A Tool to Reduce Defects due to Dependencies between HTML5, JavaScript and CSS3

by

Manit Singh Kalsi

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved June 2016 by the
Graduate Supervisory Committee:

Kevin Gary, Chair
Adam Doupe
Timothy Lindquist

ARIZONA STATE UNIVERSITY

August 2016

ABSTRACT

One of the most common errors developers make is to provide incorrect string identifiers across the HTML5-JavaScript-CSS3 stack. The existing literature shows that a significant percentage of defects observed in real-world codebases belong to this category. Existing work focuses on semantic static analysis, while this thesis attempts to tackle the challenges that can be solved using syntactic static analysis. This thesis proposes a tool for quickly identifying defects at the time of injection due to dependencies between HTML5, JavaScript, and CSS3, specifically in syntactic errors in string identifiers. The proposed solution reduces the delta (time) between defect injection and defect discovery with the use of a dedicated just-in-time syntactic string identifier resolution tool. The solution focuses on modeling the nature of syntactic dependencies across the stack, and providing a tool that helps developers discover such dependencies. This thesis reports on an empirical study of the tool usage by developers in a realistic scenario, with the focus on defect injection and defect discovery times of defects of this nature (syntactic errors in string identifiers) with and without the use of the proposed tool. Further, the tool was validated against a set of real-world codebases to analyze the significance of these defects.

For Mom, Dad and dearest Sister.

ACKNOWLEDGMENTS

I would like to express my thanks to Dr Kevin Gary, the chair of my committee. His unending support as an excellent advisor, critic and mentor made my work a success. I wholeheartedly thank him for his guidance and his time, without which this thesis would not have been possible. I would also like to thank the members of my committee Dr Timothy Lindquist and Dr Adam Doupé for their valuable support and time with this research.

I would also like to thank all the student developers in the HEAL lab who acted as beta testers for the tool. I also wholeheartedly thank all the students who spent their valuable time to be a part of my research study. I also thank my family and all my friends for their never ending support.

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER

1 INTRODUCTION 1

Modern Day Web Application 1

Syntactic Dependencies 3

Current Practices of Handling Syntactic Dependencies 6

Defects Due to Syntactic Dependencies 7

Research Questions 9

Thesis Overview 11

2 LITERATURE REVIEW 12

Static Analysis of Web Applications 13

JavaScript Analysis 15

CSS Analysis 19

Most Common Defects 21

Prior Work 24

Prior Validation 24

Experiment Setup 25

Protocol 26

Source Code 26

Environment 26

CHAPTER Page

Participants 27

Results 27

Observations 28

Discussion 31

3 DEPENDENCY AND ERROR MODELING 33

Dependency Model 33

HTML5 to HTML5 34

HTML5 to JavaScript 34

HTML5 to CSS3 35

JavaScript to HTML5 36

JavaScript to JavaScript 36

JavaScript to CSS3 36

CSS3 to HTML5 37

CSS3 to JavaScript and CSS3 37

Error Taxonomy 37

Contributions 40

4 IMPLEMENTATION 42

Parsing HTML5 44

Parsing CSS3 47

Parsing JavaScript 48

Integrating Parser Results 50

Output Verbosity 51

CHAPTER Page

Export Results 52

HTML Viewer 54

Rule-based Analysis 55

Recommendation for Fixes 56

Integrated Plugin Development for Eclipse 57

5 VALIDATION 58

Experiment 1: Testing Against Real World Codebases 58

Discussion 61

Experiment 2: User Study 62

Protocol 62

Environment 65

Participants 65

Results 66

HTML5toJS Defects 66

HTML5toCSS3 Defects 68

JStoHTML5 Defects 70

External File Dependency Defects 71

Observations 72

Aggregate Statistical Analysis 73

Limitations 74

Discussion 75

6 CONCLUSION AND FUTURE WORK 77

CHAPTER Page

REFERENCES..... 80

APPENDIX

A LIST OF FEATURES TO BE IMPLEMENTED IN USER STUDY 85

B CODEBASES CONSIDERED FOR RQ1 VALIDATION 90

C EXPERIMENT TWO: RAW DATA 92

LIST OF TABLES

Table		Page
1.	Dependency Model	34
2.	Experiment on Real World Codebases	60
3.	Average Delta Observed per Defect Type	74
4.	Codebases Considered for RQ1 Validation	91
5.	HTML5toJS Dependency Error Raw Data	93
6.	HTML5toCSS3 Dependency Error Raw Data	93
7.	JStoHTML5 Dependency Error Raw Data	94
8.	External File Dependency Error Raw Data	95

LIST OF FIGURES

Figure Page

1. HTML, JavaScript and CSS from Nov 15th, 2010 to Feb 1st, 2016 3
2. Data Comparison Between Nov 15th, 2010 and Feb 1st, 2016..... 3
3. Syntactic Dependencies Across the HTML5-JS-CSS3 Stack 4
4. Console Tab of Google Chrome Dev Tools..... 6
5. HTML DOM and CSS Style View of Firebug 7
6. User Study 1..... 29
7. User Study 2..... 29
8. User Study 1..... 30
9. User Study 2..... 30
10. Error Classification 38
11. Error Taxonomy..... 39
12. Flow of a Web Page Rendering on a Browser 43
13. Visual Overview of the Tool Process 44
14. Speed Comparisons of Popular JavaScript Engines 48
15. Plain Text Format Snippet Part 1 52
16. Plain Text Format Snippet Part 2 52
17. JSON Format Snippet 53
18. HTML Viewer 54
19. HTML Viewer 55
20. HTML5toJS Dependency Errors..... 67
21. HTML5toCSS3 Dependency Errors 69

Figure Page

22. JStoHTML5 Dependency Errors 70

23. External File Dependency Errors 72

CHAPTER 1

INTRODUCTION

In my research I seek to create models and tools to improve productivity of the modern web developer. The modern web developer repeatedly deals with syntactic dependencies in the DOM and render trees of the browser, and lacks the tool support to identify and resolve the dependencies rapidly. This method and tool is important as it addresses a common, prevalent issue for developers in a manner tied directly to productivity, thereby saving time and improving quality.

The following sections discuss the topics that will help the reader better understand the concepts behind the research problem and put them in a better position to get to the crux of the research. The discussion begins with modern day web applications and how the web has evolved, how this evolution has given rise to syntactic dependencies that are difficult to keep track of, how it impacts the developers and how are such defects significant. The final section will revisit the research questions and discuss them in detail.

1.1 MODERN DAY WEB APPLICATIONS

The web was built with the purpose of document sharing [1], with the server generating HTML pages with almost no JavaScript or CSS. This means that most of the content was static and the server was the main source of all the client side documents. The minimal JavaScript and CSS ensured very little interaction between the Document Object Model (DOM) and the scripts and stylesheets. As the web evolved, this trend began to change. The client side is no more a “thin-client”, instead, a lot of functionality is client driven.

Static HTML pages generated by a server gave way to dynamic web pages with a lot of user interaction, runtime DOM manipulation, and client-server interaction. Mesbah, et al. [2, pp. 210] have very aptly described modern day web application as ".. stateful asynchronous client/server communication, and client-side runtime manipulation of the DOM tree ..". Therefore, with the help of these characteristics, it is very clear to see that modern day web applications are front-end heavy, meaning more of HTML5, JavaScript, and CSS3, written by developers. Statistics have shown this claim to be true. httparchive.org [3] has analyzed almost half a million websites gathered solely based on the Alexa Top 1,000,000 sites [4]. As can be seen from Figure 1, over the past six years, the size of HTML5 has increased by almost 200%, the size of JavaScript has increased by almost 300% and the size of CSS has increased by almost 300%. The clearly shows how rapidly the size of HTML5, CSS3 and JavaScript is growing on the client side. It is also important to note here that this data is just for the past 5.5 years. If this trend continues, then we should expect a similar growth over the next 5 years or so. To help the user better understand the comparison, Figure 2 shows a side by side comparison of the data from Nov 15th, 2010 vs the data from Feb 1st, 2016.

Figure 1. HTML,JavaScript and CSS from Nov 15th,2010 to Feb 1st,2016 respectively[3]

Figure 2. Data comparison between Nov 15th, 2010 and Feb 1st, 2016 [3]

To summarize, this shows that the size of HTML5, JavaScript and CSS3 has been increasing. This means that the front end web developer has to deal with huge codebases that span across three different kinds of languages.

1.2 SYNTACTIC DEPENDENCIES

As a web developer, this shift towards the front end stack means that the developer has to write code in HTML5, CSS3, and JavaScript. These are three different languages with

their own characteristics; HTML5 is a markup language that is used to create the structure of the web pages; CSS3 is style-sheet language that is used to provide presentation to the HTML5 document; JavaScript is a dynamically typed interpreted language that provides dynamic nature to the otherwise static HTML5 documents. JavaScript gives the developer an ability to manipulate the underlying page structure (the DOM) at runtime.

While writing a front-end web application, a developer has to keep a track of the DOM in the HTML5, the associated JavaScript, and the associated CSS3 stylesheets as well. Because of this interplay, there are a lot of syntactic dependencies created. Figure 3 can help better illustrate the concept of syntactic dependencies.

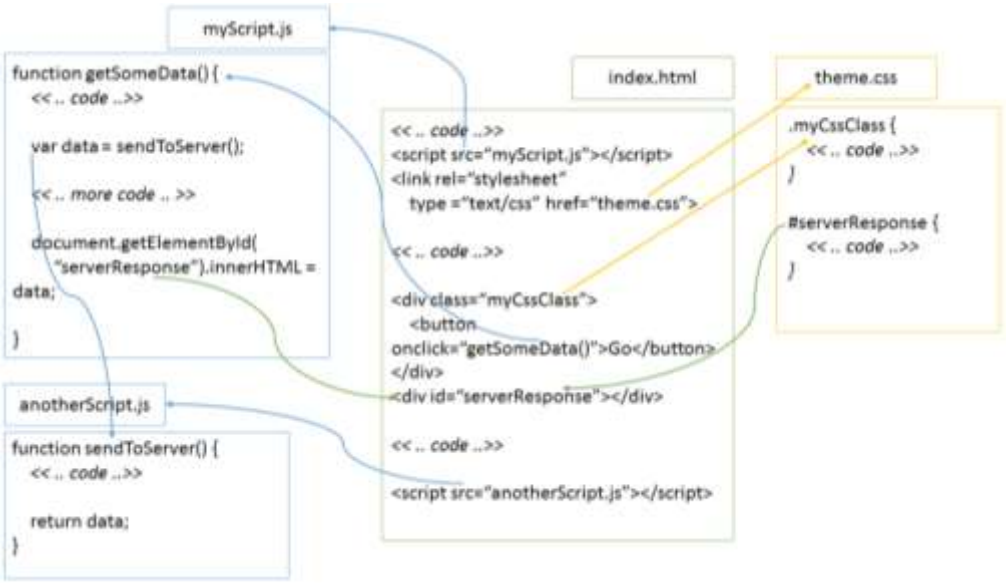


Figure 3. Syntactic Dependencies across the HTML5-JS-CSS3 stack[5, pp. 4]

Given the nature of these languages and the dependencies between them, modern day web applications are prone to having defects. And as more and more behavior is moving

to the front-end, errors in this technology stack are no longer cosmetic faults but significant defects that impact the correctness of the application. The developer has to keep track of how the DOM accesses the JavaScript and vice-versa, how the DOM accesses the CSS3 stylesheet and vice-versa, how the JavaScript accesses the CSS3 stylesheet and vice-versa. Figure 3 illustrates the concept at a very small scale. For example, the button in `index.html` has an `onclick` event that is handled by the `getSomeData()` method in the `myScript.js` file. Similarly, the `myScript.js` file attempts to manipulate the content of the DOM by accessing the `serverResponse` DOM element by its ID. The div that contains the button uses a style `myCssClass` as defined in the `theme.css` file. So even in about 20 lines of code, it is very easy to see the nature of these dependencies. And the developer can easily make an error resolving these dependencies. These errors can range from typographical errors to non-existent constructs like ID and functions, etc. It becomes very difficult for the developer to keep track of such dependencies as the size of the codebase increases.

From the statistics shown above in figures 1-2, it is clear that the size of the codebase is indeed very large, and the example of Figure 3 demonstrates that dependencies between the component technologies are pervasive. It is essential to note that in all of the modern web applications, these three languages work in parallel, making them prone to dependency related defects. Ocariza, et al. [6] found that most of such defects (specifically JavaScript) are injected by the programmers in the code itself. I will further explore the errors found in real-world codebases and how it maps to these dependencies in Chapter 2.

1.3 CURRENT PRACTICES OF HANDLING SYNTACTIC DEPENDENCIES

In most scenarios, such defects are not caught even during testing (unit and integration). Currently, the most common method used by developers to test such dependencies is to run the code in the browser, see the results and look for any error message in the console or any undesirable behavior in their application. The most common tools for this purpose is either the Google Chrome Dev tools (Figure 4) or Firefox Firebug (Figure 5). There are different components of these tools that help the developer map the dependencies and find the defects if any. In most cases, any JavaScript related defect is directly caught by looking at the “console” view. But CSS related defects can only be found through a visual inspection of either the functionality of the module or through a code inspection.

Figure 4. Console tab of Google Chrome Dev tools

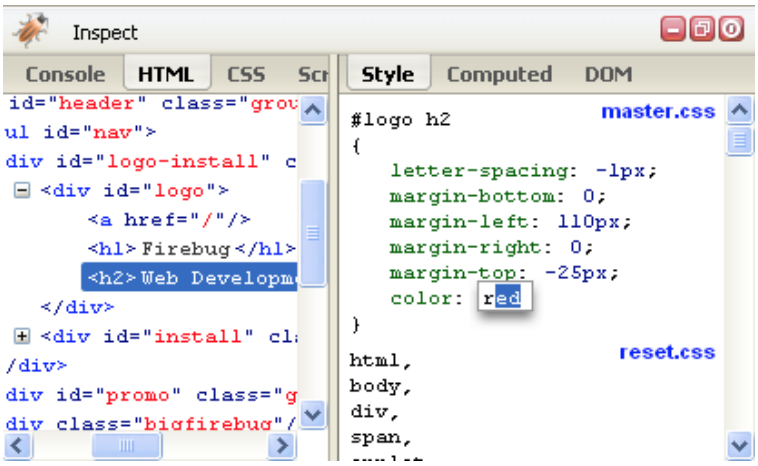


Figure 5. HTML DOM and CSS Style view of Firebug

This round trip is not only an overhead adding to the development time and cost, but is also very error-prone. When a developer injects a defect, the only way for them to discover this injection is by opening the web application on a browser, inspect the code by using either Google Chrome dev tools or Firefox Firebug, and then find the defect. Even the fix validation would require the developer to go through the same round trip. Also, because these languages do not include a "compile" step, this round trip is a way for the developers to look for syntax errors. This impacts developer efficiency a lot as the developer is expected to spend a significant amount of time during their development to go through this round trip technique of development.

1.4 DEFECTS DUE TO SYNTACTIC DEPENDENCIES

"A defect is an instance in which a requirement is not satisfied." [7, pp. 745] The dependencies across the HTML5-JavaScript-CSS3 stack make it highly prone to defects committed by developers. One of the most common errors developers make is to provide incorrect string identifiers across the HTML5-JavaScript-CSS3 stack. The existing

literature [6][8][10] shows that a significant percentage of defects observed in real-world code bases belong to this category. I further investigate the existing observations in Chapter 2. It is important here to note that such defects do exist and are caused by the dependencies across the stack.

The literature [9] shows that 80 percent of the defects are caused by 20 percent of the modules. In this case, when I talk about defects generated due to dependencies in the HTML5-JavaScript-CSS3 stack, the main causes of those defects can be traced back to the DOM. In fact, an empirical study of client-side JavaScript bugs has shown that 65% of the bugs are DOM related [6]. In another study, the authors have observed that DOM manipulation is one of the most common usages of JavaScript in modern web application and have recommended static analysis tool designers to consider the DOM as a component in their tools [10]. This is a clear indication that most of the defects are caused by interaction between the DOM and JavaScript. Although no similar studies have been found for DOM and CSS interactions, it is easy to extend these results and expect a similar behavior for DOM and CSS interactions.

When a developer introduces a defect into the code, this activity is termed as *defect injection*. And when the defect is found by either the same developer or some other developer/user, this activity is termed as *defect discovery*. I contend that the existing web developer toolset and practices make this delta between defect injection and defect discovery a relatively large amount of time. This is because of the round trip between the IDE/text editor to the browser's dev tools to inspect the console and/or the behavior of

the application to determine the existence of a defect. Even when changes are made, this round trip is a mandatory step to determine the status of the defect. This necessary evil adds to the delta between defect injection and defect discovery, thereby affecting developer productivity. Researchers recommend reducing this delta to as minimum as possible: “In order to eliminate defects from the product it is necessary to address their prevention, or detection and resolution as soon as possible after their injection during development and maintenance.” [7, pp. 746]. It is also recommended that such defects be found and fixed before delivery to avoid cost: “Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase.” [9]. This thesis focuses on defect injection and defect discovery as an in-phase activity rather than across phases. This is due the nature of front end web development. This micro optimization will help save many seconds per developer and when aggregated over the entire development team it will amount to a lot of valuable time.

1.5 RESEARCH QUESTIONS

The discussion and findings above are a strong motivation towards answering research problems surrounding developer productivity and the significance of syntactic dependency defects in modern day web applications. These motivations are discussed in detail in the literature review chapter. I now present the research questions that this research targets:

- RQ1: How significant are syntactic errors in string identifiers referencing DOM elements in the HTML5-JavaScript-CSS3 stack?

- RQ2: What is the delta (time or cost) between defect injection and defect discovery in new HTML5-JavaScript-CSS3 style applications?
- RQ3: Does a dedicated just-in-time syntactic string identifier resolution tool significantly reduce delta time/cost from RQ2 for a significant portion of real problems (RQ1)?

