WebMate: A Tool for Testing Web 2.0 Applications

Valentin Dallmeier · Martin Burger · Tobias Orth · Andreas Zeller
Saarland University – Computer Science
Saarbrücken, Germany
{dallmeier, mburger, orth, zeller}@st.cs.uni-saarland.de

ABSTRACT

Quality assurance of Web applications is a challenge, due to the large number and variance of involved components. In particular, rich Web 2.0 applications based on JavaScript pose new challenges for testing, as a simple crawling through links covers only a small part of the functionality. The WEBMATE approach automatically explores and navigates through arbitrary Web 2.0 applications. WEBMATE addresses challenges such as interactive elements, state abstraction, and non-determinism in large applications; we demonstrate its usage for regular application testing as well as for cross-browser testing.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Algorithms

Keywords

JavaScript, web applications, test generation

1. INTRODUCTION

In the past years, major browser producers such as Google and Mozilla have considerably improved the performance of their JavaScript engines. As a consequence, it is now possible to build rich interactive applications based on the interplay of HTML, CSS and JavaScript. The appeal of these so-called *web applications* is their ease of use: no installation is required and all data is stored centrally on a server. As a result, web applications today are abundant and JavaScript has become the core language for the dynamic web. Being a highly dynamic language, JavaScript is notoriously hard to debug and cross-browser inconsistencies make it difficult to ensure the correctness of large web applications across different browsers.

Due to the large number and variety of involved components, the only feasible approach for quality assurance of web applications is *testing*. In particular, for a project with more than one release cycle, we need *automated tests* that can be rerun after each change. Existing tools such as Selenium [5] allow to write such tests by remote-controlling a browser and injecting JavaScript code that tests the outcome of user interactions. The main effort when writing system-level tests is usually spent on navigating to the desired state and establishing the test setup. Even for simple tests in small applications, the setup part can consist of several steps: open the start page, enter login data, open the menu, and then navigate to a submenu. What is worse is that if only one of the elements on this path is changed, the test is broken. Due to the lack of proper refactoring tools, small changes in the user interface often break large parts of the test suite which causes considerable maintenance costs. As a result, *most companies only use manual testing or do not test at all.*

In this paper, we address this problem with WEBMATE—an approach to *automatically explore and navigate through arbitrary Web 2.0 applications.* WEBMATE analyzes the web application under test, identifies all functionally different states, and is then able to navigate to each of these states at the user's request.

Figure 1 summarizes how WEBMATE works. The sole mandatory input to WEBMATE is the URL of the web application to test. (If the application requires special inputs such as login data, then the user has to provide this data to WEBMATE.) Using the URL as a starting point, WEBMATE explores the web application and learns a *usage model* that captures how a user can interact with the web application. In this step, WEBMATE examines all buttons, links, forms or any other element with a JavaScript event handler that can be triggered by a user interaction. The resulting usage model represents these interactions as a graph where nodes correspond to different states of the application, and edges represent user interactions. Users of WEBMATE can leverage the usage model to systematically generate executions of the application and to perform analyses in all states recorded in the model.

Systematic exploration of a dynamic web applications is a challenging task. In Section 2, we discuss the most important challenges and the solutions as implemented in WEBMATE. In Section 3, we present a small case study that investigates the effectiveness of WEBMATE with two subjects. Section 4 shows how WEBMATE helps to automate testing for cross-browser incompatibilities. We briefly discuss related work in Section 6 and conclude with ideas on the future of WEBMATE in Section 7.

