Web Application Modeling for Testing and Analysis

William G.J. Halfond
Advisor: Alessandro Orso
College of Computing
Georgia Institute of Technology
{whalfond, orso}@cc.gatech.edu

Abstract

The goal of my work is to improve quality assurance techniques for web applications. I will develop automated techniques for modeling web applications and use these models to improve testing and analysis of web applications.

Categories and Subject Descriptors

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

General Terms

Reliability, Verification

Keywords

Web applications, testing, analysis, modeling

Hypothesis

Automated modeling of web applications can be used to improve testing and analysis of web applications.

1. RESEARCH PROBLEM MOTIVATION

Testing and analysis techniques for web applications have become vitally important. Many companies provide a diverse range of services, such as banking, communication, and shopping, via their web applications. The ability to provide these services with a high level of reliability and quality is essential to the success of these companies.

Many of the techniques developed for software testing and analysis are not directly applicable to modern web applications. For example, many test-input generation techniques assume that a software module has a set of explicitly defined interfaces. For web applications, this set does not exist since web application interfaces are implicitly defined by the control flow of a web application. Other characteristics of web applications, such as the use of HTTP or generated object programs, can make it difficult to identify message passing or know when a test suite is adequate. More generally, many of the abstractions used by traditional software quality techniques, such as control-flow, data-flow, test-adequacy criteria, and interface definitions, are defined differently for web applications and this complicates the application of traditional testing and analysis techniques to web applications.

The growing complexity of modern web applications also creates difficulties for many techniques developed for early web applications. Since early web applications were comprised of primarily static HTML pages, it was sufficient to focus exclusively on the web pages themselves. Many techniques employed HTML validators to statically check HTML pages or used web crawlers to ensure that web pages displayed correctly at runtime. Modern web applications have become more complex and can dynamically generate HTML content, interact with external systems, and combine data from multiple sources. Many current techniques that are targeted at web applications cannot account for these features and are therefore limited with regards to the types of web applications they can analyze.

To improve testing and analysis for web applications, I propose the development of a set of techniques for modeling web applications. This set of techniques will focus on identifying aspects of modern web applications that have made the application of existing testing and analysis techniques challenging; for example, control flow, data flow, interface definitions, and test-adequacy criteria. Many testing and analysis techniques depend on these software abstractions, and developing analyses that identify them will facilitate the improvement and application of these techniques for web applications. I will also develop and implement new testing and analysis techniques that use information from the models and evaluate the performance of these techniques on a set of web applications.

In the rest of this document I describe my proposed research in more detail. First, I provide background information on web applications in Section 2. In Section 3, I explain the proposed methodology, evaluation methods, and expected contributions of the dissertation work. Section 4 describes the preliminary results, and Section 5 surveys related work in the area.

2. BACKGROUND

A *web application* is a software system that is available to users over the Internet. It is composed of a set of *web components*, which are the modules that implement the application's functionality, such as logging in, displaying an order, or processing a request. Each component provides a *root method*, which is the entry point for the component and is the first method called when the component executes.

FSE-16 Doctoral Symp., November 10, Atlanta, Georgia, USA

To invoke the functionality of a component, a client (e.g., a web browser or another component) submits an HTTP request to the target component. The request, which we call an *interface invocation*, contains zero or more name-value pairs called *input parameters* (IP). A web server receives the invocation on behalf of the target component and calls the root method with the invocation as an argument. After a component is invoked, it can access and use the set of IPs contained in the invocation. An IP is accessed by calling one of a set of special API functions, called *parameter functions* (PF); given the name of an IP, a PF returns the IP's value. We refer to the set of IPs accessed by a component during a particular execution as an *accepted interface*. A component has multiple accepted interfaces if it accesses different sets of IPs along distinct paths of execution.

During execution a component can interact with other components and perform a wide variety of operations. A component's most typical operation is to generate HTML pages by writing HTML tags and content to its output stream. These pages are returned to the end user and displayed in a browser. A component can also issue HTTP commands to the client. These commands can be informative or instruct the client to do certain operations, such as update a cached web page or redirect to another location on the web.

In practice, developers use different general-purpose programming languages and frameworks to implement components. For Java, the most popular web application framework is the Java Enterprise Edition (JEE), which provides a basic component interface called a servlet. Developers implement servlets by writing Java code or providing a Java Server Page (JSP). Developers also implement components using the Active Server Page (ASP) framework in the .NET platform; Perl, Python, and Ruby scripts; and, in general, any application that uses the Common Gateway Interface (CGI). The specific API calls that serve as PFs vary by language and framework; however, the HTTP format used to transmit and encode IPs is the same for all web applications.

3. METHODOLOGY & CONTRIBUTION

The goal of my research is to improve quality assurance techniques for web applications. To do this my dissertation work will focus on the development of automated techniques for modeling web applications. I hypothesize that this model, called the *web application model*, can be used to improve testing and analysis techniques for web applications. To evaluate my hypothesis, I will develop techniques that use the web application model and compare the effectiveness of these against techniques that do not use the model. The following objectives will be part of my research:

1. Create automated techniques to model web applications.
2. Develop and evaluate the following techniques:
   (a) Static analysis technique for identifying errors related to inter-component communication.
   (b) Test-input generation technique that uses information from the model to test web applications.
   (c) Test-adequacy criteria for web applications based on coverage of features of the model.

