allow *direct* and easy retrieval of DOM elements. For instance, elements can be accessed based on their tag name, ID, and class names, and the DOM can be traversed using the `parentNode` and `childNodes` properties. In addition, modern browsers provide APIs such as `querySelector` for retrieving DOM el-

Table 4.1: List of commonly used CSS selector components.

Component	Description
Tag Name	The name of the tag associated with an element. *Examples*: `div`, `span`, `table`, etc.
ID	The id associated with an element. This is prefixed with the # symbol. *Example*: If a `div` element has an id of "myID", a CSS selector that can retrieve this element is `div#myID`.
Class Name	The name of a class to which an element belongs. This is prefixed with a period. *Example*: If a `span` element belongs to the "myClass" class, a CSS selector that can retrieve this element is `span.myClass`.
Is Descendant	A space character indicating that the element described by the right selector is a descendant of the element described by the left selector. *Example*: To find all `table` elements belonging to class "myClass" that are descendants of a `div` element, we use the selector `div table.myClass`.
Is Child	The "`>`" character, which indicates that the element described by the right selector is a child of the element described by the left selector. *Example*: To find all `tr` elements that are children of the `table` element with id "myID", we use the selector `table#myID > tr`.
Is Next Sibling	The "+" character, which indicates that the element described by the right selector is the next sibling of the element described by the left selector. *Example*: To find all `tr` elements that follow another `tr` element, we use the selector `tr + tr`.

ements using patterns called *CSS selectors*. CSS selectors follow a well-defined grammar [167], and serve as a unified way of retrieving DOM elements; for example, retrieving a `DIV` DOM element with ID "news" translates to `"DIV#news"` in CSS selector syntax. Table 4.1 shows some of the commonly used components that make up a CSS selector.

Once an element is retrieved using the CSS selector, JavaScript code can use the reference to that element to access its properties, add new or remove/modify existing properties, or add/remove elements to/from the DOM tree.

Running Example. Here, we describe the running example that we will be using throughout this chapter to simplify the description of our design. The running example is based on a bug in Drupal involving jQuery's Autopager [92] extension, which automatically appends new page content to a programmer-specified DOM-element A snippet of the simplified JavaScript code is shown in Figure 4.1. In the `pagerSetup()` function, the programmer has set the display ID suffix to

```
1 function pagerSetup() {
2   var display = "catalog_view";
3   var content = "p.pages span";
4   appendTo(display, content);
5 }

7 function appendTo(display, content) {
8   var view_selector = "div#view-display-id-" + display;
9   var content_selector = view_selector + " > " + content;
10  var pageToAdd = "<div>New Content</div>";
11  var pages = $(content_selector);
12  var oldContent = pages0.innerHTML;
13  pages0.innerHTML = oldContent + pageToAdd;
14 }
```

Figure 4.1: JavaScript code of the running example.

"catalog_view" (line 2), and the DOM element where the page is added as "p.pages span" (line 3). These inputs are passed to the `appendTo()` function, which sets up the full CSS selector describing where to add the new page through a series of string concatenations (lines 8-9). In this case, the full CSS selector ends up being "div#view-display-id-catalog_view > p.pages span".

The above JavaScript code runs when the DOM state is as shown in Figure 4.2. Note that in this case, the CSS selector will *not* match any element in the DOM state. As a result, `$()` returns an empty set in line 11; hence, when retrieving the old content of the first matching element via `innerHTML` (line 12), an "undefined" exception is thrown. The undefined exception prevents the Autopager to successfully append the contents of the new page (line 10) to the specified element in line 13.

For this particular bug, the fix applied by the programmer was to change the string literal "div#view-display-id-" to "div#view-id-" in line 8. This, in turn, changes the full CSS selector to "div#view-id-catalog_view > p.pages span", which is valid in the DOM in Figure 4.2.

Challenges. The interaction between two separate languages (i.e., the DOM and the JavaScript code) makes web applications highly error-prone – something established in both our study of unhandled JavaScript exceptions [128] and our bug report study in Chapter 2. Since most JavaScript faults are DOM-related as per the results of the latter study, the majority of JavaScript faults originate from errors that

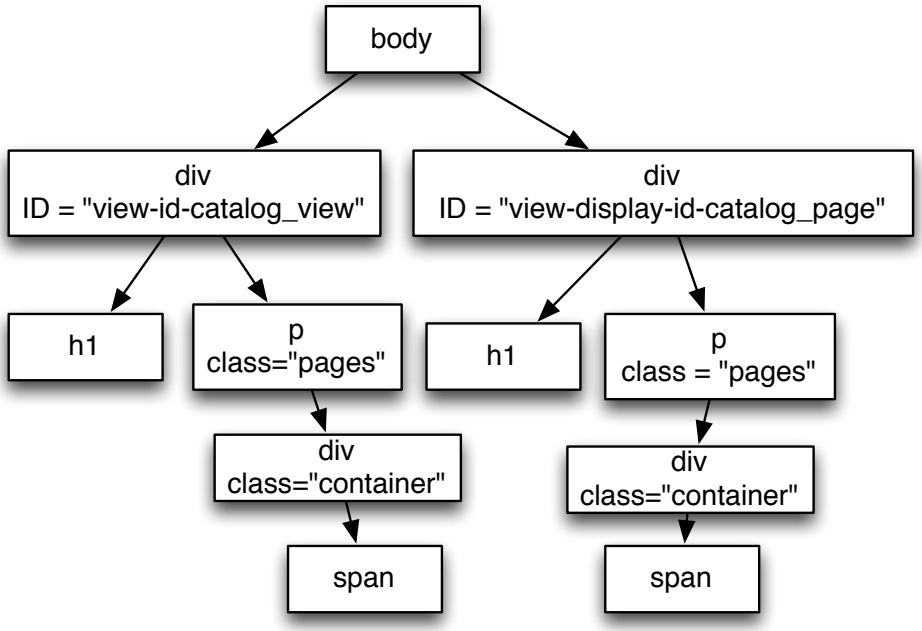


Figure 4.2: The DOM state during execution of the JavaScript code in the running example. For simplicity, only the elements under `body` are shown.

eventually propagate into the parameter or assignment value of a DOM method-/property. The running example provides an example of a DOM-related fault, as the error present in the example eventually propagates into the DOM method `$()`, which retrieves DOM elements through CSS selectors.

DOM-related faults, by definition, involve the propagation of the error to a DOM method's parameter, or a DOM property's assignment value. Therefore, the fix likely involves altering the code responsible for setting up this parameter or assignment value. The challenge, of course, is in answering the following question: *how should the code be modified to repair the fault?* Answering this question requires knowledge of (1) the location in the code that needs to be altered; and (2) the specific modification that needs to be applied to that location. For the first task, the origins (i.e., the backward slice) of the parameter or assignment value must be traced. For the second task, the specific replacement parameter or replacement assignment value must be inferred, and the way in which this replacement should be incorporated into the code must be determined.

While these challenges are difficult and sometimes impossible to tackle with arbitrary method parameters or assignment values (because they require programmer *intent* to be inferred), parameters and assignment values to DOM methods/properties – as well as the DOM itself – are more structured. Hence, for these kinds of parameters or assignment values, the problem of inferring programmer intent reduces to finding replacements that satisfy this structure. Therefore, we study common patterns in how programmers fix DOM-related faults, to prioritize the fix suggestions we infer. This study is presented in the next section.

4.3 Common Developer Fixes

To better understand how programmers implement fixes to DOM-related JavaScript faults, we analyze 190 *fixed* bug reports representing DOM-related faults and analyze the developer fixes applied to them. Our goal is to answer the following research questions:

RQ1 (Fix Categories): What are the common fix types applied by programmers to fix DOM-related JavaScript faults?

RQ2 (Application of Fixes): What modifications do programmers make to JavaScript code to implement a fix and eliminate DOM-related faults?

The above questions will help us determine how programmers typically deal with DOM-related faults. This understanding will guide us to design our automated repair algorithm.

4.3.1 Methodology

We perform our analysis on 190 fixed JavaScript bug reports from eight web applications and four JavaScript libraries (see Table 4.2). Note that these bug reports are a subset of the bug reports we explored in Chapter 2. However, the analysis conducted here is new and is not part of that bug report study.

In the initial version of our bug report study, we analyzed a total of 317 fixed JavaScript bug reports; note that the conference paper on which this current chapter is based was written before we extended our bug report study from 317 to 502 bug reports, which is why we were only able to look at a subset of bugs (317

Table 4.2: List of applications used in our study of common fixes.

Application	Description
Moodle	Learning Management System
Joomla	Content Management System
WordPress	Blogging
Drupal	Content Management System
Roundcube	Webmail
WikiMedia	Wiki Software
TYPO3	Content Management System
TaskFreak	Task Organizer
jQuery	JavaScript Library
Prototype.js	JavaScript Library
MooTools	JavaScript Library
Ember.js	JavaScript Library

out of 502) from our study. Of these 317 bug reports, about 65%, or 206 of the bugs were DOM-related JavaScript faults. Further, we found that in 92% of these DOM-related JavaScript faults (or 190 bugs), the fix involved a modification of the client-side JavaScript code. We consider only these 190 bugs in this study, as our goal is to find repairs for DOM-related JavaScript faults that involve the JavaScript code.

To answer the research questions, we perform a qualitative analysis of the fixes applied by the programmers to each bug report. To do so, we manually read the portions of each bug report documenting the fix applied (e.g., developer comments, discussions, initial report descriptions, fix descriptions, patches, etc.). Based on this analysis, we devise a classification scheme for the bug report fixes so that we can group the fixes into different, well-defined categories, to answer RQ1. Our analysis of the code patches and/or fix descriptions helps us answer RQ2.

4.3.2 Results

Fix Categories. We found that the fixes that programmers apply to DOM-related JavaScript faults fall into the following categories.

Parameter Modification, where a value that is eventually used in the concatenation of a DOM method/property parameter or assignment value is modified. This is done either by directly modifying the value in the code, or by adding calls to modifier methods (e.g., adding a call to `replace()` so that the

string value of a variable gets modified). This category makes up 27.2% of the fixes.

DOM Element Validation, where a check is added so that the value of a DOM element or its property is compared with an expected value before being used. This category makes up 25.7% of the fixes.

Method/Property Modification, where a call to a DOM API method (or property) is either added, removed, or modified in the JavaScript code. Here, modification refers to changing the *method* (or property) originally called, not the parameter (e.g., instead of calling `getElementsByClassName`, the method `getElementsByTagName` is called instead). This category makes up 24.6% of the fixes.

Major Refactoring, where significantly large portions of the JavaScript code are modified and restructured to implement the fix. This category makes up 10.5% of the fixes.

Other/Uncategorized, which make up 12% of the fixes.

As seen in the above fix categories, the most prominent categories are Parameter Modification and DOM Element Validation, which make up over half (52.9%) of the fixes. Therefore, we focus on these categories in our work. Although we do not consider *Method/Property Modifications* in our repair approach, our algorithm can be adapted to include this class of errors, at the cost of increasing its complexity (see Section 4.7).

Application of Fixes. We next describe how programmers modify the JavaScript code to apply the fixes. We discuss our findings for the three most prominent fix categories – Parameter Modification, DOM Element Validation, and Method/Property Modification.

Parameter Modification: We found that 67.3% of fixes belonging to the Parameter Modification fix category involve the modification of string values. The vast majority (around 70%) of these string value modifications were direct modifications of string literals in the JavaScript code. However, we also found cases where the string value modification was applied by adding a call to string modification methods such as `replace()`.

We also analyzed the DOM methods/properties whose parameters are affected

by the modified values. For string value modifications, the methods/properties involved in multiple bug report fixes are `getElementById()`, `$()` and `jQuery()`; together, fixes involving these methods comprise 51.4% of all string value modifications. For non-string value modifications, fixes involved modification of the numerical values assigned to elements' style properties, particularly their alignment and scroll position.

DOM Element Validation: 75.5% of fixes belonging to this category are applied by simply wrapping the code using the pertinent DOM element within an `if` statement that performs the necessary validation (so that the code only executes if the check passes). Other modifications include (1) adding a check before the DOM element is used so that the method returns if the check fails; (2) adding a check before the DOM element is used such that the value of the DOM element or its property is updated if the check fails; (3) encapsulating the code using the DOM element in an `if-else` statement so that a backup value can be used in case the check fails; and finally (4) encapsulating the code in a `try-catch` statement. The most prevalent checks are `null`/`undefined` checks, i.e., the code has been modified to check if the DOM element is `null` or `undefined` before it is used, which constitutes 38.8% of the fixes in the DOM Element Validation category.

Method/Property Modification: 53.2% of these fixes involve changing the DOM method or property being called/assigned; the rest involve either the removal of the method call or the property assignment (e.g., remove a `setAttribute` call that changes the class to which an element belongs), or the inclusion of such a call or assignment (e.g., add a call to `blur()` to unfocus a particular DOM element). Of the fixes where the DOM method/property was changed, around 44% involve changing the event handler to which a function is being assigned (e.g., instead of assigning a particular method to `onsubmit`, it is assigned to `onclick` instead).

Summary of Findings. Our study shows that the most prominent fix categories are Parameter Modification and DOM Element Validation. Our analysis also shows the prevalence of string value modifications and `null`/`undefined` checks when applying fixes. In addition, most parameter modifications are for values eventually used in DOM methods that *retrieve* elements from the DOM, particularly the `$()`, `jQuery()` and `getElementById()` methods. These results motivate our fault model choice in Section 4.4 as well as our choice of *possible sickness classes* in

Section 4.5.2.

4.4 Fault Model

In this work, we focus on DOM API methods that retrieve an element from the DOM using CSS selectors, IDs, tag names, or class names, as we found that these were the common sources of mistakes made by programmers (Section 4.3). These DOM API methods include `getElementById()`, `getElementsBy-TagName()`, `getElementsByClassName()`, `querySelector()`, and `querySelectorAll()`. We also support DOM API wrapper methods made available by commonly used JavaScript libraries including those in jQuery (e.g., `$()` and `jQuery()`); Prototype (e.g., `$$()` and `$()`); and tinyMCE (e.g., `get()`), among others. For simplicity, we will refer to all these DOM API methods as the *direct DOM access*, which is in line with terminology we used in Chapter 3.

We further focus on DOM-related faults that lead to *code-terminating* failures, which means the DOM API method returns `null`, `undefined`, or an empty set of elements, eventually leading to a null or an undefined exception (thereby terminating JavaScript execution). However, our design can also be extended to apply to DOM-related faults that lead to *output-related* failures, i.e., those that lead to incorrect output manifested on the DOM. Such faults would require the programmer to manually specify the direct DOM access. In contrast, with code-terminating DOM-related faults, the direct DOM access can be determined automatically using the AUTOFLOX tool proposed in Chapter 3. Thus we focus on this category of faults in this work.

The running example introduced in Section 4.2 is an example of a fault that is encompassed by the fault model described above. Here, the direct DOM access is the call to the `$()` method in line 11, which returns an empty set of elements. It is code-terminating because the fault leads to an undefined exception in line 12.

4.5 Approach

In this section, we describe our approach for assisting web developers in repairing DOM-related faults satisfying the fault model described in the previous section. Figure 4.3 shows a block diagram of our design, which consists of three main

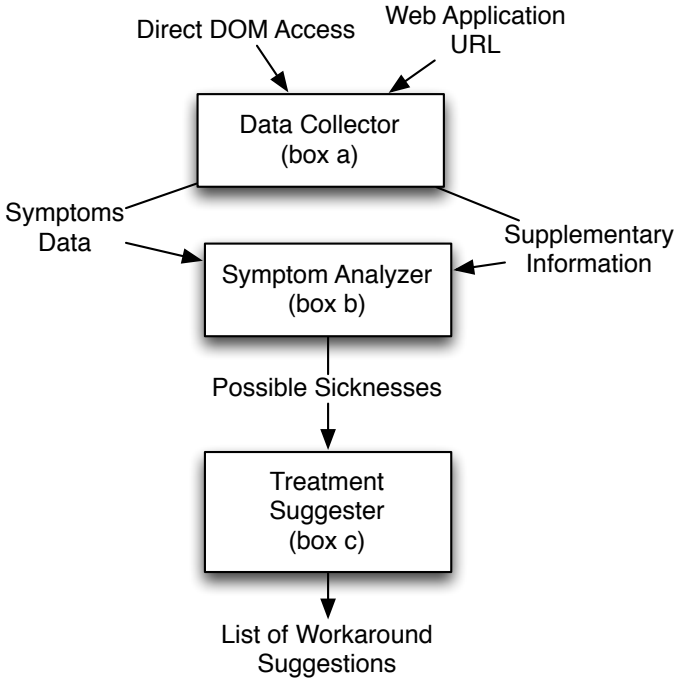


Figure 4.3: High-level block diagram of our design.

components: (1) the *data collector*; (2) the *symptom analyzer*; and (3) the *treatment suggester*. These components are described in Sections 4.5.1–4.5.3.

Our approach assumes the parameter (or the array index) of the direct DOM access is incorrect. This is inspired by the results presented in Section 4.3, which demonstrated the prevalence of Parameter Modification fixes. As such, our approach attempts to find *valid replacements* for the original parameter or array index, where a valid replacement is a parameter that matches at least one element in the DOM. Once the valid replacements are found, our approach analyzes the code context to determine what *actionable message* to suggest as a potential repair to the programmer.

4.5.1 Data Collector

The main purpose of the data collector module is to gather dynamic data that may reveal the *symptoms* present in the web application. In general, symptoms are defined as any indications of abnormalities in the intended behaviour of the program with regard to DOM accesses. We consider the following as symptoms in our design based on our fault model:

- **Symptom 1**: The direct DOM access is returning `null`, `undefined`, or an empty set of elements. This leads to a "null" or "undefined" exception eventually.
- **Symptom 2**: The index used to access an element in the list of elements returned by the direct DOM access is out of bounds. This is only applicable to DOM methods that retrieve a list of elements (e.g., `getElementsBy-TagName()`, `$()`, etc.). This eventually leads to an "undefined" exception.

The data collector collects the direct DOM access' line number, and the name of the function containing it. This data is provided by the user (manually) or gathered automatically using a tool such as AUTOFLOX [134]. The data collector module also collects the following *supplementary information*, which can help infer the context under which a particular symptom is appearing:

- The dynamic execution trace of the JavaScript program, with each trace item containing the line number of the executed line, the name of the function containing the line, and the names and values of all in-scope variables at that line. It also includes the lines in the body of a loop, and a list of `for` loop iterator variables (if any). The data describing which lines are part of a loop are used by the treatment suggester to infer code context, to determine what actionable repair message to provide to the programmer; more details are in Section 4.5.3.
- The state of the DOM when the direct DOM access line is executed. For instance, in the running example, the DOM state in Figure 4.2 is retrieved; The DOM state will be used by the symptom analyzer to determine possible replacements for the direct DOM access parameter (if any); in particular, if the direct DOM access is returning `null` or `undefined` (i.e., Symptom 1),

this means that the parameter to the direct DOM access does not correspond to any element in the current DOM state, so our technique can look at the current DOM state to see if there are any reasonable replacements that *do* match an element (or a set of elements) in the DOM.

4.5.2 Analyzing Symptoms

The *symptom analyzer* (Figure 4.3, box b) uses the data gathered by the data collector to come up with a list of *possible sicknesses* that the web application may have. Each possible sickness belongs to one of the following classes:

- **String**: A [variable | expression | string literal] has a string value of `X`, but it should probably have string value `Y`. This sickness triggers Symptom 1.
- **Index**: An array index has a numerical value of `X`, but it should fall within the allowed range `[Y--Z]`. This sickness triggers Symptom 2.
- **Null/Undefined**: A line of code `X` accessing a property/method of the DOM element returned by the direct DOM access should not execute if the DOM element is [`null` | `undefined`]. This sickness can trigger both Symptoms 1 and 2.

These classes are based on the results of our study of common bug report fixes. In particular, the "String" and "Null/Undefined" classes account for Parameter Modification and DOM Element Validation fixes, respectively. The "Index" class is included because in some cases, an undefined exception occurs not because of retrieving the incorrect element, but because of using an out-of-bounds index on the returned array of DOM elements.

The symptom analyzer takes different actions depending on the symptom in Section 4.5.1 as follows:

1. **String Replacement**: Assume that the program suffers from Symptom 1. This implies that the string parameter being passed to the direct DOM access does *not* match any element in the DOM – i.e., the program may be suffering from the "String" sickness class, as described above. Our design will look for potential replacements for these parameters, where the replacements are

determined based on the current DOM state. Each potential replacement represents a possible sickness belonging to the "String" class.

2. **Index Replacement**: Assume that the program suffers from Symptom 2. This implies that the program may be suffering from the "Index" sickness class, as described above. This step is only taken if the direct DOM access corresponds to a method that returns a *set* of DOM elements. Our approach will determine the allowed range of indices, representing a possible sickness belonging to the "Index" class.
3. **Null/Undefined Checks**: By default, our design additionally assumes a possible sickness belonging to the "Null/Undefined" class.

Each of the above cases will be described in detail. Because CSS selectors provide a unified way of retrieving elements from the DOM, we will only describe how the possible sicknesses are determined for the case where the parameter to the direct DOM access is a CSS selector (as in the case of the running example).

Case 1: String Replacement. The main assumption here is that the string parameter passed to the direct DOM access is incorrect; we call this parameter the ***erroneous selector***. Hence, the goal is to (1) look for potential replacement parameters that *match* an element (or a set of elements) in the current DOM state (i.e., are *valid replacements*), and (2) suggest only the most viable replacements so as to not overwhelm the programmer; therefore our approach assumes that the replacement will be relatively close to the original, erroneous selector (i.e., only one component of the original selector is assumed incorrect by any given replacement). Algorithm 1 shows the pseudocode for this step. The sub-steps are described below in more detail.

Dividing Components: The first step is to divide the erroneous selector into its constituent components, represented by **C** (line 1). In essence, **C** is an ordered set, where each element c_i corresponds to a selector component ($c_i.comp$); its matching component type ($c_i.type$; see Table 4.1); and its level in the selector, where each level is separated by a white space or a $>$ character ($c_i.level$). The erroneous selector itself is retrieved from the direct DOM access (*dda*) which is input to the algorithm. For example, consider the erroneous selector in the running example: "div#view-display-id-catalog_view $>$ p.pages span". This selector contains the

Algorithm 1: Parameter Replacement

```
Input: trace: The dynamic execution trace
Input: dda: The direct DOM access
Input: dom: The current DOM state
Output: listOfpossibleSicknesses: A list of possible sicknesses
1  C ← {c_1, c_2, ..., c_N};
2  GSS ← {(s_1, l_1), (s_2, l_2), ..., (s_K, l_K)};
3  foreach c_i ∈ C do
4      LSS_i ← match(c_i, GSS);
5  end
6  VS ← ∅;
7  foreach c_i ∈ C do
8      PVE ← {dom.root};
9      for j ← 0 to c_i.level do
10         nextElems ← ∅;
11         foreach e ∈ PVE do
12             all ← e.getElementsByTagName("*");
13             foreach f ∈ all do
14                 if f matches level j of erroneous selector then
15                     nextElems.add(f);
16                 end
17             end
18         end
19         PVE ← nextElems;
20     end
21     foreach e ∈ PVE do
22         newElems ← ∅;
23         if level after c_i.level is the "descendant" then
24             newElems ← getAllDescendants(e);
25         end
26         else if level after c_i.level is the "child" then
27             newElems ← getAllChildren(e);
28         end
29         else if level after c_i.level is the "next sibling" then
30             newElems ← getNextSibling(e);
31         end
32         foreach f ∈ newElems do
33             if f has c_i.type then
34                 newSelector ← dda.erroneousSelector.replace(c_i.comp, c_i.type of f);
35                 VS.add(newSelector);
36             end
37         end
38     end
39     foreach selector ∈ VS do
40         if e ← selector(dom) ∉ dom then
41             VS.remove(selector);
42         end
43     end
44 end
45 PR ← replacementsFinder(VS, LSS_1, LSS_2, ..., LSS_N);
46 foreach rep ∈ PR do
47     possibleSickness ← craftPossibleSickness(rep);
48     listOfPossibleSicknesses.add(possibleSickness);
49 end
```

following components: (1) the tag name "`div`"; (2) the "has ID" identifier "`#`"; (3) the ID name "`view-display-id-catalog_view`"; (4) a ">" character indicating that the next component is a child of the previous; (5) the tag name "`p`"; (6) the "has class" identifier "`.`" (i.e., a dot character); (7) the class name "`pages`"; (8) whitespace indicating that the next component is a descendant of the previous one; and (9) the tag name "`span`".