As a recap, I revisit these research questions in light of the concepts discussed in the previous sections in this chapter. Developers often commit the mistake of using incorrect string identifiers across the HTML5-JavaScript-CSS3 stack. For example, assume an "id" attribute declared for an HTML5 element is `foobar`. Developers commit the mistake of accessing it as `fooBar` or `Foobar`, etc. Also, as the code grows larger and larger, with new changes being added, it is difficult to keep track of such identifiers and these errors become significant. RQ1 attempts to identify the significance of such errors, defined as the severity of the defect on system behavior.

There are different development environments and toolchains used by developers for front-end applications which impact the delta between defect injection and defect discovery. The typical developer uses Chrome Developer tools or Firefox Firebug to test changes to code from a text-based IDE. The time it takes to round-trip test DOM-related development in these tools is considered the standard for defect injection and defect discovery. RQ2 attempts to define the typical delta distribution.

RQ3 attempts to reduce the delta observed in RQ2 by providing the developers with a dedicated just-in-time syntactic string identifier resolution tool which will help the developer to discover the defect quicker than traditional methods in practice today.

1.6 THESIS OVERVIEW

This thesis will attempt to contribute towards helping the developers manage dependencies with the help of a tool that maintains a symbol table of the dependencies identified above. In Chapter 2, I present the current state of research and literature around static analysis in web applications. The proposed solution is in the class of static analysis tools but focuses on a very specific problem. In Chapter 3, Dependency and Error Modeling, I discuss the dependencies presented above in much more detail. Further, an taxonomy of errors is presented that is mapped to these dependencies to help provide better error messages to the developers to help resolve the defects quickly. Chapter 4 discusses about how the tool is implemented and the flow of the tool. Chapter 5, the validation chapter describes about how the validation was conducted in order to answer the three research questions presented above. In Chapter 6, I summarize the main contributions, limitations and present future work related to this research.

CHAPTER 2

LITERATURE REVIEW

Front end web application development is changing very rapidly in the industry today. Every year new frameworks, libraries, and technologies related to HTML5, CSS3 and JavaScript are launched, while existing ones evolve rapidly. It is difficult for the research community to keep pace with such rapidly changing environment. Some researchers have tried to provide a comparative evaluation of commonly used frameworks to help other researchers and practitioners to choose from a plethora of available frameworks. Gizas et al. [11] have done a comparative evaluation and discussed quality and performance which might help developers decide which JavaScript framework to pick. Graziotin et al. [12] have provided a framework which might help researchers and practitioners do a comparative analysis of JavaScript frameworks. This shows the variety and magnitude of various frameworks available today.

There has been some research related to HTML5, JavaScript, and CSS3 in isolation. Some researchers have spanned across two of the three technologies. Almost little to no research exists across these three technologies, spanning across the entire stack. Most research is concentrated on either JavaScript code semantic analysis or CSS code semantic analysis. Also, most of the current research focuses on security, performance, and optimizations. There is almost no literature that talks about syntactic static analysis across HTML5, CSS3, and JavaScript in light of software engineering concepts like defect injection and discovery. Existing work [13] [14] [15] [16] [17] focuses on semantic static analysis, while this research project attempts to tackle the challenges that

can be solved using syntactic static analysis. The following sections discuss briefly the existing research around static analysis of front-end web applications. Then I discuss the nature of most common front end web application defects. In section 2.5, I discuss the prior work which has been extended by this research project.

2.1 STATIC ANALYSIS OF WEB APPLICATIONS

Most research around static analysis of web applications is focused on vulnerability detection and defect detection. The web applications discussed in these research projects are focused on PHP based web applications. Medeiros et al. [18] proposed a tool called Web Application Protection (WAP). This tool addresses flow-based security issues related to confidentiality and integrity. The tool uses a static analysis approach to find vulnerabilities by generating an abstract syntax tree and doing taint analysis. Scholte et al. [19] discuss a tool called IPAAS (Input Parameter Analysis System), which is an automated input validation tool. The main purpose of this tool is to prevent input validation related vulnerabilities. The approach is a combination of dynamic and static analysis, where the vulnerability validators are applied at run time. Artzi et al. [20] discuss a tool called Apollo that is used to find bugs in web applications. The bugs that the tool intends to capture are of two types, execution failures (crashes) and HTML failures (malformed HTML). The tool uses dynamic test generation and explicit-state model checking. Shar et al. [21] proposed a solution to develop a fine-grained vulnerability prediction approach for web applications. The discussion in this paper is built on top of their previous work which was a tool called PhpMiner [22]. The approach is based on input validation and input sanitization. Hybrid program analysis is used to

make the best of both static analysis and dynamic analysis. Both supervised and semi-supervised learning methods are used to tackle the problem of limited training data. Jovanovic et al. [23] also explore static analysis for detection of vulnerabilities in PHP based web applications. They have used flow-sensitive, interprocedural and context-sensitive dataflow analysis. To improve correctness and precision, they use alias and literal analysis. Their tool, called Pixy, is an open source prototype. Using the tool, they have discovered 15 previously unknown vulnerabilities and reconstructed 36 known vulnerabilities with a false positive rate of 50%. Huang et al. [24] used static analysis and runtime inspection to detect vulnerabilities and enhance the security of web applications. Their approach exploits information flow and uses lattice-based static analysis algorithm. During the analysis, if any code block is considered vulnerable, a runtime guard is inserted to enhance the security of the web application. Their proposed solution is implemented as a tool called WebSSARI (Web application Security by Static Analysis and Runtime Inspection).

All the examples above use static analysis for vulnerability detection and defect detection in web applications. All papers presented above are focused on PHP. The focus of this thesis is in the domain of web applications, but it targets modern web applications. The examples above were discussed to present the current state of art in static analysis and web applications.

2.2 JAVASCRIPT ANALYSIS

There are certain papers that draw attention to the need to improve tools for JavaScript developers. Andreasen et al. [25], in their position paper, discuss three major JavaScript tools used for various purposes. The tools discussed cover dataflow analysis, code refactoring and code coverage testing. The tools are TAJS [26], JSRefactor[27] and Artemis[28]. The paper discusses how each of these three areas are a challenge in themselves when it comes to JavaScript, owing to the nature of the language. Dataflow analysis for JavaScript is interesting because it helps find type related errors and dead-code. Pointer-based static analysis cannot find dead code and hence, often leads to false data. Also, pointer-based static analysis is context-insensitive, whereas, dataflow analysis is partially context-sensitive. Though there are such shortcomings with pointer analysis, various other research endeavors[15] [17] have shown that pointer analysis alongside other techniques can help overcome the drawbacks of pointer analysis.

Andreasen et al. [13] acknowledge the need for better tools for JavaScript programmers. On the other hand, the authors also discuss the challenges with static analysis of JavaScript due to the dynamic nature of the language and the heavy use of libraries. Although there has been previous work done to achieve determinacy using dynamic analysis, the technique proposed can help to integrate determinacy in static analysis, and can help in avoiding the drawbacks of dynamic determinacy analysis. The author discusses an analysis technique that combines selective context and path sensitivity, constant propagation and branch pruning. Two major tools used for JavaScript analysis have been compared, TAJS and WALA. Previous work has shown that WALA [29] [30]

is not capable of analyzing jQuery because it uses pointer analysis. On the other hand, TAJS [26] uses dataflow analysis, which is what the authors recommend. Based on the results it can be said that determinacy information can be inferred and exploited in static analysis of JavaScript code. Therefore, high precision data flow analysis is a promising approach for semantic static analysis of JavaScript code.

Jensen et al. [16] discuss a technique that uses static analysis to reason about data and control flow in JavaScript apps. The work is built on top of TAJS [26]. The main goal of this work is the ability to show the absence of errors and find dead and unreachable code. They claim their work to be the very first in data and control flow analysis of JavaScript web apps. This paper discusses the need of modeling the HTML DOM and Browser API to handle object properties and function parameters. Even though such models are needed, they further discuss the challenge of non-standard implementations and variations in browser implementations. Their work has some significant results showing that dataflow analysis using models for the DOM and Browser API can be really helpful.

Static analysis of JavaScript code using pointer analysis has certain drawbacks, namely, lots of false positives (due to flow and context insensitivity) and inability to work in the presence of dead code. Madsen et al. [17] discuss an approach that uses pointer analysis alongside *use analysis* to tackle such issues. Also, this approach helps to use static analysis for JavaScript in the presence of libraries and frameworks. The basic idea behind the combination of these techniques is to be able to discover properties of returned objects from libraries. The use analysis is done in two ways; partial inference (in the

presence of stubs) and full inference (in the absence of stubs). The purpose of stubs is to describe all objects, functions, and properties, which can help to establish flow information and include that in static analysis. They use Andersen-style *points-to analysis*, which is relatively straightforward technique, but is flow and context-insensitive, and field-sensitive. Their entire work is focused on Windows 8 applications. Their technique is very elaborative and seems to be a good fit for semantic static analysis of JavaScript applications in the presence of libraries and frameworks.

Bajaj et al. [14] discuss the implementation of a tool called *Dompletion* that provides automated code completion suggestions by analyzing DOM structures and JavaScript code. The authors mention that there is no existing work that discusses such a tool. The approach is to extract various DOM states from the application and infer patterns from the observed DOM tree. Then, the tool captures and analyzes all JavaScript code that interacts with the DOM and it reasons about the consequences of such interactions on the DOM state. Finally, it provides code completion suggestions. The implementation of the tool is also done in JavaScript and the target IDE is Brackets. The tool implementation approach is as follows:

1. DOM Analysis
2. JS Code Analysis
3. Suggestion generation

To improve the time and space complexity of the tool, compression of the list of suggestions is done using the following:

1. Eliminate duplicate DOM element locators

2. Combine DOM element locators with similar IDs

3. Combine DOM element locators with similar classes

The concept of DOM element locators is similar to the syntactic string identifiers that this thesis is addressing. This paper touches a small portion of such identifiers and uses them only for code completion purposes.

Schäfer et al. [15] discuss another JavaScript code completion tool called Pythia. Their approach is similar to Madsen, et al. [17] in that they also combine static analysis and usage-based property inference. The major difference is that Madsen, et al. used stubs to generate property inferences, but they are using dynamic analysis of libraries/frameworks using the test suites provided by the libraries/frameworks to generate models to infer property information. They use static analysis to infer properties from user code. They use usage-based property inference to tackle incomplete programs that are under development. They use dynamic analysis to establish models which can then be used with usage-based property inference to tackle the analysis of JS code in libraries/frameworks/native APIs. They also use Andersen-style points-to static analysis. Their tool is written in JavaScript and uses WALA.

All the papers above discuss the current state of art in semantic JavaScript analysis, with TAJS and WALA being the most well-known. These papers tackle the difficult problem of static analysis of JavaScript which is a dynamically typed language. This thesis, however, is focused on syntactic HTML5, JavaScript, and CSS analysis and targets a

whole different set of problems related to developer productivity when compared with the papers above.

2.3 CSS ANALYSIS

Even though CSS is used widely for various purposes, very little literature talks about static analysis of CSS. The limited existing literature on CSS focuses on CSS refactoring and duplicate CSS rules [31] [32] [33] [34] [35] [36] with the aim to reduce CSS rules to optimize rendering of the HTML pages.

Bosch et al. [31] discuss a tool that automatically refactors CSS files to reduce the size of the style sheets. This refactoring preserves the rendering semantics and does not affect the web page style rendering. Their techniques are based on static analysis of semantic relations between CSS selectors and media queries. Their results showed that the average size reduction of CSS files was 7.75% with a maximum of 17.83%. Mazinanian et al. [32] discusses an automated approach to remove duplicate CSS code by detecting three different types of CSS declaration duplication. Their tool further suggests presentation-preserving refactoring opportunities that can help reduce the size of the CSS files. Their results showed that average size reduction was 8% and the maximum size reduction was 35%. Bosch et al. [33] discuss a tool that detects unnecessary property declarations in CSS files based on semantical relations between CSS selectors. Their tool observed 4.95% of unnecessary property declarations in CSS files. Hague et al. [34] propose a tree rewriting approach to remove redundant CSS rules. Their tool *TreePed* uses static analysis to find and remove redundant CSS rules by using a tree rewriting model.

Geneves et al. [35] were one of the first ones to statically analyze CSS files. Their approach also uses tree logics for statically detecting unused and redundant CSS rules. Mesbah et al. [36] discuss a tool called *CILLA*, which finds unmatched and ineffective selectors, overridden declaration properties, and undefined class values. Their results show 60% of unused CSS code in various web applications.

All these papers discussed the aforementioned approach of CSS static analysis from a semantic perspective and their results show that CSS files are often bloated and contain many unused CSS rules. The existing literature on CSS focuses on CSS refactoring and duplicate CSS rules with the aim to reduce CSS rules to optimize rendering of the HTML pages. This thesis instead focuses on unused CSS rules because of syntactic defects in string literals. More often than not, unused CSS code does not directly produce defects and hence unused rules are ignored. There can also be several cases where an ID or selector is referenced in the HTML but is undefined in CSS and vice-versa. Hence, there is a need for a tool that can help figure such dependencies. There is no way for the developers to find any errors with the CSS rules because the browser does not report errors on the console. This results in several defects in CSS files going unnoticed which in turn can have unwanted effects like latency in page rendering (performance), dead code (maintenance), and multiple rules over the same elements which would lead to latency. The only way for a developer to know if CSS is broken is by testing it in the browser. This case is worse than that of JavaScript because JavaScript throws error messages in the console of the browser. Hence, if the developer is not careful enough, they might not notice a defect related to CSS. Moreover, CSS techniques like hide/show

are a common alternative used by developers instead of DOM manipulation. These are further reasons to investigate dependencies between CSS and HTML.

2.4 MOST COMMON DEFECTS

The existing literature validates our hypothesis that syntactic errors exist in modern web applications. In an empirical study conducted by Ocariza, et al. [10] on JavaScript errors, they noted the following:

> For example, the error message "C is null" was encountered in the Yahoo application. Subsequent analysis revealed that the error was caused by a typographical error in the value of the "id" attribute of a *div* element in the DOM. The incorrect id caused the getElementById method to return a null value, which, in this case, was assigned to the variable "C". The variable "C" was later used to update the class name of the div element, causing a null exception to be thrown. ([10], pp. 104)

This is a classic example of syntactic error caused by string identifiers in modern web applications. Moreover, it is important to note that such a defect is caused by dependency between JavaScript and HTML5; avoiding the defect requires the developer to trace the set of IDs used in the DOM to their respective references in JavaScript. The authors categorized the above defect under the umbrella of `NullExceptions`. Among other categories were `Undefined Symbol` and `Syntax Errors`. Further, they report these findings: "null exception errors make up 9.3%; undefined symbol errors make up 28.4%; and syntax errors make up 4.1%" ([10], pp. 105). That amounts to 41.8% of the errors that they observed. Taking into account the Yahoo Application example quoted above, it

is safe to conclude that a certain significant percentage of defects contributing to 41.8% of the errors are caused by syntactic errors and typographical errors in string identifiers. In the same paper, the authors observed: “There is a significant correlation between the total number of distinct null exception errors and the number of functions called dynamically by the web application.” ([10], pp. 107). Further, they note that: “... null exception errors are often caused by failed accesses to the DOM of the web application …” ([10], pp. 107). Although they have not categorized it further, but it can be seen that these failed DOM accesses can be because of three major reasons:

1. Invalid syntax/selector
2. Typographical errors
3. Deleted DOM elements

This is evidence concluding the existence of defects due to syntactic errors, specifically in string identifiers. In another paper, Ocariza, et al. [6] found that “approximately 14% 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.” ([6], pp. 61) and “Interestingly, around 7% of bugs resulted from syntax errors in the JavaScript code that were made by the programmer.” ([6], pp. 62). Hence, taking into account both of these observations, it can be seen that syntax related issues cause a total of 21% defects in JavaScript code. Another common category of defects found by the authors was ““Incorrect Method Parameter” faults account for around 74% of JavaScript faults” ([6], pp. 59). Some of these errors are caused by methods invoked by events on DOM elements and are also a source of dependency related defects. Using manual classification scheme for quantitative analysis, and a

qualitative reading of defect reports for qualitative analysis, the authors' claim is that 65% of all JS defects are DOM related, and 80% of such defects are of high impact or severity. In another paper, Ocariza et al. [37] further show that 79% of JavaScript errors are DOM-related errors using fault localization approach for JavaScript-based web applications.

Although very little literature exists that illustrates the same concepts related to CSS, it is not very difficult to find certain common patterns that can be applicable to CSS as well. Mazinanian, et al. [32] stated that: “While CSS has a relatively simple syntax, some of its complex features, such as inheritance, cascading, and specificity , inherently make both the development and maintenance of CSS code cumbersome tasks for developers” ([32], pp. 496). I contend that because of this nature, it makes CSS error-prone.

A dedicated just-in-time syntactic string identifier resolution tool coupled with JavaScript and CSS lint checkers can help map these dependencies and prevent the kind of errors discussed above. It will also help to significantly reduce the delta (time or cost) between defect discovery and defect injection. Ocariza, et al. [8] found that amongst the most common fixes used to fix JavaScript faults, a certain percentage is where the developers directly modified string literals, thereby prompting the need for a tool that could help prevent such defects altogether, i.e., reducing the delta to 0. In another paper [6], the authors report that “We found that DOM-related faults have an average triage time of 26.4 days, compared to 44.4 days for non-DOM-related faults. On the other hand, DOM-related faults have an average fix time of 90.8 days, compared to 66.8 days for non-

DOM-related faults." ([6], pp. 62). The proposed tool can help reduce this delta significantly for DOM-related faults.

