2012 19th Working Conference on Reverse Engineering

Automated Acceptance Testing of JavaScript Web Applications

Natalia Negara, Eleni Stroulia
Department of Computing Science
University of Alberta
Edmonton, Canada
{*negara,stroulia*}*@ualberta.ca*

***Abstract*—Acceptance testing is an important part of software development and it is performed to ensure that a system delivers its required functionalities. Today, most modern interactive web applications are designed using Web 2.0 technologies, many among them relying on JavaScript. JavaScript enables the development of client-side functionality through the dynamic modification of the web-page's content and structure without calls to the server. This implies that server-side testing frameworks will necessarily fail to test the complete application behaviors.**

In this paper we present a method for automated acceptance testing of JavaScript web applications to ensure that required functionalities have been implemented. Using an intuitive, human-readable scripting language our method allows users to describe user stories in high level declarative test scripts and to then execute these test scripts on a web application using an automated website crawler. We also describe a case study that evaluates our approach in terms of capabilities to translate user stories in automated acceptance test scripts.

***Keywords*-Acceptance testing, JavaScript, Automated Testing, Crawling, Ajax, Web testing, User Story**

I. Introduction

Acceptance testing is an essential activity in the software-development lifecycle. Its objective is to validate that the software meets the client requirements. To that end, acceptance tests are driven by, and correspond to, user stories. Automated acceptance tests can also serve as a part of a regression test suite, to verify that the client requirements are not violated by evolution of the software.

The rapid evolution and increasing popularity of Web 2.0 technologies have recently boosted the development of complex dynamic web applications [1], where considerable amount of application logic is moved to the browser via such web technologies as JavaScript. JavaScript is a dynamic programming language that is widely used for creating modern web applications and is responsible for client-side computations and dynamic updates of the web page. These characteristics (dynamic DOM content and structure of web pages) make JavaScript-enabled applications harder to test.

Software testing, an admittedly difficult task, is even more challenging in the context of modern web applications, due to their characteristics [2]. A number of approaches [3], [4] have been suggested for automated testing of dynamic web applications by "exploring" their behaviors. These methods are not suitable for the acceptance-testing task, as they automatically generate test cases based on the exhaustive analysis of dynamic DOM states of a web application, without being aware of its intended behavior. Other frameworks and methods have been proposed that can be adopted for acceptance testing, however they either require substantial programming expertise [5] or do not have complete testing solutions for Ajax-based (Asynchronous JavaScript and XML) web applications [6], [7].

To address the challenge of acceptance testing of dynamic web applications, one has to address two related problems. First, one has to be able to exercise and record the behavior of JavaScript-enabled web applications. Crawljax is such an open-source tool [8]. It was designed to exercise Ajax-based web applications at the client-side, i.e., through the browser, and, in the process, it constructs a flow-graph of the application's behavioral states, corresponding to changes in the DOM (Document Object Model) tree underlying the web page. Previously, Crawljax has been successfully applied to perform invariant-based automatic testing [9], security testing of web widgets interactions [10], regression testing [11] and cross-browser compatibility testing [12] of Ajax applications. The second problem that needs to be addressed involves the specification of the intended behavior of the web application and the comparison of a mechanism to compare it against its actual behavior; this is the problem we address with this work, relying on Crawljax.

The rest of the paper is organized as follows: Section I provides an overview of the related work. The proposed approach along with the challenges encountered during the implementation of *CrawlScripter* are described in Section II. Section III-D presents a case study. In Section IV we conclude and present ideas for future work.

II. Related Work

A number of methods have been proposed for web-application testing. To test Ajax web applications Marchetto *et al.* has recently suggested a method to dynamically extract DOM states into a finite state machine and to analyze semantically interacting events sequences in order to automatically generate test cases [3]. To overcome the problem of exponentially growing search space of semantically interacting event sequences, the authors later suggested to use search-based algorithms to extract maximally diverse event

1095-1350/91 $25.00 © 4891 IEEE
DOI 10.1109/WCRE.2012.41

sequences of different length [4]. This technique reduces the size of generated test cases, since event sequences are not repeated. In contrast to our approach, the above methods focus on finding faults in a web application, instead of establishing that it delivers the desired functionalities.

