Recording and Replaying Navigations on AJAX Web Sites

Alberto Bartoli, Eric Medvet, and Marco Mauri

DI[3] - University of Trieste
Via Valerio 10, Trieste, Italy
bartoli.alberto@units.it

Abstract. Recording and replaying user navigations greatly simplifies the testing process of web applications and, consequently, greatly contributes to improving usability, robustness and assurance of these applications. Implementing such replaying functionalities with modern web technologies such as AJAX is very hard: the GUI may change dynamically as a result of a myriad of different events beyond the control of the replaying machinery and even locating a given GUI element across different executions may be impossible.

In this work we propose a tool that overcomes these problems and is able to handle real-world web sites based on AJAX technology. Recording occurs automatically, i.e., the user navigates with a normal browser and need not take any specific action. Replaying a previously recorded trace occurs programmatically, based on several heuristics that make the tool robust with respect to DOM variance while at the same time maintaining the ability to detect whether replaying has become impossible—perhaps because the target web site has changed too much since the recording. The entire procedure is fully transparent to the target web site. We also describe the use of our tool on several web applications including Facebook, Amazon and others.

1 Introduction

The ability to record and replay GUI navigation sequences has become an essential component of testing procedures for modern software [11]. The need for incorporating similar procedures in web applications is becoming more and more urgent, given the richness of their user interfaces and their ever more stringent requirements in terms of usability, robustness and assurance [9]. Unfortunately, modern web technologies such as AJAX make programmatic interaction with client-side GUIs very hard, due to the stateful and highly dynamic nature of the DOM that determines the actual GUI appearance. Client-side code constructs the DOM and manipulates it as a result of a myriad of different events, that may be triggered by user actions but also by asynchronous interactions between the browser and the server. Indeed, even the seemingly trivial task of identifying the elements in a DOM that may affect navigation is actually very challenging and is still partly unsolved [2,6]. Repeating the same sequence of user actions against

M. Brambilla, T. Tokuda, and R. Tolksdorf (Eds.): ICWE 2012, LNCS 7387, pp. 370–377, 2012.

the same initial DOM, moreover, typically results in a different DOM: attributes of individual elements may change across different executions, for example if they include a form of session identifier, and the very same DOM structure may change because the server usually serves different contents at different times. The complexity of replaying a browser session is magnified further by the fact that, in practice, HTML elements usually do not have any form of identity that persists across browsing sessions. It follows that finding programmatically the "same" HTML element accessed in a previous session may be very challenging. Indeed, replaying a browsing session may even be impossible, for example because the web application at the target web site has changed between the registration and the attempted replay.

In this work we propose a method and a tool for recording and replaying traces of user interactions with web applications, which may be of great help in a number of testing activities as pointed out above. Key properties of our contribution are: (i) the creation of a trace is fully automatic: the user merely navigates into the target web site without taking any specific action or issuing any dedicated command; (ii) the process is fully transparent to the target web application, that need not be modified in any way; (iii) the replaying algorithm is highly robust to DOM variance, while at the same time maintaining the ability to notify the user in cases the replay has become impossible to perform—for example because the target web application has changed too much between the record and the replay. We are not aware of any similar tool with all these properties. The tool that implements our method is able to cope with real-world web applications based on AJAX technology and is able to record and replay such actions as login and file upload. These actions are key components of the workflow of many web applications but are notoriously difficult to handle programmatically and are often missing from web analysis tools such as crawlers and vulnerability scanners [3].

2 Related Work

A tool for recording and replaying traces of web application navigations is proposed in [8]. The proposed approach requires that the user explicitly marks every action to be recorded, by right clicking on the desired web element and then choosing the event to register from a menu presented by an instrumented web browser. This approach is manual and, it seems fair to say, quite cumbersome to use. Our proposal, in contrast, is entirely automatic because the user need not take any specific action during the recording process.

Recording and replaying functionalities are an essential component of the approach to web application testing proposed in [9]. Recording is done by replacing dynamically the handler of each event in the page with a handler that merely records its activation and then invokes the original handler. This approach allows recording only events for which the page author has defined handlers using the so called version 0 of the DOM. This technique of defining method handlers is long deprecated and many modern web application no longer use it.

Our approach is radically different in the sense that, essentially, it does not place any requirement on the DOM and, in particular, it does not attempt to discover the event-handler relationships defined in the page being recorded. As a technical but important detail, we also remark that the cited work cannot handle multi-page web applications, which in many cases prevent the handling of the login step. Similar comments apply to [4], except that in this case multi-page web site are supported.

