WebMate: Generating Test Cases for Web 2.0

Valentin Dallmeier, Martin Burger, Tobias Orth, and Andreas Zeller

Saarland University, Computer Science Department, Saarbrücken, Germany
{dallmeier,mburger,orth,zeller}@st.cs.uni-saarland.de
WWW : http://www.st.cs.uni-saarland.de/

Abstract. Web applications are everywhere—well tested web applications however are in short supply. The mixture of JavaScript, HTML and CSS in a variety of different browsers makes it virtually impossible to apply static analysis techniques. In this setting, *systematic testing* becomes a real challenge. We present a technique to *automatically generate tests* for Web 2.0 applications. Our approach systematically explores and tests all distinct functions of a web application. Our prototype implementation WEBMATE handles interfaces as complex as Facebook and is able to cover up to 7 times as much code as existing tools. The only requirements to use WEBMATE are the address of the application and, if necessary, user name and password.

Key words: test case generation, automate testing, Web 2.0, web applications

1 Introduction

In the software industry there is a strong trend towards replacing classic desktop applications with *web applications*—programs that are accessed via a browser and are typically run on a central server. Web applications are popular because they are easy to use and easy to maintain. The user only needs a browser, and the developer only has to maintain a single installation of the application. As a result, the cost of running a web application is relatively low compared to classic applications. On the other hand, quality assurance for web applications is difficult.

The user interface of a web application typically consists of JavaScript, HTML and CSS. This technology mix is notoriously difficult to debug. For instance, JavaScript is a dynamically typed language which makes static code analysis difficult. Therefore, existing techniques that can statically determine type errors cannot be applied. Another reason why debugging web applications is hard is that—despite existing standards—there are subtle implementation differences across browsers. As a result, code that works in one browser may not work in other browsers. Overall, testing web applications requires a lot of manual effort. As software release cycles are getting shorter and shorter, implementing effective quality assurance is difficult and therefore often ignored. As a consequence, users are faced with bad software that sometimes only fails to work correctly, but

sometimes also inadvertently leaks private data as witnessed by recent security breaches of popular web pages.

One way to alleviate this problem is to use *automated quality assurance*. As illustrated above, due to the complexity of the technology mix we cannot use static analysis or approaches like symbolic verification. Hence, the only technique that remains is *testing*. The technique of choice to verify that a web application is correct is *system testing*, which is able to check functional and non-functional requirements. However, manually creating and maintaining such tests again requires a lot of effort.

In the last years, we have seen a number of tremendously successful approaches that *automatically generate tests*. The majority of these techniques are being developed for individual programming languages, such as Java. In this paper, we investigate how existing approaches to test case generation can be applied in the context of Web 2.0 applications. Our work makes the following contributions:

1. We identify and discuss the main challenges when generating test cases for Web 2.0 applications (Sections 2 and 3).
2. We present WEBMATE, a tool that systematically explores web applications (Section 4). WEBMATE works fully automatically—it will only need credentials if parts of the web application are protected by a login.
3. We evaluate the effectiveness of WEBMATE and compare its performance to existing crawlers[1] (Section 5). While exploring the application, WEBMATE learns a *usage model* that describes all different functions of the web application. In one instance, our prototype is able to achieve up to 40 % code coverage.
4. We present an application where WEBMATE generates test cases to automatically test the compatibility of a web application in different browsers (Section 6).

We discuss related work in Section 7 and close the paper with concluding remarks and future work in Section 8.

2 Background: Test Case Generation

Automatic test case generation derives test cases from the program's code, its behavior, or other artifacts like a formal specification. The majority of these approaches generates tests for individual components of a program, such as methods or classes—so-called *unit tests*. On the other hand, *system tests* check the whole program at once by generating inputs for the user interface.

Existing approaches differ mainly in the concrete test generation approach and the resulting requirements:

[1] A crawler is a tool that systematically visits all pages of a web site and a web application, respectively.