CoScripter was originally developed for automating browsing interesting paths through a web site [6], recording a user's actions demonstrating these paths was introduced. CoScripter records user actions performed in the web browser, such as clicking on links and buttons or providing input for fields. The recorded actions are described in a human readable and editable textual script that can be shared with other users and later replayed. CoScripter consists of two main components: a Firefox[1] browser plug-in that records and replays the user actions, and a repository where the users can share, edit, and rate scripts. Recording of actions in CoScripter is done by adding event listeners to the events generated in response to the user's interaction with the HTML DOM elements. When a new action is detected, a step is added to the script by filling a template that corresponds to the type of the action [13]. To parse scripts, CoScripter uses an LR(1) parser, which converts the textual representation of a script into web command objects. CoScripter uses heuristics to guess labels for a large number of input elements in the DOM structure. For example, to detect a label of a textbox, the algorithm might look for the text to the left of the textbox, so the label could be found in the DOM structure between the textbox' parent element and the textbox itself. As opposed to our system, CoScripter does not support actions that rely on DHTML or Ajax updates which make it not applicable for writing automation tests for Ajax-enabled web applications. Also, being a Firefox plug-in, CoScripter is limited to one browser only.

Mahmud and Lau [7] applied CoScripter for web testing and developed CoTester, a system for automated web-application testing. CoTester extends CoScripter by implementing assertions and a machine learning algorithm for subroutine identification in the test scripts. A subroutine in CoTester is a group of test steps that represent a higher level action like "Log in to the system". Subroutines are intended to help test maintenance, and also to better reflect the structure of a test script. However, CoTester which is built upon CoScripter, is available only for the Firefox browser, and does not support handling of Ajax requests.

One of the most commonly used tools designed for acceptance testing is Fit (Framework for Integrated Test) [5], which is based on JUnit[2] testing framework. Acceptance tests using Fit are jointly created by clients and developers: clients write test cases in the form of HTML tables and for each HTML table programmers write individual Java classes called "fixture" which must conform to certain Fit conventions. A fixture class reads the content of an HTML table and automatically generates acceptance tests, which are then executed by the system on a tested program. In contrast to our approach, Fit requires expertise of developers to create automation tests, since it is necessary to write fixture classes for every type of HTML table.

Other tools for web-application acceptance testing include Cucumber[3], a behavior-driven development framework, and the Robot Framework[4], which uses a keyword-driven testing approach, with a test case represented as a plain text or HTML file. Both frameworks do not provide complete testing solutions to handle Ajax-requests in web applications.

Additionally, in contrast to the aforementioned frameworks, *CrawlScripter* can be accessed through any web browser and does not require any software installation or specific configuration on the client's machine.

III. *CrawlScripter*

In this section, we discuss the architecture of the *CrawlScripter* tool, which enables developers to specify automated acceptance tests for Javascript-enable web applications and executes them using Crawljax.

A. Crawljax

Crawljax [8], developed by Mesbah *et al.* for automatically crawling and testing Ajax-based web applications, can crawl any Ajax web application by identifying the "clickable" elements on a web page rendered at the client's browser, firing events appropriate for these clickable elements, and filling the input widgets with data. These capabilities make Crawljax a powerful testing tool.

The main components of Crawljax are the:

- "Embedded Browser", which executes JavaScript and supports the technologies required by Ajax;
- "Robot", which is responsible for filling input data on the embedded browser;
- "Controller", which detects state changes and updates the State Machine; and
- "Finite State Machine", which is responsible for maintaining the state-flow graph and a pointer to the current state.

