Delft University of Technology
Software Engineering Research Group
Technical Report Series

Research Issues in the Automated Testing of Ajax Applications

Arie van Deursen and Ali Mesbah

Report TUD-SERG-2009-032

TUD-SERG-2009-032

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: *This paper is a pre-print of:*
Arie van Deursen and Ali Mesbah. Research Issues in the Automated Testing of Ajax Applications. In *Proceedings 36th International Conference on Current Trend in Theory and Practice of Computer Science (SOFSEM)*, pp. 16-28. Lecture Notes in Computer Science 5901, Springer-Verlag, 2010.

Research Issues in the Automated Testing of Ajax Applications

Arie van Deursen and Ali Mesbah

Delft University of Technology
{arie.vandeursen,a.mesbah}@tudelft.nl

Abstract. There is a growing trend to move desktop applications towards the web. This move is made possible through advances in web technologies collectively known as Asynchronous JavaScript and XML (Ajax). With Ajax, the classical model of browsing a series of pages is replaced by a JavaScript engine (running in the browser) taking control of user interaction, exchanging information updates with the web server instead of requesting the complete next page. The benefits of this move include no installation costs, automated upgrading for all users, increased interactivity, reduced user-perceived latency, and universal access, to name a few. Ajax, however, comes at a price: the asynchronous, stateful nature and the use of JavaScript make Ajax applications particularly error-prone, causing serious dependability threats. In this paper, we evaluate to what extent automated testing can be used to address these Ajax dependability problems. Based on an analysis of the current challenges in testing Ajax, we formulate directions for future research.

1 Introduction

There is a growing trend to move applications towards the Web. Well-known examples include Google's mail and office applications including spreadsheet, word processing, and calendar applications. The reasons for this move to the web are manifold and include:

- No installation effort for end-users.
- Automatic use of the most recent software version by all users, thus reducing maintenance and support costs.
- Universal access from any browser on any machine with Internet access;
- Possibility to share data and enrich user interaction with information available at the server.
- In-depth insight for software developers in which features are actually used.
- Customization per user, based on, e.g., earlier experience with the application
- Fast innovation cycles, since releasing and deploying a new version is instantaneous.

In an interesting recent blog post [23], McKenzie also argues that the *conversion rate* for web applications is better, i.e., the percentage of site visitors that

actually *purchase* a software product is higher for license-based web applications than for applications they have to download and install. McKenzie furthermore argues that web applications solve the problem of software piracy, simply by the fact that there is no software anymore that is to be downloaded and (illegally) distributed.

One of the implications of this move to the web, is that *dependability* [3] of web applications is becoming increasingly important [11, 29]. Dependability is affected by many factors, including the level of testing, the skills of the developers involved, and the actual software technology used.

For today's web applications, one of the key technologies used is Ajax, an acronym for "Asynchronous JavaScript and XML" [15]. With Ajax, web-browsers not just offer the possibility to navigate through a sequence of HTML pages, but enable rich user interaction via graphical user interface components.

While the use of Ajax technology positively affects user-friendliness and interactiveness of web applications [26], it comes at a price: Ajax applications are notoriously error-prone due to, e.g., the stateful and asynchronous nature as well as the use of (untyped) JavaScript (see Section 3).

In this paper, we will explore how testing can be used to improve the dependability of Ajax applications. In particular, we first provide an abstract view on what exactly is comprised by Ajax. We do this in Section 2, by means of an *architectural style* capturing the essential elements of Ajax applications. Next, in Section 3, we offer a survey of our work on the automated testing of Ajax applications. In particular, we discuss a plugin-based tool infrastructure called Atusa, which can be used to detect a range of faults typically occurring in Ajax applications. We conclude the paper with an analysis of open issues and research problems in the area of automated testing of Ajax applications.

2 Defining Ajax

Ajax potentially brings an end to the classical *click-and-wait* style of web navigation, providing the responsiveness and interactivity level end users usually expect from desktop applications. In a classical web application, the user has to wait for the entire page to reload to see the response of the server. With Ajax, however, small delta messages are requested from the server, behind the scenes, by the Ajax engine and updated on the current page through modifications to the corresponding 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.