- **Randomized testing** uses simple *randomized algorithms* to generate tests based on structural properties of the program. The work by Ciupa et al. [5] implements random testing for EIFFEL programs and uses invariants specified in the code to validate the test result.
- **Constraint-based testing** uses *symbolic execution* to simulate the execution of the program with symbolic values. From these executions, the techniques derive formulas that describe conditions on the program's input such that specific code regions are being exercised. With the help of a constraint solver, these formulas are solved to find valid input values. The scalability of these approaches is usually limited by the constraint solver. New approaches try to leverage this problem by combining concrete and symbolic execution [11].
- **Search-based testing** uses *machine learning algorithms* to efficiently navigate the search space for test inputs. The advantage of using genetic algorithms over constraint-based approaches is their ability to achieve maximum code coverage also for system tests, for example when generating inputs for user interfaces [9].
- **Model-based testing** requires a *model* that specifies the expected behavior of the program. The model is then used to derive test cases that cover as much of the specification as possible. One instance of such a tool is SPECEXPLORER from Microsoft. It allows to generate tests from specifications written in SPEC# [2] in order to verify that the code complies to the model.

With the exception of model-based testing, all of the above approaches cannot verify the correctness of the test outputs. An authority that decides whether a given test output is correct is called a *test oracle*. The lack of proper oracles for testing purposes is usually referred to as the *oracle problem*. This problem is one of the biggest challenges for automatic test case generation. Many approaches therefore only look for runtime exceptions when executing generated tests and disregard the program output.

Another issue for *unit test generation* is that many generated tests use test inputs that would never occur in reality. To circumvent this problem, we can generate tests on the system level rather than on the unit level; on the system level, inputs are no longer under the control of the program and therefore the program should be able to handle any input without raising an exception.

In the past few years, test case generation has made significant advances. Modern search-based approaches are able to achieve high coverage quickly by generating tests on the unit level [8] as well as on the GUI level [9].

3 Generating Tests for Web 2.0 Applications

In the scope of this work, we define the term *web application* to denote a program that requires a browser to be accessed by the user, and that is executed on a central server or, alternatively, in the cloud. Hence, web applications consist of two parts: a client-side part and a server-side part: on the client side, the web application uses HTML, CSS and JavaScript to implement the user interface; on

the server side, a wide variety of different programming languages and techniques is used. Thus, while the technology mix on the client side is fixed, there is a large number of different platforms and languages on the server side.

Can we transfer existing test generation approaches to web applications? Due to the technology mix and the distributed nature of web applications, generating tests on the unit level is difficult. Since unit test case generation is typically very close to the code, it has to be re-implemented for every language and platform on the server side. This causes a considerable effort which in turn makes the approach difficult to implement for a large number of applications. To alleviate this problem, we can restrict the test generation to system tests for an interface that consists of HTML, CSS and JavaScript. Our approach analyses *only those parts of the application that are executed in the browser* (black-box approach). Starting from the application's landing page, our approach systematically analyses all elements of the user interface on the current web page and continues exploration until no new elements and new web pages, respectively, are found.

This approach has the advantage that it avoids the heterogeneity on the server side. However, it also comes with a number of new challenges:

Recognize the user interface. In order to generate system tests, we need to identify all elements of the user interface. In traditional web applications, this only includes buttons, links and forms. However, in modern Web 2.0 applications, arbitrary elements can be associated with a JavaScript snippet that will be executed, for instance, when the user clicks on the element. With JavaScript, these so-called *handlers* can be added dynamically, which makes identifying user interface elements difficult

Distinguish similar states of the user interface. In order to generate test cases efficiently, we have to be able to identify *similar application states*. Otherwise, thousands of tests would be generated for one and the same function. Rather than generating tests for the whole data-set of the application (for instance, a social graph in a database), we would like to generate tests for all functions. In traditional web applications, it is possible to distinguish application states based on the URL of the browser only. In Web 2.0 applications, however, this is no longer possible since many JavaScript handlers change the state of the page in the background; thus, without modifying the URL.

Hidden state. To generate tests, we only use those parts of the application state that are visible to the browser. Since virtually all web applications also *store information at the server*, parts of the application state may change at any point in time. For example, in an email application, a new email may pop up at any time. Such state changes often also change the state of the user interface and the testing tool has to be able to cope with these changes.

Since the test case generation is restricted to those parts of the state that are visible to the browser, it may be unable to reach all of the web application's functions. On the other hand, this approach has the advantage that it does not require access to the code that runs on the server and is therefore easily applicable to a large number of projects.

