Delft University of Technology
Software Engineering Research Group
Technical Report Series

Crawl-Based Analysis of Web Applications: Prospects and Challenges

Arie van Deursen, Ali Mesbah, and Alex Nederlof

Report TUD-SERG-2014-015

TU Delft

SERG

TUD-SERG-2014-015

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication in *Science of Computer Programming*, 2014, special issue on Program Understanding.

Crawl-Based Analysis of Web Applications: Prospects and Challenges

Arie van Deursen[a,*], Ali Mesbah[b], Alex Nederlof[a]

[a] *Delft University of Technology, The Netherlands*
[b] *University of British Columbia, Canada*

Abstract

In this paper we review five years of research in the field of automated crawling and testing of web applications. We describe the open source Crawljax tool, and the various extensions that have been proposed in order to address such issues as cross-browser compatibility testing, web application regression testing, and style sheet usage analysis.

Based on that we identify the main challenges and future directions of crawl-based testing of web applications. In particular, we explore ways to reduce the exponential growth of the state space, as well as ways to involve the human tester in the loop, thus reconciling manual exploratory testing and automated test input generation. Finally, we sketch the future of crawl-based testing in the light of upcoming developments, such as the pervasive use of touch devices and mobile computing, and the increasing importance of cyber-security.

Keywords: Test automation, web crawling, software evolution.

Personal Message to Paul Klint, from Arie van Deursen

From 1990–1994 and 1996-2005 I had a great time working in the research group headed by Paul Klint at CWI. The work on Crawljax described in this paper mostly dates from after my period at CWI. Nevertheless, the success of Crawljax owes a lot to Paul.

Paul has set an example to many by his enthusiasm for programming. Paul is always programming: in Spring and in Summer, in Lisp, in ASF, in ASF+SDF, in C, in ToolBus script, in Java, and, these days, in Rascal. Even this week (early August 2013), while many of us are secretly contributing to this special issue devoted to him, Paul committed to `GitHub` *every* day.

It is this enthusiasm for programming that has inspired many of his students and co-workers. Thank you Paul for great times at CWI: Your influence goes well beyond the papers you have written. The Software Engineering Research Group at Delft University of Technology has been shaped by your approach to research.

1. Introduction

Modern society critically depends on highly interactive web applications, which hence must be reliable, maintainable, and secure. Unfortunately, the increasing complexity of today's web applications poses substantial challenges into their dependability.

While static analysis of client and server code of web applications can provide valuable insight in their dependability, the highly dynamic nature of today's client-side (JavaScript) code makes dynamic analysis indispensable.

One of the key technologies facilitating these dynamic web applications is Ajax,[1] an acronym for "Asynchronous JavaScript and XML". With Ajax, web-browsers not only offer the user navigation through a sequence of HTML

*Corresponding author

[1] Jesse Garret. "Ajax: A New Approach to Web Applications". February 18, 2005. `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications`

pages, but also responsive rich interaction via graphical user interface components by means of asynchronous processing.

While the use of AJAX technology positively affects user-friendliness and interactiveness of web applications [1], it comes at a price: AJAX applications are notoriously error-prone due to, e.g., their stateful, asynchronous, and event-based nature, the use of (loosely typed) JAVASCRIPT, the client-side manipulation of the browser's Document-Object Model (DOM), and client-server communication based on deltas rather than the exchange of full pages [1].

In our research during the past five years we have gained considerable experience with the use of *crawl-based* dynamic analysis of web applications [2]. In particular, we have developed CRAWLJAX,[2] a tool that can click through an arbitrary web application in order to build up a model of the potential user interactions [3] [4]. Subsequently, this model can be validated against *invariants*, expressing desirable properties (such as the use of valid HTML code only) the system under test should have at any state [5] [6].

The goal of this paper is to explore the prospects and challenges of crawl-based analysis. To that end, we first provide a brief survey of related work, covering our own CRAWLJAX work as well as work by others. Based on that survey, we subsequently explore some of the key open problems in crawl-based analysis, laying out avenues for further research.