2. ANALYZING WEB 2.0 APPLICATIONS

In essence, WEBMATE is a crawler for Web 2.0 applications with the additional feature of being able to replay interactions to reach different states. At the heart of WEBMATE lies an engine that is able to remote control a browser, trigger interactions and inspect the cur-

JSTools'12, June 13, 2012, Beijing, China

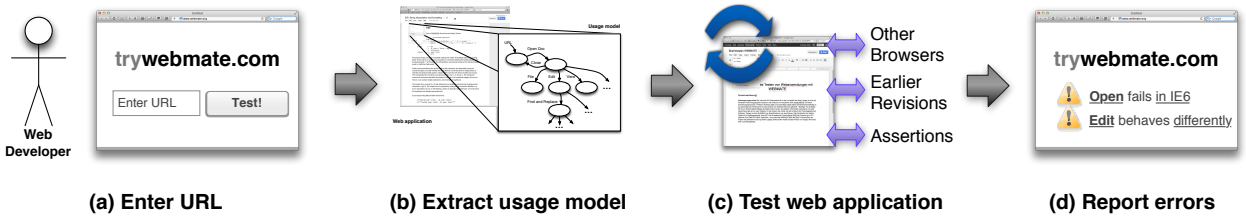

Figure 1: How WEBMATE works. Given a URL as input (a), WEBMATE analyzes the web application and learns a *usage model* of how a user can interact with the web application (b). Users of WEBMATE can then use this model to systematically explore the functionality of the web application and run different analyses such as cross-browser compatibility checks, regression tests or code analyses (c). Results of the analysis can then be reported back to the user (d).

rent state of the document object model (DOM). When *analyzing* a web application, WEBMATE repeatedly triggers actions on the web page and extracts the state of the DOM after each action. From this data, WEBMATE generates the application's usage model, a graph where nodes correspond to states of the application, and a transition represents a single interaction with the application. Each state comprises information about all user interface elements and WEBMATE systematically triggers interactions until all elements are explored and the analysis stops. The resulting usage model contains all distinct states identified by WEBMATE and all the possible ways to navigate between them.

Web crawlers have been around for many years. However, they only work well for traditional URL based web applications. Effectively crawling Web 2.0 applications poses a number of challenges, both conceptually and from a technical point of view. In this section, we briefly summarize the most important challenges and sketch the solutions implemented in WEBMATE.

Interactive elements. To interact with the application, WEBMATE needs to recognize all elements on a web page the user can interact with. This includes all links, buttons and input elements. In addition, the tool also has to consider all elements that have a JavaScript event handler configured. Such event handlers can either be added statically in the HTML document, or attached dynamically using JavaScript. Unfortunately, there is no way to detect if an element has a dynamically attached event handler. However, there is a number of JavaScript libraries such as JQUERY and PROTOTYPE which keep track of which event handlers were added to an element. WEBMATE leverages this feature and supports event handlers attached with JQUERY and PROTOTYPE.

State abstraction. WEBMATE distinguishes different states of the application using an abstraction function over the state of the DOM. Abstraction is necessary since otherwise WEBMATE would consider two pages with slightly different content (e.g. a different heading) as distinct states. For a web application that is backed by a database, using the verbatim DOM would cause WEBMATE to explore the whole database. However, as a testing tool we would like WEBMATE to maximize coverage of the application while visiting as few states as possible. Our current implementation therefore employs an abstraction over all the user interface elements on the web page. The idea behind this is that the current state of the user interface reflects the state of the application. For some cases, this abstraction does not work well and therefore WEBMATE is unable to cover all functionality. However, in first experiments this approach has proven to produce concise usage models that yield good coverage (see Section 3).

Application size. When learning the usage model, WEBMATE tries to explore all interaction elements for every state of the application. For more complex applications, a single state may easily contain hundreds of interaction elements. Typically, many of these elements exercise the same functionality. For example, the result of a search on Amazon contains a link to the product page of every matching article. WEBMATE is perfectly able to visit all these links, this has a serious impact on the runtime of the analysis and at the same time mostly executes the same functionality. To solve this problem, WEBMATE currently employs a set of filtering techniques based on analyzing links and JavaScript code in order to identify elements that are likely to trigger the same functionality.

Non-determinism. As stated above, WEBMATE uses an abstraction over the DOM to estimate the state of the application. Since the majority of web applications also stores state information on the server, there is a (usually large) portion of the state that is invisible to WEBMATE but may influence the behavior of the application. As a consequence, the usage model contains a certain degree of non-determinism: Triggering an action several times may lead to different states and therefore some states in the model may become inaccessible. Unfortunately, there is no easy solution to this problem. To completely avoid non-determinism, we would need to reset the server-side state after every action. For real-world applications, this is not feasible. Therefore, WEBMATE uses heuristics to identify states that are not reachable and disregards them for further exploration.