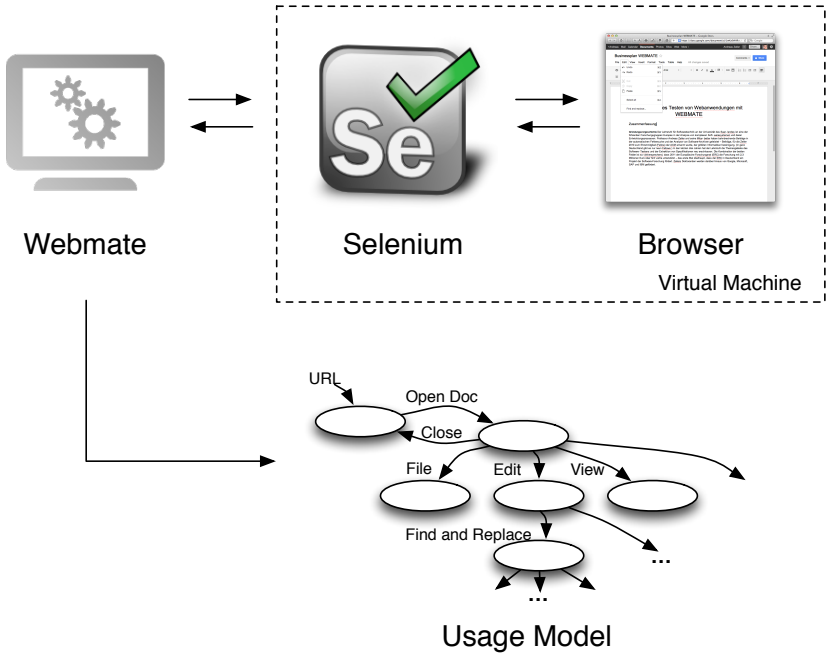

Fig. 1. WEBMATE employs Selenium to remote control the browser and learns a *usage model* that captures all possible ways to interact with the application. For security reasons, Selenium and the browser are sand-boxed in a virtual machine.

4 WebMate: A Test Case Generator for Web 2.0 Applications

WEBMATE is a prototype that implements the above approach to generate test cases for web applications. The main parts of WEBMATE are depicted in Figure 1: WEBMATE uses Selenium [6] to remote control the browser, interact with the web application, and extract the state of the user interface. In this setting, the state of the user interface comprises all elements that can be used to interact with the application. Using these techniques, WEBMATE is able to generate tests in two steps:

1. **Exploration.** In the first step, WEBMATE systematically tests all elements of the user interface in all different states of the application and derives a so-called *usage model*. This model captures the logical order of all possible ways to interact with the application. In essence, the usage model is a finite state automaton where states map to different states of the user interface and transitions between states are triggered by interacting with the application. Exploration ends as soon as WEBMATE is unable to find new states and all user interface elements have been explored.

2. **Testing.** In the next step, WEBMATE generates tests depending on what exactly the user of WEBMATE wants to test. For instance, WEBMATE is able to generate tests that systematically cover all states and interactions of the usage model learned in the exploration phase. For example, the cross-browser application described in Section 6 requires to visit all states of the usage model.

Since mouse clicks (and other movements like hovering) are the dominant way to interact with a web application, WEBMATE is able to trigger almost all user interface elements on its own. A problem, however, occurs with forms: in order to submit a form, all of its input fields must contain valid values. WEBMATE employs a set of heuristics to guess the range of values for an input field. Still, these heuristics fail as soon as a form requires complex input such as a pair of user name and passwords. For these cases, WEBMATE provides a way to specify input data for individual forms.

WEBMATE implements the following solutions for the challenges described in Section 3:

Recognize the user interface. WEBMATE recognizes all statically specified elements of the user interface by analyzing the HTML source code of the application. Dynamically added JavaScript handlers are also supported if they are added using one of the popular JavaScript libraries JQUERY or PROTOTYPE. If a web application uses other means to dynamically attach event handlers, WEBMATE will be unable to identify these elements as part of the user interface and therefore cannot include them in the analysis.[2]

Distinguish similar states of the user interface. To identify similar states of the user interface, WEBMATE uses an abstraction over all elements in the user interface. This abstraction characterizes the state of the application based on what functions are currently available to the user. Since WEBMATE is a testing tool, this abstraction makes sure that *two pages with the same functionality are mapped to the same state* and WEBMATE will visit each state only once. As any abstraction, this method also leads to a loss of information. In some cases, WEBMATE is therefore unable to explore all functions of the application.