2. Crawling Interactive Web Applications

2.1. Challenges

Web crawlers are almost as old as the World Wide Web itself. The first crawler was implemented by Matthey Gray in the spring of 1993. It was called the "Wanderer" and its goal was to measure the size of the web[3]. Soon after that in 1994, the first crawlers that indexed the web appeared [7].

As the web evolved it became less about document sharing and more about interactive content to even full-blown applications. JAVASCRIPT, the dominant browser language, can dynamically generate or load content. Because of this, crawling the web by just following links is not sufficient anymore [8]. To be able to crawl and fetch the dynamic content of a web application, a crawler has to interact with JAVASCRIPT in the browser.

With JAVASCRIPT-enabled crawling, the result of a crawl is a model of the user interaction: A click on some element in the browser can bring the web application in a given state, and exhaustively attempting to execute all possible clicks builds a model of the ways in which a user can interact with the application.

This introduces a number of challenges:

State Explosion: Any click can result in a new state. Even a small web application can have an infinite number of states (think of a simple TODO-list application with states for every possible todo item). Furthermore, content may be time based, or may differ per visitor.

State Navigation: Even though browsers have page forward and backward functionality build in, this is only tied to the application state if the developers choose to. And even if they do, it is a cumbersome error-prone task. This is why web applications often have a different state model than the one that can be derived from the URLs, making the navigation hard to automate [9]. This means that crawlers cannot expect to go to the previous state when they press the back button. They need a more robust system of navigating through the application.

Triggering State Changes: State changes can be caused by many kinds of events in a web page. Clickables are not limited to `<a href="example.com">` elements. JAVASCRIPT allows one to add a click handler to practically any HTML element. Besides clicks, other events may cause a state change, such as hover, mouse-in, mouse-out, drag and drop, double click and right click, as well as touch and touch-gesture events for tablets and smartphones.

To reach all possible states, the crawler could invoke all possible events on all possible elements. But even then the combination of those elements might be the key to going to the next state. For example, some applications

[2] `http://crawljax.com/`
[3] http://www.mit.edu/people/mkgray/growth/

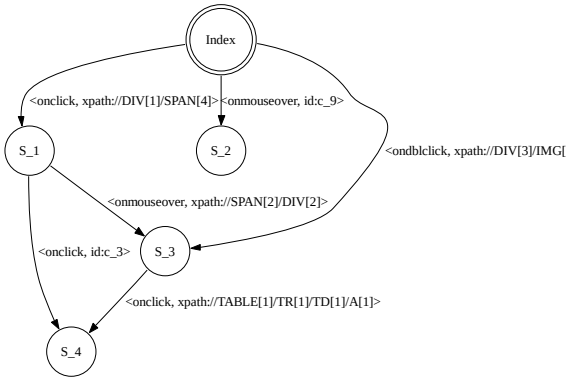

Figure 1: An inferred state-flow graph.

have special states for when a user holds a keyboard key and then click an element. The challenge for crawlers is to either try many of these combinations, or to be smart and discover which elements are listened to by JavaScript. Although finding which elements in JavaScript have listeners is possible, this does not cover the case of input combinations.

Unreachable States: The term "The Deep Web" comes from the traditional crawlers meaning the part of the web that cannot be found by following links [8] [10]. Although JavaScript-enabled crawlers can find more, they face some of the same barriers. The simplest example is that of a user name and password box. Other examples include states that can only be found by entering specific search criteria. Lastly, there are the sites that hide pages by not linking them to any other page and not making them searchable.

2.2. Crawling JavaScript*-based Applications*

To facilitate fully automatic testing we have developed a crawler for JavaScript-based applications [3] [4].

Such a crawler needs to be able to deal with client-side JavaScript code execution, identify elements that can be clicked, keep track of client-side updates to the Document Object Model (DOM) tree, deal with delta-communication between the browser and the server in which only parts of the DOM tree are exchanged instead of full HTML documents, and with the various types of events that users can apply (click, double click, drag and drop, ...).