Ajax gives us a vehicle to build web applications with a *single-page web interface*, in which all interactions take place on one page. Single-page web interfaces can improve complex, non-linear user work-flows [39] by decreasing the number of click trails and the time needed [38] to perform a certain task, when compared to classical multi-page variants.

Another important aspect of Ajax is that of enriching the web user interface with interactive components and widgets. Examples of widgets, which can all co-

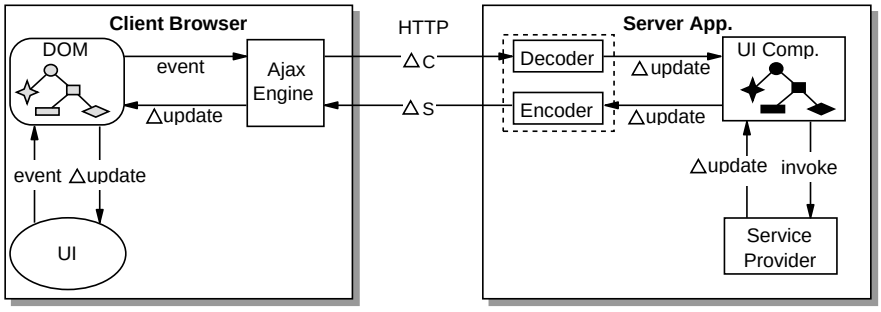

Fig. 1. Processing View of a SPIAR-based architecture.

exist on the single-page web interface, include auto-completion for input fields, in-line editing, slider-based filtering, drag and drop, rich tables with within-page sorting, shiny photo albums and calendars, to name a few. These are all web user interface components that are made possible through extensive DOM programming by means of JAVASCRIPT and delta client/server communication.

In most classical web applications, a great deal of identical content is present in page sequences. For each request, the response contains all the redundant content and layout, even for very marginal updates. Using AJAX to update only the relevant parts of the page results, as expected, in a decrease in the bandwidth usage. Experimental results have shown a performance increase of 55 to 73% [35, 24, 38] for data transferred over the network, when AJAX is used to conduct partial updates.

In our earlier work we have proposed SPIAR, an *architectural style* [14, 31] for AJAX [26]. SPIAR results from a study of different major AJAX frameworks, investigating their salient architectural properties, key elements, and constraints on those elements required to achieve the desired properties. Such a style captures the essence of AJAX frameworks and can be seen as an abstract model of different architectural implementations.

In SPIAR three types of architectural elements can be identified: processing (e.g., browser, AJAX engine, server application, service provider, user interface components), connectors (e.g., events, delta update, push channels), and data (e.g., representation, representational model, delta messages).

Given the processing, data, and connecting elements, we can use different architectural views to describe how the elements work together to form an architecture. Figure 1 depicts the processing view of an SPIAR-based architecture based on run-time components rendering as in, e.g., Echo2[1]. The view shows the interaction of the different components some time after the initial page request (the engine is running on the client). User activity on the user interface fires off an event to indicate some kind of component-defined action which is delegated to the AJAX engine. If a listener on a server-side component has registered itself with the event, the engine will make a delta-client message of the current state

[1] `http://echo.nextapp.com/site/echo2`

changes with the corresponding events and send it to the server. On the server, the decoder will convert the message, and identify and notify the relevant components in the component tree. The changed components will ultimately invoke the event listeners of the service provider. The service provider, after handling the actions, will update the corresponding components with the new state which will be rendered by the encoder. The rendered delta-server message is then sent back to the engine which will be used to update the representational model and eventually the interface. The engine has also the ability to update the representational model directly after an event, if no round-trip to the server is required.

Table 1. Constraints and induced properties in AJAX applications.

	User Interactivity	User-perceived Latency	Network Performance	Simplicity	Scalability	Portability	Visibility	Data Coherence	Reliability	Adaptability
Single-page Interface	+									
Asynchronous Interaction	+	+								
Delta Communication	+	+	+		–		–	+		
Client-side processing	+	+	+							
UI Component-based	+			+					+	+
Web standards-based				+		+			+	
Stateful	+	+	+		–		–			
Push-based Publish/Subscribe		+	+		–		–	+		+

Architectural constraints can be used as restrictions on the roles of the architectural elements to induce the architectural properties desired of a system. Table 1 presents an overview of the constraints and induced properties. A "+" marks a direct positive effect, whereas a "–" indicates a direct negative effect. SPIAR rests upon these constraints, which are chosen to retain the properties such as user interactivity, data coherence, and scalability.