Hidden state. The server-side part of the state is invisible to WEBMATE. If the state on the server changes, some states in the usage model may no longer be available to WEBMATE and the user interface would become non-deterministic from WEBMATE's point of view. As this problem is unavoidable, WEBMATE will tolerate these issues when exploring the application and will report errors when generating tests in the test phase.

In essence, WEBMATE is a tool to systematically explore all ways to interact with a web application and thus covers as much of the application as possible.

[2] In JavaScript, there is no official, uniformly way to retrieve a dynamically attached event handler. However, all major JavaScript libraries offer their own way to retrieve attached handlers. Therefore, for the time being, WEBMATE has to specifically support each library.

Hence, WEBMATE basically generates *executions*. When combined with generic oracles such as cross-browser compatibility checks (see Section 6 below), WEBMATE is able to fully automatically detect errors in web applications.

5 Evaluation: How does WEBMATE Improve Test Coverage?

To evaluate WEBMATE's effectiveness when analyzing Web 2.0 applications, we compare the coverage achieved by WEBMATE to that achieved by a traditional crawler for a set of subjects. From the point of view of the test case generator, we would like to achieve as much coverage as possible to test all parts of the program.

WEBMATE provides a first implementation of solutions to the challenges for Web 2.0 test case generation as described in Section 3. Since WEBMATE is still an academic prototype, we cannot expect it to achieve full coverage (i.e., 100 %). It is also difficult to measure how much coverage the tool could possibly achieve. Since WEBMATE generates tests on the system level, the tests can only reach those parts of the code that are reachable via the user interface. In this setting, full coverage would only be possible if all parts of the program are actually reachable by the user interface, which is hardly feasible for most applications. Hence, in this evaluation, we focus on the *relative* improvement of the coverage and *not on absolute* values.

5.1 Experimental Setup

For this experiment we use five open-source subjects. Table 1 lists the names and project sizes for all subjects. We chose our subjects to cover a variety of project sizes and domains. Our subjects range from small projects (1,448 lines of code) to fully blown web applications (65,692 lines of code). To facilitate our setup, we restrict the evaluation to subjects implemented in Java.

For our experiments we chose SCRAPY (`http://scrapy.org`) as a representative for traditional crawlers. SCRAPY is a popular tool that extracts data from web sites and is also used to provide load tests for web applications. In contrast to other crawlers, SCRAPY also allows to specify credentials. Since all our test programs require a login, this is a crucial feature to provide meaningful results in our evaluation.

To compare the coverage achieved by WEBMATE and SCRAPY, we measure the amount of lines covered. To get line coverage, we use Cobertura [14] to instrument each subject before it is executed. At runtime, the instrumented program logs all executed lines and calculates coverage at the end of the program run.

Both SCRAPY and WEBMATE get as inputs the landing page of each subject and credentials for a user account. To evaluate a single subject, we first run SCRAPY to collect coverage, reset the database of the subject to the state before SCRAPY was run, and then run WEBMATE to again collect coverage. Both crawlers are configured to only analyse pages that are part of the application.

Table 1. Subjects for the experimental evaluation. Project size is determined as the sum of the lines in all classes that are loaded at runtime.

Name	Homepage	Domain	Project Size (lines of code)
DMS	`dmsnew1.sourceforge.net`	document management	13,513
HippoCMS	`www.onehippo.com`	content management	65,692
iTracker	`www.itracker.org`	document management	15,086
JTrac	`www.jtrac.info`	task management	6,940
ScrumIt	`scrum-it-demo.bsgroupti.ch`	project management	1,448

Table 2. Results of the experimental evaluation. WEBMATE achieves up to seven times better coverage than SCRAPY.