The approach we propose is based on a *state-flow graph*, which provides a model of the user interface of the web application. The nodes are user interface states, and are represented by the run time DOM tree belonging to the state. Edges correspond to transitions from one state to another, and are labeled with the identification of the DOM-tree element clicked (typically through an XPath expression) to reach the next state. An example state-flow graph is shown in Figure 1.

The proposed crawling approach includes the following steps:

- *Identifying Clickables*

  Since JavaScript code can be used to make any DOM element clickable, identification of clickable elements needs to be done dynamically. A list of *candidate clickable* is determined statically (e.g., all "div" or "href" tags), after which they are tried dynamically. If a click event leads to a modified DOM tree, the element is considered clickable.

- *Comparing States*

  While in principle a new DOM tree is considered a new state, the amount of change matters, since some changes may be less relevant. One approach we use is string-based Levenshtein distance [11], where changes are considered relevant if the distance is beyond a given threshold. Alternatively, we pipeline a series of "comparators", each eliminating one level of irrelevant detail (such as timers, counters, colors, particular names, etc.) or subtrees of the DOM tree [12].

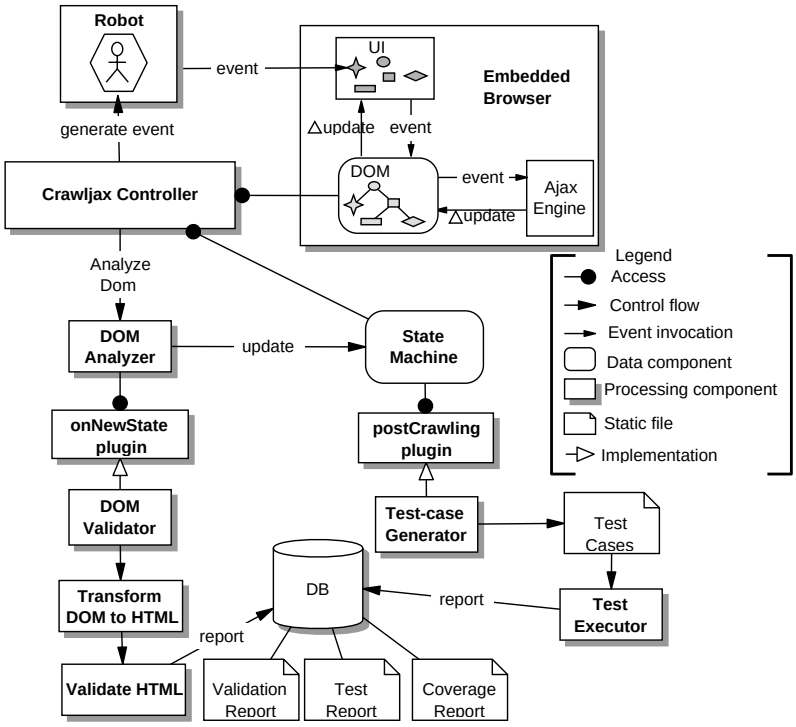

Figure 2: Processing view Crawljax

- *Recurse*

  When a new state is identified, crawling needs to recurse to process the elements on the next state. To determine clickables already processed on the new page, a *diff* algorithm is applied to guide the crawling process to the newly added elements.

- *Backtracking*

  To navigate back after completing a recursive call, one would like to use the browser's Back-button. Unfortunately, this assumes correct registration of client state in the browser's history stack, which in practice is seldom done correctly. Thus, we record the path taken to a particular state, and replay that path in order to backtrack to a previous state.

- *Providing Input Data*

  Data entry points can be filled with random input values, or with one or more custom input values for specific states or input fields. This process needs to be repeatable and configurable.

2.3. Crawljax

The approach is implemented in an Java-based open source tool called CRAWLJAX[4], available via GitHub. It builds upon WebDriver[5] to provide an embedded browser, and hence offers support for Firefox, IE, and Chrome. An architectural view of the processing elements of CRAWLJAX is provided in Figure 2.