As a black-box technique, WEBMATE cannot guarantee coverage of all functionality that may be reachable by the user interface. We are working on integrating search-based techniques [6] into WEBMATE that leverage server-side code coverage to direct test case generation.

3. CASE STUDY: COVERAGE

In this section we present a small case study to investigate the effectiveness of an early prototype version of WEBMATE. To measure effectiveness, we collect code coverage while WEBMATE explores the application's usage model. To put the resulting coverage values into context, we compare them to coverage values achieved by humans.

Our technical setup is as follows: Table 2 lists the two subjects in our study: JTRAC is a small issue tracker and HIPPO is a content management system. Both projects are web applications written in Java and are based on the APACHE WICKET web framework. We specifically chose these projects as WICKET, in contrast to other

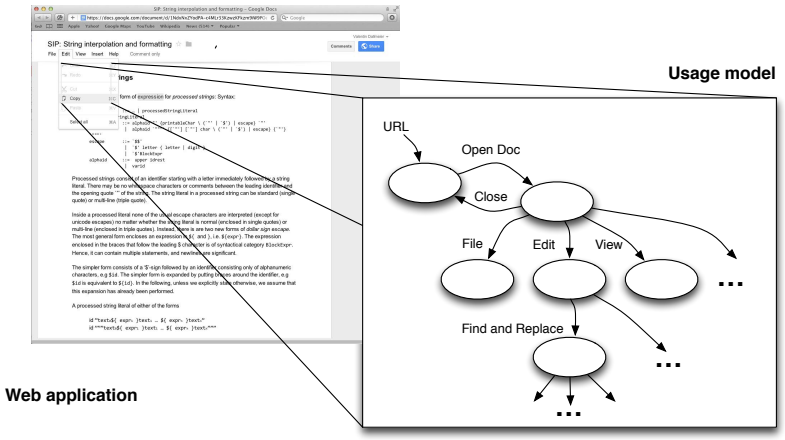

Figure 2: Simplified example of a usage model. States in the model correspond to different states of the application, whereas transitions in the model occur due to user interactions.

web frameworks, does not allow to encode logic into the HTML templates. Instead, all logic must be encoded in Java, and therefore the coverage results are more precise.

Both HIPPO and JTRAC require user authentication to access most functions. To access this functionality, we generate user accounts and provide the account data to both WEBMATE and the human user. To measure code coverage, we instrument each subject using Cobertura [10]. For our case study, we ran WEBMATE for 30 minutes on HIPPO and 10 minutes for JTRAC recording the coverage for both subjects. One of the authors of this paper then used the same amount of time to manually explore each application trying to cover as much functionality as possible.

The results of our case study are summarized in Table 1. For HIPPO, WEBMATE achieves significantly less coverage than manual exploration. This is mostly due to form inputs that HIPPO expects when creating new documents, which cannot be synthesized by WEBMATE at the moment. On the other hand, for JTRAC WEBMATE is able to cover almost the same amount of lines than manual exploration. In contrast to HIPPO, JTRAC expects much lass structured input which is why WEBMATE achieves much better results. Judging from our experiences with WEBMATE so far, we expect that input generation is one of the biggest challenges to achieve acceptable coverage. Section 5 briefly discusses our ideas for working on this and other issues.

Overall, the results of our case study are encouraging but we by no means claim that they are representative. Our choice of subjects is biased since we were specifically looking for Java subjects to be able to use Cobertura for coverage. Another issue is that our results do not include coverage of JavaScript code executed in the browser. Since modern web applications contain a considerable amount of JavaScript, server-side code coverage alone may be misleading. However, in this case study we focus on comparing manually and automatically achieved coverage. Since we use the same measurement in both cases, we consider the impact of missing JavaScript coverage to be negligible for the overall tendency of our results. Nevertheless, future evaluations of WEBMATE will have to include JavaScript coverage.