	Coverage (percent)	
Name	WEBMATE	SCRAPY
DMS	19.3	10.8
HippoCMS	42.0	11.1
iTracker	33.2	7.2
JTrac	28.6	18.6
ScrumIt	38.5	5.5

5.2 Results

Table 2 lists the results of our experiments. Each line shows the coverage achieved by WEBMATE and SCRAPY for one subject. For all the subjects in this experiment, WEBMATE is able to achieve a higher coverage than SCRAPY. Thus, we can conclude that WEBMATE is more effective when generating tests for Web 2.0 applications than SCRAPY. The approach implemented in WEBMATE can significantly increase the coverage of generated tests: For ScrumIt, WEBMATE is able to achieve seven times as much coverage as SCRAPY. For HippoCMS and iTracker, coverage is still four times as high. Both applications make heavy use of JavaScript and dynamic HTML, which is why SCRAPY fails to achieve good coverage values.

Absolutely speaking, WEBMATE achieves between twenty (DMS) and forty (HippoCMS) percent coverage. For an automated testing tool these values are acceptable, but they are still not good enough for the tool to be of practical value. When analyzing the results of this experiment, we had several insights that lead to new ideas how to further improve WEBMATE. Some of these ideas are discussed in Section 8.

5.3 Threats to Validity

As any experimental study, the results of our experiments are subject to threats to validity. When discussing those threats, we distinguish between threats to internal, external and construct validity:

Threats to external validity concern our ability to transfer our results to other programs. As our study only includes five subjects, we cannot claim that the results generalize to arbitrary applications. In our experiments, the degree of coverage achieved by WEBMATE differs strongly between subjects, so we cannot make any predictions as to how WEBMATE would perform for other applications. Nevertheless, our results show that modern web applications need new approaches to test case generation in order to achieve acceptable coverage values.

Threats to internal validity concern the validity of the connections between independent and dependent variables in our setting. Since the selection process for the subjects in this study was not randomized, our sample is not independent. The authors of this study may have unintentionally preferred applications that make uncharacteristically strong use of JavaScript and hence are difficult for SCRAPY to analyze. However, in order to succeed in the web today, a web application has to provide a good user experience and therefore has to make heavy use of dynamic techniques. On the long run we expect the vast majority of web applications to employ a high degree of dynamic techniques.

Threats to construct validity concern the adequacy of our measures for capturing dependent variables. In our setting, the only dependent variable is coverage which is measured as the set of lines executed when running the program. The number of executed lines is directly connected to the control flow of the program and is the industry standard of measuring coverage. To measure this value, we use an open-source tool that is used by many other projects and hence can be expected to provide correct results. Overall it is safe to say that our measures for capturing dependent variables are adequate.

6 Application: Cross-browser Compatibility

Besides generating tests, a testing tool needs to generate oracles (see Section 2) in order to classify the outcome of each test. The oracle decides if the behavior of the program is correct or not. For semantic tests, oracles typically check the correctness of return values, for example "If there are three items in the shopping cart, the total sum is the sum of the individual prices for all three items." Without further information it is not possible to generate such oracles automatically. However, there is a number of problems for which *automatic oracles* can be generated. In this section, we present an application of WEBMATE where it is possible to provide such an automated oracle.

For the success of a web application it is vital that the application works correctly in all major browsers. A web application is said to be *cross-browser*

Table 3. Market shares of the major browsers first quarter 2012 [13].

Version	Market share [%]	Total [%]
IE 8.0	27.02	27.02
IE 9.0	12.82	39.84
Chrome 16.0	9.88	49.72
IE 6.0	7.38	57.10
Chrome 17.0	6.18	63.28
Firefox 9.0	5.54	68.82
IE 7.0	4.73	73.55
Firefox 10	4.43	77.98
Safari 5.1	3.37	81.35
Firefox 3.6	3.22	84.57
Firefox 8.0	2.82	87.40
Opera 11.x	1.45	88.85
Firefox 11	1.19	90.04
Safari 5.0	1.08	91.12
Firefox 4.0	0.64	91.76
Firefox 6.0	0.63	92.39
Firefox 7.0	0.56	92.95
Firefox 3.0	0.55	93.50
Firefox 5.0	0.51	94.02
Firefox 3.5	0.48	94.50
Chrome 14.0	0.47	94.97
Chrome 15.0	0.39	95.36

compatible if it is rendered identically and works correctly in all browsers. In practice, maintaining cross-browser compatibility is a real challenge. In total, there are five major browser vendors; for almost all browsers, there is more than one version available and some browsers are provided for different platforms. The distribution of market shares in Table 3 shows that for many browsers more than one version is actively used. As a consequence, in order to support 95 % of the customers, the developers of a web application have to test 22 different browsers. These tests are necessary because different browsers (and also different versions of the same browser) behave differently despite the fact that there are existing standards for the most important technologies. For quality assurance, this has serious consequences: For instance, to test a small web application with just 10 pages, a tester would have to manually compare 220 pages. Moreover, to avoid regression errors, these tests would have to be carried out after each change to the application.