Thus, as can be seen from above, it is important to establish dependencies across HTML5, JavaScript and CSS3. This research project also focuses on how such a dependency model can help significantly reduce the delta between defect injection and defect discovery in front-end web applications.

2.5 PRIOR WORK

Gupta et al. [5] [38] proposed a dependency model for establishing the dependencies across the HTML5, JavaScript and CSS3 stack. A modified version of this model has been used by this research project and is discussed in the Dependency and Error modeling chapter. The completeness and soundness of this approach was validated by conducting user studies and observing precision and recall in finding defects related to dependencies. These studies were repeated as a part of prior validation for this research work. The prior validation also helped make the prior work results stronger. As a consequence of the prior validation, I had some questions which were the main motivation for this research. These are discussed in section 2.5.16.

2.5.1 PRIOR VALIDATION

In this section I present the results of the prior validation experiments conducted. These experiments were an exact rerun of the experiment designed in [38]. These experiments helped to reaffirm the observations from [38] in terms of the soundness and completeness

of this approach by using precision and recall as measures. The findings from prior validation helped pave the way for this research and also helped to modify the dependency model that was used in this research.

2.5.2 EXPERIMENT SETUP

This experiment was an exact repeat of the experiment as defined in [38] with two different populations. The experiment is focused on measuring the productivity and efficiency of developers using "HJCDepend" in discovering and removing defects by calculating the following metrics [38]:

- Precision: The fraction of retrieved defects that are relevant:

$$precision = \frac{|\{relevant\ defects\} \cap \{retrieved\ defects\}|}{|\{retrieved\ defects\}|}$$

  This tells us the accuracy of the developer in finding defects.

- Recall: The fraction of relevant defects found by a developer out of the total defects that are present in a body of code:

$$recall = \frac{|\{relevant\ defects\} \cap \{retrieved\ defects\}|}{|\{relevant\ defects\}|}$$

  This would tell us what percentage of the total defects present in the body of code was found by the developer.

- Defect discovery rate: The average time taken by a developer to find a defect in a body of code. This will indicate how fast a developer finds defects.

- Defect removal rate: The average time taken by a developer to remove/ fix a defect in a body of code. This will indicate how efficient a developer is in removing defects.

It is important here for the reader to note that this experiment does not answer any of the research questions. Instead, it helps us reaffirm the approach by soundness and completeness measures. Soundness is determined by precision and completeness is determined by recall.

2.5.3 PROTOCOL

Two separate user studies were conducted on two different groups of student developers. They were asked to debug an existing body of code for dependency related defects with the help of HJCDepend and another group of roughly equivalent skill set of developers that debugged the same code but without HJCDepend. Each developer was given a total of 75 minutes to do two tasks. For the first 45 minutes, they were not allowed to make any changes to the code and were asked to report as many defects as they could find in the source code. They were not aware of the total number of actual defects present in the code and were told that once they were convinced that they have found all the defects in the source code, in the remaining time of the study their second task was to remove the defects and report each defect they fixed. The second activity had to be at least of 30 minutes.

2.5.4 SOURCE CODE

The source code used for this user study is the same as used in [38]. It was a two-page web application which consisted of 2 HTML5 files (total 100 lines), 4 JavaScript files (total 85 lines) and 3 CSS3 files (total 142 lines). It was seeded with dependency related defects.

2.5.5 ENVIRONMENT

The participants had to use Windows 7 desktops with similar configurations. The group not using the tool had the freedom to use any IDE and any tool for the activity. The group using the plugin had to use Eclipse IDE.

2.5.6 PARTICIPANTS

A pre-survey was conducted to recruit participants. The identity of the participants was kept anonymous. Based on the responses, the participants were sampled into two groups with equivalent skill set. The study was done with two different groups of participants, one with n=33 and the other with n=19.

2.5.7 RESULTS

There were two different user studies conducted with different groups of participants. The results are not aggregated and are present separately for each user study. The group using the plugin is referred to as group B and the group not using the plugin is referred to as group A. The first user study had n=33 and the second one had n=19 participants.

In the first user study, the group using the tool had a high precision of 81.22% compared to 68% of the group not using the tool. The recall was also higher for the group using the tool. It was 62% as compared to 43.36% of the group not using the tool. The average time taken to report valid defects for the group using the tool was 60.62 seconds as compared to 97.65 seconds for the group not using the tool.

In the second user study, the group using the tool had a high precision of 77.4% compared to 72.5% of the group not using the tool. The recall was also higher for the group using the tool. It was 51.36% as compared to 36.25% of the group not using the tool. The average time taken to report valid defects for the group using the tool was 99.64 seconds as compared to 160.59 seconds for the group not using the tool.

2.5.8 OBSERVATIONS

For both the user studies, recall is always higher for the group using the plugin, and always over 50%. That means the group using the plugin is able to find more valid defects. Similarly, for the both the user studies, precision is also always higher for the group using the plugin, and always higher than 70%, meaning the group using the plugin found 7 valid defects in every 10 defects that they reported. On an average, people who used the plugin found valid defects faster. For both user studies, the group using the tool found defects 1.61 times faster.

It can be seen in Figure 6 and Figure 7 that the group using the plugin found more valid defects. The group using the plugin is represented by the red color and the group is represented by the blue color.

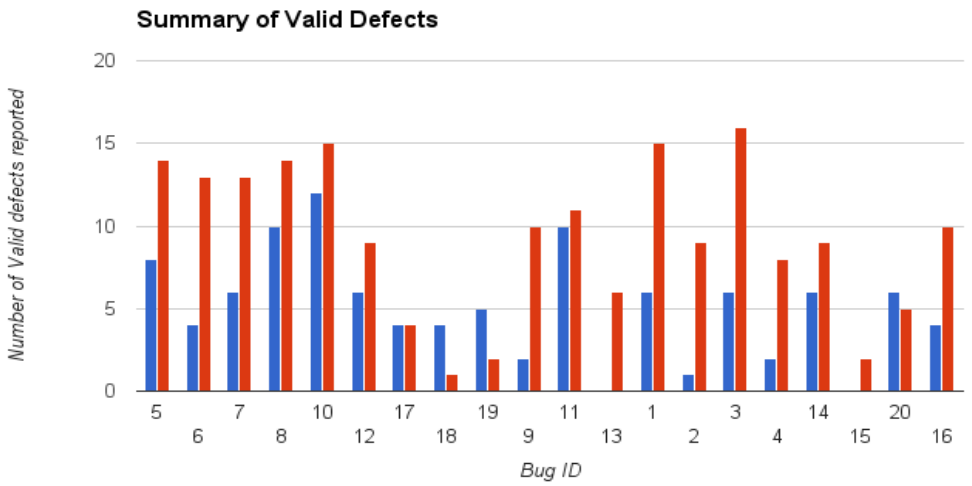


Figure 6. User Study 1

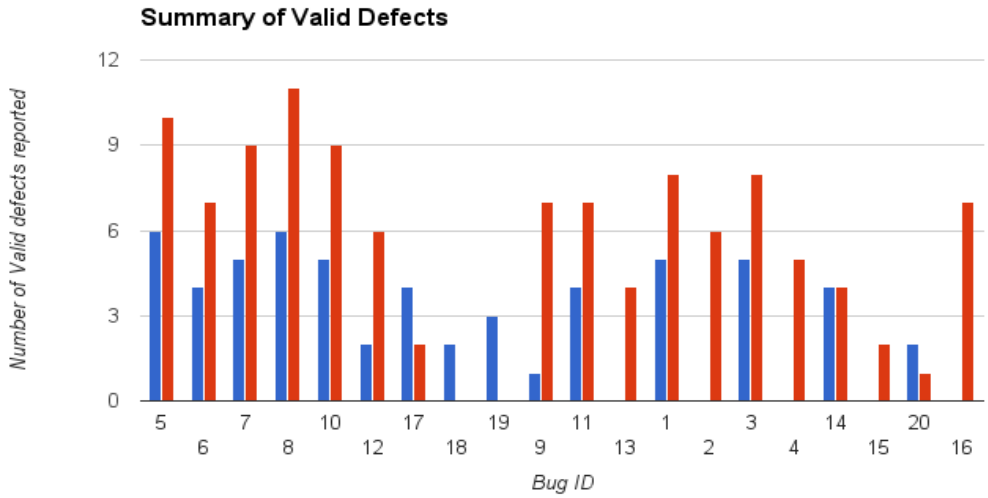


Figure 7. User Study 2

Figure 8 and Figure 9 show the average time taken to report defects. The red color represents the group using the plugin and the blue color represents the group not using the plugin.

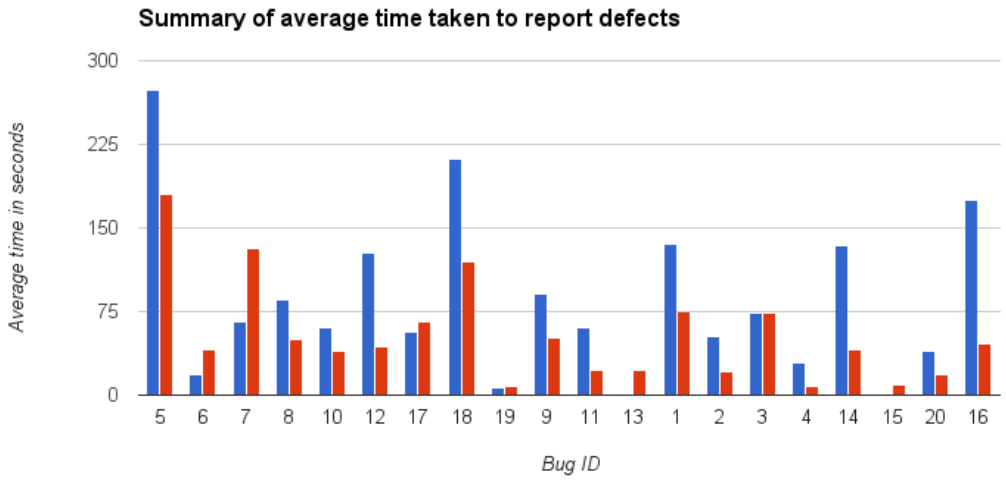


Figure 8. User Study 1

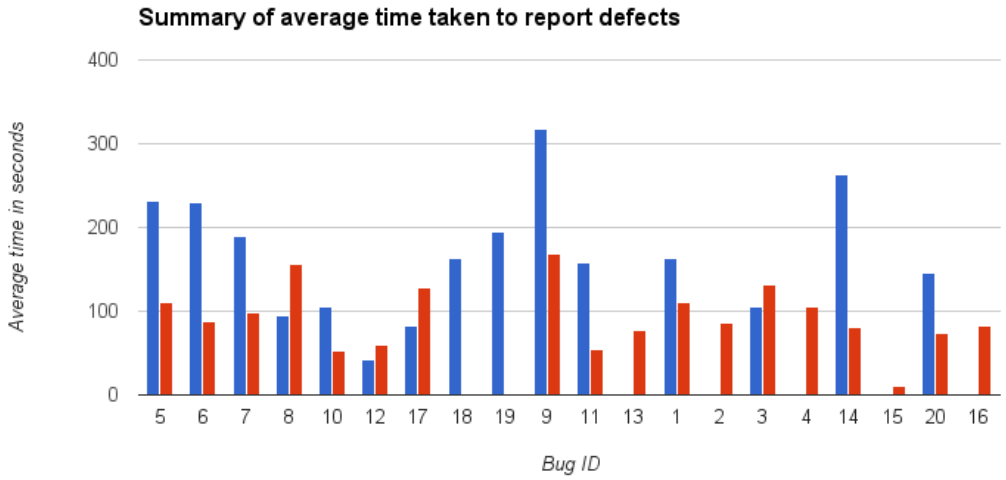


Figure 9. User Study 2

For both user studies, the *p* value computed was 0.1 which is greater than desired value of 0.05, but not greater than 0.1. The null hypothesis for this experiment is that the means observed for the two groups are the same, suggesting the time taken is not related to using the HJCDepend tool. The alternate hypothesis is that the means are different, or

that the time taken is related to using the HJCDepend tool. The observed value shows weak evidence that the null hypothesis does not hold, meaning not very strong evidence that the means observed for the two groups are different from each other. The *d* value computed was 0.5 which signifies medium effect size and medium practical significance. These two values show that the results do not present very strong evidence, but are still significant enough to not be discarded.

For User Study 1, Group A takes 41.3 seconds on average to fix the defects, compared to 38.7 for Group B. For User Study 2, Group A takes 59.3 seconds and Group B takes 38.7 seconds.

2.5.9 DISCUSSION

There were two main limitations in these user studies. First, the *p* value and *d* value computed for both the user studies imply that a larger sample size is needed to provide stronger evidence. Second, the study was conducted with student developers who may not be at the same skill level as the professional developers.

The prior validation experiment shows that this particular approach towards dependency management is effective in terms of soundness and completeness. As a result of conducting these studies, I had a few questions that lead to motivation for this research. The questions were:

- How can the dependency model be improved?

- Are there any other measures more impactful than precision and recall to measure developer productivity?
- How does this approach compare to newer literature?

These questions were a motivation towards taking the next steps for this research. This lead to the focus on using delta between defect injection and defect discovery as a measure as compared to precision and recall. The inaccuracy in self-reported data was also a factor contributing towards the focus on delta between defect injection and defect discovery. The next chapter discusses the modified dependency model and the error taxonomy that has been used. The modified dependency model and an error taxonomy also required a new tool to be implemented. The validation approach was entirely focused on measuring defect injection and defect discovery times. The implementation of the tool is discussed in chapter 4 and the validation is discussed in chapter 5.

CHAPTER 3

DEPENDENCY AND ERROR MODELING

The HTML5-JavaScript-CSS3 stack for front end development is very tightly coupled. To successfully render a web application on a browser, these three languages have to be parsed and loaded correctly. The discussion of how this works is beyond the scope of this research, but it is important to understand here that the developer has to deal with the dependencies between these three languages. It is important for the developer to understand these dependencies and tackle any defects generated because of them. Gupta [38] talks about dependency analysis between HTML5-JavaScript-CSS3 and provides a dependency model. Based on this dependency model, we further extend the concept of dependencies to establish an error taxonomy to categorize the dependencies more succinctly in order to provide better error messages for the developers using this static analysis tool. In the next section we briefly touch upon the dependency model as explained by [38], and then we discuss the error taxonomy used to further translate the dependency model for explaining the dependency errors to the developer using the tool.

3.1 DEPENDENCY MODEL

Table 1 shows the dependency model that this research project is using based on the model as defined in [38].

To → From	HTML5	JavaScript (JS)	CSS3
HTML5	No dependencies identified.	1. Links from HTML5 to JS files. 2. Event listeners in HTML5 file.	Class attribute in HTML5 elements.
JavaScript	Through Document object	1. Function calls within a function. 2. Global variables.	JavaScript adding CSS3 class to DOM assuming it exists in one of the CSS3 files included in the webpage.
CSS3	Other CSS3 selectors like id, tag name.	No dependencies identified.	No dependencies identified.

Table 1. Dependency Model (from [38])

The next few paragraphs discuss these dependencies briefly as presented in [38] and then map these dependencies to the proposed error taxonomy in our research.

3.1.1 HTML5 to HTML5

As identified in [38], there might be certain dependencies within the HTML5 document between different tags depending on the structure of the HTML5 document. Such dependencies are not in the scope of this research and hence, are not investigated further.

3.1.2 HTML5 to JavaScript

There are two major categories of dependencies between HTML5 and JavaScript:

1. Links from HTML5 to JS files.
2. Event listeners in HTML5 file.

The dependency because of Links originates from the code where the HTML5 document includes a reference to a JavaScript file. A simple example to illustrate this dependency is shown below:

```
<script src=”js/someJavaScriptFile.js></script>
```

The dependency because of event listeners arises because of user interaction with an element in the HTML5 document that has a corresponding event listener bound whose definition is in one of the associated JavaScript files. An example of such a dependency is shown below:

```
<input type=”button” onclick=”myFunction()”>
// .. in the JavaScript file .. //
function myFunction(){
      alert (“A button on the webpage was clicked”)
}
```

3.1.3 HTML5 to CSS3

The most common CSS selector used in defining CSS rules is the class selector. A class selector is identified as a dependency in the model. An example of such a dependency is shown below:

```
<div class=”myStyleByClass”></div>
/* in the CSS file */
.myStyleByClass {
   position: relative;
}
```

3.1.4 JavaScript to HTML5

This dependency exists because of the ability to reference HTML5 element through the document object. Such references are used to get a handle onto the HTML5 element and perform actions like event binding, DOM manipulation, etc. If the reference does not exist, it will lead to JavaScript errors. An example of such a dependency is shown below:

```
var htmlElement = document.getElementById("myDivContainer");
```

3.1.5. JavaScript to JavaScript

The dependency model discussed in [38] presents various dependencies that might exist within a JavaScript file. However, such dependencies are not in the scope of this research. There has already been some research done in the field of JavaScript static analysis as presented in the literature review chapter.

3.1.6 JavaScript to CSS3

This dependency is an extension of the dependency between JavaScript and HTML5. We have already seen that JavaScript can access HTML5 elements using the document object. Once a reference has been obtained to any element, using JavaScript code, we can assign a CSS class to the HTML5 element. An example of such a dependency is shown below:

```
document.getElementById("myElement").className = "myCssClass";
```

3.1.7 CSS3 to HTML5

The two common CSS selectors used for defining CSS3 rules are class selectors and id selectors. The class selector has been identified as a dependency in HTML5 to CSS3 dependency. The other dependency is generated due to id selector. This dependency arises by accessing an HTML5 element using an id selector in a CSS3 rule. An example of such a dependency is shown below:

```
<div id=”myStyleById”></div>
/* in the CSS file */
#myStyleById {
    margin: 5px;
}
```

3.1.8 CSS3 to JavaScript and CSS3

For the scope of this research, I have not identified any relevant dependencies between CSS3 and JavaScript. Although there are dependencies like accessing a CSS class from a JavaScript file, but that has already been covered in JavaScript to CSS3 dependency.

