Software Analysis for the Web: Achievements and Prospects

(Invited Paper)

Ali Mesbah
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

***Abstract*—The web has had a significant impact on our lives. A technology that was initially created for sharing documents across the network has evolved into a strong medium for developing and distributing software applications. In this paper, we first provide a concise overview of the evolution of the web itself. We then focus on some of the main industrial and research achievements in software analysis and testing techniques geared toward web apps, in the past two decades. We discuss static, dynamic, and hybrid analyses approaches, software testing and test adequacy techniques, as well as techniques that help developers write, analyze and maintain their code. Finally, we present some of the current and future challenges and research opportunities ahead in this field.**

I. Introduction

Many existing software systems have been and continue to be migrated to the web, and many new domains are developed, thanks to the open and ubiquitous nature of the web. The web has emerged as a successful platform for software development and delivery with benefits that include no installation costs, automatic upgrades for all users, and universal access and execution from any machine with Internet access, through the web browser.

The web is one of the most fascinating inventions of our time. It has had a significant impact on many aspects of our society, from business, education, government, entertainment, to our social and personal lives. It provides a unique platform for software engineering with benefits that include instantaneous delivery for all users and "write once, run anywhere" through universal access and execution from any device connected to the Internet. Because of these benefits, any application that can be written as a web app, is likely to eventually be written as a web app. Web-based services such as GitHub and StackOverflow have already revolutionized the way developers write software. Many existing software systems have been and continue to be migrated to the web, and many new domains are developed, thanks to the open and ubiquitous nature of the web.

The web is an example of a continuously evolving technology that causes its applications to age quickly. Unlike traditional software, web apps are heterogeneous (i.e., JavaScript, CSS, HTML, etc). The dynamic inter-dependencies between these languages, and their distributed asynchronous client/server nature, pose many challenges for developers.

In 2007, Jazayeri [47] wrote a Future of Software Engineering (FOSE) paper in which he presented and predicted some trends in web app development.

Today, many of those predictions have come true, but more interestingly, the web has evolved even further and faster than anyone could have imagined. For instance, JavaScript has become the most popular language on GitHub [37] and a recent survey of more than 26K developers conducted by Stack Overflow found that JavaScript is the most-used programming language [110]. JavaScript is now used not only inside the browser, but also as a language for building desktop and server-side applications, thanks to the Node.js platform. After almost 10 years since Jazayeri's paper, we set out to explore the achievements and prospects of the software engineering research for the web.

In this paper, we focus on two closely related software engineering areas, namely, software analysis and software testing. We look at the research in the past two decades, but mainly focus on modern web apps, highlighting some of the main challenges they post and recent achievements in techniques and tools for web analysis and testing. Finally, we discuss potential future research directions.

We, by no means, claim that this paper represents all the relevant and noteworthy research performed in the area of web analysis and testing. Nevertheless, we hope that this paper will provide an overview of some of the achievements and potential challenging areas for future exploration.

We organize the rest of this paper as follows. Section II depicts a concise overview of the evolution of the web, from its inception to today. Section III discusses some of the technical challenges posed by modern web apps today for analysis and testing. Section IV presents research advancements and industrial tools for web analysis and testing. Section V provides additional areas in which opportunities and challenges exist for future research. Finally, Section VI concludes the paper.

II. The Web Evolution

A. Static Hypertext Documents

Around three decades ago, a simple but powerful client/server architecture [33] was developed, in which resources could be linked together and easily accessed through web browsers, using a handful of crucial concepts, such as the Uniform Resource Identifier (URI), HyperText Markup Language (HTML), and HyperText Transfer Protocol (HTTP) [17]. In the early nineties, the web was merely composed of linked static hypertext documents. Upon sending a request to the server, the server would simply locate and retrieve the corresponding web page on the file-system, and send it back to the browser. The browser would then use the new web page to refresh the entire interface.

B. Dynamically Generated Pages

After a wider adoption of the web, more complex web apps began to flourish, moving from static pages on the file-system to dynamically generated pages served by smarter web servers. The first server-side dynamic web pages were often created with the help of languages such as Perl, typically though the Common Gateway Interface (CGI). As the web matured, more server-side scripting languages appeared, such as PHP, Python, Ruby, JavaServer Pages (JSP), and Active Server Pages (ASP). The ability to generate web pages contributed to the separation of concerns (presentation, business logic, data) and realization of multi-tier architectures for web apps.

C. REST Architecture

At this stage, many of the initial concepts forming the backbone of the web (e.g., HTTP, URL, HTML) and additional recommendations such as Cascading Style Sheets (CSS) and the Document Object Model (DOM) were standardized through the World Wide Web Consortium (W3C). Additionally, an architectural style of the web, called the Representational State Transfer (REST), was proposed and interestingly published first at ICSE 2000 [33], capturing the essence of the main features of the web architecture. REST specified a layered client-stateless-server architecture in which each request is independent of the previous ones, inducing the property of scalability.

D. Rich Internet Applications

Soon it became apparent that HTML was not designed for creating interactive Graphical User Interfaces (GUI). Classical web apps are, inherently, based on a *multi-page* user interface model, in which interactions are based on a synchronous page-sequence paradigm. While simple and elegant in design for exchanging documents, this model has many limitations for developing modern web apps with user friendly human-computer interaction. The main limitations can be summarized as follows:

- Low lever of user interactivity;
- Redundant data transfer between the client/server;
- High user-perceived latency;
- Passive browser engine: there is hardly any application-specific client-side processing.

The concept of Rich Internet Applications (RIA) was proposed [6] as a response to these issues. The common ground for all RIAs is an intermediate layer of code introduced between the user and the server, which acts as an extension of the browser, usually taking over responsibility of server communication and rendering the web user interface. Examples include Adobe Flex (based on Flash), and Microsoft Silverlight. One of the main issues with such technologies was their non-standard, proprietary nature that required the installation of specific plugins by the user. [1]

E. JavaScript and Dynamic DOM

By the year 2005, web browsers had advanced their support for technologies such as the DOM, and JavaScript, which allowed developers to mitigate some of the limitations of heavy-weight RIAs. The term Ajax, was coined to highlight and give a name to a new breed of web apps that could be seen as a further evolution of the classical web:

- standards-based presentation using XHTML and CSS;
- dynamic display and interaction using the DOM;
- data interchange and manipulation using XML/JSON;
- asynchronous data retrieval using XMLHttpRequest (XHR);
- and JavaScript binding everything together.

This brought an end to the classical *click-and-wait* style of web navigation, enabling developers to provide the level of responsiveness and interactivity end-users expected from desktop applications. With Ajax, small updates are requested from the server behind the scenes, and updated on the current page through modification made by JavaScript code to the DOM-tree. This in sharp contrast to the classical *multi-page* style, in which after each state change a completely new DOM-tree is created from a full page reload.

This new model enables *single-page* web interfaces, which can improve complex, non-linear user workflows by decreasing the number of click trails and the time needed to perform a certain task, when compared to classical multi-page variants. Examples of early adopters of this model include Gmail and Google Docs.

By 2010, more and more of the state of the web app started was being off-loaded to the client, transforming JavaScript from an often neglected scripting language that was used for small client-side validation, to a popular programming language for building sophisticated large applications.

III. Current Challenges

The web is an excellent example of an evolving technology that causes its applications to age. It started as a simple static page-sequence client/server system. web apps based on the classical model of the web and the technologies available in

[1]Note that Java Applets in the nineteens were a previous not-so-successful attempt at targeting the same issue.