4. CROSS-BROWSER TESTING

When you develop a web application, you have to check for *cross-browser compatibility:* you have to ensure that your application *functions correctly across all, or at least the majority of web browsers.*[1] This includes aspects like correct rendering of all page elements as well as correct behavior of dynamic content, no matter what web browser, what browser version, or what operating system the user would use.

Table 1: Coverage results for JTRAC and HIPPO.

	Coverage (percent)	
Name	WEBMATE	Manual
JTrac	30.53	31.49
Hippo	42.85	62.34

Table 2: Subjects included in the coverage study. Project size is determined with David A. Wheeler's SLOCCount.

Name	Homepage	Size (LOC)
JTrac	`http://www.jtrac.info`	13,476
Hippo	`http://www.onehippo.com`	46,371

4.1 Classic Cross-Browser Testing

If you want to ensure cross-browser compatibility for your simple web site with ten static pages for 95 % of all your potential visitors, you would have to open and check those pages in 22 different browsers [9]. Thus, you would have to check 220 rendered pages one by one, and you would have to repeat that process after each change to your site to check for possible regressions. While this already would be a very tedious and time-consuming task, you would require access both to browsers on different platforms, and to ancient versions, which are not simply available on modern computers, such as IE 6. Furthermore, if your web site would contain forms that require user input, just to open those pages would not be sufficient. You rather would have to interact with the site, which would make the process even more complex and laborious—not to mention modern web applications, which practically are made of dynamic content that requires complex user interaction.

There are many tools available that help check for cross-browser compatibility issues. These are either based on *screenshots*, requir-

[1]An intranet application where all users would use the same browser software could be an exception.

ing you to manually inspect all the screenshots one by one (220 in our example); or *VNC-based*, allowing you to interactively test dynamic content, such as AJAX-powered interfaces, in different remote browsers (22 in our case).

All these tools are a long way from providing an automatic diagnosis. Currently, there are two services available that provide *web consistency testing* [2] [3, 8]. In contrast to many of the services that provide classic cross-browser testing, both services allow to crawl a web site. That way, they are able to compute a diagnosis not only for a single page, but for an entire web site. However, the underlying crawlers are restricted to hyperlinks as they occur in static web sites; thus, they are not able to cover a typical Web 2.0 application.

In the following, we will see how WEBMATE combines the techniques described in Section 2 with web consistency testing, and thus is able to automatically compute different comprehensive diagnoses for Web 2.0 applications. While WEBMATE's diagnoses include cross-browser compatibility issues among others, we will concentrate on the former.

4.2 Cross-Browser Testing with WebMate

Whenever you want to run a software test, you need an *oracle* that determines whether a given test has passed or failed. In the case of classic cross-browser compatibility testing, that oracle is a human being who manually examines, for instance, screenshots. However, if you want to run tests *automatically,* you will need an oracle that firstly is able to *unassistedly* decide whether a given behavior is correct, and that secondly can be executed at will—typically, a proper software program.

Tools that implement web consistency testing obviously require an oracle of the latter type. For this purpose, web consistency testing takes advantage of the way many developers approach web development: they write code and keep checking the effects in their favorite browser. Thus, in that *reference browser,* it is guaranteed that a page looks and behaves as expected; that behavior serves as *reference*. Having this baseline data at hand, we are able to check against all respective browsers for which a cross-browser diagnosis is to be computed; the latter we call *x-browsers*. Taking advantage of both this automatable oracle and its feasibility to explore Web 2.0 applications, WEBMATE is able to automatically derive cross-browser issues for these in three steps:

Collecting reference data. First of all, WEBMATE has to collect both the reference data in the reference browser and the corresponding data in all x-browsers. For this purpose, WEBMATE uses the techniques described in Section 2 to explore all states in each browser. Furthermore, for each state the tool serializes the DOM, and for all elements of interest, it annotates the DOM with rendering information, like the element's position and other properties that could affect its visual representation.