Crawljax provides a plug-in mechanism enabling developers to influence the crawling behavior. The stages at which plug-ins can be attached are displayed in Figure 3.

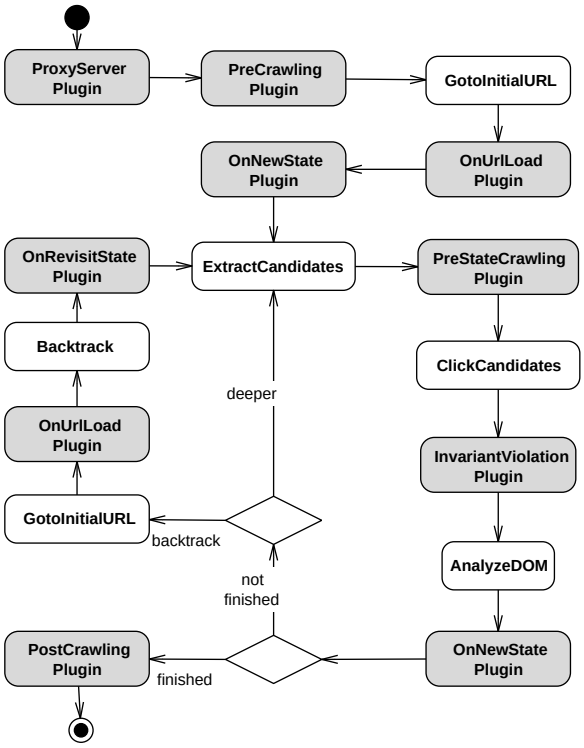

Figure 3: Plug-ins Invocation Flow.

3. A Survey of Research in Crawl-Based Application Analysis

To illustrate the potential of crawl-based analysis of web applications, we provide a brief survey of some of the most important research results of the last few years that has focused on support for automated testing and software evolution.

3.1. Software Testing

Crawling an AJAX application amounts to trying to exercise every possible user interface element of a web application. To turn this into a *test*, some form of *oracle* is needed indicating whether an execution is correct.

A first approach we propose is the use of *invariants* over the DOM-trees of the state-flow graph [5] [6]. These invariants can be *generic* in the sense that they apply to any web application. Examples include the requirement that

[4] http://crawljax.com
[5] http://seleniumhq.org/docs/03_webdriver.html

any DOM should be composed of valid HTML, that there are no broken links, and that all element ID attributes are unique.

Alternatively, such invariants can be application-specific expressing, e.g., the structure of a particular collection of states, or the interaction between two GUI components such as a table-of-contents pane and an actual pane showing the contents. This requires the identification of relevant differences between two derived graphs, and visualization of those differences in order to mark them as regression or as desired modification [12]. Another type of oracle can be inferred from earlier runs. Such runs can be used to obtain inspiration for invariants (in the spirit of Daikon [13]). Furthermore, they can be applied for the purpose of web application regression testing [14].

The derived state-flow graphs can be also used for doing cross-browser testing [15] [16]. In different crawling sessions, models are derived for different (versions of) browsers. Again, relevant changes among the graphs have to be identified and visualized. These changes can reside within the states themselves (different DOM details for conceptually the same state), or at the level of the graph itself (when certain states are unreachable for a particular browser).

Another recent direction is using a dynamic execution trace obtained from crawling for mutation testing in JavaScript applications [17]. This way, the fault finding ability of JavaScript test cases can be automatically assessed.

3.2. Software Evolution

The insight obtained through crawling can be used beyond testing purposes. An example is the analysis of Cascading Style Sheet (CSS) code [18]. Through Crawljax, dynamically applied CSS rules can be identified, and their actual use can be analyzed. An empirical study identified an average of 60% unused CSS selectors in deployed applications [18].

The subject of analysis can also be the JavaScript code actually used in web applications, by intercepting the JavaScript code through a proxy server. This has been used to identify JavaScript-specific *code smells*, and their occurrence [19]. The results indicate that lazy object, long method/function, closure smells, coupling between JavaScript, HTML, and CSS, and excessive global variables are the most prevalent code smells.