Crawljax creates a *state-flow* graph that models the possible navigational paths of a web application as a sequence of states (corresponding to the DOM states of the application user interface) and transitions between them (corresponding to the clickable elements whose behavior links one state to the next). The state-flow graph is built incrementally. It is initialized to contain only a single state, the DOM of the first application home page. Crawljax starts to crawl over the application and new DOM states are produced and added to the graph as a result of DOM-tree changes. These changes

[1] www.mozilla.org
[2] http://www.junit.org/
[3] http://cukes.info/
[4] http://code.google.com/p/robotframework/

are caused either by server-side changes sent to the client, or by client-side events. Crawljax defines a set of *candidate clickable elements* which fire events of different types, e.g `click, mouseOver, submit`. After an event is fired on a candidate clickable element, Crawljax compares the DOM tree as it was before the event against the DOM tree resulting after the event. The comparison between two DOM trees is done by computing the Levenstein [14] distance between them. A new DOM state is created and added to the state-flow graph if there are changes detected in a resulting DOM tree. To identify an already discovered DOM state and to eliminate duplicate states, the tool computes hashcodes for each DOM tree and compares every new state with any other state in the state-flow graph. Also, a new edge annotated with the event, whose firing led to the new DOM state, is added to the graph. Finally, the current pointer of a state machine is set to the newly added DOM state. The process is repeated until no new states can be found.

B. The CrawlScripter Scripting Language

The scripting language of *CrawlScripter* consists of a library of high level instructions (input by the user) that can be used to write test scripts that would be executed by the crawler on the web site to be tested. These instructions are close to natural language, therefore users of *CrawlScripter* are not required to have programming knowledge to write automated acceptance tests. The grammar of *CrawlScripter*'s scripting language is shown in figure 1.

$< Instruction >$
$\rightarrow < Event > | < Input\_spec > | < Assertion >$
$< Event > \rightarrow Click < Element >< Label >$
$< Input\_spec > \rightarrow < Input\_type >< Label >< Value >$
$< Assertion >$
$\rightarrow < Assert\_type >< Label >< Failure\_message >$
$< Tag > \rightarrow < Tag\_name >< Attribute >< Value >$
$< Element > \rightarrow link \mid button \mid checkbox \mid < Tag >$
$< Input\_type > \rightarrow enter\ text \mid select\ from\ drop-down$
$< Assert\_Type > \rightarrow\ present \mid not\ present$
$< Label > \rightarrow$ "$String$"
$< Tag\_name > \rightarrow$ "$String$"
$< Tag > \rightarrow$ "$String$"
$< Attribute > \rightarrow$ "$String$"
$< Value > \rightarrow$ "$String$"
$< Failure\_message > \rightarrow$ "$String$"

Figure 1. The *CrawlScripter* Language

CrawlScripter supports three types of instructions: events, input specification and assertions. Most of the actions performed by users in the web applications can be represented as web commands that consist of one verb that corresponds to the action performed, and one or two nouns (arguments upon which the action will be performed) [15]. Each instruction in *CrawlScripter* begins with the verb phrase that expresses the action to be performed, followed by the nouns or, in case of an assertion's failure message, a phrase. Event instructions represent Crawljax's *click* event types on the elements of the web page and as an argument require to specify element's label to be clicked on (for example, `Click link "Search"`). Input-specification instructions require to provide as arguments the name (as an attribute in DOM structure) of an input field element and a value to be filled in the field (for example, `Enter text "searchField", "dog"`. Assertions are used to verify the presence or absence of a DOM element on the web page. Assertion instructions take as arguments the element's identifier (or name) to be checked for presence (or absence), which is followed by the failure message (for example, `Assert is present "searchField", "Search field is not present"`). In case of the assertion failure the provided message is displayed to the user.

CrawlScripter parses the test scripts and extracts from each instruction the type of the action to be performed on the web page and its arguments. This information is then compiled into JavaScript functions.