Comparing against reference. In a second step, WEBMATE compares the previously collected data between the reference and each x-browser; basically, the tool checks for each element of interest whether it is rendered *similary* in the respective x-browser. We cannot check for identical rendering, as this would render the approach useless: Firstly, for most applications a small difference in a previously specified tolerance range is acceptable. Secondly, to provide a diagnosis of high quality we avoid to report consequential errors. For instance, if a container element is misplaced, all the contained child elements would be misplaced as well. Therefore, before we check those child elements, we normalize their position relatively to the parent container; only then, we compare the relative positions of each child.

[2]Web consistency testing subsumes cross-browser and functional testing as well as regression testing. Most importantly, it aims to *automate diagnosing*; for instance, web consistency testing automatically checks for rendering issues across different browsers instead of leaving that task to developers.

Figure 3: Calendar widget properly rendered in the reference browser. WEBMATE highlights the correct boundaries of the widget to make comparison easier for the developer.

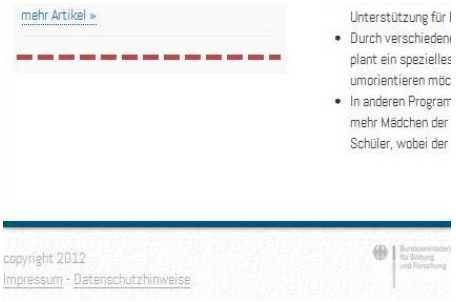

Figure 4: In one of the x-browsers, the calendar widget is rendered in principle, but with a height of 0. Here, WEBMATE draws a line to indicate the invisible widget's position.

Producing a diagnosis. The diagnosis provided by WEBMATE includes both a textual description of the issue, as well as two screenshots that show how the state is rendered in the reference and the respective x-browser. In the textual representation, the tool uses an unique XPATH expression that identifies the affected element. In the screenshots, WEBMATE highlights the affected element as rendered in the respective browser. Figure 3 shows an extract of how a calendar widget is rendered in the reference browser; in Figure 4, you can see that the calendar is missing in one of the x-browsers.

In addition to rendering issues as exemplarily shown above, WEBMATE is able to report *missing functionality;* for instance, a state could be not reachable at all, because a web shop's checkout button may accidentally be inactive in one of the x-browsers. As WEBMATE knows all the interactions available in the reference browser, it is able to report such issues—which no other approach is capable of.

Lastly, WEBMATE runs checks like HTML and CSS validation for each state to detect further browser-specific issues. That way, WEBMATE is able to automatically compute a comprehensive cross-browser compatibility analysis for state-of-the-art Web 2.0 applications.

5. FUTURE WORK

In an area where manual testing still is the norm, automatic test generation can tremendously assist developers in achieving and maintaining the quality and reliability of their systems. By systematically covering JavaScript functionality as well as Web pages, WEBMATE is applicable out of the box to a wide range of Web applications; by checking against other browsers, WEBMATE can easily detect a wide class of errors that take developers hours to check.

The easy availability of automatic test generation for Web applications allows for other novel ways of testing. Rather than checking against other browsers, WEBMATE may also check against earlier versions of the application and thus detect regression errors. A small set of well-placed assertions may suffice to easily detect arbitrary run-time errors for all applications with a Web interface.

That being said, WEBMATE still faces a number of challenges, all related to test generation. First and foremost, WEBMATE so far is a pure black-box testing tool without any feedback from the server side whether functionality in the code has been covered or not. In future work, we will extend WEBMATE to include search-based testing techniques in order to achieve even better coverage, in a way similar to search-based system testing [6]. To that end, we need to integrate coverage information for JavaScript as well as for the server-side parts of the application. The second major goal is to devise smart methods for string generation, e.g. generating sensible inputs for fields such as names, addresses, ZIP codes, or likewise. Both challenges are feasible in principle as in practice; and their solution will help turning WEBMATE into a general-purpose testing framework for web applications.

6. RELATED WORK

Research on analyzing and testing has gained momentum in the last decade. The body of related work can be grouped into client-side and server-side techniques.