Concerning the replaying process, the method for searching a target element proposed in [9] implicitly assumes that the replay will occur without any content variation. While our heuristics allow identifying the correct element, the approach in [9] would often select other elements during the replay.

The problem of determining whether two serialized DOM trees represent the same page is essential in [5,6,7,10]. The goal of the cited works is to define a distance between two DOM trees and then compare that distance against a threshold in order to detect if the two DOM trees represent the same page. In contrast, in this work we aim at locating a given element across different DOM trees.

3 System Architecture

Our system consists of two separate applications: the *trace recorder*, that records the actions executed by the user during a browsing session, and the *trace replayer*, that replays a browsing session previously recorded by the trace recorder. The system is fully *transparent* to both the user and the target web site. That is, the user navigates with a normal browser and need not take any specific action for recording, and the target web site need not be modified in any way. A fundamental characteristic of both or our tools is the use of a real browser (Firefox), which is very important to ensure compatibility with real-world web applications and to provide the user with a familiar interface. The trace replayer reads the data previously saved in a trace and pilots the browser programmatically, so as to replay the user actions on the target web site automatically.

3.1 Trace Recorder Architecture

The trace recorder consists of: (a) a web application that we developed and that we call *Observer*; (b) a *proxy*; and (c) a browser.

The Observer is composed of a server side code (Observer-S) and a client-side code executed by the browser (Observer-C). Observer-C records all the DOM events generated by the user and periodically sends a description of these events to the Observer-S, that saves the corresponding descriptions into a file—the *trace*.

The proxy, placed in between the browser and the target web site, performs two actions: (i) injects the Observer-C code into all the pages sent to the browser; and (ii) redirects part of the web traffic so as to enable communication between Observer-C and Observer-S without violating the *same origin policy* implemented by modern browsers.

The browser fetches our JavaScript code injected by the proxy—i.e., Observer-C—and executes this code locally. The results produced by Observer-C are sent to Observer-S through the URL `/GWT-Observer`. The proxy is configured so as to reroute any traffic to `/GWT-Observer` toward the server in our control that actually executes the Observer-S code. In other words, the browser is tricked into believing that Observer-C is fetched from the target web site and communicates with that site. This fairly complex structuring allows circumventing the *same origin policy (SOP)* implemented by modern browsers, which would prevent any communication from Observer-C to a server in our control [1].

Our tool is able to handle also encrypted `https` traffic by configuring the proxy to act as a man in the middle between the browser and the target web application.

A trace contains a sequence of *event descriptions*, each of them consists of: (i) type of the DOM event (e.g. click, wheel, etc.); (ii) time at which the event occurred; (iii) description of the target element of the event. The description of the target element contains its `tag name`, e.g. `span` or `div`, its `text` content, its `x` and `y` positions and the possible values of the `id`, `name`, `src`, and `type` attributes.

We also defined, in addition to those defined by the DOM standard, two *synthetic* event types, `write` and `select`, to represents respectively the typing of a text inside an input field and the selection of a choice from a drop-down list as a more compact representation of sequences of events that have actually occurred.

3.2 Trace Replayer Architecture

The goal of the trace replayer is to read the trace created by the trace recorder and reproduce the registered events using the browser. The reproduction of the trace is performed by the trace replayer driving the browser, through Webdriver[1], a browser automation framework that enables to manipulate a real browser programmatically.

We introduce a distinction between *relevant* and *irrelevant* events. A relevant event is any event whose replay is essential to properly replay the entire trace, while replaying an irrelevant one is not essential to properly replay the trace. The trace recorder registers events that could be irrelevant because whether a given event is relevant depends on the events that follow that event. For example, events generated for selecting a text field inside a form are irrelevant, because the subsequent event of typing inside that field implies the selection of the text field itself.

The trace replayer preprocesses the trace as follows: (i) filter out all the irrelevant events from the trace; (ii) shorten the trace by introducing *synthetic events* wherever possible.

After preprocessing, the trace is replayed according to the following algorithm: for each event E in the sequence S, search the associated target element T'_E in the

[1] `http://seleniumhq.org/projects/webdriver`

opened web page; if T'_E is found, replay the event using WebDriver; otherwise, repeat the search for a predefined amount of times, waiting for a fixed amount of time (half a second) after each attempt. This waiting heuristics copes with the case in which the searched element is created dynamically by JavaScript code. If this repeated search fails, trace replayer simply aborts the replay signaling the error.

