Rumadai: A Plug-In to Record and Replay Client-Side Events of Web Sites with Dynamic Content

Asım Yıldız
Özyeğin University
Istanbul, Turkey
asim.yildiz@ozu.edu.tr

Barış Aktemur
Özyeğin University
Istanbul, Turkey
baris.aktemur@ozyegin.edu.tr

Hasan Sözer
Özyeğin University
Istanbul, Turkey
hasan.sozer@ozyegin.edu.tr

***Abstract*—Reproducing user events when testing web pages is challenging because of the dynamic nature of the web content and potential dependency on third party content providers. We present Rumadai, a Visual Studio plug-in, that helps web programmers test web pages by recording and replaying client-side events. Rumadai injects code into web pages to be deployed at servers. The injected code, written in JavaScript, records user events as well as client-side dynamic content requests (e.g. via Ajax or Jsonp) and their responses. Recorded events and responses are then sent to a remote database via HTTP POST. Web page developers can query the saved client data, again using Rumadai seamlessly from Visual Studio, to replay all or a subset of events on a browser.**

***Keywords*-Web testing; Plug-in; JavaScript; User event record&replay.**

I. Introduction

Modern web sites are complicated. They depend on dynamic content computed at the server-side (e.g., using PHP, ASP), at the client-side (e.g., JavaScript), and at external sources (e.g., using Jsonp). They also rely on asynchronous client-server communication. These features make testing and debugging of web applications a challenging task.

Capture-and-replay tools record and deterministically reproduce user events at the developer-site. They are adopted to preserve the integrity of web applications [1] and to support the debugging process [4,5]. Currently, these tools either run as browser plug-ins [2,3], or they employ customized browsers [4]. There are also stand-alone systems and proxies [5] used for the same purpose. However, the use of a proxy breaks the end-to-end security enforced by HTTPS [4]. Also, dynamic content obtained from external sources may not be captured by proxies.

We introduce *Rumadai*, a capture-and-replay tool that runs as a plug-in of an Integrated Development Environment (IDE), instead of a browser. Due to the complexity of web pages, programmers at large use IDEs to develop web projects. Furthermore, many enterprise web projects are being developed in teams since the scale of these projects has increased substantially over the years. With Rumadai, the developer does not have to leave the development environment, either to prepare the web page for recording events or to replay the captured events. In addition, logging of captured events is facilitated by a web service. This way, multiple developers can register for a single project and the pool of events can be made accessible to a team of developers/testers. The logging service is also accessed and controlled within the development environment. Hence, testing, event capturing, recording, replaying and debugging tasks can all be performed through a uniform interface. A distinguishing feature of Rumadai is that it provides replayability even when a web document obtains data from external sources, e.g. using cross-domain communication.

We implemented Rumadai as a Visual Studio plug-in. There is no fundamental reason why we could not implement our tool as a plug-in of another IDE, such as Eclipse. Visual Studio has been our preference because of our previous experiences in using it. Integrating Rumadai into the IDE has been a smooth process. It took us less than one man-day to complete the integration (i.e., reading the tutorials, adding the necessary menus, making hook calls, etc.) – we have significant experience in C#, but had no prior experience in writing a Visual Studio plug-in. Rumadai is available at **http://srl.ozyegin.edu.tr/projects/rumadai**.

II. System Overview

Rumadai has the following workflow:

1) A developer registers to the Rumadai logging service using the domain name of the web project and a username & password. Recorded events can be obtained later using the same registration information. Multiple users can register for the same web project; a user may register to multiple projects.
2) The developer injects the event-capturing script into her web page documents using the Rumadai menu.
3) Optionally, the developer may have unique identifiers generated for all elements on the current web page documents that do not already have id's. Having id's associated with the document elements makes it more efficient to find a component during replay.
4) The developer deploys the web pages to the server.
5) A client requests the web documents from the server. Code injected at step 2 runs inside the client browser. Events are captured.

978-1-4673-1820-4/12/$31.00 © 2012 IEEE

6) Captured events are sent to the Rumadai logging service. They are saved in a database.
7) The developer browses the events using Rumadai, without having to leave the development environment.
8) The developer selects a subset of the events, replays them to visualize what happened at the client-side.