Finding the Global String Set: The next step is to determine the *string set* corresponding to each component (lines 2-5). The string set refers to the list of locations, in the JavaScript code, of the origins of all the parts that make up a particular string value. For instance, consider the erroneous selector in the running example, whose final string value is "div#view-display-id-catalog_view > p.pages span". This entire string is made up of a concatenation of the following strings: (1) "`div#view-display-id-`" in Figure 4.1, line 8; (2) "`catalog_view`" in line 2; (3) " > " in line 9; and (4) "`p.pages span`" in line 3.

The algorithm first determines the *global string set*, which refers to the string set of the entire erroneous selector; in Algorithm 1, this is represented by **GSS** (line 2). The global string set is found by recursively extracting the dynamic backward slice of each concatenated string value that makes up the erroneous selector (using the dynamic execution trace) until all the string literals that make up the erroneous selector have been included in the string set. Note that the slice extraction process is a dynamic one, and is hence precise. However, it may be unable to resolve the origin of every variable in the code e.g., because a variable gets its value from an external XML file. Unresolved portions of the erroneous selector are left as "gaps" in the string set.

The **GSS** consists of an ordered set of tuples of the form (s_i, l_i), where s_i is a string value and l_i is the location in the JavaScript code where that value originated (i.e., line number and enclosing function). Each tuple represents an element in the string set. In the running example, given the string set of the erroneous selector just described above, the ordered set of tuples will be as follows: {("`div#view-display-id-`", 8), ("`catalog_view`", 2), (">", 9), ("`p.pages span`", 3)}.[23] Note that a gap in the string set is likewise represented

[23]For simplicity, we omit the enclosing functions.

as a tuple; the string value s_i is retained, but the location l_i is left undefined, and a special variable is used to store the *earliest* expression from which the unresolved string value originated.

Finding the Local String Sets: Once the global string set is found, the *local string set* of each component – represented by $\mathbf{LSS}_i$ – is inferred (lines 3-5). In essence, this procedure matches each erroneous selector component c_i with the corresponding elements in the global string set (line 4). For example, consider the id name component "`view-display-id-catalog_view`" in the running example. If *startIndex* and *endIndex* refer to the index range of the characters from the global string set element that belong to the local string set, then the string set of this component is {((“`div#view-display-id-`”, 8), *startIndex*: 4, *endIndex*: 19), ((“`catalog_view`”, 2), *startIndex*: 0, *endIndex*: 11)}.

Finding Valid Selectors: Lines 6-44 of Algorithm 1 looks for *valid selectors* (**VS**) in the current DOM state. This portion of the algorithm iterates through each component c_i of the erroneous selector and assumes that c_i is incorrect; it then traverses the current DOM state's tree to see if it can find new CSS selectors (i.e., those in which the component assumed to be erroneous is replaced by a different value) that match an element in the current DOM state. This procedure is carried out for each component of the erroneous selector; hence, by the end of this procedure, each component will have a corresponding set of CSS selectors (may be empty).

Precisely, to find the valid selectors, the algorithm first looks for possibly valid elements, represented by **PVE** (lines 8-20). These are the elements that match the original selector *up to and including* the the selector level $c_i.level$, neglecting the component being assumed erroneous. For instance, suppose in the running example, the tag component "`p`" of the erroneous selector is assumed as incorrect by our design. This component is found in level 2 of the erroneous selector. Hence, our design traverses the DOM to look for elements that match the selector up to level 2 neglecting "p" – i.e., elements that match "`div#view-display-id-catalog_view > .pages`".

Once **PVE** is found, the algorithm (lines 21-38) checks if the element does indeed contain a corresponding replacement for the component that was assumed to be incorrect (e.g., if an ID is being replaced, the element must have an ID) (line

32-37). In our example, "p" – which is a tag component – was assumed incorrect so the verification will pass for all elements in **PVE** because every element has a tag name. It also checks if the element contains any descendants, children, or siblings, depending on the structure of the erroneous selector (lines 22-31). Again, in the running example, the next level (level 3) of the erroneous selector must be the "descendant" of the first two levels, because of the whitespace between the level 2 components and the level 3 components; hence, the check will pass for an element if it contains any descendants. If both checks pass, the corresponding component is used to create a new selector; each new selector is stored in **VS**. Finally, for each new selector, a final verification step is carried out to ensure that the new selector is indeed valid in *dom* (lines 39-43).

In summary, for the running example, our design looks for matching selectors of the form "div#view-display-id-catalog_view > <**NEW-TAG**>.pages span". Similarly, if the ID component "view-display-id-catalog_view" were assumed incorrect, the algorithm looks for matching selectors of the form "div#<**NEW-ID**> > p.pages span". In the latter case, two matching valid selectors are found: "div#view-id-catalog_view > p.pages span" and "div#view-display-id-catalog_page > p.pages span"

Inferring Possible Replacements: To determine the possible sickness, our design determines if any element of the local string set of each component ($\mathbf{LSS}_1$, $\mathbf{LSS}_2$, ..., $\mathbf{LSS}_N$) can be replaced to match one of the valid selectors in **VS**. This is accomplished by the `replacementsFinder()` function (line 45). The basic idea is as follows: for each component string set element, assume that this element is incorrect, then determine if any of the valid selectors can provide a replacement string value for that element. We accomplish this matching with the valid selectors through the use of a string constraint solver (see Section 4.5.4).

Let us go back to the running example. Suppose the design is currently considering the "`view-display-id-catalog_view`" component, whose local string set was found earlier. Also, as mentioned, two valid replacement selectors were found for this component. Our design goes through each element in the local string set to look for possible replacements. First, it assumes that the first string set element – namely (("`div#view-display-id-`", 8), *startIndex*: 4, *endIndex*: 19) – is incorrect; hence, it checks if any of the valid selectors is of the form

"`div#<`**`NEW-STRING`**`>catalog_view > p.pages span`" – i.e., the erroneous selector with the string "view-display-id-" replaced. In this case, the constraint solver will find one matching selector: "`div#view-id-catalog_view > p.pages span`". Next, our design will move on to the second local string set element and perform the same procedure to find the following matching selector: "`div#view-display-id-catalog_page > p.pages span`".

Case 2: Index Replacement. In this step, our design assumes that the index used to access the list of elements returned by the direct DOM access is incorrect. To check whether this assumption holds, our approach records the size of the array returned by the direct DOM access; this is determined based on the value of an instrumented variable added to the JavaScript code to keep track of the size. The erroneous array index used, if any, is also recorded.

The erroneous array index is compared with the size to see if it falls within the allowed range of indices (i.e., [0–size-1]). If not, our approach will package the following as a possible sickness (belonging to the "Index" sickness class), to be added to the list of possible sicknesses: "*An array index has a numerical value of* `X` *that does not fall within the range* `[0-Z]`"; here, `X` is the erroneous array index, and `Z` is `size-1`.

Case 3: Null/Undefined Checks. By default, our design packages a possible sickness belonging to the "Null/Undefined" class to account for cases where the repair is a DOM Element Validation. In essence, this means the line of code must be wrapped in an `if` statement that checks whether the DOM element being used is `null` or `undefined`. If code termination was caused by a null exception (or undefined exception), the following is packaged and added to the list of possible sicknesses: "*A line of code* `X` *accessing a property/method of the DOM element returned by the direct DOM access should (probably) not execute if the DOM element is* `null` *(or* `undefined`*)*".

4.5.3 Suggesting Treatments

Once the symptom analyzer has found a list of possible sicknesses, each of these possible sicknesses is examined by the *treatment suggester* (Figure 4.3, box c). The goal of the treatment suggester is as follows: given a possible sickness, cre-

ate an *actionable* repair message that would prompt the programmer to modify the JavaScript code in such a way that the symptom represented by the possible sickness would disappear. In order to accomplish this, the code context, as inferred from the supplementary information retrieved by the data collector, is analyzed. This module handles each possible sickness class separately, as described below.

String class. Possible sicknesses belonging to the "String" class require the string value `X` at some line in the JavaScript code to be replaced by another string value `Y`. If applied correctly, this would cause the parameter at the direct DOM access to be valid, so the direct DOM access would no longer output `null`, `undefined`, nor an empty set of elements (i.e., Symptom 1 disappears). As we discovered in Section 4.3.2, most Parameter Replacement fixes are direct string literal replacements; hence, at first, it may seem straightforward to simply output a message prompting the programmer to directly perform the replacement. However, there are several cases that may make this suggestion invalid, for example:

1. The string value is not represented by a string literal, but rather, by a variable or an expression. Recall that when calculating the string set, gaps may exist in this string set due to string values originating from sources external to the JavaScript code, or due to values not originating from string literals. Hence, a simple "replace" message would be inappropriate to give out as a suggested repair;
2. The string value may be in a line that is part of a loop. Here, a "replace" message may also be inappropriate, since the replacement would affect other (possibly unrelated) iterations of the loop, thereby possibly causing unwanted side effects.

To account for these cases, before outputting a repair message, our approach first examines (a) the string set element type (i.e., is it a variable, expression, or string literal?), and (b) the location (i.e., inside a loop?). Through this analysis, the treatment suggester can provide a suggested repair message. The algorithm essentially performs a "background check" on the code suffering from the bug to determine what message to output. For example, if our design finds that a string set element is in a line inside a loop, and this line executed multiple times a message

Table 4.3: List of output messages.

Type	Description
REPLACE	*Replace the string literal X with Y in line L*
REPLACE AT ITERATION	*Wrap line L in an if statement so that the string literal X instead has value Y at iteration I*
OFF BY ONE AT BEGINNING	*Change the lower bound of the for loop containing line L so that the original first iteration does not execute*
OFF BY ONE AT END	*Change the upper bound of the for loop containing line L so that the original last iteration does not execute*
MODIFY UPPER BOUND	*Change the upper bound of the for loop containing line L so that the loop only iterates up to iteration I (inclusive)*
EXCLUDE ITERATION	*Skip iteration I of the for loop containing line L by adding a 'continue'*
ENSURE	*Ensure that the string value at line L has value Y instead of X*. This is a fall back message which is given if a precise modification to the code cannot be inferred by the suggester. Thus, our suggester is conservative in that it only provides a particular suggestion if it is certain that the suggestion will lead to a correct replacement.

such as "replace at iteration" or "off by one" – will be given. The complete list of messages is presented in Table 4.3.

When the running example is subjected to the treatment suggester algorithm, the possible sicknesses found by the symptom analyzer will lead to two REPLACE messages being suggested, one of which is the fix described in Section 4.2: *Replace the string literal "div#view-display-id-" with "div#view-id-" in line 8*. The other message is a spurious suggestion: *Replace the string literal "catalog_view" with "catalog_page" in line 2*.

Index and Null/Undefined class. For the "Index" class, the suggestion is always as follows: *Modify the array index in line L to some number within the range [0–Z]*. For the "Null/Undefined" class, the suggestion depends on whether the exception was a null exception or an undefined exception. If the exception is a null exception, the following message is given: *Wrap line L in an if statement that checks if expression E is null*. Here the expression E is inferred directly from the error message, which specifies which expression caused the null exception. An analogous message is given if the exception is "undefined".

4.5.4 Implementation: Vejovis

We implemented our approach in a tool called VEJOVIS, which is freely available for download [126].

The data collector is implemented by instrumenting the JavaScript code using RHINO and running the instrumented application using CRAWLJAX [110]. For the symptom analyzer, we use the string constraint solver HAMPI [88] for `replacementFinder()` (see Algorithm 1, line 45), which looks for viable replacements among the valid parameters found. The symptom analyzer treats the valid parameters found as defining the context-free grammar (CFG).

In keeping with our goal of providing as few suggestions as possible, VEJOVIS allows the users to modify a parameter called the *edit distance bound*. The edit distance bound is a cutoff value that limits the suggested replacement strings to only those whose edit distance with the original string is within the specified value. We use Berico's [26] implementation of the Damerau-Levenshtein algorithm to calculate the edit distance.

4.6 Evaluation

To evaluate the efficacy of VEJOVIS in suggesting repairs for DOM-related faults, we answer the following research questions.

RQ3 (Accuracy): What is the accuracy of VEJOVIS in suggesting a correct repair?

RQ4 (Performance): How quickly can VEJOVIS determine possible replacements? What is its performance overhead?

We perform a case study in which we run VEJOVIS on real-world bugs from eleven web applications. To determine accuracy (RQ3), we measure both the precision and recall of our tool. To calculate the performance (RQ4), we compare the runtimes of VEJOVIS with and without instrumentation.

4.6.1 Subject Systems

The bugs to which we subject VEJOVIS come from eleven open-source web applications, studied also in Section 4.3; hence, these bugs represent *real-world* DOM-related faults that occurred in the subject systems. We choose two bug reports

randomly from the set of bugs that satisfy our fault model, for each of the eleven web applications, for a total of 22 bugs. Note that TaskFreak is not included among the applications studied, as we only found 6 JavaScript bugs from that application, none of which fit the fault model described in Section 4.4. Descriptions of the bugs and their corresponding fixes (henceforth called the *actual fixes*) can be found online [126]. It took programmers an average of 47 days to repair these bugs after being triaged, indicating that they are not trivial to fix. We had to restrict the number of bugs to two per application as the process of deploying the applications and replicating the bugs was a time and effort intensive one. In particular, most of the bugs were present in older versions of the web applications. This presented difficulties in installation and deployment as some of these earlier versions are no longer supported.

4.6.2 Methodology

Accuracy. We measure accuracy based on both *recall* and *precision*. In the context of this experiment, recall refers to whether our tool was able to suggest the "correct fix" – that is, whether one of the suggestions provided by VEJOVIS matches the *actual* developer fix described in the corresponding bug report. Hence, in this case, recall is a binary metric (i.e., either 0% or 100%), because the actual fix either appears in the list of suggestions, or it does not. Note that in some cases, the suggested fix is not an exact match of the applied fix, but is semantically equivalent to it, and is considered a match. Precision refers to the number of suggestions that match the actual fix divided by the number of suggestions provided by VEJOVIS. Again, since there is only one matching fix, precision will be either 0 (if the correct fix is not suggested), or $\frac{1}{\#Suggestions}$. This metric is a measure of how well VEJOVIS prunes out irrelevant/incorrect fixes.

To measure the above metrics, we first replicated the bug, and ran VEJOVIS with the URL of the buggy application and the direct DOM access information (i.e., line number and enclosing function) as input; for the libraries, the bugs are replicated by using the test applications described in the bug reports. VEJOVIS outputs a list of suggestions for the bug, which we compare with the actual developer fix to see if there is a match. Based on this comparison, we calculated the recall and

precision for that particular attempt. In our experimental setup, the suggestions are sorted according to the edit distance of the replacement string with respect to the original string, where replacements with smaller edit distances are ranked higher. Suggestions for "null" or "undefined" checks are placed between suggestions with edit distance 5 and those with edit distance 6. In the event of a tie, we assume the worst case i.e., the correct fix is ranked lowest among the suggestions with the same edit distance.

To test our assumption that the replacement parameter closely resembles the original parameter, we control the edit distance bound (defined in Section 4.5.4) for VEJOVIS. We first run our experiments with an edit distance bound of *infinity*, which, means the suggestions given do not have to be within any particular edit distance relative to the original string being replaced (i.e., no edit distance bound assigned). Then, to observe how this bound affects VEJOVIS' accuracy, we re-run our experiments with a smaller edit distance bound of 5. We choose the value 5 based on a pilot study.

Performance. For each bug, we measure the performance overhead introduced by VEJOVIS' instrumentation by comparing the corresponding web application with and without instrumentation. This evaluates the performance of the data collection phase of VEJOVIS. We also measure the time it takes for VEJOVIS to generate the repair suggestions. This evaluates the performance of the symptom analysis and treatment suggestion phases of VEJOVIS.

4.6.3 Results

Accuracy. Table 4.4 shows the results of our experiments when the edit distance bound is set to infinity i.e., no bound is assigned (numbers outside parentheses). The "Accurate" column of Table 4.4 indicates, for each bug, whether the actual fix appears among the list of repairs suggested by VEJOVIS (i.e., the recall was 100%). As the results show, assigning no bound causes VEJOVIS *to accurately suggest repairs for 20 out of the 22 bugs, for an overall recall of 91%*. The only unsuccessful cases are the second bugs in Roundcube, where, the correct replacement selector is ":focus:not(body)", and jQuery, where the correct replacement selector is "[id="nid"]"; VEJOVIS does not currently support these CSS selector syntax .

Table 4.4: Accuracy results, with edit distance bound set to infinity i.e., no bound assigned. BR1 refers to the first bug report, and BR2, the second bug report (from each application). Data in parentheses are the results for when the edit distance bound is set to 5.

Application	Accurate?		Precision	
	BR1	BR2	BR1	BR2
Drupal	✓ (✗)	✓ (✗)	3% (0%)	25% (0%)
Ember.js	✓ (✓)	✓ (✓)	50% (50%)	33% (100%)
Joomla	✓ (✓)	✓ (✓)	1% (25%)	1% (100%)
jQuery	✓ (✓)	✗ (✗)	1% (25%)	0% (0%)
Moodle	✓ (✓)	✓ (✓)	3% (33%)	3% (100%)
MooTools	✓ (✗)	✓ (✓)	50% (0%)	50% (50%)
Prototype	✓ (✓)	✓ (✓)	17% (50%)	50% (50%)
Roundcube	✓ (✓)	✗ (✗)	1% (25%)	0% (0%)
TYPO3	✓ (✓)	✓ (✓)	1% (100%)	100% (100%)
WikiMedia	✓ (✗)	✓ (✓)	4% (0%)	1% (100%)
WordPress	✓ (✓)	✓ (✓)	3% (7%)	1% (50%)

Note that in three of the successful cases, the repair suggestion does not exactly match the actual fix, but rather is equivalent to (or close to) the actual fix.

First, in the second TYPO3 bug, the actual fix documented in the bug report is to add a check to ensure that the NodeList `valueObj`, which is populated by a direct DOM access call to *getElementsByName()*, has a length greater than 0, thereby preventing the use of `valueObj[0].value` from throwing an undefined exception. VEJOVIS, in contrast, suggested an alternate but equivalent fix with no side effects, namely adding a check to see if the expression `valueObj[0]` is undefined before trying to access one of its properties.

Second, in both the first Moodle bug and the second Prototype bug, VEJOVIS provides the fallback "ENSURE" suggestion. In the Moodle bug, VEJOVIS suggests the following: *Ensure the value of variable* `itemid` *is "id_itemname" instead of "itemname"*; this is because the string literal "itemname" originated from an anonymous function, which our implementation currently does not support, leaving a gap in the string set. Nonetheless, this simplifies the debugging task for the programmer, as it points her directly to the problem – i.e., the string "itemname", located somewhere in the JavaScript code, needs to be changed to "id_itemname". Similarly, in the Prototype bug, VEJOVIS suggests the following: *Ensure the expression* `id.replace(/[\.:]/g, ''\\$0'')` *has value "outer.div" instead of "outer\\$0div"*. Again, while VEJOVIS is not able to provide the exact fix, it points the programmer to the relevance of the *replace()* method

Table 4.5: Rank of the correct fix when suggestions are sorted by edit distance. The denominator refers to the total number of suggestions. Top ranked suggestions are in bold.

Application	Rank	
	BR1	BR2
Drupal	31 / 40	**01 / 04**
Ember.js	**01 / 02**	**01 / 03**
Joomla	**01 / 88**	**01 / 88**
jQuery	02 / 108	–
Moodle	02 / 37	**01 / 37**
Moodle	02 / 02	**01 / 02**
Prototype	**01 / 06**	**01 / 02**
Roundcube	04 / 79	–
TYPO3	**01 / 187**	**01 / 01**
WikiMedia	06 / 24	**01 / 71**
WordPress	13 / 30	**01 / 170**

in the fix. These results show that even in cases when VEJOVIS is unable to fully resolve the origins of the erroneous selector's string values, it still provides meaningful suggestions that facilitate debugging and are hence useful to the programmer.

Among the successful cases, the average precision ("Precision" column of Table 4.4) is approximately 2%; on average, this translates to VEJOVIS providing 49 suggestions for each bug, with a maximum of 187 total suggestions for the first TYPO3 bug. The high number of suggestions motivated us to implement the simple ranking scheme based on edit distance, as described in Section 4.6.2.

Table 4.5 shows, for each bug, the rank of the actual fix among the list of suggestions provided by VEJOVIS; only the cases where the actual fix appears among the list of suggestions are considered. As shown in the table, the correct fix appears as the first suggestion in 13 out of the 20 cases, and as the second suggestion in three more cases. In fact, for the WordPress bug, the correct fix is tied for first place among 13 bugs; we listed its rank as 13 because we consider the worst case. Hence, despite providing a large number of suggestions on average when the edit distance bound is set to infinity, *our simple ranking scheme based on edit distance ranked most of the actual fixes near the top of the list of suggestions*.

As mentioned, the above results were obtained with an edit distance bound of infinity. To quantify the effects of using a finite bound, we re-ran the above accuracy experiment with an edit distance bound of 5. The results are shown in parentheses in Table 4.4. As the results show, assigning a bound of 5 decreases

Table 4.6: Performance results.

Application	Crawl Time w/o Instrumentation (s)	Crawl Time with Instrumentation (s)	Average Treatment Time (s)
Drupal	28.5	49.6	1.8
Ember.js	10.8	16.3	0.8
Joomla	57.0	85.0	6.1
jQuery	12.0	19.0	4.1
Moodle	46.9	59.6	7.4
MooTools	13.3	19.1	0.9
Prototype	12.0	17.4	8.6
Roundcube	25.1	34.7	3.4
TYPO3	39.9	72.8	6.5
WikiMedia	15.9	24.8	5.4
WordPress	20.2	33.8	7.0
Average	-	39.3	4.7

the number of successful cases from 20 to 16, where four additional cases became unsuccessful because the actual fix required replacing the original parameter with another parameter that is more than an edit distance of 5 away. However, the precision jumps dramatically to 36% with this bound, which translates to around 3 suggestions given for each bug on average. Hence, assigning a finite edit distance bound can significantly decrease the number of suggestions, which makes the list of suggestions more manageable; however, this comes at the cost of lower recall of 73% (as compared to 91%).

Performance. There are two sources of performance overhead in VEJOVIS: (1) instrumentation overhead, and (2) symptom analysis and treatment suggestion overhead. Table 4.6 shows the results. The time taken with and without instrumentation during the data collection phase of VEJOVIS are shown in the second and third columns of the table. The time varies per application, ranging from 16.3 to 85.0 seconds, for an average of 39.3 seconds. The time in the symptom analysis and treatment suggestion phases is shown in the last column. The average time for these phases is 4.7 seconds, ranging from 0.8 to 7 seconds. Thus, on average, VEJOVIS takes less than one minute (44 seconds) to find the correct fix, with a worst-case time of 91.1 seconds for Joomla.

4.7 Discussion

Extensions. First, VEJOVIS suggests treatments belonging to the Parameter Modification and DOM Element Validation categories as mentioned in our empirical study of common fixes in Section 4.3. While these together constitute more than half of the fix types we found in the study, another common fix category is Method/Property Modification, in which a DOM API method or property is added, removed or replaced with another method/property. We do not incorporate this fix category in our design; however VEJOVIS can be extended to account for this category. For instance, it is possible, in some cases, to reduce the problem of replacing DOM methods to replacing CSS selectors. As an example, replacing `getElementById(``str'')` with `getElementsByClassName(``str'')` can be thought of as replacing the CSS selector "#str" with ".str".

Second, the results of our evaluation show that while VEJOVIS accurately predicts the actual fix in almost all of the bug reports analyzed, the number of suggestions provided can be large, thereby lowering its precision. In our evaluation, we showed that ranking the fixes based on edit distance makes the actual fix rank high in many cases. We are currently exploring more intelligent ways to perform this ranking; for example, based on the textual patterns of the strings.

Threats to Validity. An *external validity threat* is that the bugs we analyzed come from only 11 web applications. However, the systems considered were developed for different purposes and hence, represent a reasonable variety. Further, the corresponding bug reports have been fixed by the developers, and are therefore representative of the issues encountered in practice.

An *internal threat* to validity is that we have assumed the fixes described in the bug reports are correct as many experienced developers are typically involved with providing patches for these bugs. Nonetheless, to mitigate this threat, we carefully analyzed the patches provided in the bug reports and have tested the fixes on our own platforms to see if they are sound.

Additionally, the bugs we considered in our evaluation were taken from the bug report study in Section 4.3, which may be a potential source of bias. This threat can be mitigated by considering other applications, which we plan to do in the future. As for repeatability, VEJOVIS is available [126], and the experimental subjects are

open source, making our case study fully repeatable.