JavaScript-based crawling is essential to identify the part of the web that is only accessible via JavaScript. This is part of the "hidden web", the part of the web that is not reachable through search engines. The size of the JavaScript-induced part of the hidden web has been analyzed: As it turns out, over 60% of the states in a sample of 500 web applications are hidden, and can only be accessed via the browser [20].

4. Research Directions in Crawl-Based Analysis

4.1. Benchmarking

In order to move the field of JavaScript-enabled crawling and analysis forward, a shared dataset is required.

As a first step, we have collected crawl results of over 4000 online web applications, randomly selected from the Internet [21]. The result is a collection of state-flow graphs including the DOM-tree for each dynamic state.

Through this dataset, we seek to answer such questions as (1) how many states can only be found by executing client-side code; (2) what sort of errors are typically found in such states, if any; (3) how can these faults be detected; (4) to what extent does their detection require full crawling.

Our first results indicate that 90% of the web sites investigated conduct JavaScript-enabled DOM manipulations after the initial load, that half of the web applications contain non-unique id-fields in their DOM, and that for half of the applications style sheets are loaded too late on the page resulting in unnecessary re-rerending [21]. Further research is needed to expand the scope of this benchmark to more sophisticated web development problems.

Turning crawl-results into static state-flow graphs has the additional benefit of allowing researchers relying on statically obtained web pages to work with dynamically obtained pages as well. In this way, crawl-based analysis of stored states may provide a useful sweet spot between purely static analysis and dynamic web application analysis.

4.2. Guided Crawling

A general random crawler that exhaustively explores the states can become mired in limited specific regions of the web application, yielding poor coverage. As an alternative to exhaustive random crawling methods, such as depth or breadth- first, a crawling strategy can guide the crawler to more ”relevant” states. For instance, efficient guided strategies [22] try to discover relevant states in the shortest amount of time. Feedback-directed exploration algorithms [23] try to guide the crawler towards maximizing JavaScript code coverage, page diversity and path diversity at runtime. Research in guided crawling of web applications has just started and results indicate this to be a promising direction to pursue.

4.3. Example-Based Crawling

While crawling aims at full automation, bringing humans in the loop may be advantageous in several settings. In particular, humans may have the domain knowledge to see which interactions are more likely than others, and they may be able to use domain knowledge to enter data into forms.

This gives rise to *example-based crawling*, in which human interactions are recorded, and used as seed to generate further crawls. In the field of testing, this would create a blend between fully manual *exploratory testing* [24], resulting in traces that are subsequently further explored automatically. Likewise, in agile projects, the team may manually create acceptance tests (which may or may not be automated). These will typically correspond to the most common happy path. Crawl-based analysis can subsequently expand these to automatically test bad weather behavior.

Such example-based crawling may also be beneficial for testing of mobile applications, which faces many challenges [25]. Such apps may support gesture-based or drag-and-drop input which is hard to trigger automatically from a crawler. Instead, such gestures can be recorded interactively, and then used in a subsequent automated phase.

4.4. Model-Based Web Application Analysis

While much can be learned from the states that are only visible after JavaScript-enabled user interactions, the sequence of events in the state-flow graph is another potentially useful resource for analysis.

This raises several questions. The first is whether the current state-flow graph is the most suitable for conducting model-based testing. We conjecture that further abstractions on the state-flow graph are desirable, such as the introduction of superstates, or the use of hyper edges (as used in graph grammars) to represent recursive behavior in web applications.

Another question is which test generation algorithm to use. While the state-flow graph itself maybe be huge and hard to infer, it could form the basis for deriving a substantially smaller set of test cases that traverses large parts of the state-flow graph. For this, different (state-based) coverage criteria can be used (see, e.g., [26] related to the sequences of events that should be covered

Alternatively, search-based techniques can be used. For example, Tonella et al investigate the use of search-based algorithms for Ajax event sequence generation during testing, in order to find semantically interacting event sequences [27].