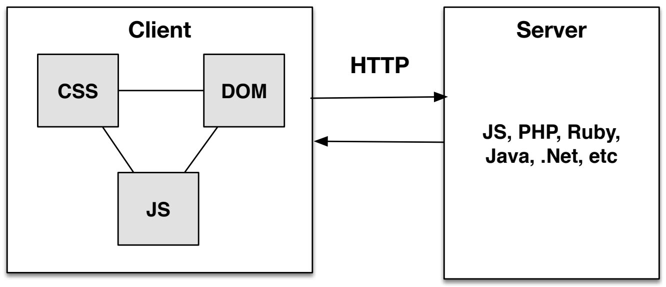

Fig. 1. web apps are heterogenous and distributed: different languages need to interact, some over the network, to realize a web app.

the early nineties, have aged and become out-dated. Over the course of the past 20 years, many web technologies (e.g., browsers, servers, web standards) have evolved. These technological advancements have made it possible to develop web systems that meet up with current user expectations, i.e., a satisfactory degree of responsiveness and interactivity.

The new changes in the evolution of the web bring not only advantages but also come with a whole set of new challenges due to its heterogeneous and distributed nature, depicted in Figure 1. Three languages, namely JavaScript, CSS, and HTML/DOM interact on the client side internally, and over the network through HTTP with at least one other language in the server side (e.g., JavaScript, PHP, Ruby, Java, etc). What we witness is that web apps built with the new models and technologies are not supported by existing tools (e.g., web crawlers), techniques (e.g., web analysis and testing), and development environments (e.g., IDEs) that have been typically slower to evolve.

Modern web apps are particularly challenging to deal with for developers, which we describe in the next subsections.

A. *JavaScript: the Difficult Parts*

JavaScript has become the lingua franca of implementing the client-side of modern web apps. It is also becoming popular as a desktop application and server-side language thanks to the Node.js platform [3].

JavaScript has a dynamic, weakly-typed, and asynchronous nature. For instance, constructs such as `eval` allow text to turn into executable code at runtime, forming a serious hindrance for providing static guarantees about the behaviour of the code [99] as well as performing proper instrumentation of the code for dynamic analysis. The weakly-type nature of the language makes it particularly challenging to apply many stablished analysis techniques that work on statically typed languages such as Java. The language also enables asynchronous callbacks through mechanism such as `setTimeout` and the XMLHttpRequest (XHR) for server communication, which are error-prone [45] and difficult to follow [5]. In addition, JavaScript is an interpreted language, meaning that there is typically no compiler that would help developers to detect erroneous or unoptimized code during development.

JavaScript also has intricate features such as prototypes [94], first-class functions, function variadicity, and closures [23]. Prototype-based programming is a class-free style of object-oriented programming, in which objects can inherit properties from other objects directly. In JavaScript, prototypes can be redefined at runtime, and immediately affect all the referring objects. The language has a very flexible model of objects and functions. Object properties and their values can be created, changed, or deleted at runtime and accessed via first-class functions. Due to such flexibility, the set of all available properties of an object is not easily retrievable statically. This poses a major challenge for scalable and precise static analysis for JavaScript [107], [31]. Empirical studies [100], furthermore, have found that most dynamic features in JavaScript are frequently used by developers and cannot be disregarded in code analysis techniques.

B. *The Dynamic DOM*

JavaScript extensively interacts with the Document Object Model (DOM) [111] to update the web page seamlessly at runtime. The DOM is a platform- and language-neutral standard object model for representing HTML and XML documents. It provides an API for dynamically accessing, traversing, and updating the content, structure, and style of such documents. Changes made dynamically, through JavaScript, to DOM elements are directly manifested in the browser's display. This allows a single-page to be updated incrementally, which is substantially different from the traditional URL-based page transitions through hyperlinks, where the entire DOM was repopulated with a new HTML page from the server for every state change. This dynamic interplay between two separate entities, namely JavaScript and the DOM, makes web apps particularly error-prone [87].

C. *Eventfulness*

JavaScript is an event-driven language allowing developers to register various event listeners on DOM nodes. While most events are triggered by user actions, timing events and asynchronous callbacks can be fired with no direct input from the user. To make things even more complex, a single event can propagate on the DOM tree and trigger multiple listeners according to the event capturing and bubbling properties of the event model [112]. Understanding these event-driven interactions in JavaScript is known to be challenging for developers [5].

D. *Cascading with Style*

Finally, CSS is used for applying a visual presentation onto the web page. CSS has a number of intricate characteristics, such as inheritance, cascading order, and selector specificity, all defined in relation to the dynamic DOM, which make the language challenging to analyze [65].

E. Inter-language Interactions

The characteristics of these languages (JavaScript, HTML/-DOM, CSS), the dynamic inter-dependencies between them, and the heterogenous, distributed asynchronous client/server nature of web apps, make web development, testing, and maintenance a daunting endeavour for developers. web apps are particularly error-prone [87], [90], [45], difficult to understand [5], [117], and challenging to analyze [107], [31], [40] and test [12], [69].

In the past two decades, software engineering research has targeted some of these changing challenges with great achievements, although still many remain to be addressed.

IV. Achievements

This section focuses on what the author believes are some of the main achievements, primarily in the past decade, in the area of software analysis and testing for the web.

A. Web Analysis

Recent research advances have made the use of static analysis on JavaScript more practical [9], [32], [48], [53], [58], [81], [89], [108]. Other techniques mitigate the analysis challenges by using a dynamic or hybrid approach [4], [38], [72], [114].

Code Analysis. WALA adapts traditional points-to analysis [107], [109] for JavaScript through correlation tracking of dynamic properties in the code. Pointer analysis has also been combined with use analysis to investigate [57] the effects of JavaScript libraries and frameworks on the applications' data flow. Using WALA [109] a partial call graph can be inferred through static analysis. Constructing (approximate) JavaScript call graphs [32] has also been proposed. Since JavaScript is such a dynamic language, these techniques yield conservative results that may not be reflective of an application's actual execution. They also ignore the JavaScript-DOM interactions completely. TAJS [51] is a whole-program data-flow analyzer for JavaScript. It includes [50] statically modelling the role of the DOM and browser in static analysis albeit with known shortcomings that result in false-positives.

Dynamic and hybrid JavaScript analysis techniques [4], [5], [84], [114] have attempted to solve some of the shortcomings of static analysis. For instance, JavaScript blended analysis [113] integrates the information gathered during both static and dynamic analyses to perform a points-to analysis of JavaScript applications. Tochal [4] performs DOM-sensitive hybrid change impact analysis for JavaScript through a combination of static and dynamic analysis. EventRacer facilitates dynamic race detection for event-driven web apps [97].

Code Smells and Refactoring Support. Code smells are patterns in the source code that indicate potential comprehension and maintenance issues in the program. Detected code smells can be refactored to improve the design and quality of the code. WebScent [85] detects client-side smells that exist in embedded code within scattered server-side code. Such smells cannot be easily detected until the client-side code is generated. After detecting smells in the generated client-side code, it locates the smells in the corresponding location in the server-side code. WebScent primarily identifies the mixing of HTML, CSS, and JavaScript, duplicate code in JavaScript, and HTML syntax errors. JSNose [73] combines static and dynamic analysis for detecting 13 smells in JavaScript code. DSLint [39] is a dynamic analysis tool to check code quality rules and best practices in JavaScript code. Detecting unused CSS code has been supported by either dynamically checking CSS rules against the DOM elements of a web app [65] or statically using tree logics [36].