3.2 ERROR TAXONOMY

Johnson, et al. [39] found in a study that developers are reluctant to use static analysis tools to find bugs because of many reasons. One of the prominent reasons was the inability of static analysis tools to provide understandable results. Developers say that it is difficult to make sense out of the error messages given by static analysis tool. Johnson, et al. [39] have also quoted some responses from developers, including: “it’s one thing to

give an error message, it's another thing to give a useful error message."([39], pp. 677) and "I find that the information they provide is not very useful, so I tend to ignore them." ([39], pp. 677). This shows that the developers need to be provided with descriptive and proper error messages that will help them understand the error and also help them understand how to fix the error. Based on the dependency model, I designed an error taxonomy that helps developers understand the error and the dependency better. The error taxonomy design helps to achieve this by building on the dependency model and providing more useful information about the dependencies and the errors.

Figure 10, shows a classification based on the dependency model and other errors that have been taken into consideration. This classification also helps us understand dependency types that do not generate any errors; namely unused dependencies.

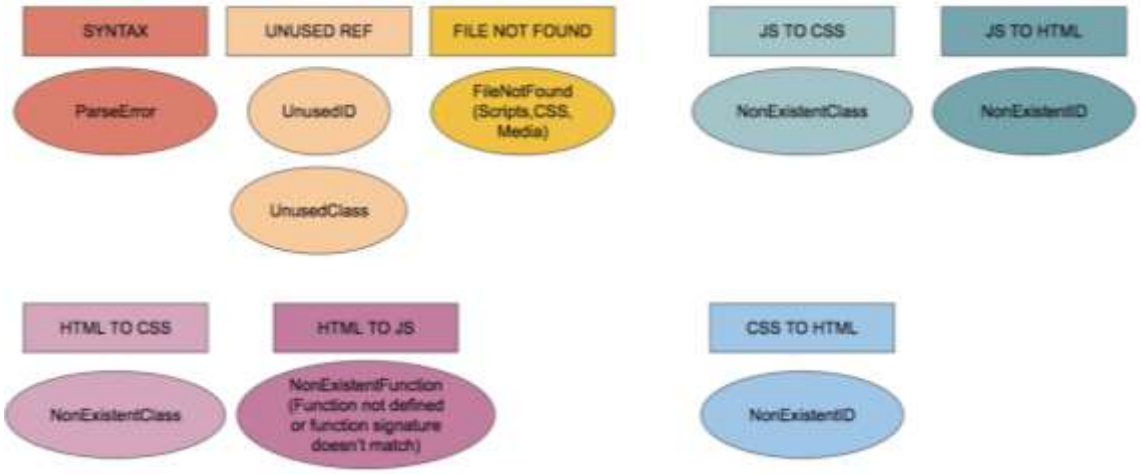


Figure 10. Error Classification

"Syntax" errors are represented by `ParseError`. "Unused Ref " refers to all those dependency constructs that are declared but are not used, and do not generate any errors. "File Not Found" categorizes all the dependencies between the HTML5 document and external files. All other classifications are extensions to the dependencies discussed in the dependency model above.

The next step is to use this classification and generate a taxonomy that will be used to describe the errors and warnings as generated by the tool. The taxonomy is presented in Figure 11.

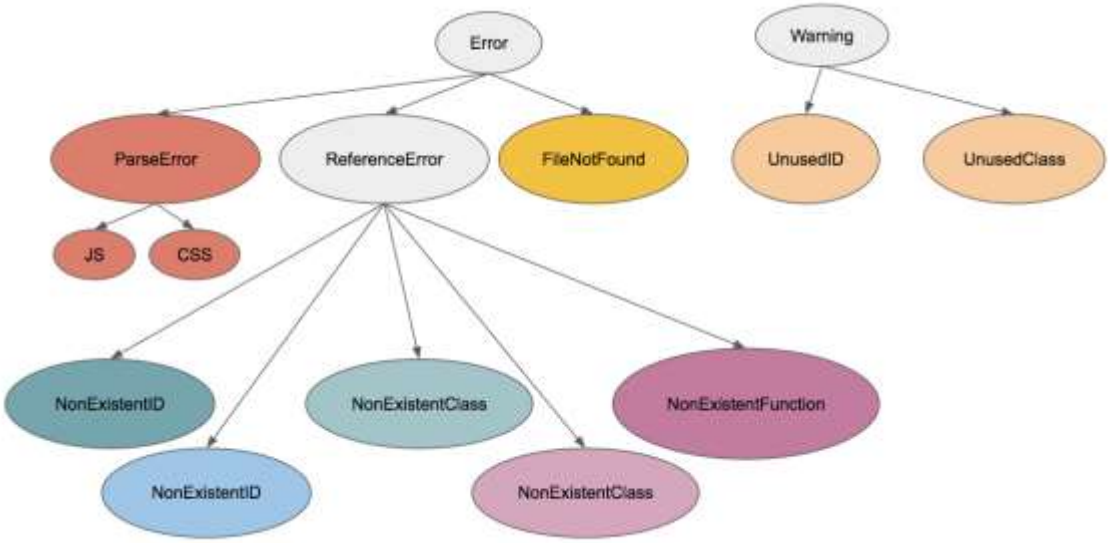


Figure 11. Error Taxonomy

The colors for each node in Figure 11 are the same as in Figure 10 to show the source of the node and its relation to the dependency model. This taxonomy categorizes the errors due to dependencies into two categories: "Error" and "Warning". Warning is caused by

those dependency constructs that do not result in errors. These constructs are defined in the code but are not used. These are represented by “Unused Ref” in Figure 10. The “Error” category includes everything else as shown in Figure 10, because all those constructs can cause errors at runtime.

3.3 CONTRIBUTIONS

Based on the dependency model discussed in section 3.1 and the error taxonomy in 3.2, I developed a static analysis tool that helps developers to handle the dependencies in the HTML5-JavaScript-CSS3 stack. This static analysis tool can be used with Eclipse during development to quickly capture the defects injected due to dependencies. As discussed in chapter one, below are the research questions again:

- RQ1: How significant are syntactic errors in string identifiers referencing DOM elements in the HTML5-JavaScript-CSS3 stack?
- RQ2: What is the delta (time or cost) between defect injection and defect discovery in new HTML5-JavaScript-CSS3 style applications?
- RQ3: Does a dedicated just-in-time syntactic string identifier resolution tool significantly reduce delta time/cost from RQ2 for a significant portion of real problems (RQ1)?

By implementing a dedicated plugin for a development environment to help the developers quickly catch defects as they are injected, I am able to answer RQ2 and RQ3 with the help of an empirical study discussed in chapter 5. Further, the error taxonomy helps in accurately determining how significant are dependency related errors in some open source codebases. This will help to answer RQ1. The results have also been

discussed in chapter 5. Before presenting these studies the next chapter describes the tool built upon this conceptual foundation.

CHAPTER 4

IMPLEMENTATION

The main focus of the implementation was to develop a static analysis tool that can generate a symbol table to manage the dependencies as discussed in Chapter 3. It is important for the reader here to note that developing an algorithm to parse HTML5, CSS3 and JavaScript was not the main focus of this thesis. To parse the codebase, popular libraries and engines were used. This chapter discusses how the implementation of the static analysis tool was achieved and the main features of the static analysis tool.

Most static analysis tools come in two flavors; a standalone command-line tool and a plugin that is installed on a preferable Integrated Development Environment (IDE). Both flavors serve different purposes. The standalone command-line tool can be integrated into the build toolchain or the continuous integration tool. The integrated plugin, on the other hand, can serve the purpose of an interactive utility within the IDE that helps developers tackle defects while they code. A similar approach was taken for this tool as well.

Before diving into the details of the implementation of this tool, it is important for the reader to understand how a browser handles a web page. When a browser receives a request for a web page, it fetches the HTML5 file first. Once the file is retrieved, it starts parsing the HTML5 document. The HTML5 tags are turned into Document Object Model (DOM) nodes in the “content tree”. Then the style data is parsed from the various style sources including CSS3 stylesheets and the inline style tags. The content tree along with the style information are merged to generated the “render tree”. The render tree then

undergoes a “layout” process and is finally painted on the browser. This painted layer is what the user sees on their browser. This flow is shown in Figure 12 below.

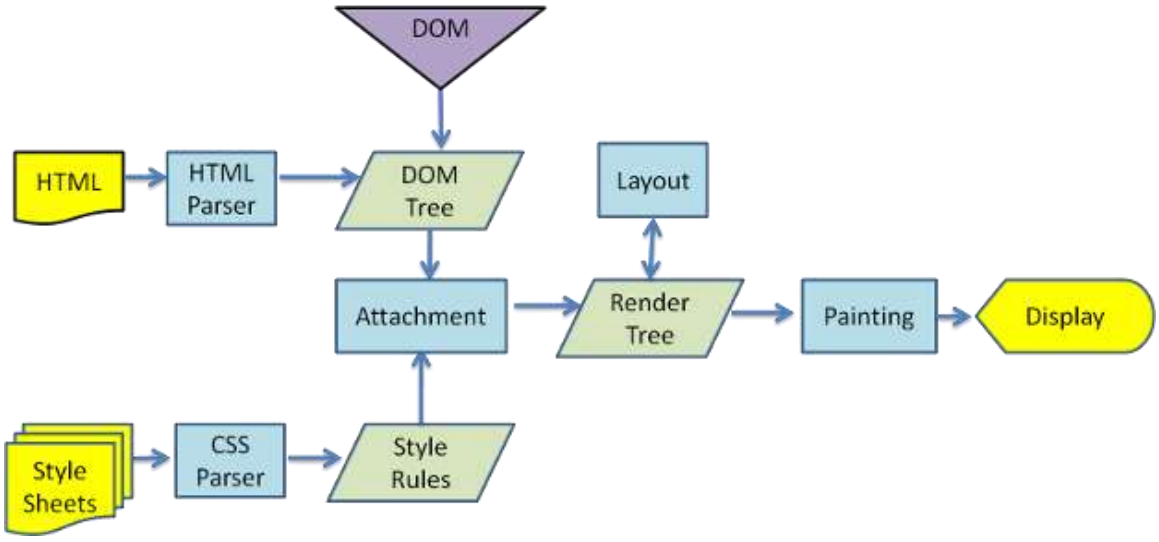


Figure 12. Flow of a web page rendering on a browser [40]

The associated JavaScript with each web page is fetched and parsed when the `<script>` tag is encountered. Based on the internal parsing optimization techniques, the HTML5 document parsing may or may not halt when the associated script files are being fetched and parsed. The general flow explains how a browser loads a web page. A similar flow design was used to develop the static analysis tool for this research.

The primary purpose of this static analysis tool is to identify dependencies among HTML5, CSS3 and JavaScript as discussed in previous chapters. To achieve this, I use different parsers to parse these languages and keep a track of dependencies. Figure 13 shows a visual overview of the tool flow. The HTML5 files in a web application are the door to the entire codebase. This tool also starts the analysis by building a list of HTML5 files in a given directory and moves from there. This way non-dependent files in the

project are ignored. The entire code for a given application is analyzed and a list of dependencies is generated, compared and the results are computed. Furthermore, metadata associated with each dependency is tracked in order to help the developer find and fix the defect as fast as possible. The metadata includes source file, line number, column number and dependency type. The secondary features of the tool include: providing verbosity in terms of output shown, exporting results in JavaScript Object Notation (JSON) and plain text format, an HTML viewer for displaying the JSON results, rule-based analysis, recommendation for fixes and integrated plugin development. In the following sections, I discuss the three different parsers used, computation of dependencies and secondary features development.

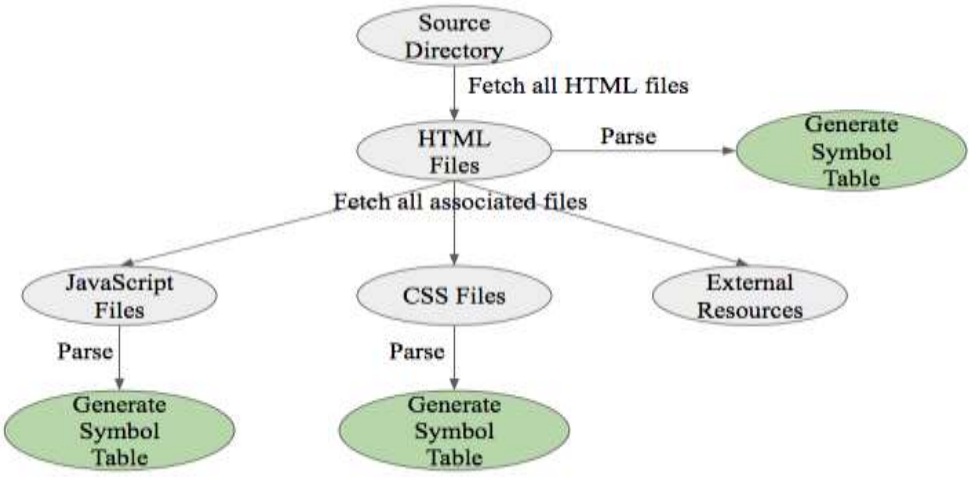


Figure 13. Visual overview of the tool flow

4.1 PARSING HTML5

For parsing HTML5, I used the Jsoup [41] parser which is a popular Java library for parsing HTML. It supports all the latest HTML5 tags and implements the WHATWG

HTML5 specification [42]. It is able to parse HTML5 from either a URL, file, or string. It parses the HTML5 into a Document Object Model(DOM). DOM is a language-independent convention for representing the objects (nodes) in an HTML5 document. It also provides a very simple API for DOM access and traversal. The jQuery-like methods and regex based selectors in the API make it very flexible. It generates a parse-tree behind the scenes but the DOM is exposed via the API as a "Document" object. Although it provides so many nice features, one major drawback is that it does not keep track of line numbers. As a result, a custom module based on Java's Matcher [43] engine was made to keep track of the line numbers and column numbers.

It is important for the reader to understand the need for using a dedicated parser for HTML5. Some readers might feel the need to simply use regex for parsing HTML5. But one very common mistake committed while parsing HTML5 is to make use of regular expressions (regex). Though it might seem a correct choice at first, it is not possible to use regex for parsing HTML5. The primary reason for this is that the underlying data structure used by regular expressions is a finite automaton, which does not have any memory and only has a finite number of internal states. Using such a data structure to parse something like HTML5, which is arbitrarily deeply nested, would mean having infinite internal states which is not possible with a finite automaton. Below are some examples [44] of valid HTML5 which show why regex cannot be used for HTML5 parsing:

- `<p>`

```
some text here
```

```
        <p>
              some inner text
              <p>
                    this is deeply nested
              </p>
        </p>
    </p>
```

- `<tag attr="value" />`
- `<a href="foo">foo</a>`

  `<!-- FIXME: <a href=" -->`

  `<a href="bar">bar</a>`

Based on the source directory that is provided for analysis, all the HTML5 files are first extracted based on the file extensions. Once all HTML5 files have been identified, each file is parsed using the Jsoup Parser. For each file, the "`body Element`" is extracted from the "`Document`" object. Convenient methods such as, `getAllElementIds`, `getAllElementClasses`, `getStyleSheetLinks`, `getMediaLinks`, `getScriptLinks` and `getEventHandlers` were implemented that recursively traverse the Element object and extract the required data. From these methods, the following data is extracted per HTML5 file:

- All "id" attributes used
- All "class" attributes used
- Links to all associated CSS3 style sheets
- Links to all associated media assets (images, videos, audio, etc.)

- Links to all associated JavaScript files
- All methods referenced through event handlers

Also, the parser allows checking for syntax errors which are stored as a Parse Error. Once the above data is available for each file, each referenced file (CSS3 file, JavaScript file or asset file) is checked for existence and path validity. If any file is not found, then a `FileNotFound` dependency error is generated. After that, each associated CSS3 file and JavaScript file is parsed and analyzed for other dependencies.

4.2 PARSING CSS3

Parsing CSS3 using regex is a possibility, but to avoid parsing defects and extract the accurate list of selectors, a CSS parser was used. CSS3 parsing was achieved using CSSParser [45] which is a Java library to parse CSS3 files. It supports CSS1, CSS2, and CSS3. It takes the CSS3 file text as input and generates a Document Object Model Level 2 Style [46] tree. It also provides the ability to choose different internal parsers as per the developers need. For the purpose of this project, I am using a SAC Parser [47] for CSS3 since the focus is on modern web applications which primarily use CSS3. Another advantage with this parser is that it allows attaching an error handler to keep track of parse errors generated per rule.

All CSS3 files associated with each HTML5 file are processed using this parser. Once the HTML5 parsing phase is complete and all valid associated CSS3 files are identified, each file is passed on to the CSS3 parser. For each CSS3 file, the following data is extracted:

- List of referenced IDs
- List of defined classes

Once this data is obtained, the static analyzer uses this information to analyze dependencies based on class and id references.

4.3 PARSING JAVASCRIPT

For parsing JavaScript, there are few JavaScript engines available which could have been used, namely Mozilla's Rhino engine, Chrome's V8 engine, etc. But I chose to use the Nashorn engine [48]. Nashorn engine is a JavaScript engine by Oracle and comes pre-bundled with Java 8 and above. It has comparable speed to Chrome's V8 engine as shown in Figure 14 [49].