4.5. Cyber-Security

In cyber-security, many vulnerabilities can be related to browsers in general, and JavaScript in particular. Examples include cross site scripting, drive by downloads, or evasion attacks, to name a few.

To illustrate this, Yue and Wang provide a study of insecure JavaScript practices [28], showing that two thirds of the web sites measured manifest insecure practices. The study is based on data from 2008, but as the use of JavaScript has increased substantially since then, we conjecture that the problem has become larger rather than smaller.

While much static analysis research is available to identify insecure practices in individual code fragments, the dynamic loading of JavaScript and the dynamic modification of DOM-trees renders static analysis insufficient. Static analysis combined with crawling promises to offer a hybrid solution, in which the crawler collects all relevant JavaScript in combination with the specific DOM-trees manipulated, after which the static analysis technique can be used to discover vulnerabilities in the resulting state.

This requires tailoring JavaScript-based crawling towards security concerns. This can include specializing the random input generator for forms towards security sensitive inputs (fuzzing), guiding the crawling so that the most

sensitive clicks are attempted first (penetration testing), and the inclusion of security-specific oracles, aimed at spotting known vulnerabilities.

CRAWLJAX is relevant to security in at least two ways: The crawling helps to unravel all JAVASCRIPT that is potentially used, instead of the possibly small subset that happens to be loaded on a first page. Subsequently, all known static JAVASCRIPT analysis techniques can be applied to the code fragments occurring in every different sate. Furthermore, crawling is aking to fuzz testing, generating not just (random or security sensitive) inputs, but also exercising all (click) events. First attempts at using JAVASCRIPT-enabled crawling have been made in our earlier work on identifying illegal web widget interactions [29], but much remains to be done.

5. Concluding Remarks

Today's web applications are more and more moving towards the single-page paradigm, in which a JAVASCRIPT engine is responsible for maintaining the DOM-represented user interface and interaction. This poses important analysis and understanding challenges, which go beyond the capabilities of state of the art static analysis tools.

In this paper, we have explored how automated crawling can help to address these challenges. In particular, we have provided a survey of five years of research in analyzing and understanding web applications through automated crawling. Furthermore, we identified a number of important and promising areas of future research in the field of dynamic analysis of modern web applications.

References

[1] A. Mesbah, A. van Deursen, A Component- and Push-based Architectural Style for Ajax Applications, Journal of Systems and Software 81 (12) (2008) 2194–2209.

[2] A. Mesbah, Analysis and Testing of Ajax-based Single-Page Web Applications, Ph.D. thesis, Delft University of Technology, 2009.

[3] A. Mesbah, E. Bozdag, A. van Deursen, Crawling AJAX by inferring user interface state changes, in: Proceedings Eight International Conference on Web Engineering (ICWE'08), IEEE, 122–134, 2008.

[4] A. Mesbah, A. van Deursen, S. Lenselink, Crawling AJAX-Based Web Applications through Dynamic Analysis of User Interface State Changes, ACM Transactions on the Web 6 (1) (2012) 3:1–3:30.

[5] A. Mesbah, A. van Deursen, Invariant-based automatic testing of AJAX user interfaces, in: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), IEEE, 210–220, 2009.

[6] A. Mesbah, A. van Deursen, D. Roest, Invariant-Based Automated Testing of Modern Web Applications, IEEE Transactions on Software Engineering 38 (1) (2012) 35–53.

[7] B. Pinkerton, Finding What People Want: Experiences with the WebCrawler, in: Proceedings Second International WWW Conference, 17–20, 1994.

[8] M. K. Bergman, White Paper: The Deep Web: Surfacing Hidden Value, The Journal of Electronic Publishing 7 (1).

[9] P. Montoto, A. Pan, J. Raposo, F. Bellas, J. López, Automated browsing in AJAX websites, Data Knowl. Eng. 70 (3) (2011) 269–283.