3 State of the Art in Ajax Testing

For traditional software, analysis and testing is still largely ad hoc [5] and already a notoriously time-consuming and expensive process [4]. Classical web applications present even more challenges [10, 2] due to their distributed, heterogeneous nature. In addition, web applications have the ability to generate different user interfaces in response to user inputs and server state.

AJAX-based web applications are not only fundamentally different from classical web applications, but also more error-prone and harder to test. The reasons for this include the stateful as well as asynchronous nature of AJAX programming, the client-side manipulation of the Document Object Model, the use of

delta-communication, and the limited possibilities for static (type) checking of JavaScript. In spite of the great success of Ajax, building dependable Ajax applications is a daunting task. Below we will discuss the current state of Ajax testing approaches.

3.1 Current Testing Approaches

The server-side of Ajax applications can be tested with any conventional testing technique. On the client, testing can be performed at different levels. Unit testing tools such as JsUnit[2] can be used to test JavaScript on a functional level. The most popular Ajax testing tools are currently capture/replay tools such as Selenium,[3] which allow DOM-based testing by capturing events fired by user (tester) interaction. Such tools have access to the DOM, and can assert expected UI behavior defined by the tester and replay the events. Capture/replay tools demand, however, a substantial amount of manual effort on the part of the tester.

Marchetto *et al.* [21] discuss a case study in which they demonstrate the effectiveness of applying traditional web testing techniques (e.g., code coverage testing [33], model-based testing [2], session based testing [12, 36]) to Ajax. Their analysis suggests that such traditional techniques have serious limitations in testing modern Ajax-based web applications. They propose [22] an approach for state-based testing of Ajax applications based on traces of the application to construct a finite state machine. Sequences of semantically interacting events in the model are used to generate test cases once the model is refined by the tester. In our recent approach [27], the focus is on automating the testing process by inferring an abstract model of the Ajax application and generating test cases automatically.

3.2 Automatic Testing of Ajax

In order to detect a fault automatically, a testing method should meet the following conditions [28, 34]: *reach* the fault-execution, which causes the fault to be executed, *trigger* the error-creation, which causes the fault execution to generate an incorrect intermediate state, and *propagate* the error, which enables the incorrect intermediate state to propagate to the output and cause a detectable output error. In addition, automating the process of assessing the correctness of test case output is a challenging task, known as the oracle problem.

Meeting these reach/trigger/propagate/oracle conditions is more challenging for Ajax applications compared to the classical ones.

One way to *reach* the fault-execution *automatically* for web applications is by adopting a web crawler to crawl through different UI states and infer a model of the navigational paths and states.

[2] `http://jsunit.net`
[3] `http://selenium.openqa.org`

3.3 Crawling Ajax

A general approach in testing the client-side of web applications has been to request a response from the server and analyze the resulting HTML page. This testing approach based on the page-sequence paradigm has serious limitations when applied to AJAX-based applications. AJAX has a number of properties making it difficult, for e.g., search engines, to crawl. We will briefly outline these challenges below.

Client-side Execution. Any search engine willing to approach such an application must have support for the execution of the scripting language. Equipping a general search crawler with the necessary environment complicates its design and implementation considerably.

Navigation. Ultimately, an AJAX application could consist of a single page with a single URL. This characteristic makes it very difficult for a search engine to index and point to a specific state on an AJAX application. For crawlers, navigating through traditional multi-page web applications has been as easy as extracting and following the hypertext links (or the `src` attribute of, e.g., image tags) on each page.

Dynamic Document Object Model (DOM). The state changes in AJAX applications are dynamically represented through the run-time changes on the DOM. This means that the source code in HTML does not represent the state anymore. Any search engine aimed at crawling and indexing such applications, will need to have access to this run-time dynamic document object model of the application.