Often, between the registration and the replay the content of the web page changes in more or less substantial way. Furthermore the only way to uniquely identify an element inside a document is optional (the *id* attribute) and in the vast majority of cases this attribute is not present, so a simple search for an element with identical content to the one in the trace will fail.

We attack this crucial problem with a series of heuristics and decide which one to use based on the element T_E to be found, as explained below. Each heuristic execution can lead to a false negative (the element exists in the page but the heuristic has not found it) as well as to a false positive (the found element is not the correct one, which may or may not exist in the page). In our experiments, described in the next section, we have not encountered any false positive or false negative. As future work, we plan to execute a broader quantitative analysis by systematically labeling a large dataset. Our heuristics are as follows:

findElementBySrc. This heuristic, used for searching media element, retrieves the first element T'_E that has the same tag name of T_E and the same value for the attribute `src`.

This heuristic could cause a false positive result if there are more media elements, in the analyzed web page, distinguishable between them only for the position relative to the page itself.

findElementByInput. This heuristic, used for searching form inputs, retrieves the first element T'_E that represents the same type of form input of T_E and has the same value for the attributes `id` and/or `name`. If none of these attributes are presents in T_E or if no element with those attribute values is found, then this heuristic compares the text content of the form inputs.

This heuristic should not be capable to generate false positives because the values of `id` are unique within a single page while those of `name` are unique within a single form. The heuristic can generate false negatives if the value of the attributes varies between different replays.

findElementByGrid. This heuristic is based on the position of the searched element T_E; it retrieves the first element T'_E that has the same tag name and whose coordinate $(x_{T'_E}, y_{T'_E})$ are similar to those of T_E.

This heuristic can generate false negatives if the position of the searched element varies too much between different replays. It can generate false positives if there is another element with the same tag name and similar coordinates that precedes, in document order, the searched element.

findElementByGridAndText. This heuristic is very similar to findElementByGrid, except that the retrieved element T'_E has also the same text content of T_E.

This heuristic can generate erroneous results in the same conditions of the previous one.

4 Experiments

We tested our tool to verify its ability to work on real web applications. Each experiment consisted of the registration of a trace on a web application and multiple replays of such trace to verify the repeatability of the reproduction. Table 1 is a summary of our experiments: it shows a line for each web application and the number of events in the corresponding registered trace. The table also shows the number of events, computed after the preprocessing of the trace.

Table 1. Summary of our experiments

Site name	# of pages	# of events		
		click	write	select
Amazon	10	7	2	0
Facebook	14	9	4	0
Google Groups	25	23	1	0
Stack Overflow	14	12	1	0
Wacko Picko	34	23	10	0
WIVET	44	29	13	1

Amazon. We registered a trace simulating the search of some products and then the addition of the desired product to the "shopping cart". In detail, we performed a search using the keyword "tablet", selected a specific model and added it to the cart. After that we performed another search using the keyword "stereo", selected a specific model and added it to the cart.

The most notable example of page variations was the search result pages: the products displayed changed between the replays. The trace replayer can withstand this type of variation thanks to the **findElementByGridAndText** heuristic. We replayed this trace several times without encountering any error.

Facebook. We registered a trace simulating a typical user interaction with the social network: (i) login; (ii) checking for new messages; (iii) adding a new event to the calendar; (iv) logout. We could perform the replays without any error. The only peculiar, but correct, behavior was the creation of several duplicated events on the Facebook calendar; this a further example of the resilience of our heuristics to page variations.

Another peculiarity of this web applications is the use of a session ID as the value of the `id` attribute of the login button. The trace replayer can cope with this kind of variation thanks to the fact that the **findElementByInput** searches by text if it cannot find the right element searching by its id.

Google Groups. We registered a trace containing the navigations on various discussion threads chosen at random, all pertaining to the "Google Web Toolkit" group of this web application, including the use of all the links that change the display mode of the discussions.

This was the web application that displayed the more pronounced page variations: for each replay the list of posts and topic displayed changed with

the additions of new contents. All the searches in this web application was performed by the **findElementByGridAndText** heuristic.

We could perform the replays without any error.

Stack Overflow. We registered a trace containing the search of various topic, the navigation of user information and the FAQ section of the web application. Like the previous web application all the searches in this one was performed by the **findElementByGridAndText** heuristic. We could perform the replays without any error.

WackoPicko. WackoPicko is a web application used to test web application vulnerability scanners [3]. It consists in a fake image shop applications that allow users to upload, comment and purchase images.