[10] A. Heydon, M. Najork, Mercator: A Scalable, Extensible Web Crawler, World Wide Web 2 (4) (1999) 219–229.

[11] V. L. Levenshtein, Binary Codes Capable of Correcting Deletions, Insertions, and Reversals, Cybernetics and Control Theory 10 (1996) 707–710.

[12] D. Roest, A. Mesbah, A. van Deursen, Regression testing Ajax applications: Coping with dynamism, in: Proceedings Third International Conference on Software Testing, Verification and Validation (ICST), IEEE, 127–136, 2010.

[13] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, C. Xiao, The Daikon system for dynamic detection of likely invariants, Science of Computer Programming 69 (1–3) (2007) 35–45.

[14] S. Mirshokraie, A. Mesbah, JSART: JavaScript Assertion-based Regression Testing, in: Proceedings of the 12th International Conference on Web Engineering (ICWE), Springer, 238–252, 2012.

[15] A. Mesbah, M. Prasad, Automated cross-browser compatibility testing, in: Proceedings of the 33rd International Conference on Software Engineering, ACM, 561–570, 2011.

[16] S. Roy Choudhary, M. R. Prasad, A. Orso, X-PERT: accurate identification of cross-browser issues in web applications, in: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, 702–711, 2013.

[17] S. Mirshokraie, A. Mesbah, K. Pattabiraman, Efficient JavaScript Mutation Testing, in: Proceedings of the International Conference on Software Testing, Verification and Validation (ICST), IEEE Computer Society, 2013.

[18] A. Mesbah, S. Mirshokraie, Automated analysis of CSS rules to support style maintenance, in: Proceedings 34th International Conference on Software Engineering (ICSE 2012), IEEE, 408–418, 2012.

[19] A. Milani Fard, A. Mesbah, JSNose: Detecting JavaScript Code Smells, in: Proceedings of the IEEE International Conference on Source Code Analysis and Manipulation (SCAM), IEEE Computer Society, 10 pages, 2013.

[20] Z. Behfarshad, A. Mesbah, Hidden-Web Induced by Client-Side Scripting: An Empirical Study, in: Proceedings of the International Conference on Web Engineering (ICWE), vol. 7977 of *Lecture Notes in Computer Science*, Springer, 52–67, 2013.

[21] A. Nederlof, A. Mesbah, 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), ACM, 4–13, URL `http://salt.ece.ubc.ca/publications/docs/icse14-seip.pdf`, 2014.

[22] S. Choudhary, M. E. Dincturk, S. M. Mirtaheri, G.-V. Jourdan, G. von Bochmann, I. V. Onut, Building Rich Internet Applications Models: Example of a Better Strategy, in: Proceedings of the International Conference on Web Engineering (ICWE), Springer, 2013.
[23] A. Milani Fard, A. Mesbah, Feedback-directed Exploration of Web Applications to Derive Test Models, in: Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering (ISSRE), IEEE Computer Society, 10 pages, 2013.
[24] E. Hendrickson, Explore It!: Reduce Risk and Increase Confidence with Epxloratory Testing, The Pragmatic Programmer, 2013.
[25] M. Erfani Joorabchi, A. Mesbah, P. Kruchten, Real Challenges in Mobile App Development, in: Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE Computer Society, 10 pages, 2013.
[26] J. Tretmans, Model Based Testing with Labelled Transition Systems, in: Formal Methods and Testing, vol. 4949 of *Lecture Notes in Computer Science*, Springer, 1–38, 2008.
[27] A. Marchetto, P. Tonella, Using search-based algorithms for Ajax event sequence generation during testing, Empirical Software Engineering 16 (1) (2011) 103–140.
[28] C. Yue, H. Wang, A measurement study of insecure javascript practices on the web, ACM Trans. Web 7 (2) (2013) 7:1–7:39.
[29] C.-P. Bezemer, A. Mesbah, A. van Deursen, Automated Security Testing of Web Widget Interactions, in: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09). Research Papers, ACM, 81–91, 2009.

TUD-SERG-2014-015
ISSN 1872-5392