4.8 Related Work

Program Repair. Program repair refers to the act of fixing bugs through automated techniques. Perhaps the best-known application of program repair is to data structures. Demsky et al. [41] use formal specifications to suggest fixes for data structures. Elkareblieh et al. [47] use programmer-specified assertions for data structure repair. However, these techniques are limited to repairing data-structures, and do not fix the underlying defect that produced the erroneous structure. While the DOM can be considered a data-structure, VEJOVIS goes beyond the DOM and actually can suggest ways to modify the JavaScript code based on the defective DOM access.

Generating fixes at the source code level has gained attention recently [13, 172, 174]. Weimer et al. [174] propose the use of genetic algorithms for repairing C programs. The main idea is to copy other parts of the program to the faulty portion of the program and check if the modified program passes the existing test cases. However, it is not clear how this technique could be applied to web applications, where the code base includes different languages such as JavaScript and HTML/DOM.

In recent work, Zhang et al. [184] propose FlowFixer, a technique to repair broken workflows in Java-based GUI applications. Similar to VEJOVIS, FlowFixer attempts to find repairs for errors that arise due to a mismatch between the code and the GUI state. However, there are two main differences between VEJOVIS and FlowFixer. First, FlowFixer is concerned with correcting the *sequence of user actions* applied to the GUI; in contrast, VEJOVIS is concerned with correcting the *code* that drives the functionality of the application. Second, FlowFixer uses random testing to find replacements; VEJOVIS is different in that it performs a systematic traversal of the DOM to find valid replacement selectors.

Web Application Repair. There has been limited work on exploring fault repair for web applications. Carzaniga et al. [33] propose automatic workarounds for web applications that experience errors in using APIs by replacing the buggy API call sequence with a functionally equivalent, but correct sequence. Samimi et al. [148] have proposed a technique for PHP code to fix errors that result in the generation

of webpages with malformed HTML; similar work has been done by Zheng et al. [187]. Neither of these techniques consider JavaScript code, nor do they apply to DOM-related JavaScript faults. In recent work, Jensen et al. [80] and Meawad et al. [107] introduce techniques to transform unsafe *eval* calls in JavaScript code to functionally equivalent, but safe constructs. This is more of a prevention than repair technique. However, they do not consider JavaScript errors, and in particular errors that lead to DOM-related faults.

4.9 Conclusion

JavaScript interacts extensively with the DOM to create responsive applications; yet, such interactions are prone to faults. In this chapter, we attempt to understand common fixes applied by programmers to DOM-related faults. Based on these findings, we propose an automated technique for providing repair suggestions for DOM-related JavaScript faults. Our technique, implemented in a tool called VEJOVIS, is evaluated through a case study of 22 bugs based on real-life bug reports. We find that VEJOVIS can accurately predict the repair in 20 out of the 22 bugs, and that the correct fix appears first in the list of fix suggestions for 13 of the 20 bugs.

Chapter 5

Detecting Inconsistencies in JavaScript MVC Applications

This chapter discusses our approach for automatically detecting identifier and type inconsistencies in applications written using JavaScript Model-View-Controller frameworks.[24] In essence, this is our first attempt to answer RQ1C from Chapter 1.1 (*Can we create a general technique that detects JavaScript faults in the presence of JavaScript frameworks? If so, how?*). Our approach has been implemented in a tool called AUREBESH[25], which we describe in the sections that follow.

5.1 Introduction

DOM-related faults – which constitute the majority of JavaScript faults as we have established in Chapter 2 – often result from a developer's incomplete or erroneous understanding of the relationship between the JavaScript code and the DOM. Partly in response to this issue, JavaScript libraries known as *MVC frameworks* have recently been developed. MVC frameworks such as AngularJS [58], BackboneJS [16], and Ember.js [87] use the well-known Model-View-Controller (MVC) pattern to simplify JavaScript development in a way that abstracts out DOM method calls. This is accomplished by giving programmers the ability to define model objects, which are then directly embedded in the HTML code (typically via

[24] The main study in this chapter appeared at the International Conference on Software Engineering (ICSE 2015) [132].

[25] AUREBESH is named after a writing system used in the *Star Wars* film series.

a double curly brace notation) such that any changes in these objects' values will automatically be reflected in the DOM, and vice versa – a process known as "two-way data binding". The frameworks thus eliminate the need for web programmers to explicitly set up DOM interactions in JavaScript.

Unfortunately, despite the apparent advantages, MVC frameworks are still susceptible to consistency issues akin to DOM-JavaScript interactions [90]. In particular, these frameworks rely on the use of *identifiers* to represent model objects and controller methods; definitions and uses of these identifiers are expected to be consistent across associated models, views, and controllers. Moreover, due to JavaScript's loose typing, which is retained in these MVC frameworks, the programmer must ensure that the values assigned to model objects and returned by controller methods are consistent with their expected *types*, depending on how they are used. Since model objects and controller methods are primarily used to represent major functionalities of the web application, any inconsistencies between these identifiers and types can potentially lead to a significant loss in functionality; hence, avoiding these inconsistencies is crucial. In addition, these inconsistencies are often difficult to detect, because (1) multiple model-view-controller groupings exist in the application, and (2) no exceptions are thrown or warnings provided in the event of an inconsistency.

To tackle this problem, we introduce an approach to automatically detect inconsistencies between identifiers in web applications developed using JavaScript MVC frameworks. Our design conducts static analysis to separate the three main components (model, view, and controller) in these applications; find the identifiers defined or used in these components; infer the types associated with these identifiers; and compare the collected identifiers and type information to determine any inconsistencies. We implement our approach in a tool called AUREBESH, which finds inconsistencies in AngularJS [58] applications, the most popular [153] JavaScript MVC framework used in practice.

Since MVC frameworks for JavaScript are fairly new, few papers have explored their characteristics. For the most part, prior work in this area does not include observations on the properties of existing MVC frameworks, but rather, proposes new MVC frameworks fitted towards a specific goal [54, 72, 162]. Other papers analyse existing JavaScript MVC frameworks, with particular focus on their

maintainability [23, 86]. *To the best of our knowledge, our work in this chapter is the first to identify the consistency issues in JavaScript applications using MVC frameworks,*[26] *and the first to propose a design for automatically detecting inconsistencies in such applications.*

We list the following as our main contributions:

- We identify consistency issues pertinent to identifiers and types that are present in JavaScript MVC applications. These consistency issues point to potential problems within the application;
- We devise a general formal model for MVC applications. This model helps us reason about the way variables and functions are used and defined throughout the application, which, in turn, allows us to more clearly define what constitutes an inconsistency among them in the application;
- We introduce an automatic approach to detect identifier and type inconsistencies in MVC applications. This approach uses static analysis, and only requires the application's client-side source code;
- We implement our design in an open-source tool called AUREBESH, which works for AngularJS applications; and
- We perform a systematic fault injection experiment on AUREBESH to test its accuracy, and we subject AUREBESH to real-world applications to assess its ability to find real bugs. We find that AUREBESH is accurate (96.1% recall and 100% precision), and can find bugs in real MVC applications (15 bugs in 22 applications, five of which were acknowledged by the developers).

5.2 Running Example

The traditional application of MVC in web applications is to provide a clear separation between the application data and the HTML output that represents them on the server side. Recent JavaScript MVC frameworks represent the next logical

[26]For simplicity, we will refer to such applications as *MVC applications* or *JavaScript MVC applications*.

```
1 <input type="text" ng-model="userName" placeholder="Type Username" />
2 <button ng-click="searchUser()">
3   List User's Favourite Movies
4 </button>
```

Figure 5.1: HTML code of the "Search" view (search.html)

step, i.e., applying the MVC model to the client-side to separate JavaScript (i.e., data and controls) from the DOM (i.e., the output).

Some popular MVC frameworks include AngularJS, BackboneJS, and Ember.js. Of these, AngularJS is the most widely used [153], with four times as many third-party modules and GitHub contributors, and over 20 times as many Chrome extension users, compared to the closest competitor, BackboneJS. Interest in AngularJS has also increased significantly since 2012, with around 50,000 questions in StackOverflow and 75,000 related YouTube videos. This is more than the corresponding items for the other two frameworks combined. For these reasons, we focus on AngularJS in this work.

We introduce the running example that we will be using throughout the chapter. This example is inspired by real-world bugs encountered by developers of AngularJS applications [83, 146]. The application – which we will refer to as *MovieSearch* – initially takes the user to the "Search" page (Figure 5.1), where the user can input the name of a user, via the `input` element. Clicking on the "List User's Favourite Movies" button leads to the "Results" page (Figure 5.2), which displays the list of movies that corresponds to the user name that has been input, as well as the number of movies in the list. In addition, clicking on the "Which User?" button in the "Results" page would display the current user name in an alert box; for example, if the user name is "CountSolo", the alert would display the message, "The user is CountSolo".

The code for this application contains two views – one corresponding to the "Search" page (Figure 5.1) and the other corresponding to the "Results" page (Figure 5.2) – implemented in HTML. It also contains two models and two controllers implemented in JavaScript, shown in Figure 5.3.

An MVC application consists of *model variables*, *controller functions*, and *groupings*. Figure 5.5 shows how model variables and controller functions are

```
1 <h3 ng-if="userData.display">
2   {{userData.intro}}
3 </h3>
4 <ul>
5   <li ng-repeat="movie in userData">
6     {{movie.name}}
7   </li>
8 </ul>
9 <div id="movieCount">
10   <ng-pluralize count="userData.count" when="movieForms"></ng-pluralize↩
       >
11 </div>
12 <br />
13 <button ng-click="alertUserName()">
14   Which User?
15 </button>
```

Figure 5.2: HTML code of the "Results" view (results.html)

defined and used, in relation to the models, views, and controllers. It also shows how these models, views, and controllers form groupings.

Model Variables. Model variables refer to the objects where the model data is stored, and are represented by identifiers defined within the scope of a particular model. These model variables are defined in models, and are used (either polled or updated) by associated views and controllers. For instance, the `Search` model in the running example defines one model variable, namely `userName` (Figure 5.3, line 4); further, the associated controller (`SearchCtrl`) and view (search.html) use this same variable in Figure 5.3, line 8, and Figure 5.1, line 1, respectively. Similarly, the `Results` model (Figure 5.3, lines 17-26) contains two model variables: `userData` and `movieForms`; these are used by the associated view (results.html) in various lines in Figure 5.2.

Controller Functions. Controller functions, as the name suggests, are functions defined in the controller. These controller functions are used in the view by attaching the function as an event handler to a view element. As an example, the `SearchCtrl` controller in Figure 5.3, lines 7-12 defines one controller function – `searchUser()` – which is subsequently used in the corresponding view (search.html) by setting it as the event handler of a button (Figure 5.1, line 2). Also, the `ResultsCtrl` controller in Figure 5.3, lines 29-31 defines the controller function `alertUserName()`, which is used in the corresponding view

```
1 var searchApp = angular.module('searchApp', 'ngRoute');
2 searchApp.controller('SearchCtrl', function($scope, $location) {
3   //MODEL - Search
4   $scope.userName = "";

6   //CONTROLLER - SearchCtrl
7   $scope.searchUser = function() {
8     var id = getUserId($scope.userName);
9     if (id >= 0) {
10      $location.path('/results/' + id);
11    }
12  }
13 });

15 searchApp.controller('ResultsCtrl', function($scope, $routeParams) {
16   //MODEL - Results
17   $scope.userData = {
18     movieList: getList($routeParams.userId),
19     intro: "Welcome User #" + $routeParams.userId,
20     display: true,
21     count: "two"
22   };
23   $scope.movieForms = {
24     one: '{} movie',
25     other: '{} movies'
26   };

28   //CONTROLLER - ResultsCtrl
29   $scope.alertUserName = function() {
30     alert("The user is " + $scope.userName);
31   };
32 });
```

Figure 5.3: JavaScript code of the models and controllers

```
1 searchApp.config(function($routeProvider) {
2   $routeProvider
3     .when('/', {
4       controller: 'SearchCtrl',
5       templateUrl: 'search.html'
6     })
7     .when('/results/:userId', {
8       controller: 'ResultsCtrl',
9       templateUrl: 'results.html'
10    })
11    .otherwise({
12      redirectTo: '/'
13    });
14 });
```

Figure 5.4: JavaScript code of the routes

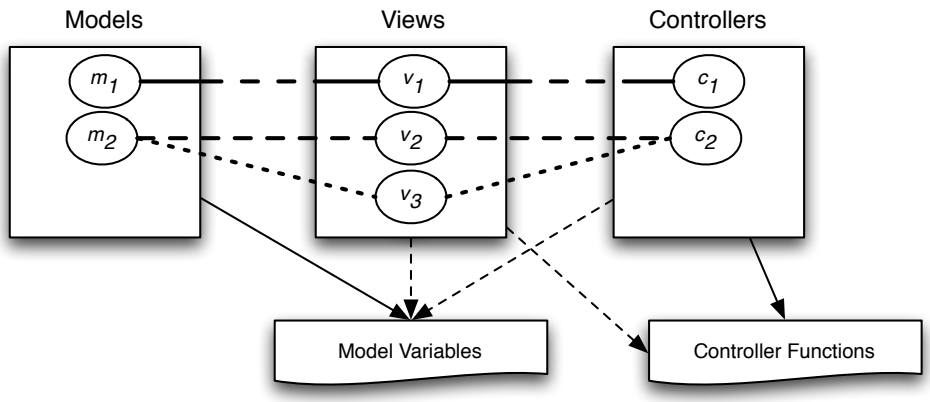


Figure 5.5: Block diagram of the def-use and grouping model for MVC framework identifiers. Solid arrows indicate a "defines" relation, while dashed arrows indicate a "uses" relation. Models, views, and controllers connected with the same line types form a grouping.

(results.html) in Figure 5.2, line 13.

Groupings. Due to the dynamic property of web applications, an MVC application can consist of multiple models, views, and controllers; hence, the programmer must specify which of these models, views, and controllers are associated with each other. Current MVC frameworks allow the programmer to specify these *(model, view, controller)* groupings by embedding the name of the model and controller in the view. These groupings can also be specified using *routers*, as in the case of the running example (Figure 5.4), which links the `Search` model and `SearchCtrl` controller with the search.html view (lines 3-6), and the `Results` model and `ResultsCtrl` controller with the results.html view (lines 7-10).

5.3 Consistency Issues

We now describe two types of consistency issues observed in MVC applications, namely *identifier consistency* and *type consistency*. We focus on these issues in this chapter.

Identifier Consistency. Model variables and controller functions are represented by identifiers in MVC applications. These identifiers are written both in the JavaScript code, when they are defined or used in the model or controller, and in the HTML code, when they are used in the view. To ensure correct operation,

(1) model variable identifiers used in the controller or view must be *defined in the model*, and (2) controller function identifiers used in the view must be *defined in the controller*. While this seems straightforward to enforce at first sight, the following factors complicate the process of maintaining this consistency.

- An identifier is repeatedly used in both the HTML code and the JavaScript code. Even though DOM interactions are abstracted out by MVC frameworks, this repeated usage of identifiers across separate languages makes the application susceptible to identifier inconsistencies. Further, the common practice of implementing models, views, and controllers in separate files – sometimes maintained by separate programmers in collaborative projects – increases the chances of such inconsistencies.

- An application typically contains multiple models, views and controllers grouped together. Hence, the programmer must ensure the consistency not just of one *(model, view, controller)* grouping, but of several groupings. Also, these groupings must be set up correctly, e.g., via the routers, or else an inconsistency may occur.

For instance, the *MovieSearch* application contains two identifier inconsistencies. First, the `ResultsCtrl` controller uses the model variable identifier `userName` in Figure 5.3, line 30, but this identifier is not defined in the `Results` model (it is only defined in the `Search` model, which is not grouped with `ResultsCtrl`); this causes the alert box to display "The user is undefined" after clicking on the "Which User?" button. Second, since the `li` element in the results.html view loops over the `userData` model variable (Figure 5.2, lines 5-7) instead of `userData.movieList`, the reference to `movie.name` in Figure 5.2, line 6 will be undefined with respect to the `Results` model; this causes blank bullet points to be displayed.

Type Consistency. In many cases, the programmer will also need to ensure that the value that is assigned to a model variable – or the value returned by a controller function – has a type consistent with that variable or function's use in the view. For example, in AngularJS, the `ng-if` attribute in the view must be assigned a

Boolean value; a type inconsistency occurs if a model variable that contains a non-Boolean value or a controller function that returns a non-Boolean value is attached to the attribute. Ensuring this consistency is complicated by the fact that JavaScript is a loosely typed language.

MovieSearch contains one such type inconsistency. In Figure 5.2, line 10, the `userData.count` model variable is attached to the `count` attribute, which expects to be assigned a value of type Number; however, `userData.count` is assigned a String in the corresponding `Results` model (Figure 5.3, line 21). This leads to the disappearance of the message that shows the number of movies, inside the `div` element with ID `movieCount` in Figure 5.2.

5.4 Formal Model of MVC Applications

We propose a more formal, abstract model for MVC-based web applications to clearly delineate all the consistency properties of such applications. This model also helps us describe our approach for automatically detecting inconsistencies.

Definition 3 (MVC Application) *An* MVC application *is a tuple* $<\mathscr{M},\mathscr{V},\mathscr{C},\Omega,\Gamma,\omega_{\mathscr{M}},\omega_{\mathscr{V}},\omega_{\mathscr{C}},\gamma_{\mathscr{C}},\gamma_{\mathscr{V}},\phi>$ *where* $\mathscr{M}$ *is the set of models;* $\mathscr{V}$ *is the set of views;* $\mathscr{C}$ *is the set of controllers;* Ω *is the set of model variables; and* Γ *is the set of controller functions. Additionally, we define the following functions.*

- $\omega_{\mathscr{M}} : \mathscr{M} \rightarrow 2^{\Omega}$ *indicates what model variables are defined in a model;*
- $\omega_{\mathscr{V}} : \mathscr{V} \rightarrow 2^{\Omega}$ *indicates what model variables are used in a view;*
- $\omega_{\mathscr{C}} : \mathscr{C} \rightarrow 2^{\Omega}$ *indicates what model variables are used in a controller;*
- $\gamma_{\mathscr{C}} : \mathscr{C} \rightarrow 2^{\Gamma}$ *indicates what controller functions are defined in a controller;*
- $\gamma_{\mathscr{V}} : \mathscr{V} \rightarrow 2^{\Gamma}$ *indicates what controller functions are used in a view;*
- $\phi : \mathscr{M} \times \mathscr{V} \times \mathscr{C} \rightarrow \{true, false\}$ *indicates the model-view-controller groupings.*

Further, a model variable in Ω and a controller function in Γ are represented by a tuple $<id,ty>$, where *id* refers to the identifier, and *ty* refers to the type (for

controller functions, this pertains to the *return* type). The function *I()* projects the *id* portion of these model variables and controller functions onto a set.

An MVC web application is *consistent* if and only if for every element $(m, v, c) \in \mathscr{M} \times \mathscr{V} \times \mathscr{C}$ such that $\phi(m, v, c) = true$, the following properties hold:

Property 1. The view and controller only use model variables that are defined in the model:

$$(\forall \mu)(\mu.id \in I(\omega_{\mathscr{C}}(c)) \cup I(\omega_{\mathscr{V}}(v)) \implies \mu.id \in I(\omega_{\mathscr{M}}(m)))$$

Property 2. The view only uses controller functions that are defined in the controller:

$$(\forall \kappa)(\kappa.id \in I(\gamma_{\mathscr{V}}(v)) \implies \kappa.id \in I(\gamma_{\mathscr{C}}(c)))$$

Property 3. The expected types of corresponding model variables in the view match the assigned types in the model or controller:

$$\begin{aligned}(\forall \mu, \rho)(\mu.id \in I(\omega_{\mathscr{V}}(v)) \wedge \rho.id \in I(\omega_{\mathscr{M}}(m)) \cup I(\omega_{\mathscr{C}}(c)) \\ \wedge\ \mu.id = \rho.id \implies \mu.ty = \rho.ty)\end{aligned}$$

Property 4. The expected and returned types of corresponding controller functions match in the view and controller.

$$\begin{aligned}(\forall \kappa, \tau)(\kappa.id \in I(\gamma_{\mathscr{V}}(v)) \wedge \tau.id \in I(\gamma_{\mathscr{C}}(c)) \wedge \kappa.id = \tau.id \\ \implies \kappa.ty = \tau.ty)\end{aligned}$$

5.5 Approach

To alleviate the consistency issues described, we propose a static analysis approach for automatically detecting identifier and type inconsistencies in MVC applications. We opt for a static instead of a dynamic approach for several reasons. First, static analysis is more lightweight than dynamic analysis, in that the application does not need to execute in order to detect the inconsistencies; this is especially

useful during the development phase, where quick relay of information about the code, such as error messages, is preferred.

Second, dynamic analysis requires user input – i.e., a sequence of user events – and it is not always clear how to choose these inputs. A dynamic approach may be suitable for tools that target specific bugs – such as AUTOFLOX (Chapter 3) and VEJOVIS (Chapter 4) – since the steps to reproduce the bug are known; in contrast, our detector is not targeting a specific bug known to exist in the program, but rather, *looking* for these bugs, without prior knowledge of how to reproduce them. This is the same reason an inconsistency detector is preferred over a mechanism that simply displays an error message when an inconsistency is encountered during execution.

There are also several challenges in designing the above detector, namely,

- **C1**: Model variables are often defined as nested objects (e.g., see Figure 5.3, lines 17-22), and the variables defined *inside* these objects, along with their types, also need to be recorded, thereby complicating the static analysis;
- **C2**: Sometimes, aliases are used in the HTML code to represent model variables defined in the JavaScript code (e.g., the `movie` variable in Figure 5.2, line 5 is an alias for `userData.movieList`, `userData.intro`, etc.). The design needs to be capable of handling these aliases;
- **C3**: Since MVC applications can contain multiple models, views, and controllers, the design needs to infer all the possible groupings to be checked; a simple comparison of *all* identifiers and types collected does not suffice.

Finally, our approach assumes that the code does not contain any instances of `eval`. This assumption is reasonable, as JavaScript MVC frameworks encourage programmers to write in a more declarative style; thus, features used in "vanilla" JavaScript such as `eval` are rarely seen in MVC applications.

5.5.1 Overview

The goal of our automatic inconsistency detector is to find all instances that violate Properties 1–4 in Section 5.4. The block diagram in Figure 5.6 shows an overview

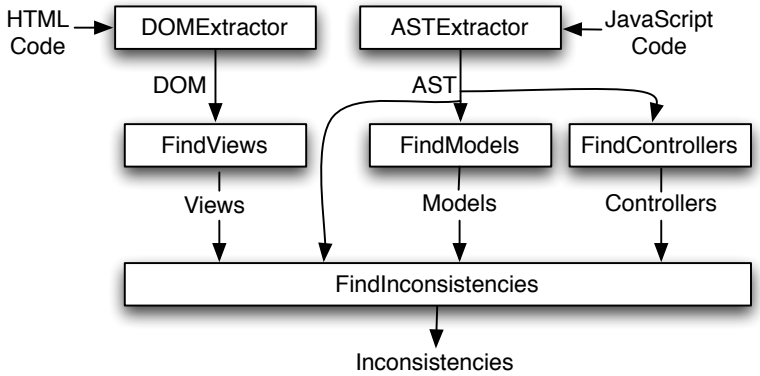


Figure 5.6: Block diagram of our approach.

of our approach. As the figure depicts, the approach expects two inputs, namely the HTML (template) and the JavaScript code. The *DOMExtractor* converts the HTML template into its DOM representation, which is used to simplify analysis of the HTML elements and attributes. Similarly, the *ASTExtractor* converts the JavaScript code into its AST representation.

The modules *FindModels*, *FindViews*, and *FindControllers* statically analyze the DOM and the AST to populate the sets $\mathscr{M}$, $\mathscr{V}$, and $\mathscr{C}$, respectively. In our approach, we chose to represent a model $m \in \mathscr{M}$ as a tuple of the form $<name, ast>$, where *name* is a unique identifier for the model and *ast* is the subtree of the complete AST extracted earlier, containing only the nodes and edges pertinent to the model; for example, the value of *ast* for the `Results` model in Figure 5.3 would be the AST representing lines 17-26. Similarly, a view $v \in \mathscr{V}$ and a controller $c \in \mathscr{C}$ are represented by $<name, dom>$ and $<name, ast>$, respectively. Section 5.5.2 describes in more detail how the above sets are populated.