C. Overall Architecture

CrawlScripter consists of three different modules: the *"Testing Engine"*, a *"Test Repository"*, and a *"Web Client"*. The architecture of *CrawlScripter* is shown with the component diagram of figure 2.

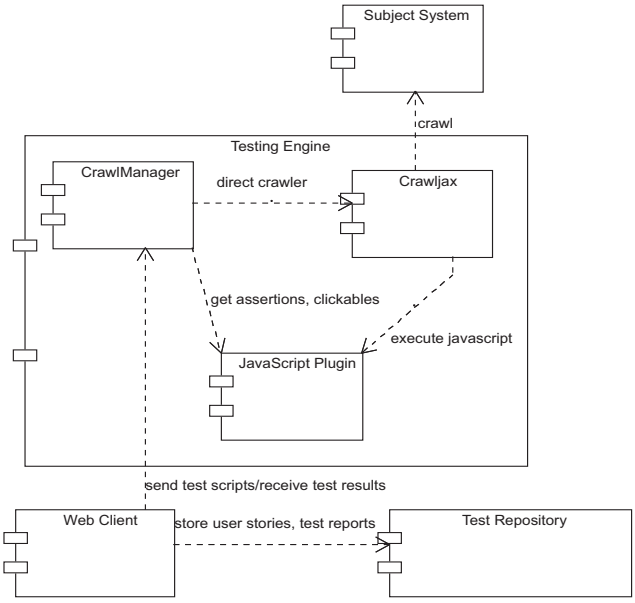

Figure 2. Architecture of *CrawlScripter*

The *Web Client* component is implemented using the JSF (Java Server Faces) framework and Ajax. In *CrawlScripter*, the client is responsible for communication with users through the user interface, for database data query and retrieval, and for interaction with the *Testing Engine*. It provides a user interface to create new acceptance tests or to edit existing ones for a selected application, to submit requests to run test scripts, and to review test reports.

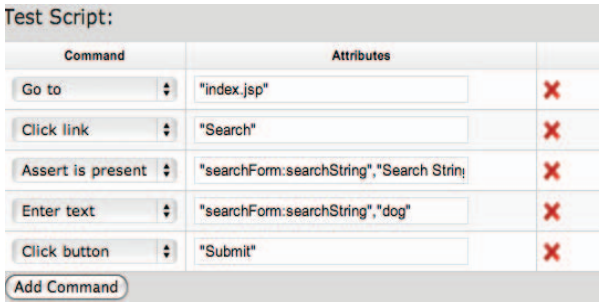

Figure 3. Creating test scripts with *CrawlScripter*

The user can add steps for the test script by clicking on the "Add Command" button, which adds a new empty line. The user can select instructions from the drop down list and specify the arguments of the instruction in the text box next to the drop down list. Figure 3 demonstrates an user interface for test script creation.

The *Test Repository* is implemented in MySQL and is used for storage and retrieval of test scripts, test reports and configuration information. The interaction of client with the repository for data query and retrieval is done using the Hibernate[5] library, which also provides the mapping of Java classes to database tables, and Java types to SQL types.

The *Testing Engine* is responsible for running test scripts on the pages of the tested JavaScript-enabled web application. The basic components of *Testing Engine* are the *CrawlManager*, the *JavaScript Plugin*, and *Crawljax*. The *Testing Engine* automatically iterates over each instruction in the test script and performs the associated action in the web application being tested.