Delta-communication. Retrieving and indexing the delta state changes from the server, for instance through a proxy between the client and the server, could have the side-effect of losing the context and actual meaning of the changes. Most of such delta updates become meaningful after they have been processed by the JAVASCRIPT engine on the client and injected into the DOM.

Events and Clickables. In AJAX, hypertext links can be replaced by elements with event-listeners, which are handled by the client engine; it is not possible any longer to navigate the application by simply extracting and retrieving the internal hypertext links. DOM Events (e.g., `onClick, onMouseOver`) can be attached to DOM elements at run-time, and as such, even a `div`-element typically used for styling or structuring can have an `onclick` event attached to it so that it becomes a *clickable* element capable of changing the internal DOM state of the application when clicked. The necessary event handlers can also be programmatically registered in AJAX. Finding these clickables at run-time is another non-trivial task for a crawler.

Despite these challenges, we have proposed [25] a new type of web crawler, called CRAWLJAX, capable of exercising client-side code, detecting and executing doorways (clickables) to various dynamic states of AJAX-based applications within browser's dynamically built DOM. While crawling, CRAWLJAX infers a *state-flow graph* capturing the states of the user interface, and the possible event-based transitions between them, by analyzing the DOM before and after firing an

event. CRAWLJAX is open source[4] and based on an embedded browser interface (with different implementations: IE, Firefox) capable of executing JAVASCRIPT and the supporting technologies required by AJAX.

3.4 Invariant-based Testing

Once we are able to derive different dynamic states of an AJAX application, possible faults can be triggered by generating UI events and identifying entry points.

In our recent work [27], we have presented an approach for automatic testing of AJAX user interfaces, called ATUSA. ATUSA is based on the crawling capabilities of CRAWLJAX and provides data-entry point detection and (pre-, in-, and post-crawling) plugin hooks for testing AJAX applications.

To tackle the oracle problem, we have proposed to use generic and application-specific structural *invariants* that serve as oracle to detect faults in and between different DOM states. Such oracles can be defined in various forms such as XPath expressions, Regular expressions, or JAVASCRIPT conditions.

3.5 Test-case Generation

While running ATUSA to derive the state machine can be considered as a first full test pass, the state machine itself can be further used for testing purposes. For example, it can be used to execute different paths to cover the state machine in different ways. To that end, we derive a test suite (implemented in JUnit) automatically from the state machine, which can be used for regression testing of AJAX applications. Figure 2 depicts the processing view of ATUSA, showing a pre-crawling DOM Validator and a post-crawling Test Case Generator as examples of possible plugin implementations.

3.6 Security Testing

AJAX applications can be composed from independent user interface components, often called web *widgets*. As any program code, widgets can be used for malicious purposes. Example scenarios include when a malicious widget changes the content of another widget to trick the user into releasing sensitive information, or even worse, listens to the account details a user enters in another widget (e.g., PayPal or Email widgets) and sends the data to a malicious site.

Testing modern web applications for security vulnerabilities is far from trivial. Traditional detection-based approaches are generally static analysis-based, which has limitations in revealing faults and violations in the dynamic distributed runtime behavior of modern rich web applications.

In our latest work [6], we propose to extend and use ATUSA for automatically spotting two types of security problems in widget interactions, namely, the case in which (1) a malicious widget changes the content (DOM) of another widget,

[4] `http://spci.st.ewi.tudelft.nl/crawljax/`

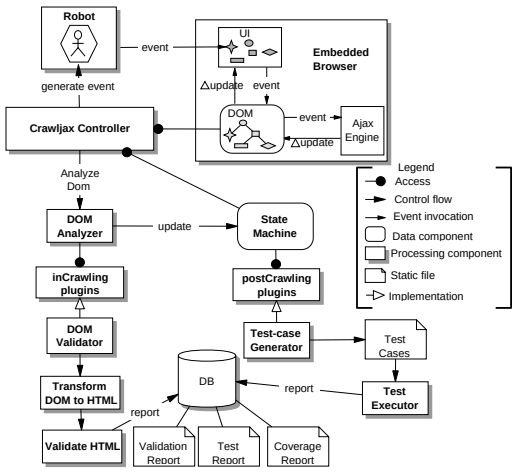

Fig. 2. Processing view of ATUSA.

and (2) a widget steals data from another widget and sends it to the server via an HTTP request.