Once $\mathscr{M}$, $\mathscr{V}$, and $\mathscr{C}$ are all populated, these sets, along with the complete DOM and AST, are input into the *FindInconsistencies* module (see Algorithm 2). The output of this algorithm is a list of inconsistencies Q. It starts by initializing the sets ϕ, as well as the "identifier inclusion functions" $\omega_{\mathscr{M}}$, $\omega_{\mathscr{V}}$, $\omega_{\mathscr{C}}$, $\gamma_{\mathscr{C}}$, and $\gamma_{\mathscr{V}}$ with the contents of $\mathscr{M}$, $\mathscr{V}$, and $\mathscr{C}$ (lines 1-7); here, all the models, views, and controllers initially map to the empty set, since the model variables and controller functions are still not known. These mappings are updated as identifiers are discovered by the *findIdentifiers()* function, described in Section 5.5.3. Likewise, the mappings

Algorithm 2: FindInconsistencies

```
    Input: 𝓜: The set of models
    Input: 𝓥: The set of views
    Input: 𝓒: The set of controllers
    Input: DOM: The complete DOM
    Input: AST: The complete AST
    Output: Q: List of inconsistencies
 1  Q ← ∅;
 2  ϕ ← {((m,v,c), false) | m ∈ 𝓜 ∧ v ∈ 𝓥 ∧ c ∈ 𝓒};
 3  ω_𝓜 ← {(m,∅) | m ∈ 𝓜};
 4  ω_𝓥 ← {(v,∅) | v ∈ 𝓥};
 5  ω_𝓒 ← {(c,∅) | c ∈ 𝓒};
 6  γ_𝓒 ← {(c,∅) | c ∈ 𝓒};
 7  γ_𝓥 ← {(v,∅) | v ∈ 𝓥};
 8  findIdentifiers(𝓜, 𝓥, 𝓒, ω_𝓜, ω_𝓥, ω_𝓒, γ_𝓒, γ_𝓥);
 9  ϕ ← findMVCGroupings(𝓜, 𝓥, 𝓒, DOM, AST);
10  foreach (m,v,c) ∈ {(m,v,c) | ϕ(m,v,c) = true} do
11      foreach mv ∈ ω_𝓥(v) ∪ ω_𝓒(c) do
12          if mv.id ∉ I(ω_𝓜(m)) then
13              Q ← Q ∪ {idMismatch(mv)};
14          end
15          else if !matchingType(mv, ω_𝓜(m)) then
16              Q ← Q ∪ {typeMismatch(mv)};
17          end
18      end
19      foreach cf ∈ γ_𝓥(v) do
20          if cf.id ∉ I(γ_𝓒(c)) then
21              Q ← Q ∪ {idMismatch(cf)};
22          end
23          else if !matchingType(cf, γ_𝓒(c)) then
24              Q ← Q ∪ {typeMismatch(cf)};
25          end
26      end
27  end
```

in ϕ are updated, in line 9, by the *findMVCGroupings()* function (Section 5.5.4). Lines 10-27 are responsible for detecting the identifier and type mismatches, and are described in detail in Section 5.5.5.

5.5.2 Finding the Models, Views and Controllers

The *FindModels*, *FindViews*, and *FindControllers* modules in Figure 5.6 populate $\mathscr{M}$, $\mathscr{V}$, and $\mathscr{C}$, respectively, by locating the corresponding structures or blocks in the HTML and JavaScript code. For example, in AngularJS, models and controllers are added as the body of the function passed to the `.controller()` method as a parameter (see Figure 5.3). Hence, in this case, locating the models and controllers

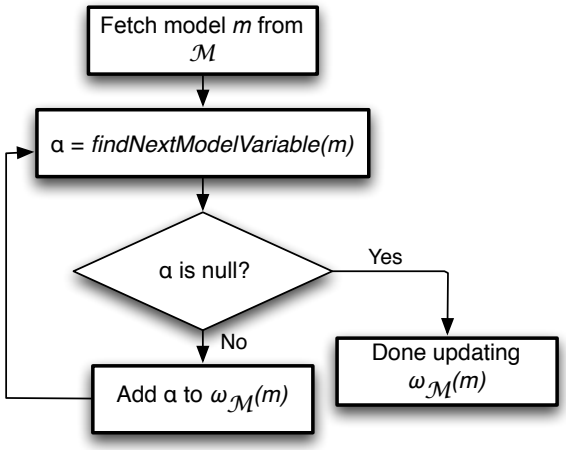


Figure 5.7: Portion of *findIdentifiers* that updates $\omega_{\mathscr{M}}$ for every model. The other "identifier inclusion functions" are updated in a similar way.

involves finding the subtrees in the AST that are rooted at a `CallExpression` for the method `.controller()`, and parsing the body of the function parameter. Similarly, views are normally saved as separate HTML files, so in most cases, finding them is tantamount to identifying these separate files.

5.5.3 Inferring Identifiers

The goal of the *findIdentifiers* module, which is invoked in Algorithm 2, line 8, is to find the model variables and controller functions that are defined or used in every model, view, and controller that were found earlier (see Section 5.5.2), thereby updating the mappings in the "identifier inclusion functions". Figure 5.7 illustrates how *findIdentifiers* looks for model variables defined in every model. A similar algorithm is used to find the model variables and controller functions used or defined in other entities.

The functions *findNextModelVariable* and *findNextControllerFunction* analyze the DOM and the AST according to the syntactic styles imposed by the MVC framework being used. In AngularJS, model variables are defined as a property of the `$scope` variable in an assignment expression (see Figure 5.3, lines 4, 17, and 23); controller functions are defined similarly, albeit the right side of the expression is a Function object (e.g., Figure 5.3, lines 7 and 29). Finding identifiers used in views, however, is trickier – although identifiers appear as attribute values of DOM

elements in many cases, they also typically appear in double curly brace notation as part of a text element (e.g., Figure 5.2, lines 2 and 6); hence, text elements in the view's DOM are also parsed since these may contain references to identifiers.

Type Inference. To find the type assigned to a model variable in the model or controller, our approach looks at the right-hand side of the assignment expression and infers the assigned type based on the AST node (e.g., if the right-hand side is a `StringLiteral` node, the inferred type is String). If the expression is too complicated and the assigned type cannot be inferred, the type is recorded as *unknown* for that identifier, so our type inference algorithm is conservative. The simplification we made for type inference requires the assigned expression to be a literal.[27] Although this may seem to be a significant limitation, note that simple assignments are commonplace in MVC applications, perhaps because MVC frameworks are designed such that applications can be programmed in a "declarative" way [59]; hence, we believe our simplification is justified (we further validate this claim empirically in Section 5.7). Note that type inference works similarly for controller functions, except that the *return value expressions* are parsed instead of assignments.

To infer the expected type of a model variable or controller function used in a view, our approach examines the attribute to which the identifier is assigned and determines if this attribute expects values belonging to a specific type. For instance, the `count` attribute in AngularJS expects a Number, so this is recorded as the expected type for `userData.count` in Figure 5.2, line 10. If the identifier has no expected types, its type is simply recorded as $\perp$, which matches all types.

There are two special cases that our algorithm must handle, namely, nested objects and aliases.

Nested Objects. To address challenge **C1** (see beginning of Section 5.5), we also model nested objects such as the `userData` and `movieForms` variables in Figure 5.3. Our approach represents nested objects as a tree. Each node in the tree represents an identifier, with an assigned type. The trees for the `userData` and `movieForms` variables are joined together in one root as they both belong to the same model. This is shown in Figure 5.8.

[27] Our design also performs some "smart parsing", e.g., it can detect concatenations of strings.

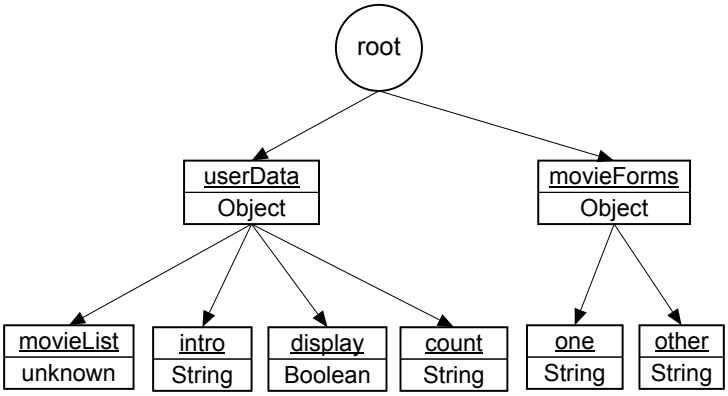


Figure 5.8: Tree representing the model variables in the `Results` model of *MovieSearch*, including all the nested objects. The identifiers are shown at the top of each node, while the types are shown at the bottom.

Analogously, if a model variable in the view uses the dot notation, then it is represented as a sequence of identifiers. For example, `userData.count` in Figure 5.2, line 10 is represented as `root` → `userData` → `count`, with expected type Number.

Aliases. An example of the use of aliases (challenge **C2**) is the `ng-repeat` directive in AngularJS, which replicates the associated HTML element in the DOM for each element of some specified collection. This directive is assigned a string value of the form "<alias> in <collection>", where <collection> is an array or an object, and <alias> is an identifier that represents each member of the array (or each property of the object) in each replication of the HTML element in the DOM.

Figure 5.2, line 5 shows an example, in which the collection is the `userData` object and the alias is `movie`. Therefore, the alias `movie` refers to every property of `userData`, namely `userData.movieList`, `userData.intro`, `userData.display`, and `userData.count`. Subsequently, the reference to `movie.name` in Figure 5.2, line 6, translates to each of these four identifiers, followed by "`.name`". These four sequences are therefore included as model variables in the results.html view – i.e., they are all added in the list that maps to the results.html view when updating $\omega_{\mathscr{V}}$.

5.5.4 Discovering MVC Groupings

At this point, the model variables and controller functions have been discovered, and have been mapped to their associated models, views, and controllers. Our approach must now find all model-view-controller combinations that can potentially appear in the application, to address challenge **C3**. More formally, our approach must find all $(m,v,c) \in \mathscr{M} x \mathscr{V} x \mathscr{C}$ such that $\phi(m,v,c) = true$, thereby updating ϕ in the process.

This procedure is carried out by the *findMVCGroupings()* function; as seen in Algorithm 2, line 9, this function takes $\mathscr{M}$, $\mathscr{V}$, and $\mathscr{C}$ as inputs, along with the complete AST and DOM. The reason the full AST and DOM are needed is that *findMVCGroupings()* will look for information in the DOM that explicitly maps a specific model or controller to a view via an HTML attribute, as well as routing information in the AST that does the same. This information, coupled with the *name* part of each model, view, and controller, allows our approach to determine all the valid groupings.

Take, for example, the router for *MovieSearch* in Figure 5.4. The first route (lines 3-7) groups the `Results` model, the `ResultsCtrl` controller, and the `results.html` view. Thus, ϕ is updated so that the model, view, and controller objects with these respective identifiers together map to *true*. In other words, if *m.name* = `Results`, *v.name* = `results.html` and *c.name* = `ResultsCtrl`, then the design sets $\phi(m,v,c) = true$. This process is repeated for all other groupings discovered.

5.5.5 Detecting Inconsistencies

The final step in our approach is to compare the model variables and controller functions within the same grouping and detect any potential inconsistencies. The pseudocode for this procedure is shown in Algorithm 2, lines 10-27.

The algorithm begins by looking for inconsistencies related to model variables (lines 11-18). Line 11 loops through every model variable used in either the view or the controller. For all such model variables *mv*, the *id* is checked to see if it also exists among the model variables defined in the corresponding model (line 12). If not, this means that Property 1 is violated and there is an identifier inconsistency,

so this inconsistency is included in Q. However, if the *id* does exist in the model, the matching model variable in the model is compared with *mv* to see if they have the same type. If the types do not match, then Property 3 is violated and there is a type inconsistency; this inconsistency is then included in Q. The algorithm for finding inconsistencies in controller functions (lines 19-26) is similar. Note that model variables with unknown types are assumed to match all types. The remaining question, then, is, how are the identifier and type comparisons made? For controller functions, the answer is straightforward – identifiers are compared based on a string comparison, and types are compared based on the assigned and returned types that were previously inferred in Section 5.5.3.

Model variables are, however, more challenging because of the possibility of nested objects. In this case, the sequence representation of *mv* is used to traverse the tree representing the model variables defined in the corresponding model. Take, for instance, the sequence `root` → `userName`, which is used in the `ResultsCtrl` controller, as per Figure 5.3, line 30. If this sequence is used to traverse the tree representing the model variables defined in `Results` (see Figure 5.8) starting from the root, our design will discover that the given sequence does not exist in the tree, and therefore, there is an identifier inconsistency. In addition, the sequence `root` → `userData` → `count` is used in the results.html view, as per Figure 5.2, line 10, and has an expected type of Number since it is assigned to the `count` attribute in `ng-pluralize`. If this sequence is used to traverse the same tree, the traversal will be successful, since the sequence exists in the tree. However, note that the expected type of the terminating node in the traversal (*count*) is String, which does not match the expected type of Number. Thus, a type inconsistency is recorded for this sequence. Finally, if `root` → `userData` → `intro` → `name` – which is one of the sequences the `movie.name` alias translated to as described in Section 5.5.3 – is used to traverse the tree, the traversal will fail, since the sequence does not exist in the tree. As a result, another identifier inconsistency will be recorded. In summary, our design is able to detect all the three inconsistencies in the *MovieSearch* running example.

5.6 Implementation

We implemented our approach in a tool called AUREBESH. It is built on top of the Ace Editor, which is the editor used for the Cloud9 IDE [7]. We have embedded Ace Editor as part of a web application that can be accessed in our website [127]. AUREBESH is implemented entirely using JavaScript, and currently supports MVC applications written with AngularJS.

To invoke the detector, we added a "Find Inconsistencies" button to the IDE, which the user must click. For every inconsistency found by the detector, an error message is highlighted on the line of code containing the inconsistency in the IDE. The user can then click on these error messages to get more details about the inconsistencies.

5.7 Evaluation

To assess the efficacy and real-world relevance of our approach, we address the following research questions:

RQ1 (Real Bugs): Can AUREBESH help developers to find bugs in real-world MVC applications?

RQ2 (Accuracy): What is the accuracy of AUREBESH in detecting identifier and type inconsistencies in MVC applications?

RQ3 (Performance): How quickly can AUREBESH perform the inconsistency detection analysis?

5.7.1 Subject Systems

In total, we consider 20 open-source AngularJS applications for our experiments, listed in Table 5.3. These applications were chosen from a list of MVC applications from AngularJS' GitHub page [60]; in particular, only the applications whose source code is available and unobfuscated are considered, since AUREBESH, in its current state, is incapable of working with obfuscated code. This is not a fundamental limitation though as AUREBESH is meant for developers to use before

Table 5.1: Real bugs found. The "Fault Type" column refers to the fault type number, as per Table 5.2.

Application	Fault Type	Error Message	Severity
Cafe Townsend	2	Undefined model variable `employee.id`	3
	2	Undefined model variable `employee.id`	3
Cryptography	3	Undefined model variable `lastWordCount`	1
Dematerializer	3	Undefined model variable `editing`	3
eTuneBook	5	Undefined controller function `doneTuneSetEditing`	3
	7	Inconsistent type for `currentFilter`	3
Flat Todo	4	Undefined model variable `showTaskPanel`	5
	4	Undefined model variable `showStatusFilter`	5
GQB	7	Inconsistent type for `download.aggregated`	3
Hackynote	3	Undefined model variable `theme.current.css`	4
	3	Undefined model variable `transition.current.css`	4
Reddit Reader	3	Undefined model variable `post`	2
Story Navigator	1	Undefined model variable `ui.columns.status`	3
beLocal	3	Undefined model variable `likeDisabled`	3
Linksupp	3	Undefined model variable `startEating`	2

obfuscating their code. As shown in Table 5.3, the applications cover a variety of sizes and application types.

5.7.2 Methodology

Real Bugs. To answer RQ1, we run AUREBESH on the 20 subject systems. For this experiment, we also ran our tool on two additional AngularJS applications developed by students for a software engineering course at the University of Victoria [181], namely beLocal and Linksupp. We analyze every error message reported by AUREBESH for these applications to see if it corresponds to a real bug. We report any true positives (i.e., error messages that correspond to real bugs) and false positives (i.e., spurious error messages) that we find.

Accuracy. To measure the accuracy (RQ2), we conduct a fault injection study

on the subject systems. An injection is performed on an application by introducing a mutation to a line of code from one of its source files (either the HTML or JavaScript code), running AUREBESH on this mutated version of the application, and then recording if AUREBESH detects the inconsistency introduced by the mutation. If the inconsistency is detected, the result of the injection is marked as "successful"; otherwise, the injection is marked as "failed".

In this experiment, we consider ten types of mutations, as seen in Table 5.2. The "expected behaviour" for a mutation describes the correct error message that AUREBESH is expected to display when running AUREBESH on an application with this mutation applied. Each of these mutation types corresponds to a *violation* of one of the four properties listed in Section 5.4; hence, the results for a mutation type give an indication of how well AUREBESH detects violations of the corresponding property.

For each application, we perform 20 injections per mutation type, which amounts to a total of 200 injections per application. However, note that a mutation type may not be applicable for certain applications (e.g., not all controllers use model variables, in which case mutation type #2 will not be applicable); this explains why several applications have fewer than 200 injections (see Table 5.3). The specific location mutated in the code is chosen uniformly at random from among the lines of code applicable to the current mutation type. For each injection, we record the number of successful detections, the number of failed detections, and the number of spurious error messages introduced by the mutation; this allows us to report both the recall (the number of successful detections over the total number of injections) and precision (the number of successful detections over the total number of error messages displayed) of AUREBESH.

Performance. We measure performance by running AUREBESH on each subject system, recording the analysis completion time and averaging over multiple trials.

Table 5.2: Types of faults injected. MV refers to "model variable", and CF refers to "controller function".

Type #	Description	Expected Behaviour	Property Tested
1	Modify the name of a MV used in *line N* of a view	Detect undefined MV in *line N*	1
2	Modify the name of a MV used in *line N* of a controller	Detect undefined MV in *line N*	1
3	For a particular MV used in *line N* of a view, delete the definition of that MV in a corresponding model	Detect undefined MV in *line N*	1
4	For a particular MV used in *line N* of a controller, delete the definition of that MV in a corresponding model	Detect undefined MV in *line N*	1
5	Modify the name of a CF used in *line N* of a view	Detect undefined CF in *line N*	2
6	For a particular CF used in *line N* of a view, delete the definition of that CF in a corresponding controller	Detect undefined CF in *line N*	2
7	For a particular MV used in the view that expects a certain type *T1*, modify the definition of that MV in *line N* of a corresponding model so that the type is changed to *T2*	Detect type mismatch in *line N* (*T1* expected but type is *T2*)	3
8	For a particular MV used in the view that expects a certain type *T1* and defined in *line N* of a corresponding model, modify the expected type to *T2* by mutating the `ng` attribute name	Detect type mismatch in *line N* (*T2* expected but type is *T1*)	3
9	For a particular CF used in the view that expects a certain type *T1*, modify the return value of that CF in *line N* of the controller to a value of type *T2*	Detect type mismatch in *line N* (*T1* expected but type is *T2*)	4
10	For a particular CF used in the view that expects a certain type *T1* and returns a value in *line N* of a corresponding controller, modify the expected type to *T2* by mutating the `ng` attribute name	Detect type mismatch in *line N* (*T2* expected but type is *T1*)	4

Table 5.3: Fault injection results. The size pertains to the combined lines of HTML and JavaScript code, not including libraries.

Application	Application Category	Size (LOC)	Successful Detections	Failed Detections	Total Injections	Recall (%)	Precision (%)
Angular Tunes	Music Player	185	40	0	40	100.00	100.00
Balance Projector	Finance Tracker	511	140	20	160	87.50	100.00
Cafe Townsend	Employee Tracker	452	160	0	160	100.00	100.00
CodeLab	RSS Reader	602	79	1	80	98.75	100.00
Cryptography	Encoder	523	120	0	120	100.00	100.00
Dematerializer	Blogging	379	186	14	200	93.00	100.00
Dustr	Template Compiler	493	80	0	80	100.00	100.00
eTuneBook	Music Manager	5042	177	23	200	88.50	100.00
Flat Todo	Todo Organizer	255	107	13	120	89.17	100.00
GitHub Contributors	Search	459	142	18	160	88.75	100.00
GQB	Graph Traversal	1170	194	6	200	97.00	100.00
Hackynote	Slide Maker	236	120	0	120	100.00	100.00
Kodigon	Encoder	948	120	0	120	100.00	100.00
Memory Game	Puzzle	181	40	0	40	100.00	100.00
Pubnub	Chat	134	120	0	120	100.00	100.00
Reddit Reader	Reader	255	120	0	120	100.00	100.00
Shortkeys	Shortcut Maker	407	120	0	120	100.00	100.00
Sliding Puzzle	Puzzle	608	40	0	40	100.00	100.00
Story Navigator	Test Case Tracker	415	117	3	120	97.50	100.00
TwitterSearch	Search	357	199	1	200	99.50	100.00
OVERALL			2421	99	2520	96.07	100.00

5.7.3 Results

Real Bugs. After running AUREBESH on the original, unaltered versions of the subject systems, AUREBESH displayed a total of 15 error messages in 11 applications. We reported these error messages to the developers, and five of them (the error messages from Story Navigator, beLocal, Linksupp, and two from Hackynote) were acknowledged as real issues and fixed. The other applications, unfortunately, are no longer maintained by the developers, so our bug reports for those applications remain unacknowledged. Nonetheless, we analyzed the 15 error messages and found that they are all true positives i.e., they all correspond to real-world bugs.

Of the 15 bugs, we found 13 identifier inconsistencies and 2 type inconsistencies; as shown in Table 5.1, each of the bugs our tool found maps to one of the fault types in Table 5.2. Note that the two error messages in Cafe Townsend, while identical, correspond to two different bugs. With regards to *why* these faults were committed, we identified the following patterns:

- **Identifier defined elsewhere** (7 cases): There are several cases where assignments representing the model variable definitions are placed not in the model itself, but inside controller functions. This applies, for example, to the model variable `lastWordCount` in Cryptography;

- **Incorrect identifier** (5 cases): In some cases, the inconsistencies arise because the programmer has typed incorrect identifiers. For instance, in Hackynote, the identifier given for a property in the nested object `theme.current` is `src`, but the identifier expected by the view is `css`;

- **Boolean assigned a string** (2 cases): The two type inconsistencies involved the programmer assigning a string to a model variable that expects a boolean value. For instance, in GQB, the `download.aggregated` variable was erroneously assigned the string value "true" instead of the boolean value `true`;

- **Identifier name not updated** (1 case): This occurs in eTuneBook. Upon inspection, it turned out that the undefined controller function `doneTuneSetEditing` in eTuneBook was defined in previous versions of the application, but was replaced with another function with a different name; the

reference to the old name remained in the view. This is an example of a regression bug.

Table 5.1 also shows the severity of the bugs, based on our qualitative assessment of these bugs; here, we use Bugzilla's ranking scheme, where 1 represents the lowest and 5 represents the highest severity. Although some of the bugs are cosmetic (e.g., the bug in Cryptography simply causes one of the labels to display as "One of possible –word permutations...", with the number next to "–word" missing), many of the bugs have considerable impact on the application. For example, the first bug in Flat Todo renders the "plus" button – which adds todos in the list – useless. A similar effect takes place in eTuneBook, where the missing controller function makes one of the buttons inoperable, thereby preventing the user from exiting edit mode. Also, the two bugs in Hackynote prevented the user from removing the theme and transition present in the slides.

Lastly, AUREBESH displayed only one false positive, in the Linksupp application. The reason is that the application uses the `$rootScope` variable to define model variables to be within the scope of *all* models. Our tool assumes that every model variable used in a view is defined only via the `$scope` variable, leading to the false positive. Nonetheless, the low number of false positives indicates that the error messages displayed by our tool are trustworthy, minimizing the effort required to filter out any spurious messages.

Accuracy. Table 5.3 shows the per-application results for the fault injection experiment. As the table shows, AUREBESH is very accurate, yielding an overall recall of 96.1%, and attains a perfect recall for eleven of the twenty applications. In addition, AUREBESH did not output any spurious messages during any of the injections; hence, AUREBESH *was able to attain an overall precision of 100% in this experiment*.

To understand what is causing the failed detections, we divide the results in terms of the properties (Section 5.4) being violated by the mutation types. As seen in Table 5.4, Properties 1, 3 and 4 have imperfect recalls. We analyzed the 27 failed detections for Property 1, which represents the consistency of model variable identifiers, and found that they all result from the usage of "filters" in conjunction with model variables in views. These filters are used in AngularJS to customize the

Table 5.4: Fault injection results per property.