6.1 Client-Side Techniques

One of the first approaches to analyzing dynamic web applications was by Benedikt et al. [2]. They introduced a framework called Veriweb which is able to automatically test simple web applications. In contrast to WEBMATE, Veriweb does not aim for exhaustively testing all actions but rather limits analysis time in order for the analysis to stop. Similar to WEBMATE, Veriweb also features a semi-automatic form filling algorithm that lets the user define mandatory inputs.

The work that is most similar to WEBMATE is a tool called Crawljax by Mesbah et al. [7]. Crawljax is able to crawl AJAX based web applications and extracts a state machine describing the application. In contrast to WEBMATE, Crawljax uses a much more low-level representation of the DOM to distinguish states. Also, Crawljax has no notion of dynamically attached event handlers and therefore has to resort to testing all elements on a web page if they are clickable. Due to these limitations we expect that WEBMATE would yield better results on modern web applications.

The work of Choudhary et al. [4] introduces a tool called Webdiff for analyzing cross-browser inconsistencies (see Section 4). Webdiff analyses the DOM and compares the output of screen shots captured in different browsers. In contrast to this, the cross-browser test application built on WEBMATE compares the rendering of individual elements in each browser and can thus be expected to provide much better results, in particular in the presence of browser-specific contents such as adds.

6.2 Server-Side Techniques

In contrast to the previous approaches, the APOLLO tool also analyses those parts of the application that reside on the server [1]. The main idea behind APOLLO is to use feedback-directed test generation to generate tests for the application. To improve the quality of the tests, APOLLO uses symbolic execution of the code at the server to direct the test generator towards unexplored areas of the code. Currently, APOLLO is able to analyze PHP code, which limits the applicability of the approach compared to WEBMATE. Nevertheless, the work of Artz et al. shows that including server side code into the analysis has a strong impact on the amount of code that can be covered by generated tests. In the future, we plan to extend WEBMATE into this direction.

7. CONCLUSIONS AND CONSEQUENCES

Web applications have become an essential part of our daily lives to shop, interact with other people, or perform financial transactions. All of these services are delivered through a combination of HTML, CSS and JavaScript. Programmers that maintain web applications have to deal with the increased testing effort this technology mixture incurs. Systematic and automated testing of web applications is by no means the standard in industry.

Our WEBMATE project alleviates this problem by providing a fully automatic means to analyze and navigate through a web application. Users of WEBMATE no longer have to manually navigate through a web page, but rather point WEBMATE to a certain state they wish to establish and the tool takes care of it. A preliminary case study shows that our early prototype already is able to achieve coverage comparable to that of a human being.

A large number of approaches from security, performance and regression testing can benefit from WEBMATE simply because they can be applied in states that previously could only be reached manually. As a proof-of-concept we show how WEBMATE can be used to automatically test an application for cross-browser inconsistencies, a task that used to be major pain for application developers.

Acknowledgments. This work was supported by the ERC Advanced Grant "SPECMATE—Specification Mining and Testing".

8. REFERENCES

[1] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of javascript web applications. In *ICSE*, pages 571–580, 2011.

[2] M. Benedikt, J. Freire, and P. Godefroid. Veriweb: Automatically testing dynamic web sites. In *In Proceedings of 11th International World Wide Web Conference (WWW 2002)*, 2002.

[3] Browsera, LLC. Browsera. http://www.browsera.com/.

[4] S. R. Choudhary, H. Versee, and A. Orso. Webdiff: Automated identification of cross-browser issues in web applications. In *ICSM*, pages 1–10, 2010.

[5] G. Code. Selenium. http://code.google.com/p/selenium/.

[6] F. Gross, G. Fraser, and A. Zeller. EXSYST: Search-based GUI testing. In *ICSE*, 2012. To appear.

[7] A. Mesbah and A. van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *Proceedings of the 31st Int. Conference on Software Engineering*, ICSE '09, pages 210–220, Washington, DC, USA, 2009. IEEE.

[8] Mogoterra, Inc. Mogotest. http://mogotest.com/.

[9] NetMarketShare. Desktop browser version market share. http://www.netmarketshare.com/.

[10] Sourceforge. Cobertura. http://http://sourceforge.net.