Some initial progress has been made toward automated transformation and refactoring support for the web. Semi-automatic refactoring support has been proposed in the form of JavaScript rename [30] and for specifying and enforcing JavaScript practices [29]. Other techniques [49], [63] transform unsafe *eval* calls in JavaScript code to functionally equivalent but without the use of *eval*. Finding refactoring opportunities in CSS code [60] has recently received attention.

Programmer Support. Helping programmers understand a web app's code and behaviour has seen some improvements in the past decade, although much remains to be done. FireDetective [117] is a Firefox add-on that captures and visualizes a trace of execution on both the client and the server side. The goal is to make it easier for developers to understand the link between client and server components. Clematis [5] facilitates the process of comprehending the dynamic behaviour of JavaScript applications using a high-level model and visualization based on a semantically partitioned trace. The technique assists developers in understanding the complex event-driven interactions in JavaScript applications, through a combination of automated code transformation, tracing, model generation, and visualization, which depicts the creation and flow of triggered events, the corresponding executed JavaScript functions, and the mutated DOM nodes. FireCrystal [93] is another Firefox extension that stores the trace of a web app in the browser. It then visualizes the events and changes to the DOM in a timeline. DynaRIA [8] focuses on investigating the structural and quality aspect of the code by collecting a trace.

B. Web Testing

There have been many advancements in software testing for the web [64]. This is an area that has attracted most of the attention from software engineering researchers. Recently, Garousi et al. conducted a systematic mapping study [35] followed by a systematic review [25] of web app testing literature. They found a large body of work addressing different challenges of web testing. We refer the reader to these studies for a complete overview. Here we discuss some of the main achievements.

Testing Classical web apps. Ricca and Tonella were among the very first researchers to publish on web testing in 2001. Their ICSE 2001 paper [98] proposed a model-based testing technique for classical web apps. This influential work won an ACM Most Influential Paper Award in 2011. Elbaum et al. were the first to propose the use of 'user session data' for web app testing. Their technique [26] collects user interactions in the form of requests send to the server, and transforms those into test cases. Subsequently, in 2005, Andrews et al. [10] proposed to manually model classical web apps as Finite State Machines (FSMs) to generate test cases from.

Testing Modern Web Apps. The effectiveness of applying traditional web testing techniques [98], [10], [26], [106] to modern web apps were assessed in a case study by Marchetto et al. [59]. This analysis suggests that such traditional techniques have many limitations in testing modern dynamic web apps. Based on this analysis, a new approach for state-based testing of modern web apps was proposed in 2008. The technique first generates traces of the application by manually interacting with the application. The traces are then used to construct a finite state model. Sequences of semantically interacting events in the model are transformed to test cases once the model is refined by the tester.

Mining Test Models. In order to automatically infer a model of a given modern web app, Crawljax was proposed [68], a technique capable of exploring event-based DOM mutations of a web app. While exploring, Crawljax infers a *state-flow graph* capturing the states of the user interface, and the possible event-based transitions between them. This crawling capability provide access to dynamic DOM states, which servers as a strong vehicle for various web analysis and testing purposes. Other similar techniques were proposed (e.g., ProCrawl [104], FeedEx [71]) for mining test models from web apps using different state and event abstraction notions.

Test Generation. A new testing technique was proposed in 2009 [67], which combined the automated exploration of Crawljax with invariant-based testing for modern web apps. In this technique, with access to different dynamic web states, the user interface can be checked against different constraints, expressed as *invariants*, which can act as oracles to automatically conduct sanity checks in any DOM state. *Generic* and *application-specific* invariants can be expressed on the DOM-tree, between DOM-tree states, and on the run-time JavaScript variables. Examples of generic invariants include the requirement that any DOM should be composed of valid HTML, that there are no broken links, and that all element ID attributes are unique. The state-flow graph automatically inferred through crawling was also used for test generation. For example, it can be used to generate different event paths and cover the application's state space in different ways. These types of test cases can be used in regression testing of web apps in which the DOM states of of a new version of the application are checked agains a previous version's [101]. The state-flow graph is also used for generating event-based and unit test cases [76].

Cross-browser Testing. With the advent of web technologies and new browsers, each with a slightly different client-side rendering of the application, the cross-browser compatibility issue became increasingly important. Cross-browser testing seeks to automatically detect inconsistencies in the behaviour and layout of a web app across multiple browsers. WebDiff [22] is a tool that analyzes the DOM as well as screen-shots of pairs of screens to locate cross-browser issues. The focus here was on identifying cross-browser differences in individual screens. Subsequently, CrossT was proposed [66] to go beyond screen-pairs and cover a larger set of the state space. The problem of cross-browser compatibility testing of modern web apps was posed as a 'functional consistency' check of web app behaviour across different web browsers and an automated solution was provided. CrossT identifies more systemic cross-browser issues that manifest in the overall behaviour of the web app. It consists of (1) automatically analyzing the given web app under different browser environments and capturing the behaviour as a state-flow graph, and (2) formally comparing the generated models for equivalence on a pairwise-basis and exposing any observed discrepancies. CrossCheck [20] combines both WebDiff and CrossT to benefit from the advantages of both approaches and X-Pert [21] improves upon previous work by using a more precise differencing technique for detecting layout issues.

Testing JavaScript. To generate test inputs for JavaScript code, Artemis was proposed at ICSE 2011 [12], a testing technique that uses feedback-directed random testing. Feedback-directed testing is a technique in which test cases are randomly generated and executed. The feedback obtained by running the generated test cases is then used to guide the test generation engine to be more effective, for example, in covering the application. Artemis randomly generates test inputs, executes the application with those inputs, and uses the gathered information to generate new test inputs. The execution feedback directs the test generator towards inputs that yield higher coverage.

There have been recent advancements in applying symbolic execution to the web. Symbolic execution is a static analysis technique that treats input variables as symbolic variables. For every program path detected, constraints are collected (called path constraints) and solved through constraint solvers. Concolic testing combines symbolic execution with dynamic analysis to overcome some of the limitations of symbolic execution and current constraints solvers. Both have been applied for testing JavaScript code recently. Kudzu [103] is a symbolic execution technique for JavaScript applications. Kudzu is built on top of a constraint solver (called Kaluza) that supports boolean, machine integer (bit-vector), and string constraints, which is used for reasoning about the parsing and validation checks that JavaScript applications perform. Kudzu is particularly focused on string reasoning and finding security vulnerabilities in JavaScript code. Jalangi [105] is a framework for light-weight source instrumentation and dynamic analysis of JavaScript code. It also provides a concolic engine for JavaScript. It handles linear integer, and string, and type constraints. SymJS [41] is a framework for automatic testing of JavaScript code. It contains a symbolic execution engine for JavaScript, and an automatic event explorer. It automatically explores events, and symbolically executes the related JavaScript code to produce test cases. Its symbolic engine is built on a symbolic virtual machine, a string-numeric solver, and a simple symbolic DOM model. ConFix [28] is an automated technique, based on dynamic symbolic execution, that generates DOM-based test fixtures for unit testing JavaScript functions. Atrina [77] infers test oracles from existing UI-level test cases to generate JavaScript unit tests.