Property	Successful Detections	Failed Detections	Total Injections	Recall (%)
1	1293	27	1320	98.0
2	560	0	560	100.0
3	268	52	320	83.8
4	180	20	200	90.0

appearance of model variables' values when displayed by the view; AUREBESH currently does not recognize these filters and ignores them when parsing, leading to the failed detection. Note that this limitation is implementation specific, and can be overcome by extending the parser.

We also analyzed the 72 failed detections for Properties 3 and 4, both of which represent the consistency of types. We found that these are caused by our assumption that the values assigned by model variables or returned by controller functions are either literals or simple expressions, and thus have types that are easy to infer. More specifically, in these 72 cases, the values assigned or returned are either complex expressions or retrieved from an external database. This prevented AUREBESH from inferring the types; since AUREBESH is conservative, it does not report the type inconsistency. Overall, these cases constitute 14% of the cases where type inference was needed. Note that in the remaining 448 cases (86% of the cases), the values were literals or simple expressions, which indicates that our assumption is valid in the majority of cases.

Performance. For each subject system, AUREBESH was able to perform the analysis in an average time of 121 milliseconds, with a worst-case of 849 milliseconds for the largest application, eTuneBook. This indicates that performance is not an issue with our tool.

5.8 Discussion

Limitations

The implementation of our approach for AngularJS has a few limitations. First, as explained in Section 5.7.3, AUREBESH currently disregards the presence of filters in views in AngularJS. Also, as mentioned in Section 5.7.3, our tool currently disregards the use of the `$rootScope` variable, which can lead to false positives.

With respect to the approach itself, a limitation is in our type inference algorithm, which assumes simple assignments and return values. Our results suggest that this assumption is reasonable; however, we also found that a considerable number (around 14%) of pertinent assignments and return values involve complex expressions or external database accesses, so a more advanced type inference algorithm is needed. Lastly, AUREBESH does not consider inheritance in MVC applications, where models are made descendants of other models to allow model variables to be inherited. Again, we have not encountered this in practice, but it can occur.

Another limitation of AUREBESH is that it works only on applications written using AngularJS. While AngularJS is the most popular client-side MVC framework, our problem formulation (Section 5.3), formal model (Section 5.4) and algorithm (Section 5.5) are all fairly generic and can be applied to other MVC frameworks with minimal modifications.

Threats to Validity

One internal validity threat regards the mutation types used in our fault injection experiment, and how representative they are of both our inconsistency model and real-world bugs. To address this issue, we selected the mutation types such that they all map to the consistency properties presented in Section 5.4. In addition, each of the 15 real-world bugs that we found in one of our experiments maps to a mutation type, as described in Section 5.7.3, giving an indication of the mutation types' representativeness.

As with any experiment that considers a limited number of subject systems, the generalizability of our results may be called into question, which is an external threat to validity. Unfortunately, since AngularJS is a fairly new framework, applications using this framework are quite scarce. Fortunately, the AngularJS GitHub page provides a list of web applications using that framework; to mitigate the external threat, we chose applications of different types and sizes from this list.

Finally, the source code of all the subject systems we considered in our experiments are all available online; further, we kept our own records of the source code of these systems that AUREBESH analyzed. Our tool AUREBESH, is also publicly

available. Hence, our experiments are reproducible.

5.9 Related Work

MVC has been applied to various domains, and one of its earliest uses can be traced back to Xerox PARC's Smalltalk [91]. The pattern has also been applied to the server-side of web applications [94], where the model and controller are implemented on the server and the view is represented by the HTML output on the client. Since the application of MVC to client-side web application programming is a fairly recent development, there are only a few papers addressing this topic. Much of the research in this area has focused on the application of the MVC model to JavaScript development, tailored towards specific application types [54, 72, 162]. Studies on JavaScript MVC frameworks' properties have been limited to an analysis of their maintainability [23, 86]. Unlike our present work, these studies do not consider the presence of consistency issues in JavaScript MVC applications, nor do they propose an approach for analyzing MVC application code.

Several papers have analyzed the characteristics of common JavaScript frameworks, such as jQuery [55, 62, 98, 147]. Richards et al. [144] and Ratanaworabhan et al. [142] analyze the effect that different frameworks have on the dynamic behaviour of JavaScript in web applications. Feldthaus and Møller [52] look at TypeScript interfaces, and propose a tool that checks for the correctness of these interfaces. In prior work, we also briefly explored the relationship between JavaScript frameworks and JavaScript faults that occur in production websites [128]. Our current work differs from these studies in that they consider non-MVC frameworks, which have different usage patterns compared to MVC frameworks.

Finally, considerable work has been done on the application of MVC on the server-side [67, 118, 123, 155], where frameworks such as Spring MVC and JSF are used. Wojciechowski et al. [177] compared different MVC-based design patterns on the server-side, and analyzed the frameworks' characteristics, such as their susceptibility to file upload issues. In contrast, our work is concerned with the client-side of web applications.

5.10 Conclusion and Future Work

In this chapter, we presented an automated technique and tool called AUREBESH, which statically analyzes applications written using AngularJS – a popular JavaScript MVC framework – to detect inconsistencies in the code. Our evaluation of AUREBESH indicates that it is accurate, with an overall recall of 96.1% and a precision of 100%. We also find that it is useful in finding bugs in MVC applications – in total, we found 15 real-world bugs in 22 AngularJS web applications.

Chapter 6

Cross-Language Inconsistency Detection

This chapter describes another technique for automatically detecting faults in MVC applications[28] While effective, our first fault detector (AUREBESH) only works for one JavaScript framework (AngularJS), and can only detect four types of inconsistencies. These limitations led us to ask the following question: *Could we design a more general fault detection technique?* Answering this question will allow us to design a technique that works for a larger set of JavaScript frameworks, and can detect a larger set of inconsistencies. Having these considerations in mind, we designed a technique called HOLOCRON,[29] which provides a more holistic response to RQ1C in Chapter 1.1.

In the sections that follow, we describe the general problem of detecting faults in JavaScript-based web applications, and we motivate our decision to apply automatic fault detection specifically to MVC applications. Thereafter, we describe our fault detection approach.

6.1 Introduction

With JavaScript's increase in popularity also comes increased expectations in JavaScript's reliability. Unfortunately, despite this greater demand for reliability, JavaScript is still notoriously error-prone [128], and, as our bug report study reveals, these errors often lead to high-impact consequences such as data loss and security

[28]The main study in this chapter is in preparation for submission to a software engineering conference.

[29]HOLOCRON is named after a continuity database used to ensure the consistency of *Star Wars* canon.

flaws [130, 145]. To mitigate this problem, web developers rely heavily on testing, and many researchers have developed tools to enhance this testing process [15, 101, 109, 116, 137]. While testing is an integral part of the software development phase, the large number of states found in web applications often renders testing insufficient in *detecting* many bugs in such applications.

An alternative to testing is static code analysis, which allows programmers to find bugs by *reasoning* about the program, without having to execute it. Several techniques have been proposed to automatically detect JavaScript bugs through code analysis. For example, JSLint [43] detects syntactic errors in JavaScript programs. Other work by Jensen et al. [78, 79] analyze JavaScript code to find type inconsistencies. Finally, in Chapter 5, we also proposed AUREBESH [132], which is a technique for automatically detecting inconsistencies in AngularJS web applications. A common issue with the above techniques is that they detect bugs based on a predefined list of inconsistency rules or bug patterns. As a result, the bugs they detect will be limited to those encompassed by these hardcoded rules. This is especially problematic for web applications which use a wide variety of frameworks and libraries, each with its own coding rules and conventions. Moreover, web frameworks typically evolve very fast, and hence hardcoded rules may become obsolete quickly.

An alternate approach for finding bugs is anomaly detection, proposed by Engler et. al. [49] and commercialized as Coverity [161]. Instead of hardcoding rules as the above techniques do, this approach looks for *deviant behaviours* in the input application's code, with these deviations providing an indication of potential bugs in the program. This approach has the advantage that it can learn rules from common patterns of behaviour, and hence the rules do not need to be updated for each framework. Anomaly-based approaches, however, only support single-language bug detection, and hence, will not be able to find bugs resulting from cross-language interactions. This makes them not particularly suitable for JavaScript-based web applications, because such applications frequently involve *cross-language interactions*; for example, HTML code often sets up event handlers by embedding JavaScript functions as attribute values, and JavaScript code often retrieves HTML elements by using DOM element selectors.

In response to the above issues, we introduce a new technique for automatically

detecting web application *inconsistencies* that is both general-purpose and cross-language. In this context, an inconsistency pertains to two web application code components where one component makes an erroneous assumption about the other component, thereby leading to a bug. Our approach is general because it makes no assumptions about which two code components are inconsistent. Further, it is cross-language because the two incompatible code components involved in the inconsistency can come from different programming languages (i.e., JavaScript and HTML). *To the best of our knowledge, we are the first anomaly-based inconsistency detection approach for web applications that is able to deal with cross-language interactions*.

As with Chapter 5, we continue to focus on detecting inconsistencies in MVC applications, which, as defined in Chapter 5, are web applications implemented using JavaScript MVC frameworks such as AngularJS, BackboneJS, and Ember.js. We focus on MVC frameworks due to their rising popularity [160], and the fact that they do not interact directly with the DOM, which makes static analysis suitable to understand such applications.

We make the following contributions in this work:

- We demonstrate that there are many inconsistency classes in MVC applications, and that there is no single inconsistency class that particularly dominates over the others. Further, many of these inconsistencies span multiple programming languages;

- We propose a general, cross-language technique for automatically detecting inconsistencies in MVC applications. Unlike prior work, our approach does not look for specific inconsistency classes, but instead uses subtree pattern matching to infer these classes. Further, it uses association rule mining to find the cross-language links between the HTML and the JavaScript code;

- We implement our technique in a tool called HOLOCRON, and we evaluate the tool on 12 JavaScript MVC applications. We find that HOLOCRON can find a total of 18 bugs in these twelve applications, many of which result from cross-language inconsistencies and hence cannot be found by other tools. Further, HOLOCRON also finds many code smells in these applications. On

average, one of two inconsistencies reported by HOLOCRON is either a bug or a code smell.

6.2 Background and Motivation

For this work, we target a general class of bugs that we call *inconsistencies*, which we formally define shortly. In addition, we focus on detecting inconsistencies in MVC applications statically. Recall from Chapter 5 that an MVC application consists of a model, which defines the application data; a controller, which defines the functions that manipulate the values of the application data; and a view, which uses the data and functions defined in the model and controller to define a user interface.

Static analysis is often sufficient for MVC applications, as they rely primarily on JavaScript bindings instead of DOM interactions; hence, even though the DOM still changes, the JavaScript code interacts primarily with these static bindings instead of directly interacting with the DOM itself. In contrast, frameworks such as jQuery rely on direct DOM interactions with JavaScript, which would force us to perform dynamic analysis to keep track of the changing DOM states.

6.2.1 Definitions

We define a ***code component*** to be any contiguous piece of JavaScript or HTML code that could span a single line (e.g., function call, HTML text, etc.) or multiple lines (e.g., function definition, view definition, etc.). These code components can be represented by subtrees of the JavaScript code's Abstract Syntax Tree (AST) or the HTML code's DOM representation; we use these subtrees in our design, as described in Section 6.3. We define an inconsistency as follows.

Definition 4 (Inconsistency) *Two code components $comp_A$ and $comp_B$ are inconsistent if $comp_A$ makes an erroneous assumption about $comp_B$ – that is, $comp_A$ incorrectly assumes that $comp_B$ possesses a particular property that it does not have – where the erroneous assumption can be implicitly inferred from the code (e.g., without having to rely on specifications). An* **inconsistency** *refers to any pair ($comp_A$, $comp_B$) that are inconsistent.*

Therefore, an inconsistency is a bug that can potentially be discovered without

the help of external specifications, which means that detecting these inconsistencies lends itself to an auotmated analysis of the web application code. An inconsistency is considered *cross-language* if $comp_A$ and $comp_B$ belong to different programming languages.

In our evaluation of AUREBESH in Chapter 5, we found that four classes of these inconsistencies occur in MVC applications. For example, we found inconsistencies in which the view components in the HTML code use variables that are erroneously assumed to be defined in the model components in the JavaScript code. In Section 6.5.3, we demonstrate through another study of bug reports that these inconsistencies abound in MVC applications, and many of these inconsistencies go much beyond the classes found in this prior study. Thus, AUREBESH will not work for these other classes, which is why a new approach is needed.

6.2.2 Motivating Examples

We introduce two examples of real inconsistencies found in open-source MVC applications. The first inconsistency comes from an AngularJS application [164], and the second inconsistency comes from a BackboneJS application [103].

AngularJS Example. For this application, the JavaScript code attempts to close a modal instance by calling the `close()` method, as follows

```
1 $modalInstance.close('close');
```

However, this leads to incorrect application behaviour (i.e., a dialog box becomes broken), as the `$modalInstance` service has been replaced in the newer version of AngularJS being used by the application by `$uibModalInstance`. In this case, the function call above incorrectly assumes that the service object being dereferenced is valid, thereby leading to the inconsistency. This example demonstrates the potential usefulness of a learning-based approach for finding these inconsistencies, as the evolution of framework APIs often modifies or introduces new coding rules.

BackboneJS Example. In this example, the JavaScript code is attempting to bind an element from an HTML view template to a layout view object by assigning the `el` property with an element selector, as shown below.

```
1 Marionette.LayoutView.extend({
2   el: '.some-view',
3   ...
4 });
```

In this case, the selector `'.some-view'` does not correspond to any elements in the HTML template, which causes the binding to fail. In other words, the view incorrectly assumes that a particular element with the class "some-view" is defined in the HTML template.

6.2.3 Challenges

One of the main challenges of this work is that we need to infer programmer intent in order to label code components as inconsistent. For example, in the AngularJS example above, how do we know that `$modalInstance` is an incorrect service name? Usually, this inference is carried out using specifications, but these specifications are typically not available in web applications. One approach is to leverage *repeated* appearances of the same code pattern to infer intent. Any deviations to this pattern are likely to be inconsistencies. Further, the more examples there are of the same pattern, and the fewer counterexamples there are, the more likely it is to be an actual pattern. For example, in the AngularJS example, there are multiple instances of `$uibModalInstance.close(...)`, which is a near-match, albeit the service name is different. This indicates that the service name `$modalInstance` is likely incorrect.

Another challenge is that we have to deal with cross-language inconsistencies, as this forces our design to infer "links" between code components coming from different programming languages. For instance, in the BackboneJS example above, our design needs to infer that the value of the `el` property needs to be a selector for an element in the HTML template. We can decide to simply hardcode this relationship in our detector, but the problem is that this link is specific to the BackboneJS framework, and will therefore not work for applications written using other frameworks. Therefore, we need a general approach to discover the link.

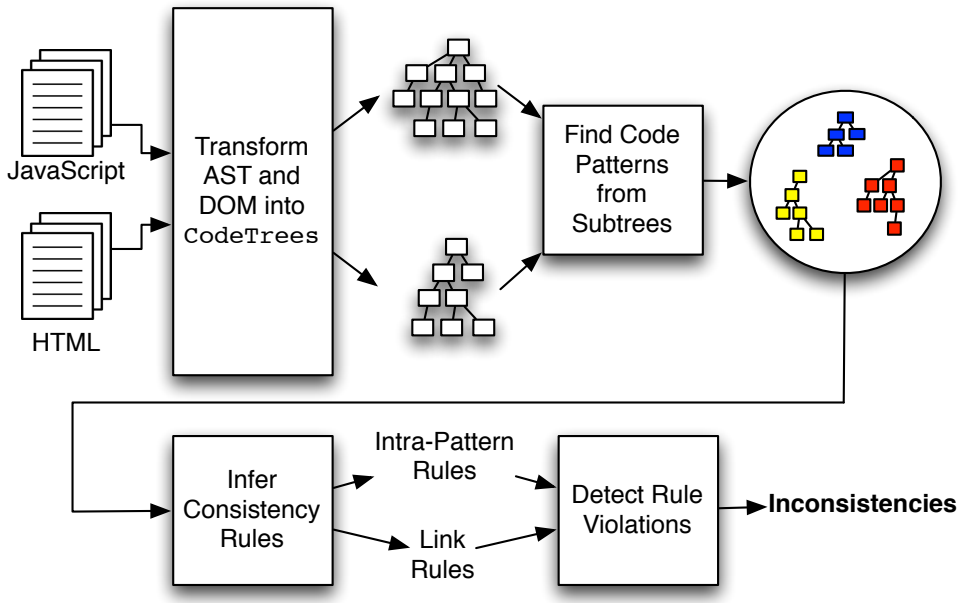


Figure 6.1: Overview of our fault detection approach

6.3 Approach

The block diagram in Figure 6.1 presents an overview of our approach. As the diagram shows, our approach takes the web application's JavaScript and HTML code as input, and transforms these pieces of code into their corresponding AST and DOM representations, respectively. As explained in Section 6.3.1, the AST and the DOM trees are transformed into another tree object called a `CodeTree`, which allows the approach to perform standardized operations on those trees. In addition to the trees generated from the input web application, our technique also retrieves the AST and DOM of other web applications that use the same framework; these web applications are retrieved from the web.

Once the `CodeTrees` are generated for the input and sample code, the approach analyzes the trees to find commonly repeated patterns in the trees (Section 6.3.2). To do so, it looks for *subtree repeats*, which by definition are subtrees that appear multiple times in the `CodeTrees`; these subtree repeats represent common *code patterns* present in the web application, and will be used to establish the consistency rules.

After finding the subtree repeats, the approach looks at each code pattern found in the previous module and formulates consistency rules based on them. Our approach looks at two levels of consistency rules. On the first level, our approach infers *intra-pattern consistency rules*, which are consistency rules defined by the individual code patterns themselves. On the second level, our approach also infers *inter-pattern consistency rules* (i.e., *link rules*), which are consistency rules inferred based on *pairs* of code patterns. As described in Section 6.3.3, these link rules allow our approach to find consistency rules that have to do with the interaction between code written in different programming languages.

Finally, once the consistency rules have been inferred, our approach finds deviations to these rules, based on a comparison between the `CodeTree` objects (which represent the AST and the DOM) and the inferred consistency rules (Section 6.3.4).

6.3.1 Transforming Code into Trees

The first module of our approach transforms the JavaScript and the HTML code of the input web application into their corresponding AST and DOM representations. More specifically, an AST is constructed for each JavaScript file (or JavaScript code within the same `script` tag), and a DOM representation is created for each HTML file. These transformations are done to simplify analysis, as trees are a well-studied data structure for which many search and comparison algorithms have been proposed. It also makes our approach easier to extend to other languages, as code-level analysis would have required complicated parsing algorithms that rely on knowledge of the syntax of specific languages.

In order to standardize the way that our approach operates on the ASTs and the DOMs, we transform them both into a data structure called the `CodeTree`. A `CodeTree` is defined as a tree $T(V,E)$, where V is the set of nodes in the tree, and E is the set of edges. For every node $v \in V$, we define the following properties.

- *v.type*: Set to "ast" if v is an AST node, or "dom" if v is a DOM node;
- *v.label*: Set to the node label. If v is an AST node, the label is set to either the node type (e.g., ExpressionStatement, Identifier, etc.), or the corresponding

identifier or literal value. If v is a DOM node, the label is set to a tag name (if v is an `Element` node), attribute name (if v is an `Attribute` node), or text (if v is a `Text` node, or an attribute value);

In addition to the above properties, for each `CodeTree` node, we also keep track of its *parent* and *childNodes*, as well as the *lineNumber*, *columnNumber*, and *sourceFile*.

6.3.2 Finding Common Patterns

The goal of our next module is to find patterns of repeating subtrees in the `CodeTrees`. These patterns will form the basis for the consistency rules. We first define the following terms.

Definition 5 (Subtree Repeats) *Let $T_1, T_2, ..., T_N$ be `CodeTrees`, and let $R(V_r, E_r)$ and $S(V_s, E_s)$ be two different subtrees of any of these `CodeTrees`. Then, R and S are defined to be* **subtree repeats** *of each other if R and S are isomorphic, where two nodes are considered equal iff they have the same type and label. Hence, each node $v_r \in V_r$ has a corresponding node $v_s \in V_s$ such that $v_r.type = v_s.type$ and $v_r.label = v_s.label$.*

Definition 6 (Code Pattern) *A* **code pattern** *C is defined as a set of subtrees, such that for every pair of subtrees $R, S \in C$, R and S are subtree repeats.*

Hence, the goal of this module is to find all the code patterns in the `CodeTrees` generated earlier in the previous module. Our technique for finding these code patterns is similar to some prior techniques [120], and in particular, to the approach used by Baxter et al. [25] to detect clones in the source code of a program using the AST. More specifically, our design looks for all *full subtrees* in each `CodeTree`, and assigns a hash value to each of these subtrees; note that a full subtree pertains to a subtree that contains *all* the descendant nodes from the subtree's root. All subtrees that hash to the same value are placed in their own hash bin. Once this process is complete, the subtrees in each hash bin are compared to detect any collisions; if there are collisions in a hash bin, the hash bin will be split to resolve the collisions. These hash bins represent the code patterns.

The difference with Baxter et al.'s technique is that when comparing subtrees, our design *abstracts out* the labels of nodes pertaining to variable and function identifiers, as well as attribute values; in other words, our design temporarily treats the labels of these nodes as the empty string, so that their original labels will not be considered in the hashing. Our design also abstracts out any labels that identify the data type of a literal node (e.g., StringLiteral, NumberLiteral, etc.). Doing so will enable our design to find intra-pattern consistency rules, as will be described in Section 6.3.3.

Using Code Examples from the Web. In addition to the target application, our design also looks for patterns that are found in example web applications taken from the web. The purpose of using these example applications is to allow patterns to appear more frequently, thereby giving our design greater confidence about the validity of the pattern found; this potentially decreases the rate of false negatives. Further, using these examples will also allow "non-patterns" to appear less frequently, percentage-wise, thereby decreasing the rate of false positives.

The example web applications retrieved must use the same framework and framework version as the target web application. To determine this, our design analyzes the `script` tags of the target web application to infer the framework, and then it looks for example web applications that attach the same framework in their script tags. To keep track of the web application, we augment each `CodeTree` node to also include the *appId*, which is a unique ID that identifies which web application the node – and hence, the tree – belongs to.

6.3.3 Establishing Rules from Patterns

Once the code patterns are found, our design then proceeds to analyze these patterns to infer consistency rules. In this case, the design looks for both intra-pattern consistency rules and inter-pattern consistency rules (i.e., link rules).

Intra-Pattern Consistency Rules

As previously mentioned, intra-pattern consistency rules are defined by individual code patterns. Algorithm 3 shows the pseudocode for finding these rules, and reporting violations to them. The main idea behind this algorithm is to *concretize* –

Algorithm 3: FindIntraPatternInconsistencies

```
    Input: C_set: The set of code patterns
    Input: t: The threshold for dominant subpatterns
    Output: PI: Set of intra-pattern inconsistencies
 1  PI ← ∅, remaining ← ∅;
 2  codePatternQueue ← {C | C ∈ C_set};
 3  while codePatternQueue is not empty do
 4      C ← codePatternQueue.dequeue();
 5      preorderNum ← getNextNodeToConcretize(C);
 6      if preorderNum < 1 then
 7          remaining ← remaining ∪ {C}; continue;
 8      end
 9      subPatterns ← ∅;
10      foreach subtree S ∈ C do
11          node ← getPreOrderNode(S, preorderNum);
12          markAsConcretized(node);
13          if subPatterns.hasKey(node.label) then
14              subPatterns[node.label].add(S);
15          end
16          else
17              subPatterns[node.label] = {S};
18          end
19      end
20      D ← getDominantPattern(subPatterns);
21      if 100 |D|/|C| >= t then
22          expected ← getPreOrderNode(D[0], preorderNum);
23          foreach code pattern CP ∈ subPatterns do
24              if CP ≠ D then
25                  foreach subtree S ∈ CP do
26                      inc ← getPreOrderNode(S, preorderNum);
27                      PI ← PI ∪ {(inc, expected)};
28                  end
29              end
30          end
31          codePatternQueue.enqueue(D);
32      end
33      else
34          codePatternQueue ← codePatternQueue ∪ subPatterns;
35      end
36  end
37  C_set ← mergeRemaining(remaining);
```

one by one – the nodes that were abstracted out in the previous module by revealing their original labels.