Rumadai has four major components: (*i*) plug-in, (*ii*) recorder, (*iii*) logging service, (*iv*) replayer. The *plug-in* is used for the steps 1, 2, 3, and 7 above. The *recorder* code injected by the plug-in is written in JavaScript. With the help of JavaScript's high flexibility and reflection features, it is possible to capture and log certain events. The script captures mouse events (click and mouseover), DOM element change events, keyboard events, Ajax requests and responses, JavaScript runtime errors, and finally cross-domain communication made using Jsonp (through JQuery's `getJSON` function). An Ajax response may be successful or it may fail. In the case of success, the contents of the returned result are recorded. The same applies to Jsonp return values. Rumadai also checks for missing images and broken links by sending HTTP GET requests to these links. The recorder does not require any modification on the client browser. We make extensive use of JQuery [6] and XMLHttpRequest [7] libraries to achieve a high level of browser compatibility.

Recorder script stores the distinguishing properties of events using in-memory string representations. Accumulated events are sent to the *logging service* periodically. In case the event is an image load, the logging service downloads the image from its source, which may be a third-party server, and saves the byte data. During replay, the saved image is shown instead of the linked one. This reduces the risk of an ill-replay because the image that resides at the external source might have changed after the recording.

Rumadai's *logging service* is a database available as a web service. We made this design decision because web sites of non-trivial sizes are developed in teams. There are typically several testers who need access to the client-side logs and several clients who submit logs. Therefore, it makes sense to have a server accessible by both the testers and clients. Recorder script talks to the logging service using HTTP POST. This is because event data may be larger than the size allowed in a GET request. However, browsers do not allow cross-domain POST requests. To remedy this problem, we embed a small *Silverlight* object together with the recorder script. Silverlight is allowed to perform cross-domain POST. An alternative approach is to divide the data into chunks small enough to fit in GET requests at the expense of increased number of requests.

Recorded events can be browsed using Rumadai's event viewer. When the developer hits the play button, the plug-in first generates a sequence of JavaScript method calls, each corresponding to an event. The generated code is then combined with helper functions to obtain a *replayer* script, which is injected into the stored web page. When the page makes an Ajax/Jsonp request, the replayer aborts the request. When a response is received, the data are replaced with what was saved during recording. This ensures true reproduction of the recording-time behavior. Replayer uses the saved identifier and DOM tree path to access DOM elements.

We use the `WebBrowser` class in the .NET library for replay. Because Rumadai plug-in is implemented in C# –a .NET language– we achieve seamless integration.

III. Limitations

Rumadai does not capture all the mouse movements; only the *mouse over* events are recorded. This is a trade-off we made considering the cost (i.e. performance overhead) of capturing/replaying all the mouse movements. For a better replay, Rumadai calibrates the mouse cursor location according to the client-side and replay-side screen resolution differences. This calibration is subject to errors in some exceptional cases. For a detailed discussion of replay-time interaction limitations, see [5].

For cross-domain communication, we only capture Jsonp calls made using *JQuery*'s `getJSON` function. There are other ways of performing cross-domain data transfer, in particular using the `<script>` tags, which Rumadai does not check. However, `getJSON` is the most popular method; we believe capturing it covers a wide range of uses.

In some cases, Rumadai may miss or artificially introduce browser-specific problems during replay, because the `WebBrowser` class we use for replay may be different than the client browser. This is a trade-off we made in favor of IDE integration. Also, replaying the mouse movements would not be possible if we used an unmodified, independent browser; browsers do not allow JavaScript to control the mouse for security reasons.

References

[1] K. Vikram, A. Prateek, and B. Livshits, "Ripley: automatically securing web 2.0 applications through replicated execution," in *Proceedings of CCS '09*, 2009, pp. 173–186.

[2] "Selenium IDE," http://seleniumhq.org, 2012.

[3] "WET," http://www.wet.qantom.org, 2012.

[4] S. Andrica and G. Candea, "WaRR: A tool for high-fidelity web application record and replay," in *Proceedings of DSN '11*, 2011, pp. 403–410.

[5] J. Mickens, J. Elson, and J. Howell, "Mugshot: deterministic capture and replay for javascript applications," in *Proceedings of NSDI'10*, 2010, pp. 159–174.

[6] "JQuery JavaScript Library," http://jquery.com/, 2012.

[7] World Wide Web Consortium, "XMLHttpRequest specification," http://www.w3.org/TR/XMLHttpRequest/, 2012.