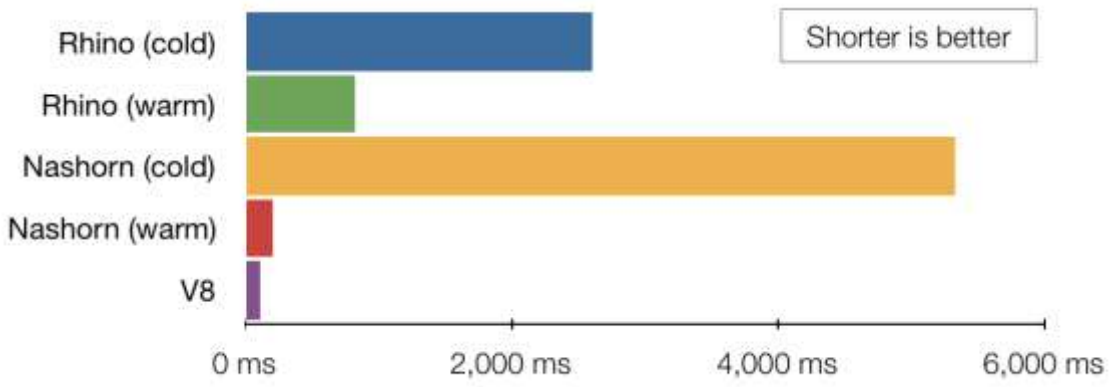


Figure 14. Speed comparisons of popular JavaScript engines [49]

It has a very nice API which provides a method to extract an Abstract Syntax Tree(AST) from the given JavaScript source code. The AST is provided as a JavaScript Object Notation(JSON) data structure which can be easily parsed to extract the information that is required. For example, for a given JavaScript source like:

```
function a() {

      var b = 5;

}
function c() { }
```

the engine returns an AST representation as follows:

```
{
      "type ": "Program ",
      "body": [{
            "type ": "FunctionDeclaration ",
            "id": {
                  "type ": "Identifier ",
                  "name": "a"
            },
            "params": [],
            "defaults": [],
            "rest": null,
            "body": {
                  "type ": "BlockStatement ",
                  "body": [{
                        "type ": "VariableDeclaration ",
                        "declarations": [{
                              "type ": "VariableDeclarator ",
                              "id": {
                                    "type ": "Identifier ",
                                    "name": "b"
                              },
                              "init": {
                                    "type ": "Literal ",
                                    "value": 5
                              }
                        }]
                  }]
            },
            "generator": false,
            "expression": false
      }, {
            "type ": "FunctionDeclaration ",
            "id": {
                  "type ": "Identifier ",
                  "name": "c"
            },
            // .. and so on
```

All JavaScript files associated with each HTML5 file are processed using this parser. Once the HTML5 parsing phase is complete and all valid associated JS files are identified, each file is passed on to the JS parser. For each JS file, the following data is extracted:

- List of referenced IDs
- List of referenced classes
- Method signatures of all methods defined

All of the above information is stored for each file along with the location of the respective file.

4.4 INTEGRATING PARSER RESULTS

Once the data from all associated files is gathered, it is compared with dependencies as and when they arise. For example, while identifying the list of IDs in a JavaScript file, those IDs are simultaneously compared to existing IDs in the associated HTML5 file. If an error is found, it is immediately stored in the Results object and retrieved later for display. Similar is the case with the CSS file processing. When a dependency is found that does not contribute to a runtime error, it is stored in the Results object as a warning. Based on the verbosity level, the user can choose to view the warning details. An example of such a warning would be “unused CSS class”.

4.5 OUTPUT VERBOSITY

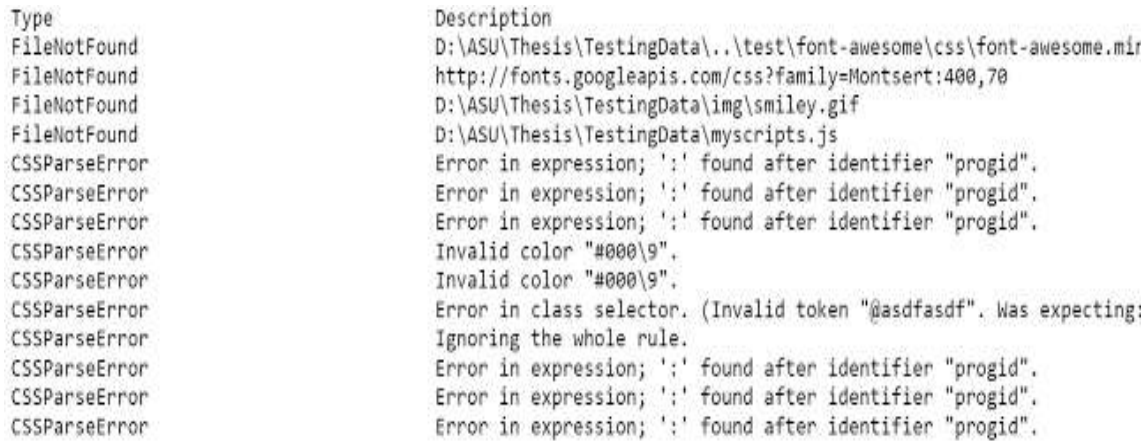
The tool supports various command line flags that help the user to modify the output as he/she desires. The list of supported flags are:

- `--help`: shows possible flags that can be used
- `--source`: pass the path of the root directory of the code to be analyzed with this flag, e.g., `--source=/User/home/testingdata/` .Note: the source directory must contain at least one HTML5 file.
- `--outputFormat`: supported formats are JSON or text, default value is text
- `--verbosity`: low, medium or high, default value is medium
- `--recommendations`: get suggestions to fix the defects, possible options are yes or no, default value is no
- `--rules`: comma separated list of rules to check. Possible options are `ParseError`, `ReferenceError`, `FileNotFound`, `Warnings` or all. The default value is all.
- `--outputFileName`: File name to save the output to. The default value is output.

Based on the flags selected, the output verbosity can be controlled. For example, under development environment the user can choose flags like `--verbosity=high --recommendations=on`, but for data extraction purposes using flags like `--verbosity=low` should be sufficient enough. Some of the other flags are discussed below.

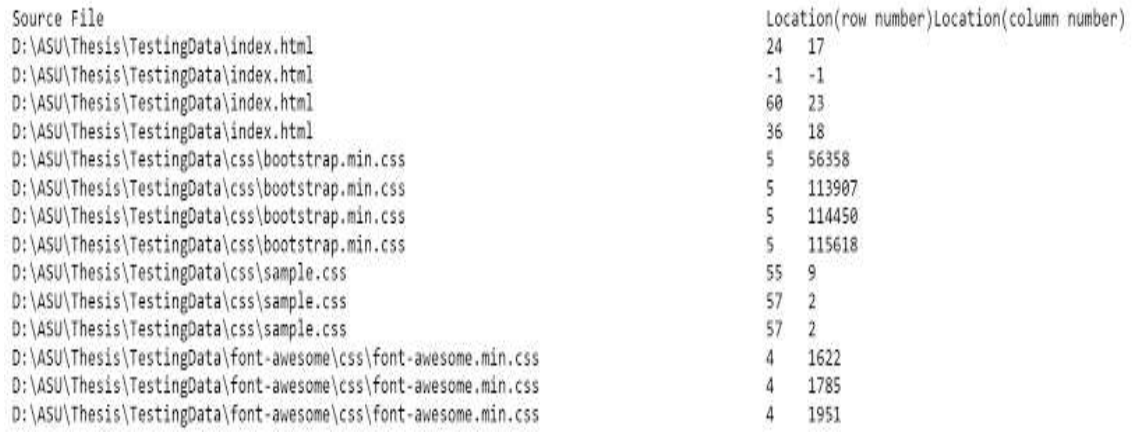
4.6 EXPORT RESULTS

The tool supports two formats to export results in: plain text and JavaScript Object Notation(JSON) format. The plain text is formatted into a column like structure to display the results in a neat way. The JSON format can be helpful in exporting the results and analyzing them in any other way. This is also useful if someone wants to expose the tool as a service-based component rather than directly using the tool. A snippet of plain text format results is shown below (the image has been cropped and displayed in two parts because of the length):



Type	Description
FileNotFound	D:\ASU\Thesis\TestingData\..\test\font-awesome\css\font-awesome.mi
FileNotFound	http://fonts.googleapis.com/css?family=Montsert:400,70
FileNotFound	D:\ASU\Thesis\TestingData\img\smiley.gif
FileNotFound	D:\ASU\Thesis\TestingData\myscripts.js
CSSParseError	Error in expression; ':' found after identifier "progid".
CSSParseError	Error in expression; ':' found after identifier "progid".
CSSParseError	Error in expression; ':' found after identifier "progid".
CSSParseError	Invalid color "#000\9".
CSSParseError	Invalid color "#000\9".
CSSParseError	Error in class selector. (Invalid token "@asdfasdf". Was expecting:
CSSParseError	Ignoring the whole rule.
CSSParseError	Error in expression; ':' found after identifier "progid".
CSSParseError	Error in expression; ':' found after identifier "progid".
CSSParseError	Error in expression; ':' found after identifier "progid".

Figure 15. Plain text format snippet Part 1



Source File	Location(row number)	Location(column number)
D:\ASU\Thesis\TestingData\index.html	24	17
D:\ASU\Thesis\TestingData\index.html	-1	-1
D:\ASU\Thesis\TestingData\index.html	60	23
D:\ASU\Thesis\TestingData\index.html	36	18
D:\ASU\Thesis\TestingData\css\bootstrap.min.css	5	56358
D:\ASU\Thesis\TestingData\css\bootstrap.min.css	5	113907
D:\ASU\Thesis\TestingData\css\bootstrap.min.css	5	114450
D:\ASU\Thesis\TestingData\css\bootstrap.min.css	5	115618
D:\ASU\Thesis\TestingData\css\sample.css	55	9
D:\ASU\Thesis\TestingData\css\sample.css	57	2
D:\ASU\Thesis\TestingData\css\sample.css	57	2
D:\ASU\Thesis\TestingData\font-awesome\css\font-awesome.min.css	4	1622
D:\ASU\Thesis\TestingData\font-awesome\css\font-awesome.min.css	4	1785
D:\ASU\Thesis\TestingData\font-awesome\css\font-awesome.min.css	4	1951

Figure 16. Plain text format snippet Part 2

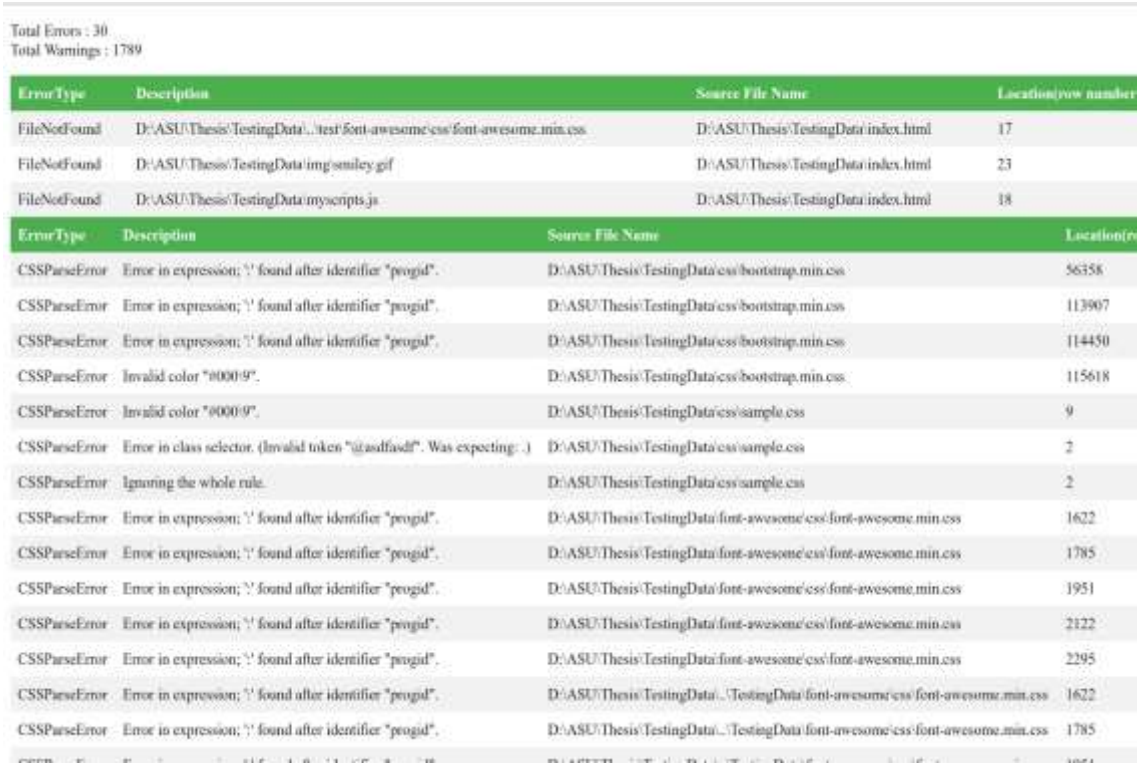
A snippet of the JSON format results is shown below:


```
{
    "ParseErrors": [{
        "errorType": "CSSParseError",
        "desc": "Error in expression; ':' found after identifier \"progid\".",
        "fileName": "D:\\ASU\\Thesis\\TestingData\\css\\bootstrap.min.css",
        "rowNumber": 5,
        "columnNumber": 56358
    }, {
        "errorType": "CSSParseError",
        "desc": "Error in expression; ':' found after identifier \"progid\".",
        "fileName": "D:\\ASU\\Thesis\\TestingData\\css\\bootstrap.min.css",
        "rowNumber": 5,
        "columnNumber": 113907
    }, {
        "errorType": "CSSParseError",
        "desc": "Error in expression; ':' found after identifier \"progid\".",
        "fileName": "D:\\ASU\\Thesis\\TestingData\\css\\bootstrap.min.css",
        "rowNumber": 5,
        "columnNumber": 114450
    }, {
        "errorType": "CSSParseError",
        "desc": "Invalid color \"#000\\9\".",
        "fileName": "D:\\ASU\\Thesis\\TestingData\\css\\bootstrap.min.css",
        "rowNumber": 5,
        "columnNumber": 115618
    }, {
        "errorType": "CSSParseError",
        "desc": "Invalid color \"#000\\9\".",
        "fileName": "D:\\ASU\\Thesis\\TestingData\\css\\sample.css",
        "rowNumber": 55,
        "columnNumber": 9
    }, {
        "errorType": "CSSParseError",
        "desc": "Error in class selector. (Invalid token \"@asdfasdf\". Was expecting: <IDENT>.)",
        "fileName": "D:\\ASU\\Thesis\\TestingData\\css\\sample.css",
        "rowNumber": 57,
        "columnNumber": 2
    }, {
        "errorType": "CSSParseError",
        "desc": "Ignoring the whole rule.",
        "fileName": "D:\\ASU\\Thesis\\TestingData\\css\\sample.css",
        "rowNumber": 57,
        "columnNumber": 2
    }, {
```


Figure 17. JSON format snippet

4.7 HTML VIEWER

Viewing the raw JSON results may not be very helpful for the user. Hence, the tool also supports an HTML Viewer that parses the generated JSON results and displays them as a pretty HTML page. It is recommended to use this viewer in a Firefox browser. Two snapshots of the HTML Viewer are shown below:

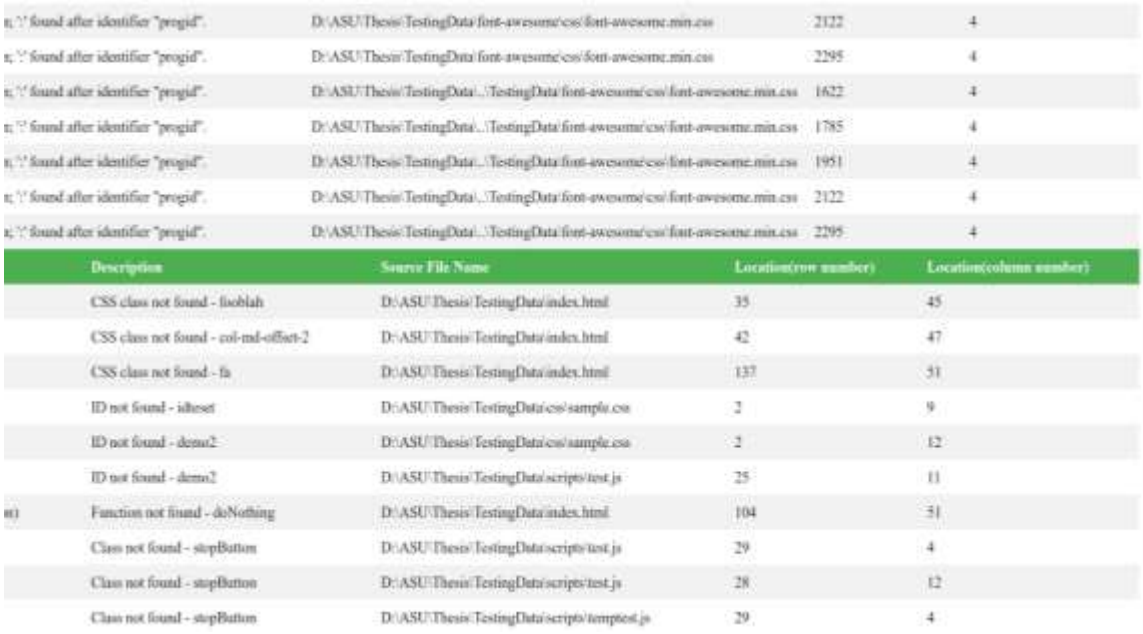


Total Errors : 30
Total Warnings : 1789

ErrorType	Description	Source File Name	Location(row number
FileNotFound	D:\ASU\Thesis\TestingData\..\test\font-awesome\css\font-awesome.min.css	D:\ASU\Thesis\TestingData\index.html	17
FileNotFound	D:\ASU\Thesis\TestingData\img\smiley.gif	D:\ASU\Thesis\TestingData\index.html	23
FileNotFound	D:\ASU\Thesis\TestingData\myscripts.js	D:\ASU\Thesis\TestingData\index.html	18