The algorithm starts by enqueuing each code pattern in a queue (line 2). For each code pattern C in the queue, the design determines the earliest node – in depth-first, pre-order – that is still abstracted out among the subtrees in C. It achieves this by calling the `getNextNodeToConretize()` function, which returns the

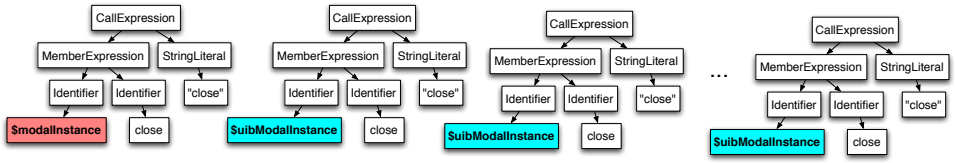


Figure 6.2: Example of an intra-pattern consistency rule violation

pre-order number of the earliest node (line 5). Once the pre-order number of the earliest node is determined, the actual nodes in the subtrees in C that correspond to this pre-order number are compared and marked as concretized (lines 11-12), and the subtrees are partitioned according to the label of the concretized node (lines 13-18). The partitions are included in an associative array called *subPatterns* (line 9).

Once the partitions are found, the algorithm looks for the *dominant pattern*, which represents the largest partition (line 20). If the number of subtrees in the dominant pattern constitutes greater than $t\%$ of all the subtrees in the original code pattern C, where t is a user-set threshold, all the subtrees belonging to the *non-dominant* patterns are considered intra-pattern inconsistencies (lines 22-32) and are discarded; here, an intra-pattern inconsistency is represented by a tuple of the inconsistent node – i.e., the node that was just concretized in the inconsistent subtree – and the expected node – i.e., the node that was just concretized in any subtree belonging to the dominant pattern (line 27). This process is repeated until there are no further nodes to concretize, after which all remaining partitions belonging to the same original code pattern at the start of the algorithm are merged (line 37).

As an example, consider the subtrees in Figure 6.2, which form a code pattern; this code pattern is found in the AngularJS example introduced in Section 6.2.2. Here, the current node being concretized is the left-most leaf node of each subtree, which, in this case, represents the name of the service being dereferenced. The subtrees are then partitioned according to the label of this concretized node. In this case, there are two partitions – one containing the left-most subtree, with the concretized node coloured red, and another containing the rest of the subtrees, with the concretized node coloured blue. The latter partition is deemed to be the dominant pattern, so the subtree in the other partition is labeled as inconsistent.

Algorithm 4: FindLinkRules

```
    Input: (C_from, C_to): Pair of code patterns
    Output: L: Set of link rules
 1  L ← ∅;
 2  foreach (S_from, S_to) ∈ C_from × C_to do
 3      i ← 1;
 4      node_from ← getPreOrderNode(S_from, i);
 5      while node_from ≠ null do
 6          j ← 1;
 7          node_to ← getPreOrderNode(S_to, j);
 8          while node_to ≠ null do
 9              if node_from ≠ node_to and node_from.label = node_to.label then
10                  lr ← (i, S_from, j, S_to);
11                  L ← L ∪ {lr}
12              end
13              node_to ← getPreOrderNode(S_to, ++j);
14          end
15          node_from ← getPreOrderNode(S_from, ++i);
16      end
17  end
```

Link Rules

In addition to finding the intra-pattern consistency rules, our design also looks for consistency rules that describe the relationship *between* code patterns. This process not only allows our design to find relationships between pieces of code in the same programming language, but also across languages, i.e., *cross-language* relationships.

All link rules are of the following form: *The ith pre-order node in Subtree S_1 is equal to the jth pre-order node in Subtree S_2.* Our design finds the link rules for each pair of code patterns (C_{from}, C_{to}), as shown in Algorithm 4. In this case, the algorithm iterates through every pair of subtrees between the two code patterns (line 2). For each of these pairs of subtrees, the algorithm goes through every pair of nodes between the two subtrees (lines 3-16), and compares the two nodes to see if they have the same label. If they have the same label, a new link rule is added to the list, where each link rule is uniquely identified by the subtree pair S_{from} and S_{to}, and their respective pre-order indices i and j. The link rules found by running this algorithm will then be used to find link rule violations, as described in the next subsection.

6.3.4 Detecting Violations

Violations to the intra-pattern consistency rules are detected in conjunction with finding those rules, as described in Section 6.3.3. For the link rules, we make a distinction between unconditional link rule violations and conditional link rule violations, as we describe below.

Unconditional Link Rule Violations

A link rule violation is unconditional if the link rule is violated by a code component (represented by a subtree) regardless of where the component is located in the code. An example of an unconditional violation is the BackboneJS example in Section 6.2.2.

To determine whether a link rule lr is violated, our design goes through each pair of code patterns C_{from} and C_{to}, as before. It then determines which pairs of subtrees between C_{from} and C_{to} satisfy lr. There are two ways in which a subtree in C_{from} can be marked as an inconsistency in this module. First, if a subtree $S_{from} \in C_{from}$ does not satisfy the link rule lr when paired with any subtree $S_{to} \in C_{to}$, *and* a large percentage[30] $pv\%$ of the other subtrees in C_{from} satisfy lr at least once, then S_{from} will be considered an inconsistency.

For instance, the left box in Figure 6.3 shows the code pattern to which the inconsistent code in the BackboneJS example (from Section 6.2.2) belongs. As indicated by the arrows in this figure, almost each subtree in this code pattern corresponds to a class attribute definition in the HTML code (right box in Figure 6.3); the only subtree that does not have a corresponding class attribute definition is the one with the node highlighted in red (``some-view'').This subtree is therefore labeled as an inconsistency, assuming that $pv \leq 75\%$.

In addition, if a subtree $S_{from} \in C_{from}$ does not satisfy the link rule lr when paired with a *specific* subtree $S_{to} \in C_{to}$, and a large percentage of the other subtrees in C_{from} satisfy the link rule lr with S_{to}, then S_{from} will also be considered an inconsistency.

[30]This is a parameter chosen by the user.

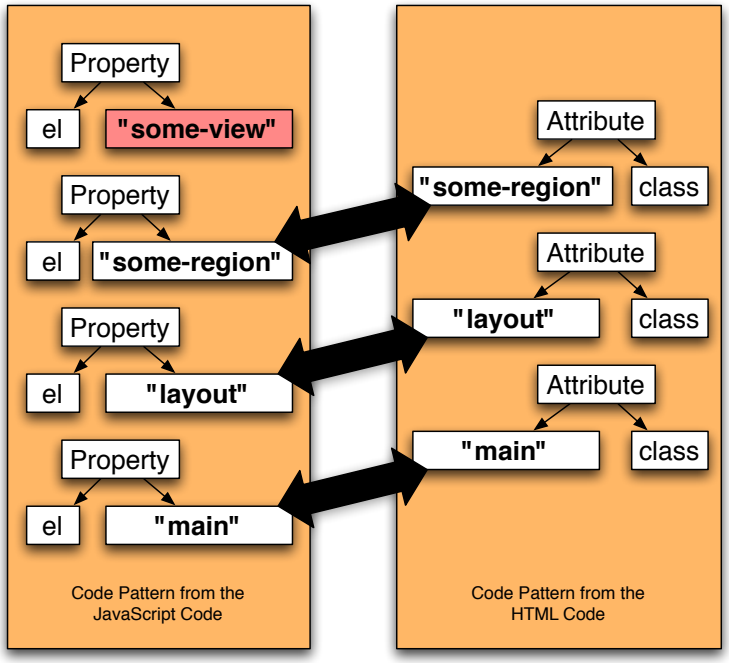


Figure 6.3: Example of an unconditional link rule violation. The subtrees are slightly altered for simplicity.

Conditional Link Rule Violations

A link rule violation is conditional if the link rule is violated *given that* the code component is located in a specific area in the code. For example, suppose a view V in the HTML code is associated with a model M in the JavaScript code. Further, suppose that the following link rule has been found: *The identifier <x> in the subtree with pattern* ***ng-model='<x>'*** *is equal to the identifier <y> in the subtree with pattern* ***$scope.<y>***. In this case, if there exists no subtrees *in the model* M with pattern **$scope.<y>** that matches a certain subtree *in the view* V with pattern **ng-model='<x>'**, then this latter subtree is considered a violation of the link rule (i.e., V is using an identifier that is never defined in the corresponding model M). Note how this link rule violation only occurs *given that* the subtrees being compared are located in M and V. Therefore, this is considered a conditional link rule violation.

To find the conditional link rule violations, we use a well-known data mining

technique called *association rule learning* [6]. This technique takes a set of *transactions* as input, where each transaction contains a set of *items* that apply to that transaction. Based on an analysis of these transactions, the technique looks for rules of the form $\{a_1, a_2, ..., a_n\} \Rightarrow \{b_1, b_2, ..., b_m\}$, where both the left and right side of the implication are subsets of all the items. In addition, the technique only reports rules that match or exceed a particular confidence value, i.e., the percentage of transactions that follow the rule.

Hence, when finding the conditional link rule violations between pairs of code patterns C_{from} and C_{to}, we create a transaction for each subtree pair (S_{from}, S_{to}). The items included in each transaction include all the link rules satisfied by the subtree pair, as well as the ancestor nodes of the root of each subtree; these ancestor nodes dictate which areas in the source code the subtrees are located. We use the apriori algorithm [5] to infer association rules with a confidence value greater than a user-set parameter *cv%*; we are particularly interested in association rules of the following form

$$\{an_{from}, an_{to}\} \Rightarrow \{lr\}$$

where an_{from} and an_{to} are ancestor nodes of the subtrees S_{from} and S_{to}, respectively, and lr is a link rule. Once these association rules are found, they are compared against each subtree pair to determine which subtree pairs do not satisfy them; these non-satisfying subtree pairs are then reported as inconsistencies.

6.4 Implementation

We implement our technique in an open-source tool called HOLOCRON.[31] This tool is implemented in JavaScript as a plugin for Brackets, which is an IDE for web development developed by Adobe [3]. To use the tool, the user only needs to specify the top folder of the target web application. The output of the tool is a list of the inconsistencies found; each inconsistency is shown to the user as a message identifying the inconsistent line of code, and an example of the expected behaviour based on the consistency rule.

[31] http://ece.ubc.ca/~frolino/projects/holocron/

The JavaScript code is parsed into an AST using Esprima [71], and the HTML code is parsed into its DOM representation using XMLDOM [82]. For finding the association rules, we adopt an existing implementation of the apriori algorithm [152].

6.5 Evaluation

We now evaluate the relevance and effectiveness of HOLOCRON by answering the following research questions:

RQ1 (Prevalence of Inconsistencies): Do inconsistencies occur in MVC applications and if so, what are the characteristics of these inconsistencies?

RQ2 (Real Bugs): Can HOLOCRON be used by developers to detect bugs in real-world MVC applications?

RQ3 (Performance): How quickly can HOLOCRON detect inconsistencies?

6.5.1 Subject Systems

For our experiments, we consider four open-source applications for each of the three main MVC frameworks (AngularJS, BackboneJS, and Ember.js), for a total of 12 applications. These three frameworks are the most widely used JavaScript MVC frameworks, experiencing a 538% growth in popularity from January 2013 to April 2016 [153] The applications are listed in Table 6.1, with the sizes ranging from 6 to 43 KB (185-1659 LOC). These applications were taken from various lists of MVC applications available on GitHub [17, 48, 60]. In particular, the applications we chose were the first four applications from each framework found from these lists which also had a GitHub repository, with preference given to those that had a working demo, as this simplified the task of reproducing any bugs found by our tool.

6.5.2 Experimental Methodology

Prevalence of Inconsistencies (RQ1). To answer RQ1, we manually analyze bug reports that have been filed for MVC applications on GitHub. More specifically,

we examine 30 bug reports for applications implemented in each of the three main MVC frameworks – AngularJS, BackboneJS, and Ember.js – for a total of 90 bug reports. We only consider fixed or closed bugs to prevent spurious reports. To find the bug reports, we use GitHub's advanced search feature, searching in particular for GitHub issues that are given the label "bug", and whose status is "closed". We perform the same search for each of the three MVC frameworks, using the keywords "angularjs", "backbone", and "emberjs", respectively. We discard any search results that correspond to applications *not* written in any of these three frameworks, as well as results that do not pertain to the web application's client-side code. We then take the first 30 bug reports that satisfy the conditions described from each of the three search results, and use those bug reports for our analysis. Note that we did not confine ourselves to the 12 subject systems for this analysis.

For each of the bug reports, we first determine whether the bug corresponds to an inconsistency, as defined in Section 6.2. If so, we determine the bug's inconsistency category, which is based on the inconsistent code components, as well as the erroneous assumption made by one of the components (see Section 6.2.1). Further, we also determine whether the bug is "cross-language" – that is, whether the bug results from an inconsistency between multiple programming languages.

Real Bugs (RQ2). For RQ2, we run HOLOCRON on the subject systems described in Section 6.5.1 and record all the inconsistencies reported by the tool. We examine each of these reported inconsistencies to determine if it corresponds to a real bug (i.e., it is indicative of an error that leads the application to a failure state). Based on a pilot study, we set the intra-pattern violation threshold t to 90%, the unconditional link rule violation thresholds pv to 95%, and the conditional link rule violation threshold cv to 85%. Finally, we use example code from five open-source web applications to train the analysis with more samples. These five applications include the other three subject systems using the same framework, as well as two additional applications found on GitHub [17, 48, 60]. We report the number of bugs found by HOLOCRON, as well as its precision (i.e., number of bugs per inconsistency reported). We also measure the number of code smells identified by HOLOCRON.

Performance (RQ3). We measure the amount of time it takes for HOLOCRON to

run on each subject application. We report the aggregate times in each run, as well as the runtime of each module for the slowest run. We run our experiments on a Mac OS/X 10.6.6 machine (2.66 GHz Intel Core 2 Duo, with 4 GB of RAM).

6.5.3 Prevalence of Inconsistencies (RQ1)

Of the 90 bug reports we studied, we found that *70% of these bug reports correspond to an inconsistency*. These did not need the application's specifications to detect, pointing to the promise of a tool such as ours which finds inconsistencies. For example, one of the Ember.js applications passed a modal object to the `buildUrl()` method, even though this method, which is part of the Ember.js API, expects a string as its first parameter. This inconsistency could be inferred based on other usages of the method which were correct. The remaining 30% of the bugs, however, required prior knowledge of the application's specifications. For example, one of the bugs was caused by the fact that the programmer did not update the `display` style of an element to 'block'; in this case, prior specification was needed to establish that the programmer intended to modify the style to "block" rather than "none".

The per-framework results are summarized in Figure 6.4. As this figure shows, 73% of bug reports correspond to inconsistencies for the AngularJS and Ember.js applications, and 63% of bug reports correspond to inconsistencies for BackboneJS applications. These results suggest that inconsistencies are prevalent in web applications created using JavaScript MVC frameworks. *We further found that 35% of the inconsistencies are cross-language*. For example, one of the bugs resulted from the programmer erroneously using the `data-src` attribute instead of the `src` attribute in the HTML code, which led to incorrect bindings with the JavaScript code. Therefore, existing tools that detect bugs based on single languages alone will not be able to detect a significant percentage of these inconsistencies.

Figure 6.5 shows the distribution of inconsistency categories we found; again, an inconsistency category is uniquely identified by the two components that are inconsistent, as well as the incorrect assumption made by one of the components. As this figure illustrates, most inconsistency categories appear only once in the bug reports we studied; for example, 30 categories had only a single inconsistency

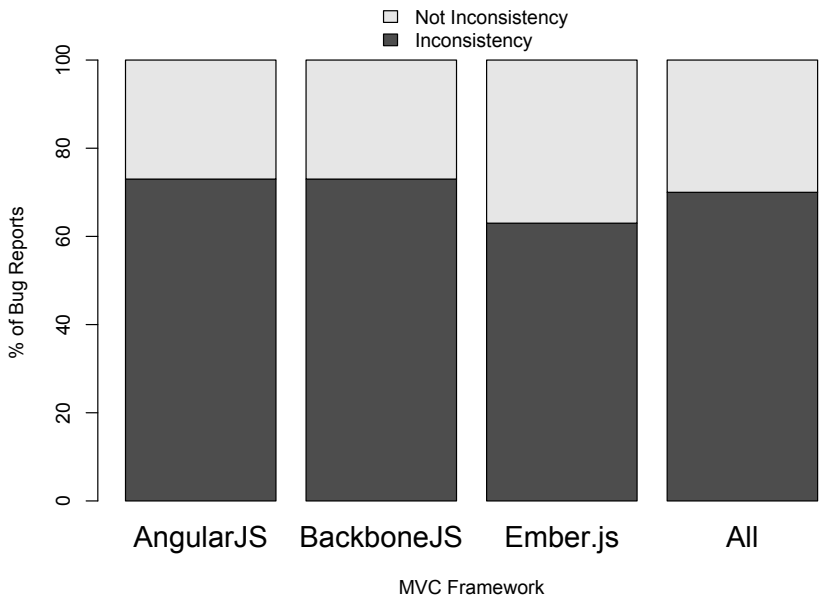


Figure 6.4: Percentage of bug reports classified as an inconsistency for each MVC framework

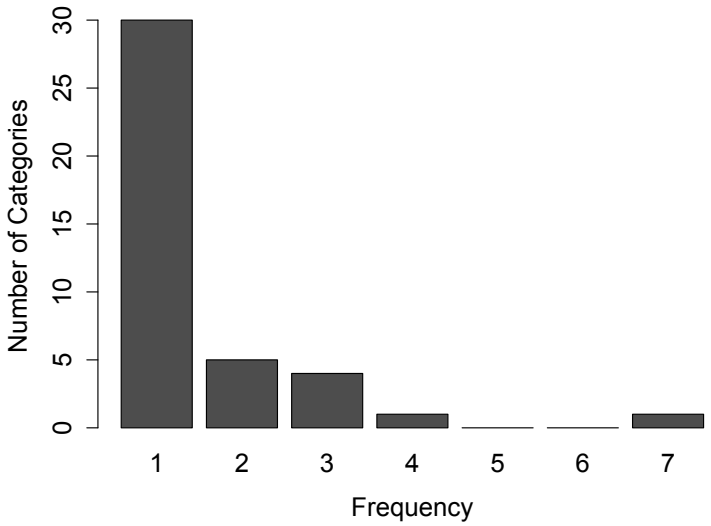


Figure 6.5: Number of inconsistency categories with a particular frequency. Most inconsistency categories have just 1-2 inconsistencies in them.

Table 6.1: Number of real bugs found per application. The size is shown in KB, with lines of code (LOC) in parentheses. The size pertains to both the HTML and JavaScript code in each application, not including libraries.

Framework	Application	Size (loc)	# of Bugs	Code Smells	Total
AngularJS	`angular-puzzle`	20 (608)	3	3	8
	`projector`	19 (569)	3	0	7
	`cryptography`	20 (582)	1	0	2
	`twittersearch`	10 (357)	1	0	1
BackboneJS	`cocktail-search`	10 (396)	2	6	13
	`contact-manager`	19 (701)	2	2	5
	`webaudiosequencer`	43 (1659)	1	7	15
	`backbone-mobile`	9 (240)	1	0	2
Ember.js	`todomvc`	8 (299)	0	2	3
	`emberpress`	21 (610)	2	11	22
	`giddyup`	12 (386)	1	2	9
	`bloggr`	6 (185)	1	0	4
OVERALL			18	33	91

each. The category with the largest share is represented by 7 bug reports, which in this case corresponds to "Incorrect Parameter Type", i.e., a JavaScript function call erroneously assumes that one of its arguments is of a particular type (i.e., string, boolean, etc.), despite other calls to the same function passing an argument of the correct type. Further, we found a total of 41 different inconsistency categories in our experiment. The large number of categories suggests that there are many different rules that are implicitly used by programmers in writing JavaScript MVC based applications. This is why it makes sense to deploy an approach such as ours that discovers the rules automatically rather than hard-code them.

6.5.4 Real Bugs (RQ2)

Table 6.1 shows the result of running HOLOCRON on the subject systems. In total, HOLOCRON *was able to detect **18 unreported bugs** from 12 MVC applications*. We have reported these bugs to the developers, with confirmation from the developers still pending, although we were able to manually reproduce the bugs and confirm them ourselves. As seen in this table, HOLOCRON was able to find a bug in *all* of the applications tested, except for one (`todomvc`). Further, HOLOCRON found 12 unconditional link rule violations, 4 conditional link rule violations and 2 intra-pattern consistency rule violations. This result demonstrates that HOLOCRON can be used by web developers to find bugs representing various types of consistency rule violations.

Further, out of 18 real bugs found, five were cross-language inconsistencies. The bug in `cryptography`, for instance, results from an assignment in the JavaScript code incorrectly assuming that an element in the HTML code has a numerical value, even though it is a string. In addition, the bug in `twitters-earch` resulted from a controller in the JavaScript code assuming that an input element in the HTML code has its type attribute defined, which is not the case. Detection of these cross-language inconsistencies is made possible by HOLOCRON looking for link rules in the applications – something which prior bug detection tools do not do.

In all, HOLOCRON was able to find bugs spanning fifteen inconsistency categories. For example, the bug in `giddyup` is caused by a property assignment erroneously assuming that it has a corresponding route assignment in the Ember.js router. In contrast, AUREBESH, which also targets MVC applications, will not be able to detect this bug because (1) it only considers four pre-determined inconsistency categories, to which this inconsistency found in `giddyup` does not belong, and (2) it only works for AngularJS applications, whereas `giddyup` is developed with Ember.js. Indeed, AUREBESH is only capable of detecting two of the inconsistencies that HOLOCRON identified, both of which come from `angular-puzzle`.

Finally, many of the bugs found have potentially severe consequences on the application. For example, the bugs in `projector` – which are caused by an incorrect assumption about the type of value being assigned to an object property – all cause the application to hang. Further, the bug in `webaudiosequence-r` causes an audio clip to no longer be playable after a sequence of input events. Thus, HOLOCRON finds bugs that can have high impact on the application.

Precision. Like most static analysis tools, HOLOCRON incurs false positives (i.e., inconsistencies that are not bugs). False positives occurred in all but one of the applications (`twittersearch`) . In total, HOLOCRON reported 91 inconsistencies, 18 of which are real bugs, meaning 73 were false positives.

To better understand the characteristics of these false positives, we manually analyzed the reasons for the inconsistencies. We found that 22 of these 73 false positives occur because of frequent usage of a certain identifier contrasted with infrequent usage of another identifier. For instance, in `angular-puzzle`, there

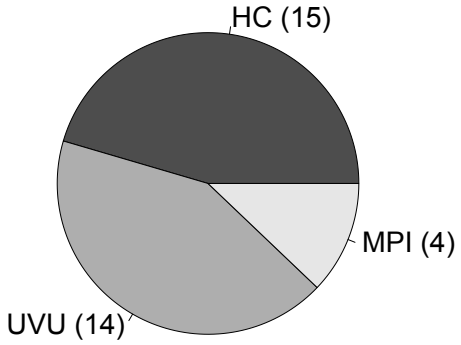


Figure 6.6: Categories of code smells found by HOLOCRON. HC stands for Hardcoded Constants, UVU for Unsafe Value Usage, and MPI stands for Multi-Purpose Identifiers

are two main arrays, both of which are accessed through the `this` identifier – `words` and `grid`. While `grid` is used almost 20 times in the code, `words` is used only twice, leading to the false positive. Additionally, 18 of the false positives occur because of frequent usage of certain kinds of literals, contrasted with infrequent usage of another kind of literal. For example, in `projector`, most of the object method calls take string literals as parameters, but there are three such calls that take number literals as parameters, which are reported as inconsistencies.

We also found that 33 of the false positves correspond to code smells, the detection of which could help the developer improve the quality of the code. The pie chart in Figure 6.6 shows the three categories of code smells discovered by our tool in these applications. The first category is "Hardcoded Constants" (HC), where constant literal values are used in calculations and method calls. This detracts from application maintainability. The second category is "Unsafe Value Usage" (UVU), which means a certain value is dereferenced without accounting for the possibility of it being `null`. For instance, in `emberpress`, one of the inconsistencies occur as a result of the `get()` method being called directly through a model object repeatedly; this is potentially unsafe if the object is `null`, and it is good practice to call the method via the `Ember` object instead (i.e., `Ember.get()`). The third category is "Multi-Purpose Identifiers" (MPI), wherein the *same* identifier is being used for multiple unrelated objects (e.g., in `angular-puzzle`, the identifier "src" is used both as a class name for a `div` element in the HTML code and as a

name for a puzzle object in the JavaScript code). This is a code smell as it detracts from the readability of the code, and can lead to confusion among programmers.