Capture and Replay. Extensive reliance on user interactions is an important characteristic of modern web apps. Capture and replay tools are used in the literature to address this issue. Mugshot [70] is a system which employs a server-side web proxy to capture events in interactive web apps. It injects code into a target web app in order to record sources of nondeterminism such as DOM events and interrupts. The recorded information is used to dispatch synthetic events to a web browser in order to replay the execution trace. WaRR [11] is another system for capturing and replaying events. Capturing is accomplished by altering a user's web browser in order to record keystrokes and mouse clicks. In the event of a failure, end users of a web app may send a record of their keystrokes to the developer for debugging purposes. Jalangi [105] is another record-replay tool that supports dynamic analysis by shadow execution on shadow values. Capture/replay tools are also integrated with debugging tools for web apps [18].

Test Adequacy Assessment. There are different ways of assessing the adequacy of a give test suite. Code coverage and mutation testing are two well-known techniques. We have witnessed many advancements in tools and techniques for assessing the quality of web app test suites recently. Code coverage is the most widely used metric for test case assessment. The idea is to measure the portion of the program code executed when the test suite is run. Code coverage is particularly useful for detecting under tested portions of the code. Many industrial tools exist today that automate measuring JavaScript code coverage.[2][3][4] All these tools focus on JavaScript code-level coverage.

DOMCovery [78] automatically extracts and measures the DOM adequacy criteria for a given test suite and generates a visual DOM coverage report. It uses a set of DOM-based test adequacy criteria for web apps for measuring coverage at two granularity levels, namely (1) inter-state: the percentage of DOM states and transitions covered in the total state space of the web app under test, and (2) intra-state: the percentage of elements covered in each particular DOM state. Test output uniqueness [7] is a set of blackbox testing criteria including five that are based on the HTML structure, and two based on the textual content of webpages. The key insight behind output uniqueness as a testing criterion is that two test cases that yield different outputs may cover two different paths in the code.

Code coverage alone does not assess the fault-finding capabilities of a test suite. Mutation testing is a fault-based testing technique to assess and improve the fault-finding quality of a test suite. The technique first generates a set of mutants—modified versions of the program—, by applying a set of well-defined mutation operators on the original version of the system under test. These mutation operators typically represent subtle mistakes, such as typos, commonly made by programmers. A test suite's adequacy is then measured by its ability to detect (or 'kill') the mutants, which is known as the mutation score. Mutation operators have been proposed [96] for HTML and Java Server Pages. Mutandis [75] is the first mutation testing technique for JavaScript programs. Mutandis leverages static and dynamic program data to rank, select, and mutate potentially behaviour-affecting portions of the program code. It uses the notion of function rank to rank JavaScript functions based on their relative importance based on the application's dynamic behaviour. The approach gives a higher probability to functions ranked higher for being selected for mutation. The insight is that if a highly ranked function is mutated, because many other functions depend on its functionality, the result of the mutation will be more likely to be observable and thus non-equivalent. To mutate a selected function, Mutandis selects from a set of JavaScript-specific mutation operators, assembled based on common mistakes made by JavaScript programmers in practice.

Another dimension related to test quality is the notion of test case robustness. A test case is believed to be fragile when a small change in the application layout causes test cases to fail. In addition, test cases created for one particular browser can easily break when executed on a different web browse. Fragile test cases require extra maintenance effort to remain functional and as such increase the overall cost of a software project. This is especially the case for test cases that check the user interface of the web app. For instance, in Selenium test cases, DOM element locators such as ID or XPath expressions are used to retrieve elements from the webpage. References to elements are needed in order simulate user actions by firing events and to assert properties of the DOM for correctness. Researchers have discussed and studied the fragility of DOM element locators [54], [55], [68], [79]. Montoto et al. [79] propose an algorithm for making XPath locators less fragile by taking into account other properties of DOM elements such as their attributes. Contextual clues [115] were proposed to mitigate the locator fragility problem in DOM-based test cases, by using a series of contextual clues based on the textual values of elements and their relative positions to locate DOM elements. LED [15] is an automated technique for synthesizing DOM element locators using positive and negative examples

[2]https://github.com/itay/node-cover
[3]https://github.com/yahoo/istanbul
[4]http://blanketjs.org

provided interactively by the developer. LED expresses the problem of synthesizing complex multi-element locators as a constraint solving problem over the domain of valid DOM states in a web app.

Server-side Code. The server-side code in languages such as Java can typically be tested with conventional software testing techniques. An interesting line of work is that by Halfond et al., which provide a test generation technique based on automated client/server interface discovery [43] and modelling input parameters using symbolic execution [42] of server-side Java code. Analyzing server-side scripting languages, such as PHP, have received some attention [102], [44], although the main focus has been on security analysis.

C. Empirical Studies

In order to understand how web developers write today's web apps and what consequences that has for software analysis and testing, various empirical studies have been conducted by analyzing web apps in the wild.

Dynamic Language Features. To gain an understanding of which dynamic features of the JavaScript language developers depend on in practice, an empirical study on 100 websites and three benchmarks [100] showed that many of the dynamic features of the language are indeed used in practice. For instance, many libraries change the builtin prototypes in order to add behaviour to different types. The study also found that object properties are changed (added/deleted) at runtime, the use of `eval` to generate and execute code is frequent, and functions take a different number of parameters at runtime than statically declared. In a follow-up study [99], the runtime behaviour of the `eval` function was examined, which revealed that between 50–80% of the 10,000 studied websites used `eval`. The study showed that JavaScript is a difficult terrain for static analysis since the dynamic features are prevalent and cannot simply be ignored to make approximate simplifications. A recent work [95] studied the use of type coercions in JavaScript by dynamically analyzing hundreds of programs; it found that type coercions are widely used (in 80.42% of executed functions) and that most coercions are harmless.

Insecure Inclusions. Insecure practices of using JavaScript on the web were analyzed [116]; 66% of the 6,800 websites studied were found to contain insecure practices pertaining to the inclusion of JavaScript files from external domains. In addition, around 44% of the sites used `eval` to generate and execute JavaScript code on the client-side. The study concluded that developers need to adopt and apply safer alternatives that exist today to reduce potential security risks. In a related study [86], a large-scale analysis was conducted of three million pages of the top 10,000 Alexa sites in search of the trust relationships of these sites with their JavaScript library inclusions. The study showed that even top-sites trust remote JavaScript providers that could be compromised and serve malicious code.

Dynamic DOM States. Through the execution of JavaScript code in browsers, the DOM tree representing a webpage at runtime can be incrementally updated without requiring a URL change. This dynamic DOM manipulation has a significant impact on traditional web analysis techniques that treat web apps as a sequence of linked static HTML pages. In order to gain an understanding of the prevalence and extent of dynamic DOM manipulated through JavaScript in practice, a study was conducted [16] on real-world websites. The study revealed that dynamic DOM is prevalent in online web apps today. From the 500 websites they analyzed, 95% contained client-side dynamic DOM content, and on average, 62% of the analyzed web states of those websites were dynamic DOM. The study shows that today's web apps rely heavily on client-side code execution, and HTML is not just created on the server, but manipulated extensively within the browser through JavaScript code.

Another study [80] aimed at understanding the software engineering implications of this change by looking at deviations from many known best practices in such areas of performance, accessibility, and correct structuring of HTML documents. The study assessed to what extent such deviations are manifested through client-side JavaScript manipulation only. To this end, a large scale study was conducted, involving automated JavaScript-enabled crawling of over 4,000 websites, resulting in over 100,000,000 pages, and around 1,000,000 unique client-side user interface states analyzed. Traditionally, each URL of a web site pointed to a single HTML document on the server, providing a one-on-one mapping between the two.