To execute an acceptance-testing procedure, the user submits a request to run a (number of) selected test script(s). The *Web Client* retrieves from the *Test Repository* the test scripts written by the user and sends them to the *Testing Engine* via a REST API. In addition to the test scripts, the API also specifies a callback URL and the URL of the web application to be tested. On the *Testing Engine*, the test scripts are parsed into a Test Suite object. For each test script in the Test Suite, the *CrawlManager* starts a *Crawljax* crawler with a *JavaScript Plugin*. During the test execution, when the DOM structure of the web page is changed or the browser moves to a new page, `onNewState` or `onUrlLoad` method of the *JavaScript Plugin* is called respectively, and the javascript code of the current test step is executed by Selenium webdriver[6] (which is used by Crawljax) as a body of an anonymous function. Clickable elements for the current test step and any assertions that were evaluated are returned to the *CrawlManager*, which then directs *Crawljax* 'actions' by specifying *Crawljax*'s candidate clickable elements for the current web page. If *Crawljax* doesn't find a candidate clickable element, a "Failed" status is returned to *CrawlManager* for the current test step. For every new DOM state (which implies that the candidate clickable element was found and the event was fired) a screen shot is captured. The test script is executed to its completion, or until an element specified in the instruction is not found and *Crawljax* cannot proceed. The *CrawlManager* sends the results of the test scripts' execution along with the screen shots back to the *CrawlScripter* client using the provided callback URL. The client saves the test execution results in the *Test Repository*.

The test results can be reviewed by the user under the "Reports" screen in the *Web Client*. Test reports are designed to provide the user with information about the results of executed test cases which can be useful for making further decisions. Each test report covers the status (passed or failed) of every instruction in a test script. Also, in a test report each successfully executed instruction is recorded with a screenshot of the current state of a web page. Screenshots are stored due to the additional graphical information they contain about the specific state of the tested web application when the result was obtained.

D. Implementation Challenges

During the implementation of *CrawlScripter* we ran into a number of technical challenges. At first, the use of Crawljax in our approach seemed to be straightforward. However, since Crawljax crawls the application by clicking on the clickables in an ad-hoc order, the hardest part in the implementation consisted in guiding Crawljax actions via instructions specified in the test scripts (click on the specific element on the current web page). In our approach test instructions are converted into JavaScript functions, which allow to execute assertions, to push elements to be clicked into the list of candidate clickable elements, and to set up values in the input fields on the page. The solution to the problem of guiding Crawljax was to develop the *JavaScript* plugin that would be able to execute JavaScript part of the test script whenever the DOM state of the page is changed or a new URL is loaded, and the *CrawlManager* that will be able to direct Crawljax on each web page by specifying clickable elements for that web page.

IV. Evaluation

To date, we have conducted two small studies to assess the applicability of our method in supporting acceptance testing and translation of functional requirements of dynamic web applications into executable test scripts. We evaluated the ease of use of *CrawlScripter* and also tried to find if *CrawlScripter* can be applied in a variety of test scenarios. From these studies we also defined opportunities for future extensions of *CrawlScripter*.

The subject of the first study is Java Pet Store (JPetStore)[7] sample Ajax-enabled application. The study consisted of

[5]http://www.hibernate.org/
[6]http://seleniumhq.org/
[7]http://java.sun.com/developer/technicalArticles/J2EE/petstore/

translating user stories for JPetStore into executable automated acceptance tests and running them on the target web application.

JPetStore is a well-known educational application designed to demonstrate the use of Java Enterprise Edition Platform in developing Ajaxified web applications. JPetStore is an Internet pet store where users can perform different actions typical to most e-stores. Users can browse the catalog of products or search for a specific item in the catalog, add a new item for sale, search for items using their location or tags, rate items or flag an item as an appropriate. The JPetStore application consists of a single page where different areas of the page are dynamically updated based on user actions using Ajax technology.

We constructed the following user stories (and corresponding tests) for JPetStore based on the functionalities described in the "About" section, which is shipped together with the JPetStore application: Browsing the Catalog, Searching the Catalog (by keyword), Rate an item, Flag as an appropriate, Add new Item for sale, Reviewing an Order, Search item by location, and Search item by tag.

The above 8 user stories were implemented as *CrawlScripter* acceptance test scripts, and they were successfully executed over the JPetStore web application.

The second case study was motivated by our need to collect citation information from a publisher's web site[8]. To that end we developed an acceptance test for verifying that each publication-title URL was associated with a citation entry. The assertion verifying successful testing, involved the clicking of a "download" button, which resulted in downloading the citation entry. This task, conceived by another member of our group not related with this project, was fairly straightforward to accomplish with *CrawlScripter*.