These results indicate that 51 out of the 91 inconsistencies that are reported by HOLOCRON correspond to either bugs or code smells. Hence, *approximately 1 out of every 2 reports are potentially useful to the developer in improving the web applications' quality*.

6.5.5 Performance (RQ3)

On average, HOLOCRON ran for 1.14 minutes for each of the 12 applications. Since HOLOCRON will typically be run prior to deployment of the application, this is an acceptable overhead. The worst-case run occurred with `webaudiosequencer` – which is also the largest application – where HOLOCRON ran for almost 8 minutes. In this case, most of the time was spent on finding the link rule violations. This is likely because of the fact that in this module, all pairs of subtree classes are compared with each other, as well as all subtrees within these classes, to find the link rules.

6.5.6 Threats to Validity

An external threat to validity is that we used a limited number of applications in our evaluation, which calls into question the generalizability of our results. Nonetheless, to ensure that our subject systems are representative of real bugs, we chose applications coming from various frameworks, sizes, and application types, as seen in Table 6.1.

In addition, for our study of bug reports in RQ1, we categorize the inconsistencies based on a qualitative analysis of the reports, some of which may not be very descriptive. To mitigate this, we also look at other aspects of the bug, including patches and commits associated with the bug.

Finally, the bugs that we studied for RQ1 are limited to fixed bugs with the label "bug" and with the status "closed"; however, many GitHub developers do not necessarily label bug reports, which means we may have missed certain bugs in our analysis. Nonetheless, we decided to choose bug reports this way because it facilitated the search for valid bugs to analyze. We mitigated this issue by simpli-

fying the keywords used to search for the bug reports, so that the search results will include a larger representation of the bugs.

6.6 Discussion

The main assumption behind our approach is that there are sufficient examples of a consistency rule that appear for it to both successfully learn the consistency rule and detect any violations to that rule. While this is the case for large applications, small applications have very few samples to learn from, and hence may incur large numbers of false positives and false negatives. To mitigate this problem, we augment our code patterns with subtrees found in code examples from other applications, as discussed in Section 6.3.2. Doing so allows our design to be more confident about the validity of the consistency rules, as well as "debunk" any consistency rules that may lead to false positives. In fact, we found that without the example code, our false positive rates more than doubled. Using more and better examples can likewise bring down the false positive rate. This is a subject of future work.

Furthermore, our main focus in this chapter is in using HOLOCRON to detect inconsistencies in MVC applications. Nonetheless, our design can be run on web applications using non-MVC JavaScript frameworks, such as jQuery. This may lead to a large number of inaccuracies, as the JavaScript code in these frameworks interacts *directly* with the DOM, which undergoes many changes throughout the execution of the web application. However, HOLOCRON may be able to detect inconsistencies *within* the JavaScript code, as well as inconsistencies between the JavaScript code and any component of the DOM that does not get modified. To detect the remaining inconsistencies, we may need to augment our static analysis approach with dynamic analysis. We plan to explore this direction in the future.

As seen in the performance results, HOLOCRON spends most of its time looking for link rule violations. This can be mitigated by limiting the number of compared subtree pairs. However, this may result in loss of coverage. Another possible way of improving the runtime is by employing a machine learning approach in which the tool would *learn* the link rules over multiple runs. By employing this approach, the tool would no longer have to find all the link rules in each run, as

some of these link rules have already been learned in prior runs. This is also a subject of future research.

6.7 Related Work

Fault Detection. Considerable work has been done on software fault detection through code analysis [69, 74, 81, 96, 171, 173]. For example, PR-Miner [95] tries to detect violations of implicit programming patterns mined by the tool, similar to our work, although unlike HOLOCRON, it only considers rules derived from pieces of code that frequently appear together. Perhaps the closest analogue to our work in this chapter is Engler et al.'s work on finding bugs based on deviant behaviour, which makes use of the notion of "belief propagation" to infer correct program behaviour [49]. The main difference with our work is that these prior techniques cannot detect cross-language inconsistencies, as they implicitly assume a single-language model.

Further, static analysis techniques such as FindBugs [73] and AUREBESH (Chapter 5) detect faults based on hardcoded rules or bug templates. Additionally, dynamic analysis techniques such as DLint [56] check consistency rules based on "bad coding practices". As shown in our evaluation (RQ1), this can lead to many missed bugs, especially for JavaScript MVC applications, as there are no specific inconsistency categories that dominate over the others. Further, the frameworks used in web applications evolve fast – thus, rules that apply today may be obsoleted tomorrow and new rules introduced.

Cross-Language Computing. An empirical study conducted by Vetro et al. [166] shows that cross-language interactions could lead to higher defect proneness, particularly for C programs. The significant number of cross-language inconsistencies we found during our bug report study strongly suggests that this also holds for JavaScript programs, and in particular, those implemented with MVC frameworks. Thus, it is important for bug-finding tools to consider cross-language inconsistencies.

Much of the work done on cross-language computing has focused on detecting the dependencies between multiple programming languages [139, 140]. Only a few techniques perform analysis in a cross-language-aware manner, including

XLL [106] and X-Develop [158], both of which perform code refactoring. Further, Nguyen et al. [122] recently proposed a tool to perform cross-language program slicing for web applications, with particular focus on PHP code and its interaction with client-side code. Unlike HOLOCRON, these proposed techniques do not deal with potential inconsistencies that occur in cross-language interactions.

6.8 Conclusions

We presented an automatic fault detection technique that finds inconsistencies in JavaScript MVC applications. Our technique analyzes the AST and DOM representations of the web application code, and it looks for both intra-pattern consistency rules and link rules; violations to these rules are thereby reported to the user. We implemented this approach in an open-source tool called HOLOCRON, and in our evaluation of open-source MVC applications, HOLOCRON was able to find 18 previously unreported bugs. In addition, while false positives do occur, many of these point to smells that can help web developers improve code quality.

Chapter 7

Conclusions and Future Work

Our main goal in this dissertation was twofold. First, we wanted to understand JavaScript bugs, in order to determine if impactful client-side JavaScript bugs do indeed plague web applications, and to have a better grasp of the causes and propagation characteristics of these bugs. To achieve this first goal, we conducted a large-scale empirical study of client-side JavaScript bug reports found in the bug repositories of open-source web applications. We found, in our qualitative analysis of 502 bug reports, that these bugs fall under a small number of fault categories; in particular, the vast majority (68%) of these bugs can be classified as DOM-related faults. Further, we found that a large number of these bugs have severe impact on the web application, and most of these severe bugs are themselves DOM-related. Finally, we identified common error patterns among the bugs we studied, and discovered that most of these errors are committed in client-side code.

Having attained sufficient understanding of JavaScript bugs, our second goal was to develop automated techniques to mitigate them. As discussed in Chapter 1, we decided to pursue fault detection and repair as our mitigation strategy instead of error prevention. While we acknowledge the importance of error prevention in potentially reducing the maintenance costs and the person-hours spent fixing bugs, we believe that fault detection is likewise important, due to the inevitability of bugs not just in JavaScript programming, but programming in general.

To achieve this second goal, we developed various JavaScript fault detection and repair techniques. More specifically, we first developed AUTOFLOX, which performs automatic localization of DOM-related faults, as well as VEJOVIS, which automatically suggests repairs for the same fault model. Subsequently, we also

developed automatic fault detection techniques for MVC applications, which do not interact directly with the DOM, but are nonetheless susceptible to inconsistencies between their various components. The first fault detection technique – AUREBESH – automatically detects identifier and type inconsistencies, and the second technique – HOLOCRON – is a generalized detector that finds both single-language and cross-language inconsistencies. All of these techniques have been implemented as open-source tools, which are available online [125].

The rest of this chapter discusses the expected impact of this dissertation, as well as future avenues of research that could be explored to extend this work.

7.1 Expected Impact

Perhaps the most important contribution of this dissertation is the fact that it places the problem of JavaScript reliability in the limelight of web application research. In the past, researchers and developers focused exclusively on the language's security and performance because initially, JavaScript was used minimally; hence, they were only concerned with ensuring that the JavaScript code did not provide a backdoor for attacks, and that it did not negatively affect the speed of the web application. However, with the rising popularity of AJAX, the situation has changed drastically, with JavaScript playing a much bigger role in the web application's functionality, thereby increasing demand for more reliable JavaScript code; and indeed, as we have demonstrated, JavaScript reliability is a real problem plaguing web applications. Further, we have demonstrated that solutions do exist for alleviating this problem. Our hope, then, is that researchers will continue to recognize this importance and strive to improve JavaScript reliability.

One of our main discoveries in this dissertation is the prevalence of DOM-related faults, many of which had high severity and were difficult to fix, as demonstrated in Chapter 2. This result from our bug report study became the driving force for the remainder of the dissertation; it inspired us to develop both automated techniques for localizing and repairing these DOM-related faults, as well as automated techniques for detecting inconsistencies in MVC applications, many of which resemble DOM-related faults inasmuch as they are cross-language. We believe the importance of this result will continue to have an impact on future research related

to the analysis and mitigation of JavaScript faults.

In particular, the prevalence of DOM-related faults suggests two important things. First, it indicates that programmers have a difficult time understanding the DOM, or at least keeping track of its many states and nodes. As discussed in Chapter 2, this is essentially a call-to-action for web developers and testers to have a more "DOM-aware" attitude when programming and testing. Furthermore, it is also a call-to-action for researchers trying to apply error prevention techniques to web applications, such as code completion and program comprehension. More specifically, such techniques must also be "DOM-aware", in the sense that they must give the JavaScript programmer a better grasp of the DOM state and state transitions. Indeed, others have begun to do work steering towards this direction, including Dompletion [19], which performs code completion for DOM API methods; Clematis [8], which is a program comprehension tool for web applications; and LED [21], which performs automatic synthesis of DOM element selectors.

Second, we believe that the prevalence of DOM-related faults is welcome news for researchers trying to develop analysis techniques for improving web application reliability. This is because the DOM is a well-structured piece of data, and this structure can be exploited to more easily understand how the JavaScript code interacts with the DOM, or debug a particular JavaScript fault. In fact, we used this observation as one of the driving principles behind VEJOVIS, which suggests fault repairs by modifying the erroneous DOM method parameter to match, not some specifications provided by the user, but rather, the current DOM's structure. This observation can also be useful for other analysis tasks, including code smell detection and test case generation, among others.

There has been considerable work done on statically analyzing JavaScript code to check for errors and code smells based on syntax. Hence, many of the tools adopted in practice that perform JavaScript analysis – including JSLint [43] and Closure [57] – are only concerned with deviations from syntax, without looking at the logic behind the program. On the other hand, our detection, localization, and repair tools look precisely at the *semantics* of the program, and try to understand these semantics. We therefore hope that the techniques we propose in this dissertation will inspire others to improve the state of JavaScript tooling, by not relegating themselves to syntax analysis, but also paying attention to JavaScript semantics.

The impact of this dissertation also goes beyond JavaScript and web applications. In Chapter 6 in particular, we explored cross-language interactions, and demonstrated how it can be possible to analyze them and, in this case, to detect inconsistencies that take place as a result of these interactions. This is an important contribution because cross-language programming – also known as heterogeneous or polyglot programming – occurs in a wide range of applications relevant to the modern world, and is applied very frequently in industrial projects [170]. For example, native Android apps are typically implemented using Java (for the activities) and XML (for the manifest file, layout, and other UI components); apps for other mobile devices are implemented in a similar fashion. In addition, cross-language interactions also occur in applications written for the cloud, as well as Internet of Things (IoT) applications [31]. Therefore, the idea behind HOLOCRON can be repurposed to also apply to these other systems.

Furthermore, while the identification of DOM-related faults as a common fault pattern is a result specific to web applications, at its core, it points to the difficulty that software programs have in interacting with their environment. For example, it is very common for programs to interact with the file system. Unlike the DOM, the file system is not written and designed specifically for the needs of the program being written; as a result, file accesses are often restricted to very specific folders whose internal structure is not as complex as that of the DOM. Nonetheless, the interaction of programs with the file system shares many characteristics with the interaction of the JavaScript code with the DOM, such as the retrieval of files using filenames to read and/or manipulate their contents, as well as the dynamic nature of the file system that results from these interactions. Hence, our knowledge regarding DOM-related faults could be used as a starting point towards understanding bugs in *other* software applications that result from their interaction with their environment.

Lastly, the dissertation also sheds light on how to properly conduct research related to reliability issues in software applications. In particular, we have demonstrated the value of conducting an empirical study first, instead of jumping to conclusions about the importance of a particular class of bugs. Conducting such a study is useful, for two reasons. First, it allows the researcher to demonstrate the importance of a particular class of bugs, which is done by establishing their

prevalence and *impact*. Second, the study allows the researcher to gain valuable insights about the *characteristics* of these bugs – including their root causes, propagation characteristics, and failure characteristics – which puts the researcher in a better position to design techniques for mitigating them. We used this approach in this dissertation, and it allowed us not only to gain better understanding of certain classes of bugs – in particular, DOM-related faults and cross-language inconsistencies – but also to create tools that make use of this newfound understanding.

7.2 Future Work

There are several ways to extend the work carried out in this dissertation, including the following.

Tool Extensions. Even though our tools were implemented primarily with client-side JavaScript in mind, many of the techniques we introduce can potentially be extended to work for other programming languages. Nevertheless, there are some challenges that can be addressed in the future in order to achieve full compatibility with other languages. For example, while the subtree pattern matching technique used in HOLOCRON can work for any abstract syntax tree (AST), a general-purpose version of HOLOCRON that supports arbitrary programming languages will need to perform syntax inference in order to generate the correct AST. Additionally, while VEJOVIS in principle can be applied to other event-driven programming languages that interact with an external entity similar to the DOM (e.g., Java code used in Android mobile apps), the type of information collected to infer the symptoms will differ depending on the features of the language.

Our fault detection tools (AUREBESH and HOLOCRON) can also be extended to work for other types of JavaScript frameworks apart from MVC. However, doing so can be challenging. In particular, one of the benefits of focusing on MVC frameworks is that it allows us to forego dynamic analysis, because these MVC frameworks rely on static bindings between the JavaScript code and an HTML template, instead of interacting directly with the DOM. On the other hand, non-MVC frameworks typically rely on direct interactions with the DOM; as a result, whenever the state of the DOM changes, the consistency relationship between the JavaScript code and the DOM also changes, which means dynamic analysis is re-

quired to keep track of these changes. Determining how to conduct this dynamic analysis is an interesting avenue for future work.

Finally, implementing our techniques as IDE extensions may also be worthwhile from the standpoint of usability. We have already started heading towards this direction, with AUTOFLOX being implemented as an Eclipse plugin (Section 3.5), and HOLOCRON being implemented as a Brackets plugin (Section 6.4). In the same way, VEJOVIS, which is already written in Java, can be extended as an Eclipse plugin, and AUREBESH, which is already written in JavaScript, can be refactored as a Node.js application to work in the Brackets IDE.

DOM Analysis. One of the recurring issues we had to deal with when designing our fault detection, localization, and repair tools was determining how to statically analyze code in the presence of a dynamic DOM. In many ways, this problem resembles traditional pointer analysis, in that it requires the code analysis tool to keep track of multiple references to the same object – in this case, a DOM element. However, analyzing the DOM can also be more challenging because unlike traditional pointers, the initial state of the DOM (i.e., the HTML code) is developed separately from the JavaScript code. As a result, while analysis of C code, for example, suffices in understanding pointers, analysis of JavaScript code does not suffice in understanding the DOM, and an accurate model of this external DOM object is required. Therefore, this calls on researchers to try to attain a stronger understanding of *DOM analysis* techniques, and explore various ways to conduct such an analysis.

JavaScript Framework Analysis. In order to address issues with JavaScript programming, many developers either create new frameworks/libraries (e.g., jQuery, AngularJS) or add new features to JavaScript by creating a superset of the language (e.g., TypeScript, Dart). For example, jQuery was written, in part, to simplify the process of making JavaScript code cross-browser compatible. In addition, TypeScript was created to help developers keep track of and enforce variable types. Therefore, the choice of a JavaScript framework or a JavaScript-based technology fundamentally affects the way that a developer writes code, which, in turn, potentially affects the reliability of the program. In that regard, it would be helpful to conduct a thorough study of JavaScript frameworks from the standpoint of re-

liability. Such a study entails determining what features recurrently degrade the program's reliability, and what features potentially enhance it. As a starting point, this study can be coupled with the results of our bug report study in Chapter 2, where the results can be used to create hypotheses (e.g., are frameworks that involve heavy cross-language interactions buggier than those that do not?).

On a related note, it would also be interesting to explore some of the ways that we can create new frameworks with the goal of improving reliability. This process will be simplified by conducting the empirical study on existing JavaScript frameworks suggested above, as the study would enable us to decide what properties our new framework would have or would not have. Nonetheless, there is a property that already stands out based on the results of this dissertation; in particular, it would be useful to design a framework that minimizes the number of cross-language interactions set up by the programmer, or at least, makes such interactions more manageable to the programmer. To some extent, MVC frameworks have succeeded in addressing part of this issue by abstracting out the DOM; however, they are still susceptible to a large number of cross-language inconsistencies, as we demonstrated in Chapters 5 and 6.

Other Application Types. The programming processes for many new application types resemble client-side web application programming. For instance, as mentioned in Section 7.1, mobile, cloud, and IoT applications involve cross-language interactions. Even without deviating from the realm of web applications, *server-side* programming is also beginning to resemble client-side programming, especially with the increase in popularity of Node.js. Hence, it is reasonable to look for ways to apply some of our techniques in this dissertation to these other domains. For example, many IoT applications also follow an event-driven execution model [93], so our method for stitching together separate asynchronous execution traces can be used to localize faults that occur in them, similar to what AUTOFLOX does. In addition, as mentioned in Section 7.1, the file system APIs used in these applications are similar to the DOM API in client-side JavaScript. Therefore, VEJOVIS' technique of suggesting repairs by comparing erroneous DOM API parameters to the DOM can be applied to these other applications; the difference, of course, is that the repairs are suggested by comparing erroneous parameters to the *file system* instead of the DOM.

Despite the above similarities, there are also differences in the way these other applications are programmed. In server-side JavaScript applications, for example, there is no DOM involved, but there is greater emphasis on JavaScript file imports, data validation, and database accesses. Further, cloud and IoT applications can run in a multi-threaded environment, and focus on distributed computing. It would therefore be interesting to see what types of bugs occur in these other applications, and if these bug categories differ significantly from those found in client-side JavaScript programs.

Bibliography

[1] R. Abreu, P. Zoeteweij, and A. Gemund. Spectrum-based multiple fault localization. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 88–99. IEEE Computer Society, 2009. → pages 50

[2] Adobe. JSEclipse, 2016. http://labs.adobe.com/technologies/jseclipse/ (Accessed: May 27, 2016). → pages 4

[3] Adobe Systems. Brackets, 2015. http://www.brackets.io (Accessed: May 16, 2015). → pages 4, 157

[4] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *Proc. Intl. Symposium on Software Reliability Engineering*, pages 143–151. IEEE Computer Society, 1995. → pages 50, 80

[5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 487–499. Morgan Kaufmann Publishers Inc., 1994. → pages 157

[6] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 207–216. ACM, 1993. → pages 157

[7] Ajax.org and Mozilla. Ace, 2015. http://ace.c9.io/ (Accessed: May 16, 2015). → pages 131

[8] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding javascript event-based interactions. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 367–377. ACM, 2014. → pages 79, 171

[9] E. Andreasen and A. Møller. Determinacy in static analysis for jquery. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 17–31. ACM, 2014. → pages 5

[10] S. Andrica and G. Candea. WaRR: High Fidelity Web Application Recording and Replaying. In *Proc. Intl. Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2011. → pages 4, 67, 79

[11] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proc. Intl. Conference on Software Engineering (ICSE)*, pages 361–370. ACM, 2006. → pages 22

[12] Aptana Inc. Aptana Studio, 2013. http://www.aptana.com (Accessed: October 10, 2013). → pages 4

[13] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE World Congress on Evolutionary Computation (CEC)*, pages 162–168, 2008. → pages 111

[14] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *Proc. Intl. Conference on Software Engineering*, pages 265–274. ACM, 2010. → pages 4, 80

[15] S. Artzi, J. Dolby, S. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 571–580. ACM, 2011. → pages 4, 49, 79, 143

[16] J. Ashkenas. BackboneJS, 2015. http://www.backbonejs.org (Accessed: May 16, 2015). → pages 113

[17] J. Ashkenas. BackboneJS: Tutorials, blog posts and example sites, 2016. https://github.com/jashkenas/backbone/wiki/Tutorials, -blog-posts-and-example-sites (Accessed: April 29, 2016). → pages 158, 159

[18] S. Bae, H. Cho, I. Lim, and S. Ryu. SAFEWAPI: web API misuse detector for web applications. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 507–517. ACM, 2014. → pages 47, 78

[19] K. Bajaj, K. Pattabiraman, and A. Mesbah. Dompletion: Dom-aware javascript code completion. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 43–54. ACM, 2014. → pages 4, 171

[20] K. Bajaj, K. Pattabiraman, and A. Mesbah. Mining questions asked by web developers. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 112–121. ACM, 2014. → pages 2, 36, 47

[21] K. Bajaj, K. Pattabiraman, and A. Mesbah. Led: Tool for synthesizing web element locators. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 848–851. IEEE Computer Society, 2015. → pages 171

[22] L. Bak and K. Lund. Dart, 2015. https://www.dartlang.org/ (Accessed: May 16, 2015). → pages 10, 22

[23] V. Balasubramanee, C. Wimalasena, R. Singh, and M. Pierce. Twitter bootstrap and AngularJS: Frontend frameworks to expedite science gateway development. In *Proceedings of the International Conference on Cluster Computing (CLUSTER)*, page 1. IEEE Computer Society, 2013. → pages 115, 140

[24] A. Bandyopadhyay and S. Ghosh. Tester feedback driven fault localization. In *Proceedings of International Conference on Software Testing, Verification and Validation (ICST)*, pages 41–50. IEEE Computer Society, 2012. → pages 79

[25] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 368–377. IEEE Computer Society, 1998. → pages 150

[26] Berico. Damerau-Levenshtein Java implementation, 2014. http://www.gettingcirrius.com/2011/06/calculating-similarity-part-3-damerau.html (Accessed: June 5, 2014). → pages 104

[27] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 308–318. ACM, 2008. → pages 10

[28] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In *Proc. Intl. Conference on Software Eng. (ICSE)*, pages 419–429. IEEE Computer Society, 2012. → pages 21

[29] B. Braun, P. Gemein, H. P. Reiser, and J. Posegga. Control-flow integrity in web applications. In *Proc. of Intl. Symp. on Engineering Secure Software and Systems (ESSOS)*, pages 1–16. Springer, 2013. → pages 3, 46

[30] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, pages 473–484. ACM, 2013. → pages 4, 79

[31] R. Buyya and A. V. Dastjerdi. *Internet of Things: Principles and Paradigms*, chapter Polyglot Programming. Elsevier, 2016. → pages 172

[32] A. Carzaniga, A. Gorla, and M. Pezzè. Healing web applications through automatic workarounds. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(6):493–502, 2008. → pages 4

[33] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 237–246. ACM, 2010. → pages 46, 83, 111

[34] S. Chandra and P. M. Chen. Whither generic recovery from application faults? a fault study using open-source software. In *Proc. Intl. Conference on Dependable Systems and Networks (DSN)*, pages 97–106. IEEE Computer Society, 2000. → pages 31, 46

[35] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proc. Intl. Conference on Dependable Systems and Networks (DSN)*, pages 595–604. IEEE Computer Society, 2002. doi:http://doi.ieeecomputersociety.org/10.1109/DSN.2002.1029005. → pages 79

[36] S. R. Choudhary, M. R. Prasad, and A. Orso. Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 171–180. IEEE Computer Society, 2012. → pages 36

[37] S. R. Choudhary, M. R. Prasad, and A. Orso. X-PERT: Accurate identification of cross-browser issues in web applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 702–711. IEEE Computer Society, 2013. → pages 36

[38] M. Cinque, D. Cotroneo, Z. Kalbarczyk, and R. Iyer. How do mobile phones fail? a failure data analysis of symbian os smart phones. In *Proc. Intl. Conference on Dependable Systems and Networks (DSN)*, pages 585–594. IEEE Computer Society, 2007. → pages 46

[39] H. Cleve and A. Zeller. Locating causes of program failures. In *Proc. Intl. Conference on Software Engineering (ICSE)*, pages 342–351. ACM, 2005. → pages 50

[40] D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008. ISBN 0596517742. → pages 2