We created a trace containing the following actions: (i) login; (ii) addition of a comment to an existing image; (iii) search of an image; (iv) purchase of an image; (v) upload of an image; (vi) logout.

Another particular aspect of this trace is the presence of both a login and an upload file steps. Many of the work cited in Section 2 are not able to perform these two actions. We could perform the replays without any error and the majority of searches was performed by the **findElementBySrc** and **findElementByInput** heuristics.

WIVET. WIVET is a benchmarking project for analyzing web link extractors. It consists in a series of pages containing links in ways that are increasingly difficult to find for automated tools. We recorded a trace containing the activation of all of the links with the exception of those involving flash applets and those using the mouse hover event as a trigger to activate the links.

This web application is almost entirely static so the various replays have not encountered any page variation. We could perform the replays without any error.

5 Conclusions

The ability to record and replay sequences of user interactions with web applications is very useful in functional testing, security testing and usability testing. We have presented a novel approach to this problem and described a tool that implements our approach. The approach is suitable for modern web applications, made up of highly dynamic contents and abundant use of AJAX technology. The tool does not require any change or configuration on the web application to be monitored, is completely non intrusive, very easy to use and supports https connections. As discussed in the related work section, the tool overcomes several limitations of earlier proposals.

Our method and tool are useful to improve web application testing by reducing the time needed to thoroughly test the web application. We plan to execute a broader quantitative analysis of our approach on a larger dataset.

Acknowledgments. This work is partly supported by eMaze[2].

[2] `http://www.emaze.net`

References

1. Same origin policy, http://www.w3.org/Security/wiki/Same_Origin_Policy
2. Bai, X., Cambazoglu, B.B., Junqueira, F.P.: Discovering urls through user feedback. In: Proceedings of the 20th ACM International Conference on Information and Knowledge Management, CIKM 2011, pp. 77–86. ACM, New York (2011), http://doi.acm.org/10.1145/2063576.2063592
3. Doupé, A., Cova, M., Vigna, G.: Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. In: Kreibich, C., Jahnke, M. (eds.) DIMVA 2010. LNCS, vol. 6201, pp. 111–131. Springer, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-14215-4_7, 10.1007/978-3-642-14215-4_7
4. Álvarez, M., Pan, A., Raposo, J., Hidalgo, J.: Crawling Web Pages with Support for Client-Side Dynamism. In: Yu, J.X., Kitsuregawa, M., Leong, H.V. (eds.) WAIM 2006. LNCS, vol. 4016, pp. 252–262. Springer, Heidelberg (2006), http://dx.doi.org/10.1007/11775300_22, 10.1007/11775300_22
5. Medvet, E., Kirda, E., Kruegel, C.: Visual-similarity-based phishing detection. In: Proceedings of the 4th International Conference on Security and Privacy in Communication Networks, SecureComm 2008, pp. 22:1–22:6. ACM, New York (2008), http://doi.acm.org/10.1145/1460877.1460905
6. Mesbah, A., Bozdag, E., van Deursen, A.: Crawling ajax by inferring user interface state changes. In: Eighth International Conference on Web Engineering, ICWE 2008, pp. 122 –134 (July 2008)
7. Mesbah, A., van Deursen, A.: Invariant-based automatic testing of ajax user interfaces. In: Proceedings of the 31st International Conference on Software Engineering, ICSE 2009, pp. 210–220. IEEE Computer Society, Washington, DC (2009), http://dx.doi.org/10.1109/ICSE.2009.5070522
8. Montoto, P., Pan, A., Raposo, J., Bellas, F., López, J.: Automating Navigation Sequences in AJAX Websites. In: Gaedke, M., Grossniklaus, M., Díaz, O. (eds.) ICWE 2009. LNCS, vol. 5648, pp. 166–180. Springer, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-02818-2_12, 10.1007/978-3-642-02818-2_12
9. Pattabiraman, K., Zorn, B.: Dodom: Leveraging dom invariants for web 2.0 application robustness testing. In: 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE), pp. 191–200 (November 2010)
10. Roest, D., Mesbah, A., van Deursen, A.: Regression Testing Ajax Applications: Coping with Dynamism. In: 2010 Third International Conference on Software Testing, Verification and Validation (ICST), pp. 127–136. IEEE (April 2010), http://dx.doi.org/10.1109/ICST.2010.59
11. Xie, Q., Memon, A.M.: Designing and comparing automated test oracles for GUI-based software applications. ACM Trans. Softw. Eng. Methodol. 16(1), 4+ (2007), http://dx.doi.org/10.1145/1189748.1189752