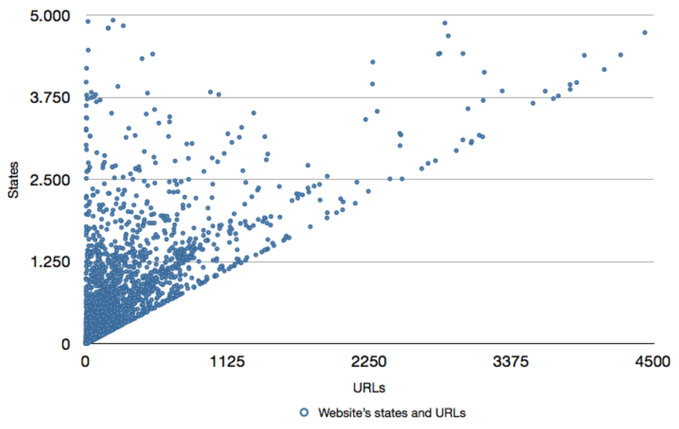

Fig. 2. Number of HTML/DOM states per URL [80].

Figure 2 depicts the relation in current web apps. There are no entries underneath the diagonal since each URL points to at least one HTML/DOM state. The results show that per URL, there are 16 dynamic DOM states, on average. This means that if a traditional static HTML analysis tool would request a certain URL once, without any context, it would miss more than half of the states. The study also found that 90% of the analyzed websites perform DOM manipulations after they are loaded into the browser. The findings also show that the majority of sites contain a substantial number of problems, making sites unnecessarily slow, inaccessible for the visually impaired, and with layout that is unpredictable due to errors in the dynamically modified DOM states. For instance, more than half of the sites contain errors such as ambiguous IDs and invalid HTML structure. These violations manifest themselves not just in HTML directly coming from the server, but require code in the browser to execute before they become visible and detectable. Consequently, tools based on static analysis, or dynamic analysis using traditional hyperlink-based crawlers, will fail to analyze a large fraction of a modern web app.

Bug Mining. A few studies have examined reported failures of web apps to characterize their nature and root causes. A characterization study [91] of the error messages printed to the browser console by JavaScript code execution of 50 websites, selected from the Alexa top 100 most visited sites revealed that runtime JavaScript errors (1) are widespread in deployed websites: an average of four JavaScript runtime error messages appear even in popular production web apps, (2) fall into a small number of categories: Permission Denied (52%), Undefined Symbol (28%), Null Exception (9%), and Syntax Errors (4%), and (3) are mostly (70%) non-deterministic, meaning that they vary from one execution to another, and that the speed of testing plays an important role in exposing such errors.

In a more recent study [87], over 300 bug reports from various web apps and JavaScript libraries were examined to understand the nature of the errors that cause these faults, and the failures to which these faults lead. The results of the study reveal that (1) around 65% of reported JavaScript faults are DOM-related — a fault is DOM-related if the parameter of a DOM access method (such as `getElementById(param)`) or the assignment value for a DOM access property is erroneous, thereby causing an incorrect retrieval or an incorrect update of a DOM element —, (2) most (around 80%) high severity faults are DOM-related; (3) the vast majority (around 86%) of JavaScript faults are caused by errors manually introduced by JavaScript code programmers, as opposed to code automatically generated by the server; (4) error patterns exist in JavaScript bug reports (such as omitting `null/undefined` checks); and that (5) DOM-related faults take longer to triage and fix than non-DOM-related JavaScript faults. Based on these findings, the authors suggest that testing efforts should target detecting DOM-related faults, as most high-impact faults belong to this category. One possible way to do this is to guide the test generation towards tests that cover DOM interaction points in the JavaScript code. This emphasis is particularly useful since DOM-related JavaScript faults often have no accompanying error messages and thus can be more difficult to find.

D. Industrial Advancements

Currently, many industrial tools exist that assist developers in parsing (e.g., Esprima[5], Rhino[6]), optimizing (e.g., Google Closure Compiler[7]), and statically analyzing JavaScript code for common syntactical errors (e.g., JSHint[8]). JSLint [2] is a static code analysis tool written in JavaScript that validates JavaScript code against a set of good coding practices. The code inspection tends to focus on improving code quality from a technical perspective. The Google Closure Compiler [1] is a JavaScript optimizer that rewrites JavaScript code to make it faster and more compact. It helps to reduce the size of JavaScript code by removing comments and unreachable code.

Many testing frameworks have been developed to help developers to write test cases for JavaScript code. For instance, QUnit[9] is a popular JavaScript unit testing framework. Jasmine[10] is a behaviour-driven development framework for testing JavaScript code. Mocha[11] is a JavaScript test framework that runs both on Node.js and the browser and has support for testing asynchronous methods. jsTestDriver[12] is a framework that automates running a JavaScript test suite in different browsers.

A challenge in testing web apps arises when the JavaScript code interacts with the DOM. In this case, an environment is needed that can support the creation and manipulation of the DOM and event-listeners. This is what current browser automation frameworks aim for. For instance, frameworks such as Selenium[13], PhantomJS[14], and SlimerJS[15] provide APIs for driving a browser instance, firing events, and accessing DOM elements at runtime. These browser automation APIs can be utilized for writing DOM-based test cases that check the behaviour of the web app (and indirectly its JavaScript code) from an end-user perspective.

Although such frameworks make it easier for the developer to write test cases, they still require a substantial level of manual effort.

V. Looking to the Future

The web is a moving target. It is continuously and rapidly evolving, making it quite challenging for developers to cope with all the technological advancements.

A. JavaScript Is Here to Stay

In 2007, Jeff Atwood predicted [13] that JavaScript would become a prominent language, in a bold statement now widely known as Atwood's law:

[5]http://esprima.org
[6]https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino
[7]https://developers.google.com/closure/compiler/
[8]http://www.jshint.com
[9]http://qunitjs.com
[10]http://pivotal.github.io/jasmine/
[11]http://visionmedia.github.io/mocha/
[12]https://code.google.com/p/js-test-driver/
[13]http://www.seleniumhq.org
[14]http://phantomjs.org
[15]http://www.slimerjs.org

"Any application that can be written in JavaScript, will eventually be written in JavaScript."

Atwood's Law

Today, JavaScript is the most prevalent language in GitHub repositories [37] and a recent survey of more than 26K developers conducted by Stack Overflow found that JavaScript is the most-used programming language [110].

Attempts at replacing JavaScript with other languages and frameworks (e.g., Java, Flash, SilverLight, Dart) have largely failed. Approaches that have embraced the language by making it easier for developers to write code have had much more success. An example of the latter is TypeScript which is a typed superset of JavaScript that compiles to plain JavaScript.

JavaScript is not only used for writing client-side code anymore. Today, many desktop and server-side applications are written in JavaScript. In fact, npm,[16] a popular JavaScript package manager, has become the largest software package repository with over 205K packages, now surpassing Maven Central (Java) and RubyGems (Ruby).

JavaScript is also being supported and used on smartphones, smartwatches, and other small devices. Advancements in JavaScript and HTML5 that allow access to native features of mobile devices (e.g, GPS, Camera) can potentially provide an alternative for building native apps that are expensive to build and maintain — build once, run on any mobile device —. We are likely to see more mobile apps written as web apps to circumvent the challenges pertaining to building and maintaining native apps for multiple platforms.

Together with HTML5, JavaScript is also a viable solution for building apps for the 'Internet of Things' [62]. JavaScript is well-suited to embedded device programming thanks to its asynchronous callbacks and native I/O support (through Node.js), which are ideal for event-driven hardware programming. Examples of JavaScript used for embedded devices include NodeBots[17], Cylon,[18] and JohnnyFive[19].