By using WEBMATE, we are able to fully automate these tests as follows. In practice, developers of web applications normally use a fixed browser to run tests while implementing the application. Hence, for this so-called *reference browser*, the application can be expected to work correctly. WEBMATE uses the reference browser to check the rendering of all *test browsers* (those browsers for which cross-browser compatibility is to be tested). If an element is rendered by a test

Fig. 2. Example of a diagnosis produced by WEBMATE. In Internet Explorer 8, the checkout button is missing. WEBMATE identifies the missing element and visualizes the problem using screen shots.

browser at a different position than in the reference browser, WEBMATE will report this problem to the user. Besides the challenges for test generation described in Section 3, there are a number of additional problems when implementing an automated cross-browser compatibility oracle:

Recognize layout problems. To detect layout problems, WEBMATE has to compare positions and sizes for each element on the web page. A simple comparison of screen shots is insufficient as dynamic content such as ads may cause large visual differences even though all relevant elements are rendered the same. Also, in practice it is often tolerable for elements to be a few pixels off, but a screen shot based comparison would not tolerate such minor deviations.

Browser specific code. To avoid incompatibilities between browsers, some web applications deploy different code based on which browser is used. Since the state abstraction of WEBMATE is based on identifying elements of the user interface, it may happen that WEBMATE inadvertently considers the same page in different browsers to be different pages. In these cases, WEBMATE is unable to match pages across browsers and therefore cannot provide a cross-browser test.

Besides layout problems, WEBMATE is also able to recognize cases where functions are present in the reference browser, but are missing in one or more test browsers. WEBMATE compares the usage model (Section 4) of the reference browser with those extracted for the test browsers. For example, if a test browser does not display the button to remove an article from a shopping cart in a web shop, the user will be unable to remove accidentally added items and therefore will likely choose another web shop. For the maintainer of the web shop, this problem is only visible in the number of aborted transactions, which could also have a variety of other causes and is difficult to investigate. For the success of a web application, it can be vital to detect such missing functionalities in a test browser. To support developers in finding such errors, WEBMATE generates a report that visualizes all deviations found in the cross-browser analysis (Figure 2).

7 Related Work

The related work for this paper can be grouped into general approaches to test case generation and testing of web applications. A summary of the most important generic test case generation approaches was already given in Section 2.

The VeriWeb project by Benedikt et al. [3] is one of the first approaches to test dynamic web applications. In contrast to WEBMATE, VeriWeb does not use state abstraction and is therefore unable to systematically explore a web application. VeriWeb uses fixed time limits to restrict analysis time, whereas WEBMATE automatically finishes the exploration as soon as all states and interactions are explored.

Mesbah et al. [12] present *CRAWLJAX*, a tool for analysing web applications implemented with AJAX. Similar to WEBMATE, Crawljax also extracts a finite state automaton that describes different states of the application. However, in contrast to WEBMATE, Crawljax does not use state abstraction but employs tree comparison algorithms to identify user interface elements that were changed by AJAX operations. Since the recursion depth for this algorithm is limited, Crawljax is also not able to detect when all states and interactions are explored. Also, Crawljax requires the user to manually identify all active elements of the user interface, whereas WEBMATE is able to automatically identify them and can thus be used for arbitrary applications without further configuration.

In the area of cross-browser testing, Choudhary et al. [4] present their tool named Webdiff. In contrast to WEBMATE, Webdiff applies a screen-shot based

comparison approach which has a number of problems in the presence of dynamic elements such as ads (see above). Also, Webdiff is not able to systematically explore a web application, which is one of the key features of WEBMATE.

Artzi et al. present APOLLO [1], a tool that analyses both the client-side and the server-side parts of a web application. APOLLO employs symbolic execution [10] and user inputs to systematically generate tests that reach previously uncovered code. Currently, APOLLO is only implemented for PHP and cannot be applied to other projects.