ErrorType	Description	Source File Name	Location(r
CSSParseError	Error in expression; '.' found after identifier "progid".	D:\ASU\Thesis\TestingData\css\bootstrap.min.css	56358
CSSParseError	Error in expression; '.' found after identifier "progid".	D:\ASU\Thesis\TestingData\css\bootstrap.min.css	113907
CSSParseError	Error in expression; '.' found after identifier "progid".	D:\ASU\Thesis\TestingData\css\bootstrap.min.css	114450
CSSParseError	Invalid color "#000\9".	D:\ASU\Thesis\TestingData\css\bootstrap.min.css	115618
CSSParseError	Invalid color "#000\9".	D:\ASU\Thesis\TestingData\css\sample.css	9
CSSParseError	Error in class selector. (Invalid token "@asdfasdf". Was expecting: .)	D:\ASU\Thesis\TestingData\css\sample.css	2
CSSParseError	Ignoring the whole rule.	D:\ASU\Thesis\TestingData\css\sample.css	2
CSSParseError	Error in expression; '.' found after identifier "progid".	D:\ASU\Thesis\TestingData\font-awesome\css\font-awesome.min.css	1622
CSSParseError	Error in expression; '.' found after identifier "progid".	D:\ASU\Thesis\TestingData\font-awesome\css\font-awesome.min.css	1785
CSSParseError	Error in expression; '.' found after identifier "progid".	D:\ASU\Thesis\TestingData\font-awesome\css\font-awesome.min.css	1951
CSSParseError	Error in expression; '.' found after identifier "progid".	D:\ASU\Thesis\TestingData\font-awesome\css\font-awesome.min.css	2122
CSSParseError	Error in expression; '.' found after identifier "progid".	D:\ASU\Thesis\TestingData\font-awesome\css\font-awesome.min.css	2295
CSSParseError	Error in expression; '.' found after identifier "progid".	D:\ASU\Thesis\TestingData\..\TestingData\font-awesome\css\font-awesome.min.css	1622
CSSParseError	Error in expression; '.' found after identifier "progid".	D:\ASU\Thesis\TestingData\..\TestingData\font-awesome\css\font-awesome.min.css	1785

Figure 18. HTML Viewer



s, ':' found after identifier "progid".	D:\ASU\Thesis\TestingData\font-awesome\css\font-awesome.min.css		2122	4
s, ':' found after identifier "progid".	D:\ASU\Thesis\TestingData\font-awesome\css\font-awesome.min.css		2295	4
s, ':' found after identifier "progid".	D:\ASU\Thesis\TestingData\...\TestingData\font-awesome\css\font-awesome.min.css		1622	4
s, ':' found after identifier "progid".	D:\ASU\Thesis\TestingData\...\TestingData\font-awesome\css\font-awesome.min.css		1785	4
s, ':' found after identifier "progid".	D:\ASU\Thesis\TestingData\...\TestingData\font-awesome\css\font-awesome.min.css		1951	4
s, ':' found after identifier "progid".	D:\ASU\Thesis\TestingData\...\TestingData\font-awesome\css\font-awesome.min.css		2122	4
s, ':' found after identifier "progid".	D:\ASU\Thesis\TestingData\...\TestingData\font-awesome\css\font-awesome.min.css		2295	4
	Description	Source File Name	Location(row number)	Location(column number)
	CSS class not found - faoblah	D:\ASU\Thesis\TestingData\index.html	35	45
	CSS class not found - col-md-offset-2	D:\ASU\Thesis\TestingData\index.html	42	47
	CSS class not found - fa	D:\ASU\Thesis\TestingData\index.html	137	51
	ID not found - idtest	D:\ASU\Thesis\TestingData\css\sample.css	2	9
	ID not found - demo2	D:\ASU\Thesis\TestingData\css\sample.css	2	12
	ID not found - demo2	D:\ASU\Thesis\TestingData\scripts\test.js	25	11
n)	Function not found - doNothing	D:\ASU\Thesis\TestingData\index.html	104	51
	Class not found - stopButton	D:\ASU\Thesis\TestingData\scripts\test.js	29	4
	Class not found - stopButton	D:\ASU\Thesis\TestingData\scripts\test.js	28	12
	Class not found - stopButton	D:\ASU\Thesis\TestingData\scripts\temptest.js	29	4

Figure 19. HTML Viewer

4.8 RULE-BASED ANALYSIS

The rule-based analysis flag basically helps the user to filter out the results and view them one category at a time. The categories were discussed as a part of error taxonomy in the previous chapter. Even though the output results are filtered as per the flag values, the internal computation is the same. This is because I do not want the analysis to disregard any dependency and miscalculate the errors. The supported values for this flag are:

- `ParseError`: this will filter out the results to show only parse errors encountered while parsing either HTML5, CSS3 or JavaScript.
- `ReferenceError`: this will filter out the results to show reference errors. Reference errors can be of various types: nonexistent class, nonexistent id, and nonexistent function.

- `FileNotFound`: this will filter out the results to show file not found errors. Note: this also checks for remote files.
- `Warnings`: this will filter out the results to show only warnings. Warnings are those dependencies that do not cause runtime errors.

4.9 RECOMMENDATION FOR FIXES

Because the research project focuses on syntactic dependencies in string identifiers, I also wanted the tool to be able to suggest fixes in case of reference errors generated due to nonexistent class and nonexistent ID. These errors arise because of a referenced class or ID does not exist in either the CSS3 or HTML5 file. Under the hypothesis that these errors are caused because of typographical errors in string identifiers, I use string matching algorithms to find the most similar string and suggest a fix to the user. The underlying algorithm used for string matching is called, Fuzzy distance similarity score [50]. This string matching algorithm is similar to the algorithms of editors such as Sublime Text, TextMate, Atom and others. The algorithm starts by comparing each character for the two given strings. For every character that matches, one point is given. And for every other match thereafter two bonus points are given. Thus, a higher score would indicate higher similarity. The string with the highest similarity is suggested as a fix for that particular defect. To illustrate a very simple example of this algorithm, let us consider two strings "`foobar`" and "`fooBar`". The user might accidentally type the first string as a class or ID. When this algorithm runs, it would give the highest score to "`fooBar`" as compared with other strings. Thus, we can see how this simple suggestion

might help the user fix the defect quickly.

4.10 INTEGRATED PLUGIN DEVELOPMENT FOR ECLIPSE

This plugin was integrated with Eclipse to provide the ease of use inside an Integrated Development Environment. Eclipse provides a Plug-in Development Environment (PDE) that helps developers to create plugins for Eclipse. These plugins are developed as a Rich Client Application (RCP). For the purpose of this research work, the plugin was made with a simple view based layout where the output of the analysis is shown in a tableview with separate rows and columns for each part of the result. The developer can also quickly jump to the location of the defect by double clicking on a particular row.

Using the implementation details discussed above, the tool that was developed was used for two different experiments to answer the research questions. The next chapter discusses the validation experiments and their results.

CHAPTER 5

VALIDATION

The research questions as discussed in previous chapters:

- RQ1: How significant are syntactic errors in string identifiers referencing DOM elements in the HTML5-JavaScript-CSS3 stack?
- RQ2: What is the delta (time or cost) between defect injection and defect discovery in new HTML5-JavaScript-CSS3 style applications?
- RQ3: Does a dedicated just-in-time syntactic string identifier resolution tool significantly reduce delta time/cost from RQ2 for a significant portion of real problems (RQ1)?

To address the research questions, I conducted two different experiments. The two different experiments target different research questions. The first experiment was to test the tool against real world codebases to answer RQ1. The second experiment involved conducting a user study to help answer RQ2 and RQ3 by focusing on measuring the delta between defect injection and defect discovery. Each of these experiments and their results are discussed below.

5.1 EXPERIMENT 1: TESTING AGAINST REAL WORLD CODEBASES

To test against real world codebases, the main task was to find repositories that did not use any JavaScript frameworks like jQuery, AngularJS, etc. GitHub was used as the source to find such repositories. GitHub also has “Issues” functionality which helped in

analyzing if dependency defects were triaged by the codebase owners. Once the code was locally cloned from the GitHub repository, I ran the tool against each one of them to find the number of dependency defects. Further, I analyzed the open issues to find out whether these defects were triaged or not. This experiment was conducted on 27th March 2016 and the results are presented in Table 2. The main inclusion criterion was to find codebases that did not use any JavaScript frameworks. Two of the three repositories used for this experiment were found in other literature as well. The Internet Explorer(IE) Test Suite was used by Jensen et. al [16] and the Google Octane Suite was used by Andreasen et al. [13] for their validations respectively. A list of all codebases that were considered for this experiment has been presented in Table 4 in Appendix B.

Repo Name	Tested File/Example	Errors	Warnings	Issues in Issue tracker	% of defects
TodoMVC	VanillaJS	2	22	3	40
IE Test Suite	@supports sample	1	0	24	4
	audiomixer	4	22	24	14.28
	blobbuilder	7	45	24	22.58
	chalkboard	1	7	24	4
	chess	17	40	24	41.46
	coloringbook	2	396	24	7.69
	compatinspector	0	0	24	0
	css3filters	11	11	24	31.42
	css3mediaqueries	1	12	24	4
	editingpasteimage	4	8	24	14.28
	eme	2	14	24	7.69
	familysearch	0	0	24	0
	fishbowl	2	20	24	7.69
	html5forms	1	65	24	4
	mandelbrot	1	15	24	4
	math	0	8	24	0
	mazesolver	5	16	24	17.24
	microphone	4	34	24	14.28
	musiclounge	0	3	24	0
	particleacceleration	2	3	24	7.69
	photocapture	0	16	24	0
	picture	1	0	24	4
	readingview	4	67	24	14.28
	setimmediatesorting	0	14	24	0
	spellchecking	0	2	24	0
	sudoku	2	101	24	7.69
	svgradientbackgroundmaker	5	36	24	17.24
	toucheffects	2	15	24	7.69
	typedarrays	3	18	24	11.11
	userselect	2	17	24	7.69
	videoformatsupport	1	19	24	4
	webaudiotuner	6	17	24	20
	webdriver	1	10	24	4
Google Octane Suite	Entire Suite(single html)	2	408	17	10.52

Table 2. Experiment on real world codebases

The defects discovered by my tool were not the same as the ones listed in the issue tracker of the repositories. On an average, 10.13% of the total defects were dependency related defects. Another interesting observation was that from the total 96 defects found, 65 of them were of the type HTML5toCSS3 dependency defects and 26 of them were CSS3toHTML5 dependency defects. This shows that HTML5toCSS3 and CSS3toHTML5 dependencies may not be easier to discover. The remaining defects included: 2 external file dependency defects and 3 CSS3 Parse errors. Further, many warnings were found showing that a lot of dead code exists in these codebases.

The main limitation of this experiment is that these set of codebases may not be an appropriate sample of the actual modern front end web development codebases, because they do not use any frameworks and libraries like jQuery, AngularJS, etc. This research currently supports only plain JavaScript. But the findings can be extended to codebases with frameworks as the nature of the dependencies stays the same regardless of the use of frameworks and libraries.

5.1.1 DISCUSSION

RQ1 as discussed in previous chapters:

- RQ1: How significant are syntactic errors in string identifiers referencing DOM elements in the HTML5-JavaScript-CSS3 stack?

The experiment discussed in section 5.1 was an attempt to answer RQ1. A detailed discussion about the prevalence of syntactic errors is not presented in this research. But running the tool against a very small subset of real world codebases shows that such defects are significant. The most interesting observation from the results of this

experiment is that none of these defects were triaged and reported in the issue trackers of these repositories. This shows that these defects might not be easy to triage when contributions to the codebase are made by multiple developers. HTML5toCSS3 and CSS3toHTML5 dependency defects were a major chunk of the total defects which shows that these dependencies specifically may not be easy to triage. A detailed analysis of such defects with a larger subset of real world codebases might present more interesting results. Across multiple developers and across various iterations of the code, many such defects might get injected and never be caught or triaged. Further, a number of warnings show that a significant amount of dead code exists in these codebases. Dead code analysis is not the focus of this research, but the existence of such warnings shows there are a lot of unmet dependencies that exist in the code and do not contribute to the functionality. This again shows that such dependencies are difficult to manage when the codebases are large, change with iterations of the product and are contributed to by various developers.

5.2 EXPERIMENT 2: USER STUDY

This experiment was conducted to answer RQ2 and RQ3. The experiment is focused on calculating the delta between defect injection and defect discovery times for dependency related defects.

5.2.1 PROTOCOL

The participants were asked to add new features to a given codebase. They were expected to spend at least 15 minutes and at most 20 minutes to understand the codebase that was

provided to them. The codebase provided to them was a single page bootstrap (http://getbootstrap.com/) template taken from GitHub: https://github.com/BlackrockDigital/startbootstrap-landing-page . The purpose of using this template was to provide a codebase that represents modern day front end codebases. The codebase was of very low complexity, with 234 lines of HTML5 and 170 lines of CSS3, excluding bootstrap CSS3 code. There was no JavaScript in the code, but as a part of the experiment, the participants had to create a new JavaScript file and write some JavaScript code. The JavaScript version 1.8 had to be used. While doing this coding exercise, the participants were expected to understand the application, understand the feature requirements, code the requirements, test them, report any defects (if any) and fix them. It was a self-paced exercise, but the participants were expected to spend at least 60 minutes on it. They were not allowed to use any external frameworks, and libraries, like jQuery, AngularJS, SASS or LESS. The participants were required to screen record their sessions using software on their machines. These recordings were later analyzed to gather data about defect injection and defect discovery. They had to record one video per feature. A total of 12 features had to be implemented. The features that the participants were supposed to implement are listed in Appendix A. These features were carefully designed to ensure that the participants inject dependency related defects so that the defect injection and defect discovery times could be captured. 4 features were designed not to inject any defect, for example:

- (Feature 4, Appendix C) Add a “See More” button in the portfolio! In line 172, add the button! Use `<a id="seemore" class="btn btn-default btn-lg"><span>See More</span></a>`

4 features were designed to definitely inject defects, for example:

- (Feature 6, Appendix C) Add some more text in “Portfolio” section! The see more button does not really do something above. Add some text that is shown once that button is clicked. Add a new `p` tag below the button in the portfolio section. Give it an id called “`portfoliodescription`”. In the method defined in step 5b, add code that will add some text to the `p` tag created in step 6a. So, in the method, add the following code:

  ```
  var elem = document.getElementById("portfolioDescription");
  elem.innerHTML = "This is some more description in the
  portfolio!";
  ```

4 features were designed to be open ended so that the participant might end up injecting defects, for example:

- (Feature 11, Appendix C) Add an Easter Egg! To the image added in step 10, add an `onclick` method. Ensure to define the method in the JavaScript file that you had created in step 2c. In this method definition, just add a `console.log` method call and pass the text: “You found the Easter Egg!”.

The reason for making these three categories was to ensure that we could gather data for the experiment. If all features were open ended, there could have been a chance of none of the participants injecting any defects.

5.2.2 ENVIRONMENT

The participants were asked to use Mac OS X as the operating system. They used the pre-installed QuickTime Player software to record their sessions. No audio recording was required for the session. It was important for the participants to record their sessions because the defect injection, defect discovery and defect fixing times needed to be observational and not self-reported. The limitations of prior work[38] and the findings from prior validation (section 2.6) extending that work was the reason to not use self-reported because it was not accurate. Also, these observational values were not done in person during the actual experiment to remove any bias in the participants' behavior. Both the groups were required to use Eclipse (Mars) for the experiment.

5.2.3 PARTICIPANTS

A pre-survey was conducted to recruit participants. The identity of the participants was kept anonymous. The pre-survey was used for exclusion of participants who could not answer two very basic questions about DOM access and CSS selectors. A total of 28 participants responded to the survey. 1 participant was excluded based on his pre-survey response. 6 participants withdrew after responding to the survey. 2 participants were excluded after the activity for not following the protocol. The participants were divided into two groups of equal skill set based on their responses to the survey questions. The group not using the tool is referred to as group A. The group using the tool is referred to as group B.

5.2.4 RESULTS

A total of 107 defects were injected by the participants out of which 28 were syntax defects, 7 were false positives, 4 were HTML5toJS dependency defects, 21 were HTML5toCSS3 dependency defects, 22 were external file dependency defects and 25 were JStoHTML5 dependency defects. The results discussed further do not include the syntax defects and the false positives. These defects were excluded from the discussion because they do not contribute towards dependency defects.

5.2.5 HTML5toJS DEFECTS

4 HTML5toJS defects were injected: 1 by a participant in group A and 3 by participants in group B. On average, the participants in group B were able to find these defects much quicker after injecting them. Figure 20 shows a visual representation of the delta values observed. Note, the values shown are log values of the actual delta observed. Log values are shown due to the range of the actual delta values. The raw data has been provided in Appendix C, Table 5.