V. Conclusion

In this paper we proposed a tool for automated acceptance testing of dynamic, JavaScript-enabled web applications. Our tool relies on Crawljax [8], an open source crawler designed to exercise modern web applications that rely on Web 2.0 technologies, such as Ajax or JavaScript. Our tool, *CrawlScripter*, enables developers to create fairly easy-to-understand automated acceptance tests, corresponding to the application user stories, and to execute them by exercising the underlying application. To evaluate our tool, we implemented tests corresponding to the user stories of a pedagogical web application and successfully executed them. In addition, we designed a test script for a single functionality of a real-world web application.

In the future, we plan to conduct more case studies to evaluate usefulness, applicability, and scalability of the proposed approach. *CrawlScripter* can be extended by introducing cross-references to other tests in order to reuse already implemented user stories (e.g. Login functionality) and to make the test scripts shorter. Another future extension of *CrawlScripter* could be the support of test scripts generalization and automation of a series of repetitive actions. Also, we plan to integrate *CrawlScripter* with the browser, so the user can add elements to the script automatically by just clicking on them on the web page to be tested.

[8]http://www.springerlink.com/

Acknowledgment

We would like to thank Kristofer Mitchell for helping in implementing the CrawlScripter. This work was supported by NSERC, AITF and IBM.

References

[1] M. Jazayeri, "Some trends in web application development," in *2007 Future of Software Engineering*, ser. FOSE '07, Washington, DC, USA, 2007, pp. 199–213.

[2] G. A. Di Lucca and A. R. Fasolino, "Testing web-based applications: The state of the art and future trends," *Inf. Softw. Technol.*, vol. 48, no. 12, pp. 1172–1186, Dec. 2006.

[3] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of ajax web applications," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ser. ICST '08, Washington, DC, USA, 2008, pp. 121–130.

[4] A. Marchetto and P. Tonella, "Using search-based algorithms for ajax event sequence generation during testing," *Empirical Softw. Engg.*, vol. 16, no. 1, pp. 103–140, 2011.

[5] R. Mugridge and W. Cunningham, *Fit for Developing Software: Framework for Integrated Tests (Robert C. Martin)*. Prentice Hall PTR, 2005.

[6] G. Leshed, E. M. Haber, T. Matthews, and T. Lau, "Coscripter: automating & sharing how-to knowledge in the enterprise," in *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, 2008, pp. 1719–1728.

[7] J. Mahmud and T. Lau, "Lowering the barriers to website testing with cotester," in *Proceedings of the 15th international conference on Intelligent user interfaces*, New York, NY, USA, 2010, pp. 169–178.

[8] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling ajax-based web applications through dynamic analysis of user interface state changes," *ACM Trans. Web*, vol. 6, no. 1, pp. 3:1–3:30, 2012.

[9] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of ajax user interfaces," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09, Washington, DC, USA, 2009, pp. 210–220.

[10] C.-P. Bezemer, A. Mesbah, and A. van Deursen, "Automated security testing of web widget interactions," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2009, pp. 81–90.

[11] D. Roest, A. Mesbah, and A. v. Deursen, "Regression testing ajax applications: Coping with dynamism," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ser. ICST '10, Washington, DC, USA, 2010, pp. 127–136.

[12] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, New York, NY, USA, pp. 561–570.

[13] A. Cypher, C. Drews, E. Haber, E. Kandogan, J. Lin, T. Lau, G. Leshed, T. Matthews, and E. Wilcox, *No Code required: giving users tools to transform the web*. Burlington, MA: Morgan Kaufmann, 2010, ch. Collaborative Scripting for the Web., pp. 85–104.

[14] V. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, 1966.

[15] G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan, "Koala: capture, share, automate, personalize business processes on the web," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, 2007, pp. 943–946.