In earlier work on WEBMATE [7], we present initial ideas and an early evaluation. This paper extends our previous work with a discussion of general issues for test case generation in modern web applications. In the current paper, we study the effectiveness of specialized test case generation techniques compared to traditional crawlers.

8 Conclusions and Future Work

The technical complexity of modern web applications poses completely new challenges for automated test case generation. On the other hand, systematic quality assurance is even more important to guarantee dependable and secure systems. In this work, we identify the most important challenges when generating tests for modern web applications and discuss possible solutions. Our prototype WEBMATE implements several of these solutions. In a controlled experiment, we investigate the effectiveness of the solutions built into WEBMATE in terms of the coverage achieved on a set of test subjects. Compared to traditional crawlers, WEBMATE is able to achieve up to seven times as much coverage and is therefore much more effective in generating tests.

Despite the promising results of our evaluation, the quality of the tests generated by WEBMATE is still not good enough to be useful in practice. In the future, we will investigate the following ideas to further improve the effectiveness of WEBMATE:

Server-side code analysis. The work of Artzi on APOLLO [1] shows that test case generation can benefit from analyzing server-side code. We plan to use code instrumentation on the server to provide feedback that allows WEBMATE to specifically generate tests for uncovered areas of the program.

Test data provisioning. To improve the generation of input data for forms, we plan to use search-based testing techniques such as genetic algorithms. As a source of information, we would also like to use the content of the web site, which often already specifies valid values for input fields.

More information about WEBMATE can be found at

`http://www.st.cs.uni-saarland.de/webmate/`

References

[1] Shay Artzi et al. "A framework for automated testing of javascript web applications". In: *ICSE*. 2011, pp. 571–580.

[2] Mike Barnett et al. "The Spec# Programming System: Challenges and Directions". In: (2008), pp. 144–152. DOI: http://dx.doi.org/10.1007/978-3-540-69149-5_16.

[3] Michael Benedikt, Juliana Freire, and Patrice Godefroid. "VeriWeb: Automatically Testing Dynamic Web Sites". In: *In Proceedings of 11th International World Wide Web Conference (WWW 2002)*. 2002.

[4] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. "WEBDIFF: Automated identification of cross-browser issues in web applications". In: *ICSM*. 2010, pp. 1–10.

[5] Ilinca Ciupa et al. "Experimental assessment of random testing for object-oriented software". In: *ISSTA '07: Proceedings of the 2007 International symposium on Software testing and analysis*. London, United Kingdom: ACM, 2007, pp. 84–94. ISBN: 978-1-59593-734-6. DOI: http://doi.acm.org/10.1145/1273463.1273476.

[6] Google Code. *Selenium*. http://code.google.com/p/selenium/.

[7] Valentin Dallmeier et al. "WebMate: A Tool for Testing Web 2.0 Applications". In: *JsTools*. To appear. 2012.

[8] Gordon Fraser and Andreas Zeller. "Generating parameterized unit tests". In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA '11. Toronto, Ontario, Canada: ACM, 2011, pp. 364–374. ISBN: 978-1-4503-0562-4. DOI: 10.1145/2001420.2001464. URL: http://doi.acm.org/10.1145/2001420.2001464.

[9] Florian Gross, Gordon Fraser, and Andreas Zeller. "Search-Based System Testing: High Coverage, No False Alarms". In: *ISSTA*. To appear. 2012.

[10] James C. King. "Symbolic execution and program testing". In: *Commun. ACM* 19.7 (1976), pp. 385–394. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/360248.360252.

[11] Rupak Majumdar and Koushik Sen. "Hybrid Concolic Testing". In: *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 416–426. ISBN: 0-7695-2828-7. DOI: http://dx.doi.org/10.1109/ICSE.2007.41.

[12] Ali Mesbah and Arie van Deursen. "Invariant-based automatic testing of AJAX user interfaces". In: *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 210–220. ISBN: 978-1-4244-3453-4. DOI: http://dx.doi.org/10.1109/ICSE.2009.5070522.

[13] NetMarketShare. *Desktop Browser Version Market Share*.

[14] Sourceforge. *Cobertura*. http://http://sourceforge.net.