In order to find DOM change violations (1), we first need to automatically detect each widget's boundary in the DOM tree. Once the boundaries are defined, we can analyze the elements receiving events and the actual changes taking place on the DOM tree to decide whether a state change is a violation.

For HTTP request violations (2), the main challenge is in coupling each outgoing request with the corresponding DOM element, from which it originated. Once we know which element is causing the request, we can analyze the behavior and decide whether a violation has occurred.

Our approach, implemented in a number of open source ATUSA plugins, called DIVA, requires no modification of application code, and has few false positives.

4 Open Research Questions

The potential of automatic testing of AJAX applications is high, but there are a number of problems and challenges that need to be addressed, related to the scalability and usability of the proposed approach. In this section, we sketch the main areas of future research we anticipate.

4.1 Invariants in Practice

Testing in ATUSA is based on the notion of *invariants*. This is a fairly weak form of an oracle, which can be used to conduct basic sanity checks on the DOM-tree or transitions in the derived GUI state machine. While initial experiments using DOM-based invariants were successful [27] a number of research questions remain.

1. While in academia the notion of design invariants is well-understood industrial practice has been reluctant to pick up the idea [8]. The premise of ATUSA is that essential design decisions can be captured into invariants. To what extent is this indeed the case? Are developers capable and willing to document these decisions by means of invariants? What is the best notation to express invariants? Are invariants sufficiently stable across different versions of an AJAX application?
2. Another premise in ATUSA is that invariants are effective in finding faults. We anticipate that the more application-specific an invariant is, the likelier it will be that it can reveal a programming fault. Is this indeed the case? How common are violations of generic invariants (concerning, e.g., HTML validity)? What sort of application-specific invariants are likely to reveal faults?
3. Ernst *et al.* have used dynamic analysis to infer "likely invariants" from execution traces [13]. An interesting question is to what extent this would be possible in our setting as well. Can we analyze the DOM in every state, and discover properties on the DOM that must always hold? Can we use the corresponding invariants for testing in subsequent versions of the AJAX application? Can we infer client-side JAVASCRIPT invariants automatically through dynamic analysis?

Note that many of these questions are empirical in nature. Therefore, in order to answer them it is necessary to to have access to several AJAX development projects, so that rigorous case studies [40] can be conducted.

4.2 Combinatorial Testing

The combinatorial explosion of the test space is one of the key problems in software testing. A system with N features each having M possibilities, leads to M^N test cases, which rapidly becomes intractable.

To deal with this problem various approaches have been proposed. A well-known method is *Category-Partition*, in which independently testable features and parameter characteristics are identified [30]. In this approach, constraints are used to limit the number of combinations that must be checked.

An alternative is *pairwise* combination testing. In this approach, not all possible combinations, but just all possible *value pairs* between two features are tested (or, more generally, k-tuples for $k < N$) [9]. In this approach, the state space grows logarithmically rather than exponentially. Furthermore, empirical evidence suggests that, in practice, faults are mostly due to two or sometimes three way interactions, making pairwise testing an effective approach [20].

The approach currently used in ATUSA investigates all possibilities, and hence suffers from the combinatorial explosion problem. In order to apply techniques such as category-partition or pairwise combination testing, we need to identify *independent* parts of the DOM-tree. Are annotations provided by developers an effective means to identify such independent parts? To what extent can independent DOM-fragments be found automatically? Are DOM-fragments a good

starting point for applying combinatorial testing? To what reductions does this lead in practice?

4.3 State Space Reduction

Related to scalability is the state space explosion problem (see, e.g., [32]). In particular, in the area of *model checking*, a significant body of research has been devoted to reducing state spaces [18].

When deriving state machines from executions, which is what we do in CRAWLJAX, an *abstraction function* is used for mapping concrete program states to abstract GUI states in our state-flow graph. Can we strengthen our abstraction function, and merge more states together? Can we reuse techniques from the area of model checking to manage our state space? Can we involve the software engineering in suggesting states to merge, for example through annotations? Can we reduce the state machine memory footprint by adopting techniques such as hashcode computation, state compression, recursive indexing [17], delta update, or Sweep Line [7]? Is the total running time reducible by using concurrent computation?