[41] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *Proc. Intl. Conference on Software Engineering (ICSE)*, pages 176–185. ACM, 2005. → pages 111

[42] K. Dobolyi and W. Weimer. Modeling consumer-perceived web application fault severities for testing. In *Proc. Intl. Symposium on Software Testing and Analysis (ISSTA)*, pages 97–106. ACM, 2010. doi:http://doi.acm.org/10.1145/1831708.1831720. → pages 80

[43] Douglas Crockford. JSLint, 2012. http://www.jslint.com (Accessed: April 18, 2012). → pages 4, 143, 171

[44] Eclipse Foundation. Eclipse, 2012. http://www.eclipse.org (Accessed: April 18, 2012). → pages 4, 67

[45] B. Eich. ECMAScript documentation, 2016. http://www.ecmascript.org/docs.php (Accessed: April 29, 2016). → pages 35

[46] S. Elbaum, G. Rothermel, S. Karre, M. Fisher, et al. Leveraging user-session data to support web application testing. *Transactions on Software Engineering (TSE)*, 31(3):187–202, 2005. → pages 46

[47] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In *Proc. Intl. Conference on Automated Software Engineering (ASE)*, pages 64–73. ACM, 2007. → pages 111

[48] EmberSherpa. Open source Ember apps, 2016. https://github.com/EmberSherpa/open-source-ember-apps (Accessed: April 29, 2016). → pages 158, 159

[49] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 57–72. ACM, 2001. → pages 143, 167

[50] M. Erfani Joorabchi, M. Mirzaaghaei, and A. Mesbah. Works for me! Characterizing non-reproducible bug reports. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 62–71. ACM, 2014. → pages 46

[51] Facebook. Flow: a static type checker for JavaScript, 2016. http://flowtype.org/ (Accessed: April 29, 2016). → pages 45

[52] A. Feldthaus and A. Møller. Checking correctness of TypeScript interfaces for JavaScript libraries. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Language and Applications (OOPSLA)*. ACM, 2014. → pages 140

[53] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers. A limit study of JavaScript parallelism. In *Proc. Intl. Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE Computer Society, 2010. → pages 3, 47

[54] J. Fujima. Building a meme media platform with a JavaScript MVC framework and HTML5. *Webble Technology*, pages 79–89, 2013. → pages 114, 140

[55] A. Gizas, S. Christodoulou, and T. Papatheodorou. Comparative evaluation of javascript frameworks. In *Proceedings of the International Conference Companion on World Wide Web (WWW Companion)*, pages 513–514. ACM, 2012. → pages 140

[56] L. Gong, M. Pradel, M. Sridharan, and K. Sen. DLint: Dynamically checking bad coding practices in JavaScript. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2015. → pages 167

[57] Google. Closure Compiler, 2013. http://developers.google.com/closure/ (Accessed: October 10, 2013). → pages 4, 171

[58] Google. AngularJS, 2015. http://www.angularjs.org (Accessed: May 16, 2015). → pages 6, 113, 114

[59] Google. What is AngularJS?, 2015. https://docs.angularjs.org/guide/introduction (Accessed: May 16, 2015). → pages 127

[60] Google. Built with AngularJS, 2015. https://github.com/angular/builtwith.angularjs.org/blob/master/projects/projects.json (Accessed: May 16, 2015). → pages 131, 158, 159

[61] K. Goseva-Popstojanova, S. Mazimdar, and A. D. Singh. Empirical study of session-based workload and reliability for web servers. In *Proc. Intl. Symp. on Softw. Reliability Eng. (ISSRE)*, pages 403–414. IEEE Computer Society, 2004. → pages 1, 3, 46

[62] D. Graziotin and P. Abrahamsson. Making sense out of a jungle of JavaScript frameworks. *Product-Focused Software Process Improvement (PROFES)*, pages 334–337, 2013. → pages 140

[63] F. Groeneveld, A. Mesbah, and A. van Deursen. Automatic invariant detection in dynamic web applications. Technical Report TUD-SERG-2010-037, Delft University of Technology, 2010. → pages 68

[64] S. Guarnieri and B. Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for JavaScript code. In *Proc. Conference on USENIX Security Symposium (SSYM)*, pages 151–168. ACM, 2009. → pages 3, 16, 78

[65] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for AJAX intrusion detection. In *Proc. Intl. Conference on World Wide Web (WWW)*, pages 561–570. ACM, 2009. doi:http://doi.acm.org/10.1145/1526709.1526785. → pages 16, 78

[66] B. Hackett and S.-y. Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 239–250. ACM, 2012. → pages 45

[67] S. Hallé, T. Ettema, C. Bunch, and T. Bultan. Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 235–244. ACM, 2010. → pages 140

[68] Q. Hanam, F. Brito, and A. Mesbah. Discovering bug patterns in JavaScript. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, page 11 pages. ACM, ACM, 2016. → pages 46

[69] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 291–301. ACM, 2002. → pages 167

[70] J. Hewitt. Firebug, 2016. http://getfirebug.com (Accessed: April 29, 2016). → pages 79

[71] A. Hidayat. Esprima, 2015. http://www.esprima.org/ (Accessed: May 16, 2015). → pages 158

[72] Y. Hongping, S. Jiangping, and Z. Xiaorui. The update version development of "wiki message linking" system-integrated Ajax with MVC model. In *Proceedings of the International Forum on Computer Science-Technology and Applications (IFCSTA)*, pages 209–212. IEEE Computer Society, 2009. → pages 114, 140

[73] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 132–136. ACM, 2004. → pages 167

[74] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy. Using web corpus statistics for program analysis. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 49–65. ACM, 2014. → pages 167

[75] J. Huggins. Selenium, 2013. http://seleniumhq.org (Accessed: October 10, 2013). → pages 67

[76] Internet World Stats. Internet growth statistics, 2016. http://www.internetworldstats.com/emarketing.htm (Accessed: May 22, 2016). → pages 1

[77] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *ACM Conf. on Comp. and Communications Security (CCS)*, pages 270–283. ACM, 2010. doi:http://doi.acm.org/10.1145/1866307.1866339. → pages 3, 47

[78] S. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. *Proceedings of the International Static Analysis Symposium (SAS)*, pages 238–255, 2009. → pages 44, 45, 143

[79] 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 Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 59–69. ACM, 2011. → pages 16, 78, 143

[80] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *Proc. Intl. Symposium on Software Testing and Analysis (ISSTA)*, pages 34–44. ACM, 2012. → pages 83, 112

[81] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 55–64. ACM, 2007. → pages 167

[82] jindw. XMLDOM, 2016. https://www.github.com/jindw/xmldom (Accessed: April 29, 2016). → pages 158

[83] jjask. Scope variable not accessible (undefined) - AngularJS, 2014. http://www.jjask.com/554996/scope-variable-not-accessible-undefined-angularjs (Accessed: May 16, 2015). → pages 116

[84] J. Jones and M. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proc. Intl. Conference on Automated Software Engineering (ASE)*, pages 273–282. ACM, 2005. → pages 4, 49, 50, 79

[85] M. Kalyanakrishnan, R. Iyer, and J. Patel. Reliability of internet hosts: a case study from the end user's perspective. *Computer Networks*, 31(1-2): 47–57, 1999. ISSN 1389-1286. → pages 1, 3, 46

[86] K. Kambona, E. G. Boix, and W. De Meuter. An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the Workshop on Dynamic Languages and Applications (DYLA)*, pages 15–23. ACM, 2013. → pages 115, 140

[87] Y. Katz. EmberJS, 2015. http://www.emberjs.com (Accessed: May 16, 2015). → pages 113

[88] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *Proc. of the Intl. Symposium on Software Testing and Analysis (ISSTA)*, pages 105–116. ACM, 2009. → pages 104

[89] S. Kim and E. J. Whitehead Jr. How long did it take to fix bugs? In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, pages 173–174. ACM, 2006. → pages 42

[90] E. Koshelko. Why you should not use AngularJS, 2015. https://medium.com/@mnemon1ck/why-you-should-not-use-angularjs-1df5ddf6fc99 (Accessed: May 16, 2015). → pages 114

[91] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Program (JOOP)*, 1(3):26–49, 1988. → pages 140

[92] lagos. Autopager jQuery extension, 2014. http://code.google.com/p/jquery-autopager/ (Accessed: June 5, 2014). → pages 85

[93] L. Lan, B. Wang, L. Zhang, R. Shi, and F. Li. An event-driven service-oriented architecture for the internet of things service execution. *International Journal of Online Engineering (iJOE)*, 11(2):4–8, 2015. → pages 175

[94] A. Leff and J. T. Rayfield. Web-application development using the model/view/controller design pattern. In *Proceedings of the International Enterprise Distributed Object Computing Conference (EDOC)*, pages 118–127. IEEE Computer Society, 2001. → pages 140

[95] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 306–315. ACM, 2005. → pages 167

[96] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *Transactions on Software Engineering (TSE)*, 32(3):176–192, 2006. → pages 167

[97] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 25–33. ACM, 2006. doi:http://doi.acm.org/10.1145/1181309.1181314. → pages 46

[98] Y. Liao, Z. Zhang, and Y. Yang. Web applications based on AJAX technology and its framework. In *Proceedings of the International Conference on Communications and Information Processing (ICCIP)*, pages 320–326. Springer, 2012. → pages 140

[99] E. Lielmanis. JSBeautifier, 2014. http://www.jsbeautifier.org/ (Accessed: June 5, 2014). → pages 67

[100] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of javascript applications in the presence of frameworks and libraries. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 499–509. ACM, 2013. → pages 4, 5

[101] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of AJAX web applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 121–130. IEEE Computer Society, 2008. → pages 49, 143

[102] A. Marchetto, F. Ricca, and P. Tonella. An empirical validation of a web fault taxonomy and its usage for web testing. *Journal of Web Engineering (JWE)*, 8(4):316–345, 2009. → pages 46

[103] MarionetteJS. Backbone Marionette, 2016. https://github.com/marionettejs/backbone.marionette (Accessed: April 29, 2016). → pages 146

[104] L. Marks, Y. Zou, and A. E. Hassan. Studying the fix-time for bugs in large open source projects. In *Proc. Intl. Conf. on Predictive Models in Softw. Eng. (PROMISE)*, page 11. ACM, 2011. → pages 22

[105] J. Martinsen, H. Grahn, and A. Isberg. A comparative evaluation of JavaScript execution behavior. In *Proc. of Intl. Conf. on Web Engineering (ICWE)*, pages 399–402. Springer, 2011. → pages 47

[106] P. Mayer and A. Schroeder. Cross-language code analysis and refactoring. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 94–103. IEEE Computer Society, 2012. → pages 168

[107] F. Meawad, G. Richards, F. Morandat, and J. Vitek. Eval begone!: semi-automated removal of eval from JavaScript programs. In *Proc. of the ACM Intl. Conference on Object Oriented Programming Systems*

Languages and Applications (OOPSLA), pages 607–620. ACM, 2012. → pages 112

[108] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 561–570. ACM, 2011. → pages 36

[109] A. Mesbah and A. van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 210–220. IEEE Computer Society, 2009. → pages 4, 49, 79, 143

[110] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. Web (TWEB)*, 6(1):3:1–3:30, 2012. → pages 67, 104

[111] J. Mickens, J. Elson, and J. Howell. Mugshot: deterministic capture and replay for JavaScript applications. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 11–11. ACM, 2010. → pages 4, 67, 79

[112] Microsoft. TypeScript, 2015. http://www.typescriptlang.org/ (Accessed: May 16, 2015). → pages 10, 22

[113] S. Mirshokraie and A. Mesbah. JSART: JavaScript assertion-based regression testing. In *Proc. Intl. Conference on Web Engineering (ICWE)*, pages 238–252. Springer, 2012. → pages 4, 79

[114] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Efficient JavaScript mutation testing. In *Proc. Intl. Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2013. → pages 21, 79

[115] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. JSeft: Automated JavaScript unit test generation. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2015. → pages 4, 79

[116] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Guided mutation testing for JavaScript web applications. *Transactions on Software Engineering (TSE)*, 41(5):429–444, 2015. → pages 143

[117] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *Proceedings of International Conference*

on Software Testing, Verification and Validation (ICST), pages 153–162. IEEE Computer Society, 2014. → pages 79

[118] R. Morales-Chaparro, M. Linaje, J. Preciado, and F. Sánchez-Figueroa. MVC web design patterns and rich internet applications. *Proceedings of the Jornadas de Ingenieria del Software y Bases de Datos*, pages 39–46, 2007. → pages 140

[119] Mozilla. Rhino, 2012. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino (Accessed: April 18, 2012). → pages 67

[120] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Clone management for evolving software. *Transactions on Software Engineering (TSE)*, 38(5):1008–1026, 2012. → pages 150

[121] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Auto-locating and fix-propagating for HTML validation errors to PHP server-side code. In *Proc. Intl. Conference on Automated Software Engineering (ASE)*, pages 13–22. IEEE Computer Society, 2011. → pages 83

[122] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Cross-language program slicing for dynamic web applications. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 369–380. ACM, 2015. → pages 168

[123] J. Nijjar and T. Bultan. Bounded verification of Ruby on Rails data models. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 67–77. ACM, 2011. → pages 140

[124] 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 *Proc. of the Conf. on Computer and Communications Security (CCS)*. ACM, 2012. → pages 3, 47

[125] F. Ocariza. Projects, 2012. http://ece.ubc.ca/~frolino/projects/ (Accessed: April 18, 2012). → pages 170

[126] F. Ocariza. Vejovis, 2014. http://ece.ubc.ca/~frolino/projects/vejovis/ (Accessed: June 5, 2014). → pages 104, 105, 110

[127] F. Ocariza. Aurebesh, 2015. http://ece.ubc.ca/~frolino/projects/aurebesh/ (Accessed: May 16, 2015). → pages 131

[128] F. Ocariza, K. Pattabiraman, and B. Zorn. JavaScript errors in the wild: an empirical study. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–109. IEEE Computer Society, 2011. → pages 3, 10, 26, 47, 50, 86, 140, 142

[129] F. Ocariza, K. Pattabiraman, and A. Mesbah. AutoFLox: an automatic fault localizer for client-side JavaScript. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 31–40. IEEE Computer Society, 2012. → pages iii, 7, 26, 49, 83

[130] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. An empirical study of client-side JavaScript bugs. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 55–64. IEEE Computer Society, 2013. → pages iii, 7, 9, 18, 50, 143

[131] F. Ocariza, K. Pattabiraman, and A. Mesbah. Vejovis: suggesting fixes for JavaScript faults. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 837–847. ACM, 2014. → pages iv, 7, 80, 82

[132] F. Ocariza, K. Pattabiraman, and A. Mesbah. Detecting inconsistencies in JavaScript MVC applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2015. → pages iv, 7, 113, 143

[133] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. A study of causes and consequences of client-side JavaScript bugs. *Transactions on Software Engineering (TSE)*, page 17 pages, 2016. → pages iii, 8, 9

[134] F. Ocariza, G. Li, K. Pattabiraman, and A. Mesbah. Automatic fault localization for client-side JavaScript. *Software Testing, Verification and Reliability (STVR)*, 26(1):69–88, 2016. → pages iii, 8, 44, 49, 94

[135] S. O'Grady. The RedMonk Programming Language Rankings: January 2016, 2016. http://redmonk.com/sogrady/2016/02/19/language-rankings-1-16/ (Accessed: May 22, 2016). → pages 2

[136] V. N. Padmanabhan, S. Ramabhadran, S. Agarwal, and J. Padhye. A study of end-to-end web access failures. In *Proc. Intl. Conf. on Emerging*

Networking Experiments and Technologies (CoNEXT), pages 15:1–15:13. ACM, 2006. → pages 3, 46

[137] K. Pattabiraman and B. Zorn. DoDOM: leveraging DOM invariants for web 2.0 application robustness testing. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 191–200. IEEE Computer Society, 2010. → pages 4, 49, 79, 143

[138] S. Pertet and P. Narasimhan. Causes of failure in web applications. Technical Report CMU-PDL-05-109, Parallel Data Laboratory, Carnegie Mellon University, 2005. → pages 3, 46

[139] R.-H. Pfeiffer and A. Wasowski. Taming the confusion of languages. In *Proceedings of the European Conference on Modelling Foundations and Applications (ECMFA)*, pages 312–328. Springer, 2011. → pages 167

[140] T. Polychniatis, J. Hage, S. Jansen, E. Bouwers, and J. Visser. Detecting cross-language dependencies generically. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 349–352. IEEE, 2013. → pages 167

[141] M. Pradel, P. Schuh, and K. Sen. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 314–324. IEEE Computer Society, 2015. → pages 15, 23, 47

[142] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn. JSMeter: Measuring JavaScript behavior in the wild. In *Proceedings of the USENIX Conference on Web Application Development (WebApps)*, pages 1–12. ACM, 2010. → pages 3, 46, 140

[143] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Proc. Intl. Conference on Automated Software Engineering (ASE)*, pages 30–39. IEEE Computer Society, 2003. → pages 79

[144] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM, 2010. doi:http://doi.acm.org/10.1145/1806596.1806598. → pages 3, 47, 140

[145] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In

Proceedings of the European Conference on Object-Oriented Programming (ECOOP), pages 52–78. Springer, 2011. → pages 3, 55, 143

[146] C. Robinson. AngularJS: If you don't have a dot, you're doing it wrong!, 2013. http://zcourts.com/2013/05/31/angularjs-if-you-dont-have-a-dot-youre-doing-it-wrong/ (Accessed: May 16, 2015). → pages 116

[147] V. Y. Rosales-Morales, G. Alor-Hernández, and U. Juárez-Martínez. An overview of multimedia support into JavaScript-based frameworks for developing rias. In *Proceedings of the International Conference on Electrical Communications and Computers (CONIELECOMP)*, pages 66–70. IEEE Computer Society, 2011. → pages 140

[148] H. Samimi, M. Schafer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *Proc. Intl. Conference on Software Engineering (ICSE)*, pages 277–287. IEEE Computer Society, 2012. → pages 4, 46, 80, 83, 111

[149] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Effective smart completion for JavaScript. Technical Report Technical Report RC25359, IBM Research, 2013. → pages 4

[150] N. Semenenko, M. Dumas, and T. Saar. Browserbite: Accurate cross-browser testing via machine learning over image features. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 528–531. IEEE Computer Society, 2013. → pages 36

[151] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 488–498. ACM, 2013. → pages 4, 79

[152] K. Sera. apriori.js, 2016. https://github.com/seratch/apriori.js (Accessed: April 29, 2016). → pages 158

[153] U. Shaked. AngularJS vs. BackboneJS vs. EmberJS, 2014. http://www.airpair.com/js/javascript-framework-comparison (Accessed: May 16, 2015). → pages 114, 116, 158

[154] G. Shu, B. Sun, A. Podgurski, and F. Cao. Mfl: Method-level fault localization with causal inference. In *Proceedings of International*

Conference on Software Testing, Verification and Validation (ICST), pages 124–133. IEEE Computer Society, 2013. → pages 79

[155] J. L. Singleton and G. T. Leavens. Verily: a web framework for creating more reasonable web applications. In *Companion Proceedings of the International Conference on Software Engineering (ICSE)*, pages 560–563. ACM, 2014. → pages 140

[156] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 253–262. ACM, 2005. → pages 46

[157] StackOverflow. StackOverflow Developer Survey 2015, 2015. http://stackoverflow.com/research/developer-survey-2015#tech (Accessed: May 16, 2015). → pages 2

[158] D. Strein, H. Kratz, and W. Löwe. Cross-language program analysis and refactoring. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 207–216. IEEE Computer Society, 2006. → pages 168

[159] A. Swartz. A brief history of Ajax, 2005. http://www.aaronsw.com/weblog/ajaxhistory (Accessed: May 22, 2016). → pages 2, 35, 41

[160] D. Synodinos. Top JavaScript MVC frameworks, 2013. http://www.infoq.com/research/top-javascript-mvc-frameworks (Accessed: May 16, 2015). → pages 144

[161] Synopsys. Coverity, 2016. http://www.coverity.com/ (Accessed: April 29, 2016). → pages 143

[162] B. Taraghi and M. Ebner. A simple MVC framework for widget development. In *Proceedings of the International Workshop on Mashup Personal Learning Environments (MUPPLE)*, pages 38–45. CEUR-WS, 2010. → pages 114, 140

[163] J. Tian, S. Rudraraju, and Z. Li. Evaluating web software reliability based on workload and failure data extracted from server logs. *IEEE Trans. Softw. Eng.*, 30:754–769, 2004. ISSN 0098-5589. doi:http://dx.doi.org/10.1109/TSE.2004.87. → pages 1, 3, 46

[164] Two Sigma. Beaker, 2016. https://github.com/twosigma/beaker-notebook (Accessed: April 29, 2016). → pages 146

[165] I. Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985. → pages 4, 49

[166] A. Vetro, F. Tomassetti, M. Torchiano, and M. Morisio. Language interaction and quality issues: an exploratory study. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 319–322. ACM, 2012. → pages 167

[167] W3C. Cascading style sheets level 2 revision 1 specification: Selectors, 2011. http://www.w3.org/TR/CSS2/selector.html (Accessed: June 5, 2014). → pages 85

[168] W3Techs. Usage statistics and market share of jQuery for websites, 2015. http://w3techs.com/technologies/details/js-jquery/all/all (Accessed: May 22, 2016). → pages 5

[169] W3Techs. Usage of JavaScript for websites, 2015. http://w3techs.com/technologies/details/cp-javascript/all/all (Accessed: May 16, 2015). → pages 2

[170] D. Wampler, T. Clark, N. Ford, and B. Goetz. Multiparadigm programming in industry: A discussion with neal ford and brian goetz. *IEEE Software*, 27(5):61, 2010. → pages 172

[171] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 35–44. ACM, 2007. → pages 167

[172] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proc. Intl. Symposium on Software Testing and Analysis (ISSTA)*, pages 61–72. ACM, 2010. → pages 111

[173] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 461–476. Springer, 2005. → pages 167

[174] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proc. Intl. Conference on Software Engineering (ICSE)*, pages 364–374. IEEE Computer Society, 2009. → pages 111

[175] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. An empirical analysis of XSS sanitization in web application frameworks. Technical Report EECS-2011-11, UC Berkeley, 2011. → pages 3, 47

[176] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, pages 1–8. IEEE Computer Society, 2007. → pages 42

[177] J. Wojciechowski, B. Sakowicz, K. Dura, and A. Napieralski. MVC model, struts framework and file upload issues in web applications based on J2EE platform. In *Proceedings of the International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET)*, pages 342–345. IEEE Computer Society, 2004. → pages 140

[178] A. Yildiz, B. Aktemur, and H. Sozer. Rumadai: A plug-in to record and replay client-side events of web sites with dynamic content. In *Proceedings of Workshop on Developing Tools as Plug-ins (TOPI)*, pages 88–89. IEEE Computer Society, 2012. → pages 4, 79

[179] Z. Yin, M. Caesar, and Y. Zhou. Towards understanding bugs in open source router software. *ACM SIGCOMM Computer Communication Review*, 40(3):34–40, 2010. ISSN 0146-4833. → pages 46

[180] C. Yue and H. Wang. Characterizing insecure JavaScript practices on the web. In *Proc. Intl. Conf. on World Wide Web (WWW)*, pages 961–970. ACM, 2009. doi:http://doi.acm.org/10.1145/1526709.1526838. → pages 3, 47

[181] A. Zagalsky. CSC485B: Startup Programming, 2015. https://github.com/alexeyza/startup-programming (Accessed: May 16, 2015). → pages 132

[182] N. C. Zakas. *Maintainable JavaScript*. O'Reilly Media, Inc., 2012. ISBN 1449327680. → pages 2

[183] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 765–784. ACM, 2013. → pages 79

[184] S. Zhang, H. Lü, and M. D. Ernst. Automatically repairing broken workflows for evolving gui applications. In *Proc. Intl. Symposium on Software Testing and Analysis (ISSTA)*, pages 45–55. ACM, 2013. → pages 111

[185] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *Proc. Intl. Symposium on Automated Analysis-Driven Debugging (AADEBUG)*, pages 33–42. ACM, 2005. → pages 80

[186] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proc. Intl. Conference on the World Wide Web (WWW)*, pages 805–814. ACM, 2011. → pages 16, 78

[187] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a z3-based string solver for web application analysis. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 114–124. ACM, 2013. → pages 112

[188] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE Computer Society, 2012. → pages 79

[189] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proc. Intl. Workshop on Predictor Models in Software Engineering (PROMISE)*, pages 9–9. IEEE Computer Society, 2007. → pages 46