New JavaScript features such as WebGL 3D will become popular as browsers continue to increase in computational speed. Powered by frameworks such as three.js,[20] we can see more applications of this technology, such as games, in-browser.

With the wide adoption of the language, which is likely to increase every year, much better tool support is needed in the near future, which provides a great opportunity for software engineering researchers to have impact in the field.

B. The Problematic DOM

The interaction between JavaScript and the DOM on the client-side is error-prone [87]. Partly in response to these issues, we have witnessed a plethora of JavaScript libraries such as jQuery and frameworks such as AngularJS, Backbone, or Ember, mainly to abstract away and facilitate the interaction between JavaScript code and the DOM. A promising direction pushes the DOM even further into a virtual/shadow entity; solutions such as ReactJS[21] and Polymer[22] are two famous examples that help developers to move towards more reusable dynamic components. Despite these efforts, DOM-JavaScript interactions pose a threat for inconsistencies in applications that are built on top of these frameworks [89]. It seems unlikely that the DOM will disappear anytime soon from the picture. Therefore, better tools and techniques are needed to support web developers in their efforts to write and analyze their code.

C. Asynchronous Programming with Callbacks

JavaScript uses an event-driven model with a single thread of execution. Programming with callbacks is especially useful when a caller does not want to wait until the callee completes. Callbacks are used to responsively handle events on the client-side by executing functions asynchronously, and, in Node.js callbacks are used on the server-side to service multiple concurrent client requests. The event loop in JavaScript prioritizes the single thread to execute the call stack first; when the stack is empty, the event loop dequeues a message from the task queue and executes the corresponding callback function.

Unfortunately, callbacks induce a non-linear control flow and can be deferred to execute asynchronously, declared anonymously, and may be nested to arbitrary levels [34]. All of these features make callbacks difficult to analyze, understand, and maintain, which developers have coined the term "callback hell" for in practice. New analysis techniques and control flow graphs are needed to adequately reason about code that uses callbacks, especially asynchronous ones.

D. Benchmarks

Despite the many achievements in the field, unfortunately the research area lacks central well-maintained benchmarks to be used for meaningful evaluations, controlled experiments, and comparative studies. Such repositories exist for other software domains, examples of which include the SIR [24] and BugBench [46], and Defect4J [52]. The field is in a real need of a coherent repository with representative subject systems of different types and sizes, with comprehensive test suites, as well as real bugs and patches. Such a repository can be used as a comparison ground for new web analysis and testing tools and techniques.

E. Mining Artifacts and Models

Leveraging feedback from existing software artifacts to guide the analysis and testing is another direction that has gained more attention recently, but can be explored much further. In the past, user-sessions [27] and user interactions [61] have been leveraged for web testing. Using the knowledge in existing test cases has been explored recently with Testilizer

[16]https://www.npmjs.com
[17]http://nodebots.io/
[18]http://cylonjs.com/
[19]http://johnny-five.io/
[20]http://threejs.org
[21]http://facebook.github.io/react/
[22]https://www.polymer-project.org/

[74]. The work is motivated by the fact that a human-written test suite is a valuable source of domain knowledge, which can be exploited for tackling challenges in automated web app test generation. It takes as input a set of Selenium test cases TC and the URL of the application, automatically infers a model from TC, feeds that model to a crawler to expand by exploring uncovered paths and states, generates assertions for newly detected states based on the patterns learned from TC, and finally generates new test cases. Other artifacts can be mined to learn more about the real challenges of web developers (e.g., StackOverflow [14]). Mining different types of repositories has been a largely under-explored opportunity for web analysis and testing thus far. For instance, npm, with more than 205K readily available JavaScript packages, and GitHub, with more JavaScript projects than any other language, are treasure mines to be exploited.

F. Cross-Language Analysis

Static analysis techniques for the web make simplifying assumptions about the code. However, due to the dynamic nature of and interactions between the languages used for creating web apps, most current static analysis techniques have many shortcomings in terms of false positives and negatives. It is surprising that despite the popularity of JavaScript, there is still a lack of robust static analysis bug detection tools, such as those for statically typed languages (e.g, FindBugs for Java). This is an area that will need further push from the researchers in the coming years.

A promising direction is the ability to handle multiple languages in static analysis. Inferring call graphs from embedded-code [82], cross-language slicing [83], and detecting inconsistencies between JavaScript and DOM [89] are three examples in this direction. We conjuncture that more such techniques will be proposed in the coming years to alleviate some of the complex inter-language dependencies in web apps. Also hybrid approaches that combine static and dynamic techniques will probably prove to be more useful for this domain, than pure static analysis.

G. Fault Localization and Automated Repair

When a fault is detected, the next natural step is to localize the fault and to repair it. AutoFlox [92] is an automated technique for localizing code-terminating DOM-related JavaScript errors — a DOM access function returns a `null`, `undefined`, or incorrect value, which then propagates into several variables and eventually causes an exception in JavaScript code execution. It takes a code-terminating line of JavaScript code as input and performs dynamic analysis and backward slicing of the web app to localize the cause of these JavaScript faults. There has been limited work on exploring fault repair for web apps. Automatic workarounds for web apps [19] replace a buggy API call sequence with a functionally equivalent, but correct sequence. Others techniques [102], [118] aim at fixing PHP errors that generate malformed HTML. Vejovis [88] tries to automatically find a fix for a JavaScript fault. Based on observations of common fixes applied by programmers to JavaScript faults, the technique provides repair suggestions for DOM-related JavaScript faults. There are many challenges and opportunities for fault localization and automated repair for the web domain, areas that need much more innovative research in the coming years.

H. Programer Support and IDEs

As web languages such as JavaScript become more prominent in use, proper IDE support becomes essential for developing and maintaining large-scale applications in practice. However, current software analysis techniques are known to have serious limitations in supporting developers to understand, write, analyze, and maintain web code. Current work on web app code smell detection, refactoring support, and code completion is scarce and industrial tools available to web developers are limited in their capabilities. As it stands today, we are not even able to extract proper control-flow and call graphs for web apps that are composed of multiple languages.

I. Green Web Development

Reducing the power consumption in web apps has many benefits, both on the client and server sides. On the client-side, especially for web apps running on smartphones, this could mean extending the battery power [56]. For the code running on the server in the cloud, this could mean more efficient CPU usages. Optimizing web code for more green development is an almost entirely unexplored area that needs more attention.

VI. Conclusion

The web has grown into a strong medium for developing software applications that can easily be distributed to end-users and executed universally through the browser. The new changes in the evolution of the web bring not only advantages, but also come with a whole set of new challenges. In this paper, we presented some of these challenges and examined research achievements in the area of web analysis and testing. We also discussed potential areas that require more attention and provide opportunities for further development and research. Unlike traditional software, web apps are built using a combination of different languages, i.e., JavaScript, HTML/DOM, and CSS on the client, which communicate with one or more languages such as Java, PHP, or JavaScript on the server. The dynamic inter-play between these languages, and their distributed asynchronous client/server nature, pose many challenges in practice. This is where software engineering research can play an important role. A promising research direction is the ability to handle multiple languages in web analysis. We are in real need of inter-language analyses to be able to handle web application code. Also hybrid approaches that combine static and dynamic techniques will probably prove to be more useful for this domain.

References

[1] Google closure compiler. https://developers.google.com/closure/.

[2] Jslint: The JavaScript code quality tool. http://www.jslint.com/.