4.4 Regression Testing

Regression testing encompasses the selective re-testing of a system to verify that modifications have not caused unintended effects and that the system still complies with its specified requirements [19]. Regression testing of web applications [37] in general and AJAX-based applications in particular is far from trivial due to the high degree of dynamism in such applications. This dynamism is usually caused by various factors such as input data from different users, server-side state, order of event sequences, etc.

In ATUSA for instance, when the generated test suite is run for regression testing, states as seen in the browser are compared with the states in the oracle (the baseline). Imagine a page that displays a date-time that changes after each retrieval. A simple string comparison would result in many false test failures. Even changing the order of followed events can result in a different state than expected according to the baseline. How can we cope with this high level of dynamism in AJAX applications when conducting regression testing? Can we implement or better yet generate intelligent oracle comparators that ignore such state differences so that we can only report real failures?

4.5 Path Seeding

Instead of starting from a single root to explore possible clicks in an AJAX application, a given sequence of clicks can be used as a starting point. From such a click trail, side paths can be explored automatically, for example within a given distance of the original sequence.

This opens various opportunities to refine the way test cases are generated. These initial sequences can be obtained from a first round of manual (acceptance) testing, for example through the use capture-and-playback tools (such as the aforementioned Selenium) or defining pre-conditions on the (e.g., DOM or JavaScript variables) states. Alternatively, the initial trails may be picked to correspond to an operational profile, and thus reflect typical usage scenarios. Subsequently, Atusa's automated capabilities can be used to expand these initial sequences to a series of closely related sequences.

A particularly intriguing route is to use *failure-inducing* paths as seeds, and then attempt to do automated fault diagnosis [1, 16]. Such failing runs can correspond to click trails generated by Atusa that lead to an invariant violation. To spot the cause of the failure, Atusa can then collect trails that are, in one way or another, similar, which do not lead to an invariant violation. Traditional spectrum-based analysis can then be used to identify the differences between these trails in terms of the underlying functionality that gets executed (by instrumenting, e.g., the underlying JavaScript code), which then can be used to localize the root cause of the fault. This amounts to combining click trail similarity with fault diagnosis: a promising direction which, however, requires further research to investigate the feasibility and benefits.

5 Concluding Remarks

As more and more applications are moved to the web, Ajax technology plays an increasingly important role in our society. Unfortunately, the state-based, asynchronous nature of Ajax in combination with the limited possibilities for static analysis of rich Internet applications, pose an increasing threat to dependability.

One way to deal with this threat is the use of automated testing. This requires the use of a crawler that can detect and follow clickable elements introduced by client-side logic. Furthermore, it requires the capability of distinguishing correct from incorrect executions, for which we propose to rely on invariants expressed over the browser's Document Object Model.

While this approach has proven successful in various case studies, a number of questions remain, related in particular to the scaleability of the approach. In order to address these concerns, in this paper we have surveyed a number of research directions and areas of future research, in which techniques from traditional testing are made to work with the specific constraints and opportunities imposed by Ajax applications.