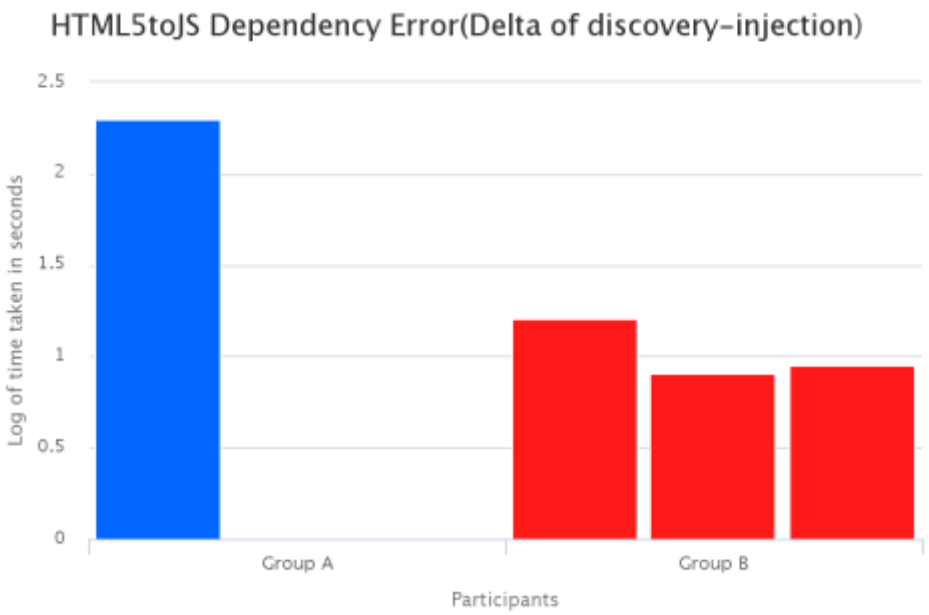


Figure 20. HTML5toJS Dependency Errors

The sample size for this particular category was too low, hence no statistical analysis was performed on this data set. The number of defects in this category is low because only one feature (feature 5, Appendix A) was associated with this category. Moreover, it asked the participants to create an `onclick` method and give it a specific name as instructed. This name had to be given in the HTML5 file and in the JavaScript file. Most participants copied the named from the HTML5 file to the JavaScript file, instead of typing the whole string identifier, thereby not injecting any defect. However, we do see that there is difference between the delta values for the two groups. The delta observed for the participant is group A was 197 seconds and the delta observed for participants in group B were 16, 9 and 8 seconds.

5.2.6 HTML5toCSS3 DEFECTS

Two different types of HTML5toCSS3 Defects were injected by the participants: the first type was injected by participants in both groups, but the second type was injected only by group B participants. The first type was injected by 10 participants in group A and 9 participants in group B. The second type was injected by 2 participants in group B. The first type was associated with feature 8 (Appendix B) and the second type was associated with feature 10 (Appendix B). The nature of these defects is the same, and hence, they are analyzed together, instead of analyzing the two types separately. The average delta observed was again much less for group B as compared to group A. The delta observed for group B was as low as 1 second. In 5 of the 10 cases in group A, participants did not even recognize that they had injected the defect. Because of this, those 5 data points were normalized for analysis. These participants did not actually discover the defect. Hence, the discovery time was replaced by the total time spent on the particular feature. Figure 21 shows a visual representation of the delta values observed. Note, the values in purple are the normalized values. The values shown are log values of the actual delta observed. Log values are shown due to the range of the actual delta values. The raw data has been provided in Appendix C, Table 6.

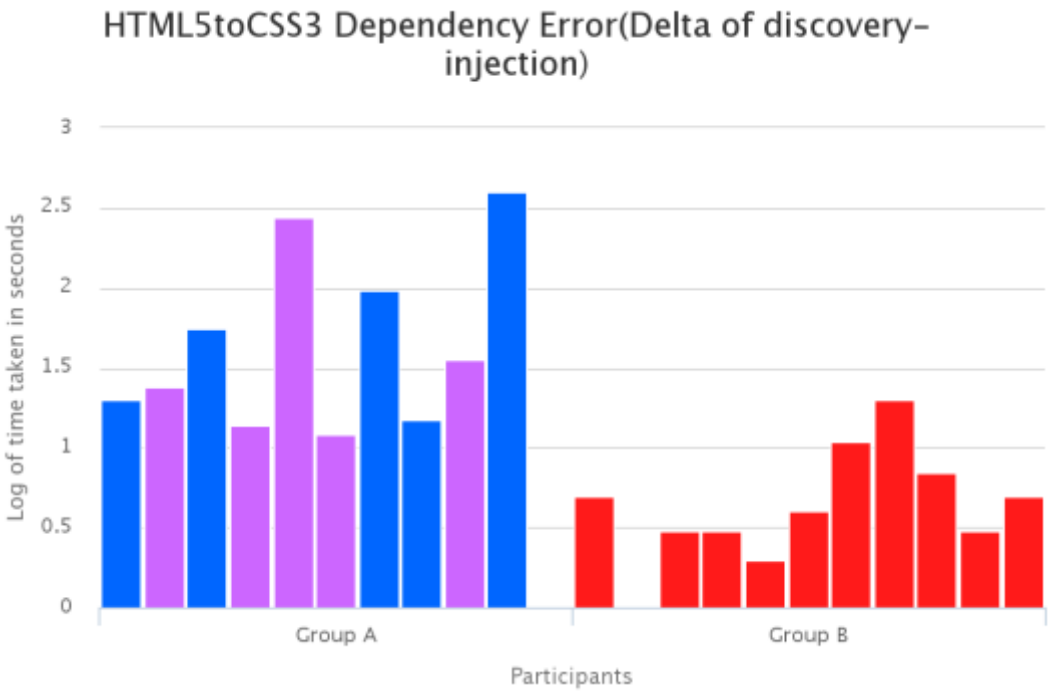


Figure 21. HTML5toCSS3 Dependency Errors

The statistical analysis of this data set results in a *p* value of 0.03. The null hypothesis for this experiment is that the means observed for the two groups are the same, suggesting the time taken is not related to using the tool. The alternate hypothesis is that the means are different, or that the time taken is related to using the tool. The observed *p* value shows strong evidence that the means observed for the two groups is different. Visual inspection of Figure 21 shows the difference readily. Note, only one group B data point is greater than any group A data points. Also, the second data point in group B is zero, which is `log(1)`. This participant took only 1 second to discover the defect after injecting it.

5.2.7 JStoHTML5 DEFECTS

Three different types of JStoHTML5 Defects were injected by the participants: the first two types were injected by participants in both groups, but the third type was injected only by group A participants. The first type was injected by 3 participants in group A and 5 participants in group B. The second type was injected by 6 participants in group A and 7 participants in group B. The third type was injected by 4 participants in group A. The first type is associated with feature 7 (Appendix B), the second type is associated with feature 6 (Appendix B) and the third type is associated with feature 12 (Appendix B). The nature of these defects is the same, hence, they are analyzed together, instead of analyzing the three types separately. The average delta observed was again much less for group B as compared to group A. Figure 22 shows a visual representation of the delta values observed. No normalization was required in this case. Note, the values shown are log values of the actual delta observed. Log values are shown due to the range of the actual delta values. The raw data has been provided in Appendix C, Table 7.

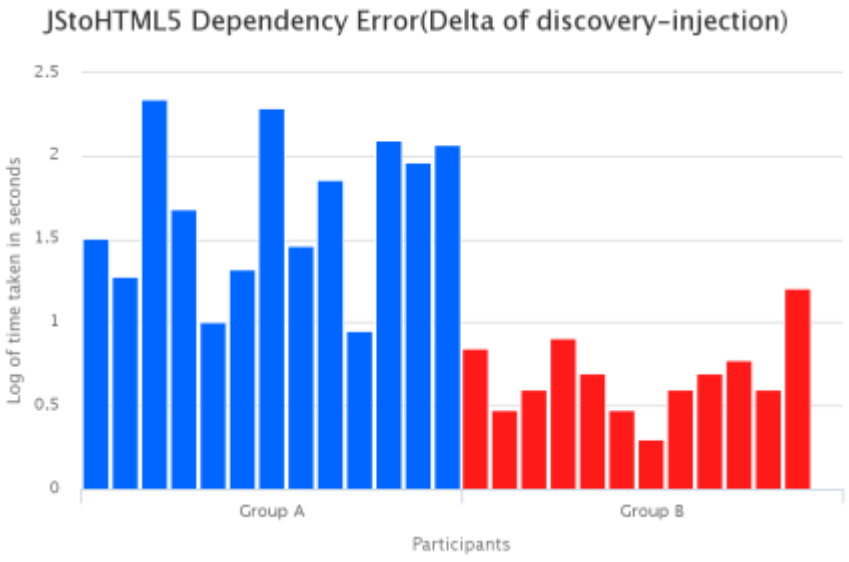


Figure 22. JStoHTML5 Dependency Errors

The statistical analysis of this data set results in a *p* value of 0.002. The null hypothesis for this experiment is that the means observed for the two groups are the same, suggesting the time taken is not related to using the tool. The alternate hypothesis is that the means are different, or that the time taken is related to using the tool. The *p* value shows strong evidence that the means observed for the two groups is different. Visual inspection of Figure 22 shows the difference readily. Note, only one group B data point is greater than any group A data points. Also, this data point is not the same as observed in section 5.2.6.

5.2.8 EXTERNAL FILE DEPENDENCY DEFECTS

Four different types of external file dependency defects were injected by the participants: the first two types were injected by participants in both groups, but the third type was injected only by group A participants and the fourth type was injected by only group B participants. The first type was injected by 10 participants in group A and 8 participants in group B. The second type was injected by 1 participant in group A and 1 participant in group B. The third type was injected by 1 participant in group A and the fourth type was injected by 1 participant in group B. The nature of these defects is the same, hence, they are analyzed together. The average delta observed was again much less for group B as compared to group A. Figure 23 shows a visual representation of the delta values observed. No normalization was required in this case. Note, the values shown are log values of the actual delta observed. Log values are shown due to the range of the actual delta values. The raw data has been provided in Appendix C, Table 8.

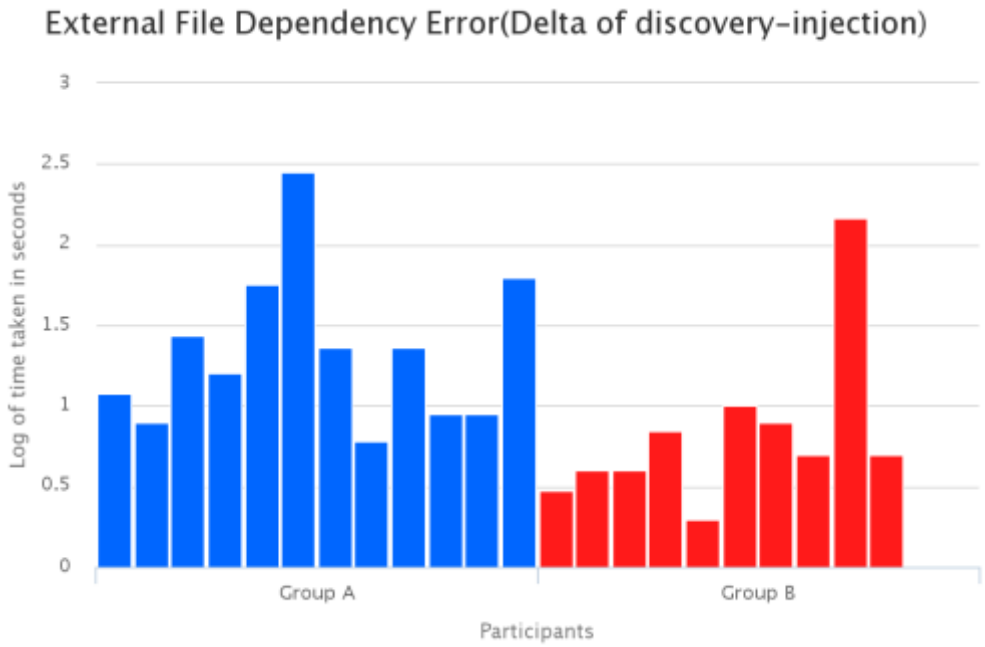


Figure 23. External File Dependency Errors

The statistical analysis of this data set results in a *p* value of 0.44. The null hypothesis for this experiment is that the means observed for the two groups are the same, suggesting the time taken is not related to using the tool. The alternate hypothesis is that the means are different, or that the time taken is related to using the tool. The observed *p* value shows very weak evidence that the means observed for the two groups is different.

5.2.9 OBSERVATIONS

There were certain patterns and behaviors of the participants that could be observed throughout this experiment. Some of this directly impacted the experiment, while the others did not. For example, the low number of defects observed in section 5.2.5 can be attributed to the fact that most participants preferred to copy string identifiers across the stack if they defined it. An example of this would be that if a participant defined an `onclick` method name is the HTML5 document, the participant just copies that name

into the JavaScript file. If it was a predefined string identifier, they would choose to type it in. An example of this would be a predefined CSS class provided with the codebase. Another interesting observation was that in the case of HTML5toCSS3 dependency defects, most participants in group A could not even identify that they had injected a defect. The JStoHTML5 defects were caused mainly because of DOM access. The participants were trying to access DOM identifier, which was either incorrect or did not exist. These defects are difficult to triage and take some time. Participants in group B were able to discover these defects in as low as 2 seconds. There was no pattern observed for fix recommendations provided to the participants using the tool. This can be partly attributed to the fact that the participants were student developers. Overall, it is very clear that this tool is very helpful for JStoHTML5 and HTML5toCSS3 dependencies. These represent a significant subset of the dependency errors that this research has been focusing on.

5.2.10 AGGREGATE STATISTICAL ANALYSIS

The average delta (discovery-injection) for each different category of defects is shown in Table 3. As can be seen from this table, group B saved approximately 52.75 seconds on average.

Defect Type	Average delta(discovery-injection) for group A in seconds	Average delta(discovery-injection) for group B in seconds	Section Number	Associated Feature Number(s), Appendix C	Observ-ed p value
HTML5toJS Defects	197	11	5.2.5	5	NA
HTML5toCSS3 Defects	116.4	5.81	5.2.6	8,10	0.03
JStoHTML5 Defects	75.84	5.58	5.2.7	6,7,12	0.002
External File Dependency Defects	44.5	19.6	5.2.8	2,9,10,12	0.44

Table 3. Average delta observed per defect type

Overall, the average delta observed for group B was 23.8 seconds compared to 76.5 for group A. Further, the *p* value computed was 0.08 which is greater than desired value of 0.05, but not greater than 0.1. The *d* value computed was 0.93, which signifies high effect size and medium high significance. These two values show that the results do present strong evidence that both the means observed are different. This means that the group using the plugin definitely saved a lot of time.

5.2.11 LIMITATIONS

First, the codebase selected was not very complex and the features to be added were also not very complex. Second, the study was conducted with student developers who may not be at the same skill level as the professional developers.

5.2.12 DISCUSSION

The research questions as discussed in previous chapters:

- RQ2: What is the delta (time or cost) between defect injection and defect discovery in new HTML5-JavaScript-CSS3 style applications?
- RQ3: Does a dedicated just-in-time syntactic string identifier resolution tool significantly reduce delta time/cost from RQ2 for a significant portion of real problems (RQ1)?

The experiment discussed in section 5.2 was an attempt to answer RQ2 and RQ3. The user study did not touch upon all of the dependencies as presented in the dependency model, but it focused on a subset. The results from this experiment show a clear difference in the two groups for the delta in defect discovery and defect injection. Even without the statistical analysis, it can be clearly seen from the visual representation of the data that the means are significantly different for the two groups. One very interesting observation from section 5.2.6 (HTML5toCSS3 dependency defects) is that certain participants never realized that they had injected a defect. This is very significant because CSS3 defects are no longer just cosmetic defects. CSS3 defects also cause functional defects. The next most significant results were observed for JStoHTML5 defects. This again is a significant subset of the dependencies because these represent the most common types of errors in front end web development: DOM access. The results for these two subsets show that such a dependency management tool is definitely helpful for developers. The aggregated statistical analysis shows a very high *d* value. This shows that the difference is significant. The graphical representation shows a clear difference in the

values for the two groups for all of the subsets. Overall, the results show that a dependency management tool is helpful for developers as it drastically decreases the delta (52.75 seconds as shown in section 5.2.10) between defect injection and defect discovery.

CHAPTER 6

CONCLUSION AND FUTURE WORK

Modern front end developers work with large scale front end web applications. The size of the codebase makes it a challenge for the developers to keep a track of the syntactic dependencies across the HTML5-JavaScript-CSS3 stack. A developer has to resort to using the developer tools within a browser and mentally keep a track of the dependencies that he/she is working with. A manual inspection of the developer tool and the running web application is prone to human error and as a result of which, a significant amount of defects due to syntactic dependencies go unnoticed. Further, the round trip between the code editor and the browser adds to the development time and cost. These factors when aggregated across an entire development team may result in significant ramifications towards development time and cost, and the final product. The time and effort saved per developer with the help of such a tool is significant. When aggregated over an entire development team, it can help save a lot of time and effort. Such an in-phase micro optimization of effort in modern day software engineering would help in drastically improving developer efficiency and developer productivity.

The dependencies discussed in this research exist in every front end web application. The results show a direct correlation with the modern front end web development industry. The results presented in section 5.1 show that none of the defects found by this tool were listed in the issue trackers of the codebases that were tested as a part of that experiment. Further, 10.13% of the defects were caused because dependency defects. This is a significant amount considering that the codebases were real world codebases. Further, the

results presented in section 5.2 show that a just-in-time syntactic dependency management tool is helpful for the front end web developers. Section 5.2.6 showed that HTML5toCSS3 dependency defects are difficult to triage. 5 of the 10 participants not using the tool could not discover the HTML5toCSS3 dependency defect. In the same section, the results show that the participants using the tool could discover the defect in as low as 1 or 2 seconds, whereas the lowest value (without normalizing) for the other group was 15 seconds. The low values for the group using the tool is almost real time. Section 5.2.7 discusses JStoHTML5 defects which essentially are caused due to incorrect DOM access. As discussed in section 2.4, DOM access are amongst the most common defects encountered in front end web applications. The results in section 5.2.7 show a very low *p* value of 0.002 which shows the impact of using the tool. DOM access defects are difficult to triage and can take a lot of time. Only one data point in the group using the tool was higher than all the values in the group not using the tool. This shows that triaging DOM access defects takes a lot of time. The aggregate statistical analysis results presented in section 5.2.10 show that the group using the tool saved 52.75 seconds on an average while dealing with syntactic dependency defects. This value is very high and when aggregated over an entire development team it will become very significant. Overall, the results show that a just-in-time syntactic dependency management tool helps the developer save time and effort while dealing with defects due to syntactic dependencies.