[3] Node.js. http://nodejs.org/.

[4] S. Alimadadi, A. Mesbah, and K. Pattabiraman. Hybrid DOM-sensitive change impact analysis for JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 321–345. LIPIcs, 2015.

[5] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding JavaScript event-based interactions. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 367–377. ACM, 2014.

[6] J. Allaire. Macromedia Flash MX-A next-generation rich client. Macromedia white paper, 2002. http://www.adobe.com/devnet/flash/whitepapers/richclient.pdf.

[7] N. Alshahwan and M. Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 181–192, New York, NY, USA, 2014. ACM.

[8] D. Amalfitano, A. Fasolino, A. Polcaro, and P. Tramontana. The DynaRIA tool for the comprehension of Ajax web applications by dynamic analysis. *Innovations in Systems and Software Engineering*, 10(1):41–57, 2014.

[9] E. Andreasen and A. Møller. Determinacy in static analysis for jQuery. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2014.

[10] A. Andrews, J. Offutt, and R. Alexander. Testing web applications by modeling with FSMs. *Software and Systems Modeling*, 4(3):326–345, July 2005.

[11] S. Andrica and G. Candea. WaRR: A tool for high-fidelity web application record and replay. In *Proceedings of the International Conference on Dependable Systems & Networks (DSN)*, pages 403–410. IEEE Computer Society, IEEE, 2011.

[12] S. Artzi, J. Dolby, S. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 571–580, 2011.

[13] J. Atwood. The principle of least power, 2007. http://blog.codinghorror.com/the-principle-of-least-power/.

[14] K. Bajaj, K. Pattabiraman, and A. Mesbah. Mining questions asked by web developers. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 112–121. ACM, 2014.

[15] K. Bajaj, K. Pattabiraman, and A. Mesbah. Synthesizing web element locators. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 11 pages. IEEE Computer Society, 2015.

[16] Z. Behfarshad and A. Mesbah. Hidden-web induced by client-side scripting: An empirical study. In *Proceedings of the International Conference on Web Engineering (ICWE)*, volume 7977 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 2013.

[17] T. Berners-Lee. WWW: Past, present, and future. *IEEE Computer*, 29(10):69–77, 1996.

[18] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *Proceedings of the Symposium on User Interface Software and Technology (UIST)*, pages 473–484. ACM, 2013.

[19] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 237–246. ACM, 2010.

[20] S. R. Choudhary, M. R. Prasad, and A. Orso. Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 171–180, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

[21] S. R. Choudhary, M. R. Prasad, and A. Orso. X-PERT: Accurate identification of cross-browser issues in web applications. In *Proceedings of the International Conference on Software Engineering (ICSE 2013)*, pages 702–711, May 2013.

[22] S. R. Choudhary, H. Versee, and A. Orso. Webdiff: Automated identification of cross-browser issues in web applications. In *Proc. of the 26th IEEE Int. Conf. on Softw. Maintenance (ICSM'10)*, pages 1–10, 2010.

[23] D. Crockford. *JavaScript: the good parts*. O'Reilly Media, Incorporated, 2008.

[24] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, Oct. 2005.

[25] S. Doaan, A. Betin-Can, and V. Garousi. Web application testing: A systematic literature review. *Journal of Systems and Software*, 91:174–201, 2014.

[26] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *Proc. 25th Int Conf. on Software Engineering (ICSE)*, pages 49–59. IEEE Computer Society, 2003.

[27] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, 2005.

[28] A. M. Fard, A. Mesbah, and E. Wohlstadter. Generating fixtures for JavaScript unit testing. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 11 pages. IEEE Computer Society, 2015.

[29] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported refactoring for JavaScript. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'11, pages 119–138, New York, NY, USA, 2011. ACM.

[30] A. Feldthaus and A. Møller. Semi-automatic rename refactoring for JavaScript. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA'13, pages 323–338, New York, NY, USA, 2013. ACM.

[31] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 752–761. IEEE Press, 2013.

[32] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE, 2013.

[33] R. Fielding and R. N. Taylor. Principled design of the modern Web architecture. *ACM Trans. Inter. Tech. (TOIT)*, 2(2):115–150, 2002.

[34] K. Gallaba, A. Mesbah, and I. Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in JavaScript. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, page 10 pages. IEEE Computer Society, 2015.

[35] V. Garousi, A. Mesbah, A. Betin Can, and S. Mirshokraie. A systematic mapping study of web application testing. *Information and Software Technology*, 55(8):1374–1396, 2013.

[36] 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*, WWW '12, pages 809–818. ACM, 2012.

[37] GitHut. A small place to discover languages in GitHub. http://githut.info, 2015.

[38] L. Gong, M. Pradel, M. Sridharan, and K. Sen. Dlint: Dynamically checking bad coding practices in JavaScript. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 94–105. ACM, 2015.

[39] L. Gong, M. Pradel, M. Sridharan, and K. Sen. DLint: Dynamically checking bad coding practices in JavaScript. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2015.

[40] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 561–570, New York, NY, USA, 2009. ACM.

[41] E. A. Guodong Li and I. Ghosh. SymJS: Automatic symbolic testing of JavaScript web applications. In *Proc. joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2014.

[42] W. G. Halfond, S. Anand, and A. Orso. Precise interface identification to improve testing and analysis of web applications. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 285–296, New York, NY, USA, 2009. ACM.

[43] W. G. J. Halfond and A. Orso. Improving test case generation for web applications using automated interface discovery. In *Proceedings of the the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 145–154. ACM, 2007.

[44] M. Hills, P. Klint, and J. Vinju. An empirical study of php feature usage: A static analysis perspective. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 325–335. ACM, 2013.

[45] S. Hong, Y. Park, and M. Kim. Detecting concurrency errors in client-side JavaScript web applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 61–70. IEEE, 2014.

[46] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, Dec. 2004.

[47] M. Jazayeri. Some trends in Web application development. In *Future of Software Engineering*, FOSE'07, pages 199–213. IEEE Computer Society, 2007.

[48] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, pages 34–44. ACM, 2012.

[49] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 34–44. ACM, 2012.

[50] 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 Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 59–69. ACM, 2011.

[51] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag.

[52] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.

[53] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: A static analysis platform for JavaScript. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 121–132. ACM, 2014.

[54] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 272–281, 2013.

[55] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Visual vs. DOM-based web locators: An empirical study. In *Proceedings of the International Conference on Web Engineering (ICWE)*, pages 322–340. Springer, 2014.

[56] D. Li, A. H. Tran, and W. G. J. Halfond. Making Web Applications More Energy Efficient for OLED Smartphones. In *Proceedings of the International Conference on Software Engineering (ICSE)*, June 2014.

[57] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 499–509. ACM, 2013.

[58] M. Madsen, F. Tip, and O. Lhoták. Static analysis of event-driven node.js JavaScript applications. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[59] A. Marchetto, F. Ricca, and P. Tonella. A case study-based comparison of web testing techniques applied to Ajax web applications. *Int. Journal on Software Tools for Technology Transfer*, 10(6):477–492, 2008.

[60] D. Mazinanian, N. Tsantalis, and A. Mesbah. Discovering refactoring opportunities in cascading style sheets. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 496–506. ACM, 2014.

[61] S. McAllister, E. Kirda, and C. Kruegel. Leveraging user interactions for in-depth testing of web applications. In *Recent Advances in Intrusion Detection*, volume 5230 of *LNCS*, pages 191–210. Springer, 2008.