References

1. R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98, Washington, DC, USA, 2007. IEEE Computer Society.

2. A. Andrews, J. Offutt, and R. Alexander. Testing web applications by modeling with FSMs. *Software and Systems Modeling*, 4(3):326–345, July 2005.
3. A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1):11–33, 2004.
4. B. Beizer. *Software Testing Techniques (2nd ed.)*. Van Nostrand Reinhold Co., 1990.
5. A. Bertolino. Software testing research: Achievements, challenges, dreams. In *ICSE Future of Software Engineering (FOSE'07)*, pages 85–103. IEEE Computer Society, 2007.
6. C.-P. Bezemer, A. Mesbah, and A. van Deursen. Automated security testing of web widget interactions. In *Proceedings of the 7th joint meeting of the European Eoftware Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering (ESEC-FSE'09)*, pages 81–91. ACM, 2009.
7. S. Christensen, L. M. Kristensen, and T. Mailund. A sweep-line method for state space exploration. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, pages 450–464. Springer-Verlag, 2001.
8. L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37, 2006.
9. D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatiorial design. *IEEE Trans. Software Eng.*, 23(7):437–444, 1997.
10. G. A. Di Lucca and A. R. Fasolino. Testing web-based applications: The state of the art and future trends. *Inf. Softw. Technol.*, 48(12):1172–1186, 2006.
11. S. Elbaum, K.-R. Chilakamarri, B. Gopal, and G. Rothermel. Helping end-users 'engineer' dependable web applications. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pages 31–40. IEEE Computer Society, 2005.
12. S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *Proc. 25th Int Conf. on Software Engineering (ICSE'03)*, pages 49–59. IEEE Computer Society, 2003.
13. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.
14. R. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, UC, Irvine, Information and Computer Science, 2000.
15. J. Garrett. Ajax: A new approach to web applications. Adaptive path, February 2005. `http://www.adaptivepath.com/publications/essays/archives/000385.php`.
16. M.-J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *PASTE '98: Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 83–90, New York, NY, USA, 1998. ACM.
17. G. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *In Proceedings of Third International SPIN Workshop*, 1997.
18. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
19. IEEE. *IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology*. IEEE, 1990.

20. D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng.*, 30(6):418–421, 2004.
21. 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.
22. A. Marchetto, P. Tonella, and F. Ricca. State-based testing of Ajax web applications. In *Proc. 1st IEEE Int. Conference on Sw. Testing Verification and Validation (ICST'08)*, pages 121–130. IEEE Computer Society, 2008.
23. P. McKenzie. Why I'm done making desktop applications. Blog post on `http://www.kalzumeus.com/2009/09/05/desktop-aps-versus-web-apps/`. Date consulted: 15 September 2009.
24. C. L. Merrill. Using Ajax to improve the bandwidth performance of web applications, 2006. `http://www.webperformanceinc.com/library/reports/AjaxBandwidth/`.
25. A. Mesbah, E. Bozdag, and A. van Deursen. Crawling Ajax by inferring user interface state changes. In *Proceedings of the 8th International Conference on Web Engineering (ICWE'08)*, pages 122–134. IEEE Computer Society, 2008.
26. A. Mesbah and A. van Deursen. A component- and push-based architectural style for Ajax applications. *Journal of Systems and Software*, 81(12):2194–2209, 2008.
27. A. Mesbah and A. van Deursen. Invariant-based automatic testing of Ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09), Research Papers*, pages 210–220. IEEE Computer Society, 2009.
28. L. Morell. Theoretical insights into fault-based testing. In *Proc. 2nd Workshop on Software Testing, Verification, and Analysis*, pages 45–62, 1988.
29. J. Offutt. Quality attributes of web software applications. *IEEE Softw.*, 19(2):25–32, 2002.
30. T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Commun. ACM*, 31(6):676–686, 1988.
31. D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.
32. M. Pezzè and M. Young. *Software Testing and Analysis*. Wiley, 2008.
33. 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.
34. D. Richardson and M. Thompson. The RELAY model of error detection and its application. In *Proc. 2nd Workshop on Software Testing, Verification, and Analysis*, pages 223–230, 1988.
35. C. W. Smullen III and S. A. Smullen. An experimental study of ajax application performance. *Journal of Software*, 3(3):30–37, March 2008.
36. S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *ASE'05: Proc. 20th IEEE/ACM Int. Conf. on Automated Sw. Eng.*, pages 253–262. ACM, 2005.
37. A. Tarhini, Z. Ismail, and N. Mansour. Regression testing web applications. In *International Conference on Advanced Computer Theory and Engineering*, pages 902–906. IEEE Computer Society, 2008.
38. A. White. Measuring the benefits of Ajax, 2006. `http://www.developer.com/java/other/article.php/3554271`.
39. J. Willemsen. Improving user workflows with single-page user interfaces, November 2006. `http://www.uxmatters.com/MT/archives/000149.php`.
40. R. K. Yin. *Case Study Research: Design and Methods*. SAGE Publications Inc, 3d edition, 2003.

TUD-SERG-2009-032
ISSN 1872-5392