There are several directions in which this research work can be extended. To begin with the codebases considered for this research did not include any JavaScript/CSS3 frameworks or libraries, but regardless, the concept of the dependencies presented is agnostic of frameworks and libraries, and can be extended to include frameworks and libraries. Further, the experiments can be validated with a set of real world developers instead of student developers. Also, this research touches upon software engineering concepts like defect injection and defect discovery in relation to front end web development. Such software engineering concepts are usually studied with systems engineering in the context of full software development lifecycle models. This work can further be extended to more software engineering concepts like defect cycle in relation to front end web development. The results from section 5.1 also showed a lot of warnings that were generated because of syntactic dependencies, which implies that there is a lot of dead code in those applications. This work can be further extended to analyze the presence of dead code and how it impacts the loading time of those web applications. The tool can be further improved from a design perspective to be real-time as opposed to the current implementation of a just-in-time tool.

REFERENCES

[1] T. Berners-Lee, “Information management: A proposal,” Unpublished manuscript. 1989.

[2] A. Mesbah and A. van Deursen, “Invariant-based automatic testing of AJAX user interfaces,” in IEEE 31st International Conference on Software Engineering, 2009, pp. 210–220.

[3] “HTTP Archive.” [Online]. Available: http://httparchive.org/. [Accessed: 07-May-2016].

[4] “Alexa Top 500 Global Sites.” [Online]. Available: http://www.alexa.com/topsites. [Accessed: 07-May-2016].

[5] V. Gupta and K. A. Gary, “Dependency Analysis Tools for the Evolving Web Application Developer.” Unpublished manuscript.

[6] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, “An empirical study of client-side JavaScript bugs,” in Empirical Software Engineering and Measurement, ACM/IEEE International Symposium on, 2013, pp. 55–64.

[7] M. E. Fagan, “Advances in Software Inspections,” in Pioneers and Their Contributions to Software Engineering, M. Broy and E. Denert, Eds. Springer Berlin Heidelberg, 2001, pp. 335–360.

[8] F. S. Ocariza Jr, K. Pattabiraman, and A. Mesbah, “Vejovis: Suggesting fixes for JavaScript faults,” in Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 837–847.

[9] B. Boehm and V. R. Basili, “Top 10 list [software development],” Computer, vol. 34, no. 1, Jan. 2001, pp. 135–137.

[10] F. S. Ocariza Jr, K. Pattabiraman, and B. Zorn, “JavaScript errors in the wild: An empirical study,” in Software Reliability Engineering, IEEE 22nd International Symposium on, 2011, pp. 100–109.

[11] A. Gizas, S. Christodoulou, and T. Papatheodorou, “Comparative evaluation of javascript frameworks,” in Proceedings of the 21st international conference companion on World Wide Web, 2012, pp. 513–514.

[12] D. Graziotin and P. Abrahamsson, “Making sense out of a jungle of JavaScript frameworks,” in Product-Focused Software Process Improvement, Springer, 2013, pp. 334–337.

[13] E. Andreasen and A. Møller, “Determinacy in static analysis for jQuery,” in ACM SIGPLAN Notices, 2014, vol. 49, pp. 17–31.

[14] K. Bajaj, K. Pattabiraman, and A. Mesbah, “Dompletion: DOM-aware JavaScript code completion,” in Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp. 43–54.

[15] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, “Effective smart completion for JavaScript,” Technical Report RC25359, IBM Research, 2013.

[16] S. H. Jensen, M. Madsen, and A. Møller, “Modeling the HTML DOM and browser API in static analysis of JavaScript web applications,” in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, 2011, pp. 59–69.

[17] M. Madsen, B. Livshits, and M. Fanning, “Practical static analysis of JavaScript applications in the presence of frameworks and libraries,” in Proceedings of the 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 499–509.

[18] I. Medeiros, N. F. Neves, and M. Correia, “Automatic detection and correction of web application vulnerabilities using data mining to predict false positives,” in Proceedings of the 23rd international conference on World wide web, 2014, pp. 63–74.

[19] T. Scholte, W. Robertson, D. Balzarotti, and E. Kirda, “Preventing input validation vulnerabilities in web applications through automated type analysis,” in Computer Software and Applications Conference, IEEE 36th Annual, 2012, pp. 233–243.

[20] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, “Finding bugs in web applications using dynamic test generation and explicit-state model checking,” Softw. Eng. IEEE Trans. On, 2010, vol. 36, no. 4, pp. 474–494.

[21] L. K. Shar, L. C. Briand, and H. B. K. Tan, “Web Application Vulnerability Prediction Using Hybrid Program Analysis and Machine Learning,” Dependable Secure Comput. IEEE Trans. On, 2015, vol. 12, no. 6, pp. 688–707.

[22] L. K. Shar, H. B. K. Tan, and L. C. Briand, “Mining SQL Injection and Cross Site Scripting Vulnerabilities Using Hybrid Program Analysis,” in Proceedings of the International Conference on Software Engineering, Piscataway, NJ, USA, 2013, pp. 642–651.

[23] N. Jovanovic, C. Kruegel, and E. Kirda, “Pixy: A static analysis tool for detecting web application vulnerabilities,” in Security and Privacy, IEEE Symposium on, 2006, p. 6–pp.

[24] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, “Securing web application code by static analysis and runtime protection,” in Proceedings of the 13th international conference on World Wide Web, 2004, pp. 40–52.

[25] E. Andreasen, A. Feldthaus, S. H. Jensen, C. S. Jensen, P. A. Jonsson, M. Madsen, and A. Møller, “Improving Tools for JavaScript Programmers,” in Proc. of International Workshop on Scripts to Programs. Beijing, China:[sn], 2012, pp. 67–82.

[26] S. H. Jensen, A. Møller, and P. Thiemann, “Type analysis for JavaScript,” in Static Analysis, Springer, 2009, pp. 238–255.

[27] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip, “Refactoring Towards the Good Parts of Javascript,” in Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, New York, NY, USA, 2011, pp. 189–190.

[28] S. Artzi, J. Dolby, S. H. Jensen, A. Moller, and F. Tip, “A framework for automated testing of javascript web applications,” in 33rd International Conference on Software Engineering, 2011, pp. 571–580.

[29] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, “Dynamic determinacy analysis,” ACM SIGPLAN Not., vol. 48, no. 6, pp. 165–174, 2013.

[30] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip, “Correlation tracking for points-to analysis of JavaScript,” in European Conference on Object-Oriented Programming, Springer, 2012, pp. 435–458.

[31] M. Bosch, P. Geneves, and N. Layaïda, “Reasoning with style,” in International Joint Conference On Artificial Intelligence, 2015.

[32] D. Mazinanian, N. Tsantalis, and A. Mesbah, “Discovering refactoring opportunities in cascading style sheets,” in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 496–506.

[33] M. Bosch, P. Genevès, and N. Layaïida, “Automated refactoring for size reduction of CSS style sheets,” in Proceedings of the ACM symposium on Document engineering, 2014, pp. 13–16.

[34] M. Hague, A. W. Lin, and C.-H. L. Ong, “Detecting redundant CSS rules in HTML5 applications: a tree rewriting approach,” in Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2015, pp. 1–19.

[35] P. Geneves, N. Layaida, and V. Quint, “On the analysis of cascading style sheets,” in Proceedings of the 21st international conference on World Wide Web, 2012, pp. 809–818.

[36] A. Mesbah and S. Mirshokraie, “Automated analysis of CSS rules to support style maintenance,” in Software Engineering, 34th International Conference on, 2012, pp. 408–418.

[37] F. S. Ocariza Jr, K. Pattabiraman, and A. Mesbah, “AutoFLox: An automatic fault localizer for client-side JavaScript,” in Software Testing, Verification and Validation, IEEE Fifth International Conference on, 2012, pp. 31–40.

[38] V. Gupta, “Dependency Analysis in the HTML5, JavaScript and CSS3 Stack,” M.S. Thesis, Arizona State University, 2014.

[39] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?,” in Software Engineering, 35th International Conference on, 2013, pp. 672–681.

[40] “How browsers work.” [Online]. Available: http://taligarsiel.com/Projects/howbrowserswork1.htm. [Accessed: 15-Jun-2016].

[41] “jsoup Java HTML Parser, with best of DOM, CSS, and jquery.” [Online]. Available: https://jsoup.org/. [Accessed: 08-May-2016].

[42] WHAT-WG, “HTML Standard.” [Online]. Available: https://html.spec.whatwg.org/multipage/. [Accessed: 07-May-2016].

[43] “Matcher (Java Platform SE 8).” [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html. [Accessed: 07-May-2016].

[44] “Can you provide some examples of why it is hard to parse XML and HTML with a regex? - Stack Overflow.” [Online]. Available: http://stackoverflow.com/questions/701166/can-you-provide-some-examples-of-why-it-is-hard-to-parse-xml-and-html-with-a-reg. [Accessed: 07-May-2016].

[45] “CSS Parser – Welcome to CSS Parser.” [Online]. Available: http://cssparser.sourceforge.net/. [Accessed: 07-May-2016].

[46] “Document Object Model (DOM) Level 2 Style Specification.” [Online]. Available: https://www.w3.org/TR/2000/REC-DOM-Level-2-Style-20001113/. [Accessed: 07-May-2016].

[47] “SAC: The Simple API for CSS.” [Online]. Available: https://www.w3.org/Style/CSS/SAC/. [Accessed: 07-May-2016].

[48] “Oracle Nashorn: A Next-Generation JavaScript Engine for the JVM.” [Online]. Available: http://www.oracle.com/technetwork/articles/java/jf14-nashorn-2126515.html. [Accessed: 07-May-2016].

[49] “Nashorn: The New Rhino on the Block,” don’t code today what you can’t debug tomorrow, 31-Mar-2014. [Online]. Available: http://ariya.ofilabs.com/2014/03/nashorn-the-new-rhino-on-the-block.html. [Accessed: 07-May-2016].

[50] “StringUtils (Apache Commons Lang 3.5-SNAPSHOT API).” [Online]. Available: https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/StringUtils.html#getFuzzyDistance(java.lang.CharSequence,%20java.lang.CharSequence,%20java.util.Locale). [Accessed: 15-Jun-2016].

APPENDIX A

LIST OF FEATURES TO BE IMPLEMENTED IN USER STUDY

1. Change the text! (find these in the HTML and edit directly)
    a. On the top left of the screen, change the text from “Start Bootstrap” to “Awesome Startup!”
    b. Change the text on the center of the screen that says “Landing Page” to “Welcome!” and the text that says “A Template by Start Bootstrap” to “To the awesome Startup!”
    c. At the bottom, change the text that says “Connect to Start Bootstrap” to “Connect with Us”.
    d. Finally, change the copyright text at the bottom from “Copyright © Your Company 2014” to “Copyright © Awesome Startup 2016”
2. Change the links! The links for Twitter, GitHub and LinkedIn are pointing to places we do not want! Instead, let’s just redirect all those links to my.asu.edu.
    a. First, find and remove the links in `href` for each of these buttons.
    b. Add `onclick` methods to each of these buttons.
    c. Next, create a new JavaScript file and include it inside the head tag. (hint: `<script src="jsfilename.js"></script>`)
    d. In this new JavaScript file, define the `onclick` methods called in step b. Add code to open a new window that redirects to https://www.asu.edu. (hint: use `window.open("https://www.asu.edu");`)
3. Add a portfolio section! In the top right of the html page, add a new list item called portfolio.

a. After line 55 in index.html, add the following (copy and paste, and ensure to keep the new lines and indents, you do not want to be a bad programmer!)

```
<li>
  <a href="#portfolio">Portfolio</a>
</li>
```

b. Now, you need to create a new section called portfolio. On line 166, add the following (again, keep the new lines and indents)

```
<a name="portfolio"></a>
  <div class="content-section-b">
    <div class="container">
      <div class="row">
        <p>This is our portfolio!</p>
      </div>
    </div>
  </div>
```

4. Add a “See More” button in the portfolio!
    a. In line 172, add the button! Use `<a id="seemore" class="btn btn-default btn-lg"><span>See More</span></a>`
5. On clicking “See More”, add some text in the portfolio section!
    a. Add an `onclick` method in the button above. Name the method “`seemoreBtnClick`”.
    b. In your JavaScript file, add method called “`seeMoreBtnClick`”.

6. Add some more text in “Portfolio” section! The see more button does not really do something above. Add some text that is shown once that button is clicked.
    a. Add a new `p` tag below the button in the portfolio section. Give it an id called “`portfoliodescription`”.
    b. In the method defined in step 5b, add code that will add some text to the p tag created in step 6a. So, in the method, add the following code :

```
var elem =
document.getElementById("portfolioDescription");
elem.innerHTML = "This is some more description in the
portfolio!";
```

7. Change the text of button whenever it is clicked! The “See More” button should say “Show Less” when clicked.
    a. In the method defined in step 5b, get a reference to the “See More” button by using the `getElementById` method in the document object. Further, get a reference to the span tag and update the `innerText` of the `span` tag to say “Show Less”.
8. The font does not look good! In the portfolio section, the font of the text looks odd and does not fit in with the whole website. Let’s use one of the pre-built fonts and add it to this section.
    a. Add the class “`lea`” to the `p` tags in the portfolio section. Just change the HTML directly!
9. The background does not look good! You do not like the background! The background does not look good! Let’s just go ahead and change it.

a. Open `css/combined.css` file.

b. Change line 768 from "`background: url(../img/intro-bg.jpg) no-repeat center center;`" to "`background: url(../img/landscape-nature-sunset-trees.jpg) no-repeat center center;`"

10. Add images to your portfolio! Add an image to the portfolio section, just below the p tag created in step 6a.

    Use the following code:

```
<div class="col-lg-5 col-lg-offset-2 col-sm-6">
<img class="img-responsive" src="img/banner-bg.png"
alt="">
</div>
```

11. Add an Easter Egg! To the image added in step 10, add an `onclick` method. Ensure to define the method in the JavaScript file that you had created in step 2c. In this method definition, just add a `console.log` method call and pass the text: "You found the Easter Egg!".

12. Add another Easter Egg! If someone is able to find the Easter egg that you created in step 11, change the color of the "See More" button. It can be any color that you wish! Just create a new CSS file, include it in your HTML file, create a new class in that CSS file that sets the background color of the button. In the `onclick` method defined in step 11, get a reference to the "See More" button by using the document object, and simply add the new class that you just created to the button.

APPENDIX B

CODEBASES CONSIDERED FOR RQ1 VALIDATION

S.no.	Name	URL	Used
1	Chrome experiments	https://www.chromeexperiments.com/	No
2	Internet explorer test drive	https://dev.modern.ie/testdrive/	Yes
3	10k apart challenge	http://10k.aneventapart.com/	No
4	Google octane suite	https://developers.google.com/octane/?hl=en	Yes
5	Sun spider suite	https://www.webkit.org/perf/sunspider/sunspider.html	No
6	Jetstream	http://browserbench.org/jetstream/	No
7	Pdfjs	https://github.com/mozilla/pdf.js	No
8	Ajax im	http://ajaxim.com/	No
9	Todo mvc	http://todomvc.com/	Yes

Table 4. Codebases considered for RQ1 validation

APPENDIX C

EXPERIMENT TWO: RAW DATA

Group	Feature #	Delta (Discovery - Injection)	Log of Delta column
A	5	197	2.294466226
B		16	1.204119983
B		8	0.903089987
B		9	0.9542425094

Table 5. HTML5toJS Dependency Error Raw Data

Group	Feature #	Delta (Discovery - Injection)	Log of Delta column
A	8	20	1.301029996
A		24	1.380211242
A		55	1.740362689
A		14	1.146128036
A		275	2.439332694
A		12	1.079181246
A		96	1.982271233
A		15	1.176091259
A		36	1.556302501
A		396	2.597695186
B		5	0.6989700043
B		1	0
B		3	0.4771212547
B		3	0.4771212547
B		2	0.3010299957
B		4	0.6020599913
B		11	1.041392685
B		20	1.301029996
B		7	0.84509804
B	10	3	0.4771212547
B		5	0.6989700043

Table 6. HTML5toCSS3 Dependency Error Raw Data

Group	Feature #	Delta (Discovery - Injection)	Log of Delta column
A	7	32	1.505149978
A		19	1.278753601
A		218	2.338456494
A	6	48	1.681241237
A		10	1
A		21	1.322219295
A		195	2.290034611
A		29	1.462397998
A		72	1.857332496
A	12	9	0.9542425094
A		124	2.093421685
A		92	1.963787827
A		117	2.068185862
B	7	7	0.84509804
B		3	0.4771212547
B		4	0.6020599913
B		8	0.903089987
B		5	0.6989700043
B	6	3	0.4771212547
B		2	0.3010299957
B		4	0.6020599913
B		5	0.6989700043
B		6	0.7781512504
B		4	0.6020599913
B		16	1.204119983

Table 7. JStoHTML5 Dependency Error Raw Data

Group	Feature #	Delta (Discovery - Injection)	Log of Delta column
A	10	12	1.079181246
A		8	0.903089987
A		27	1.431363764
A		16	1.204119983
A		57	1.755874856
A		281	2.44870632
A		23	1.361727836
A		6	0.7781512504
A		23	1.361727836
A		9	0.9542425094
A	9	9	0.9542425094
A	2	63	1.799340549
B	10	3	0.4771212547
B		4	0.6020599913
B		4	0.6020599913
B		7	0.84509804
B		2	0.3010299957
B		10	1
B		8	0.903089987
B		5	0.6989700043
B	12	148	2.170261715
B	2	5	0.6989700043

Table 8. External File Dependency Error Raw Data