[62] M. McCool, R. S. John, and R. Peri. Programming the internet of things with node.js and HTML5. http://solidcon.com/internet-of-things-2015/public/schedule/detail/40797, 2015.

[63] F. Meawad, G. Richards, F. Morandat, and J. Vitek. Eval begone!: semi-automated removal of eval from JavaScript programs. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 607–620. ACM, 2012.

[64] A. Mesbah. *Advances in Testing JavaScript-based Web Applications*, volume 97 of *Advances in Computers*, chapter 5, pages 201–235. Elsevier, 2015.

[65] A. Mesbah and S. Mirshokraie. Automated analysis of css rules to support style maintenance. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 408–418. IEEE Computer Society, 2012.

[66] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 561–570. ACM, 2011.

[67] A. Mesbah and A. van Deursen. Invariant-based automatic testing of ajax user interfaces. In *Proceedings of the 31st ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 210–220. IEEE Computer Society, 2009.

[68] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.

[69] A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering (TSE)*, 38(1):35–53, 2012.

[70] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for Javascript applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 159–174. USENIX Association, 2010.

[71] A. Milani Fard and A. Mesbah. Feedback-directed exploration of web applications to derive test models. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 278–287. IEEE Computer Society, 2013.

[72] A. Milani Fard and A. Mesbah. JSNose: Detecting JavaScript code smells. In *Proceedings of the International Working Conference onSource Code Analysis and Manipulation (SCAM)*, pages 116–125, 2013.

[73] A. Milani Fard and A. Mesbah. Jsnose: Detecting JavaScript code smells. In *Proceedings of the International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125. IEEE Computer Society, 2013.

[74] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 67–78. ACM, 2014.

[75] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Efficient JavaScript mutation testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 74–83. IEEE Computer Society, 2013.

[76] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Jseft: Automated JavaScript unit test generation. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, page 10 pages. IEEE Computer Society, 2015.

[77] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Atrina: Inferring unit oracles from GUI test cases. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, page 11 pages. IEEE Computer Society, 2016.

[78] M. Mirzaaghaei and A. Mesbah. DOM-based test adequacy criteria for web applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 71–81. ACM, 2014.

[79] P. Montoto, A. Pan, J. Raposo, F. Bellas, and J. Lapez. Automating navigation sequences in Ajax websites. In *Proceedings of the International Conference on Web Engineering (ICWE)*, volume 5648, pages 166–180. Springer, 2009.

[80] A. Nederlof, A. Mesbah, and A. van Deursen. Software engineering for the web: The state of the practice. In *Proceedings of the ACM/IEEE International Conference on Software Engineering, Software Engineering In Practice (ICSE SEIP)*, pages 4–13. ACM, 2014.

[81] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Building call graphs for embedded client-side code in dynamic web applications. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 518–529. ACM, 2014.

[82] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Building call graphs for embedded client-side code in dynamic web applications. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 518–529. ACM, 2014.

[83] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Cross-language program slicing for dynamic web applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 369–380, New York, NY, USA, 2015. ACM.

[84] H. V. Nguyen, H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen. Mining interprocedural, data-oriented usage patterns in JavaScript web applications. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 791–802. ACM, 2014.

[85] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen. Detection of embedded code smells in dynamic web applications. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 282–285. ACM, 2012.

[86] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proc. Conf. on Comp. and Comm. Security*, pages 736–747. ACM, 2012.

[87] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. An empirical study of client-side JavaScript bugs. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 55–64. IEEE Computer Society, 2013.

[88] F. Ocariza, K. Pattabiraman, and A. Mesbah. Vejovis: Suggesting fixes for JavaScript faults. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 837–847. ACM, 2014.

[89] F. Ocariza, K. Pattabiraman, and A. Mesbah. Detecting inconsistencies in JavaScript MVC applications. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 325–335. ACM, 2015.

[90] F. Ocariza, K. Pattabiraman, and B. Zorn. JavaScript errors in the wild: An empirical study. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–109. IEEE, 2011.

[91] F. Ocariza, K. Pattabiraman, and B. Zorn. JavaScript errors in the wild: An empirical study. In *Proc. of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–109. IEEE Computer Society, 2011.

[92] F. J. Ocariza, K. Pattabiraman, and A. Mesbah. Autoflox: An automatic fault localizer for client-side JavaScript. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 31–40. IEEE Computer Society, 2012.

[93] S. Oney and B. Myers. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*, pages 105–108. IEEE Computer Society, IEEE, 2009.

[94] S. Porto. A plain english guide to JavaScript prototypes. http://sporto.github.com/blog/2013/02/22/a-plain-english-guide-to-javascript-prototypes/.

[95] M. Pradel and K. Sen. The good, the bad, and the ugly: An empirical study of implicit type conversions in JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015.

[96] U. Praphamontripong and J. Offutt. Applying mutation testing to web applications. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '10, pages 132–141, Washington, DC, USA, 2010. IEEE Computer Society.

[97] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 151–166. ACM, 2013.

[98] F. Ricca and P. Tonella. Analysis and testing of web applications. In *ICSE'01: 23rd Int. Conf. on Sw. Eng.*, pages 25–34. IEEE Computer Society, 2001.

[99] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 52–78. Springer, 2011.

[100] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 1–12. ACM, 2010.

[101] D. Roest, A. Mesbah, and A. van Deursen. Regression testing ajax applications: Coping with dynamism. In *Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST)*, pages 128–136. IEEE Computer Society, 2010.

[102] H. Samimi, M. Schafer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 277–287. IEEE Computer Society, 2012.

[103] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *Proc. Symp. on Security and Privacy (SP)*, pages 513–528. IEEE Computer Society, 2010.

[104] M. Schur, A. Roth, and A. Zeller. Mining behavior models from enterprise web applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 422–432, New York, NY, USA, 2013. ACM.

[105] K. Sen, S. Kalasapur, T., and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proc. European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ES-EC/FSE'013)*. ACM, 2013.

[106] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *Proc. 20th IEEE/ACM Int. Conf. on Automated Sw. Eng. (ASE)*, pages 253–262. ACM, 2005.

[107] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 435–458. Springer, 2012.

[108] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP'12, pages 435–458. Springer-Verlag, 2012.

[109] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. *SIGPLAN Not.*, 42(6):112–122, June 2007.

[110] Stack Overflow. 2015 Developer Survey. http://stackoverflow.com/research/developer-survey-2015, 2015.

[111] W3C. Document Object Model (DOM). http://www.w3.org/DOM/.

[112] W3C. Document Object Model (DOM) level 2 events specification. http://www.w3.org/TR/DOM-Level-2-Events/, 13 November 2000.

[113] S. Wei and B. G. Ryder. Practical blended taint analysis for JavaScript. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 336–346. ACM, 2013.

[114] S. Wei and B. G. Ryder. State-sensitive points-to analysis for the dynamic behavior of JavaScript objects. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 1–26. Springer, 2014.

[115] R. Yandrapally, S. Thummalapenta, S. Sinha, and S. Chandra. Robust test automation using contextual clues. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 304–314. ACM, 2014.

[116] C. Yue and H. Wang. Characterizing insecure JavaScript practices on the web. In *Proceedings of the 18th international conference on World wide web*, pages 961–970. ACM, 2009.

[117] A. Zaidman, N. Matthijssen, M.-A. Storey, and A. van Deursen. Understanding Ajax applications by connecting client and server-side execution traces. *Empirical Software Engineering*, 18(2):181–218, 2013.

[118] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a z3-based string solver for web application analysis. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 114–124. ACM, 2013.