3.1 Objective 1: Web Application Modeling

The first objective is to create an automated approach for generating models of web applications based on static analysis of their code. Since there is no widely accepted standard for web application models, I have compiled a list of elements, based on preliminary work and experience, that are good candidates for inclusion in the model. I expect that these requirements will evolve and become more refined as the development of the techniques for Objective 2 progresses. The candidate elements are:

1. Components of the web application.
2. Links between components that communicate.
3. Accepted interfaces of each component.
4. Interface invocations of each component.
5. Sequencing information showing the relative ordering of the sending and receiving of interface invocations.

Several different types of representations will be necessary to capture the static structure and dynamic behavior of the web application. For example, elements 1–4 represent static features of the web application and could be shown with an annotated directed graph. Element 5 represents runtime behavior and could be shown using a message sequence chart.

Figure 1 shows a possible static web application model. Components (Element 1) are represented by the boxes labeled C_1, C_2, and C_3. Components that communicate via invocations (Element 2) are linked by one or more lines. The direction of the arrowhead on each line shows the direction of the invocation. For example, in the figure, the direction of the arrow from C_1 to C_2 shows that an invocation is made by C_1 to C_2. The accepted interfaces of each component (Element 3) are shown by the small boxes labeled A_1, A_2, A_3, and A_4. If a communication link accesses a specific accepted interface, the link is shown pointing to the accepted interface. Each invocation (Element 4) is shown in the box that annotates each communication link. This box shows the set of name-value pairs transmitted in the invocation.

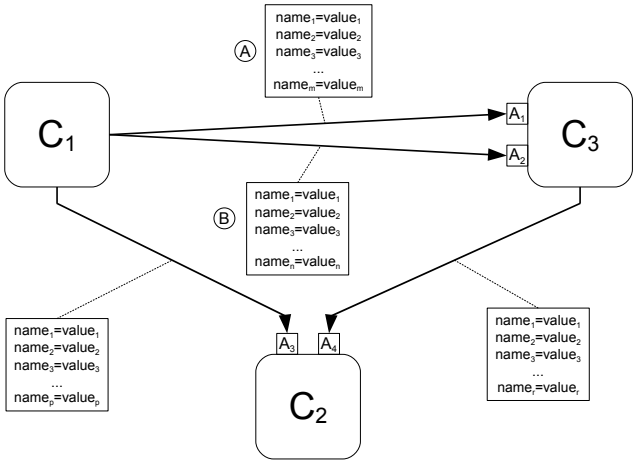

Figure 1: Static web application model.

To generate the web application models, I will develop an approach based on static analysis of the web application. This approach will analyze the code of a web application (e.g., JSP pages, servlets, and libraries) and automatically generate the corresponding models. The approach will most likely consist of a family of related techniques, each identifying a certain element in the web application model.

3.2 Objective 2: Testing and Analysis

In the second part of my dissertation work, I will develop a family of techniques that uses the information from the web application model and compare their usefulness and effectiveness against techniques that do not use this information. In the rest of this section, I outline each of the techniques I will develop and explain how I will evaluate them.

Analysis to Identify Errors in Interface Usage: For modern web applications, errors in the generation of interface invocations have become a common and serious problem [6]. In general, there are four types of errors related to interface invocations: 1) invocation of a non-existing interface, 2) incorrect invocation of an existing interface, 3) error in the domain of an individual parameter in an invocation, and 4) presence of interfaces that are not invoked by another component in the application. Most current techniques rely on testing in order to catch these errors; however, testing is inherently incomplete and would typically catch only a subset of the errors in the application.

Each of the four types of errors in interface invocations can be detected by analyzing the web application model. My technique will use the information available in the web application model to reconcile and compare the interface invocations with the accepted interfaces of the target components. Interface invocations that do not match an accepted interface will be reported to developers as a potential error.

To evaluate the analysis, I will determine its accuracy and compare its effectiveness against testing-based techniques. To evaluate effectiveness, I will use the analysis to identify errors in a set of subject web applications that contain real and seeded errors. I will then compare the identified errors against the ones found by testing-based techniques in order to determine my analysis's comparative effectiveness.

Test-input Generation: Accurate and complete information about a web application's accepted interfaces is useful for testing. Testers use this information to know which parameters must be included in a legal invocation to a web component. Incomplete information about an application's interfaces can lead to parts of the code being untested. Identifying this information in dynamic web applications is difficult because the interfaces are defined at runtime by the path of execution. To identify accepted interfaces, current test-input generation techniques for web applications rely either on purely dynamic approaches, such as web crawling, that are inherently incomplete, or require developers to manually specify all interfaces to be tested, which makes the process expensive and dependent on the developers' accuracy. The web application model can improve test-input generation as it provides a means for accurately identifying the accepted interfaces of each component in the web application.

In my work I will develop a test-input generation technique for web applications that will generate test cases by leveraging information from the web application model, such as accepted interfaces, interface domain constraints, and invocation sequences. I will evaluate this technique to determine the usefulness and effectiveness of my technique as compared to other approaches, such as test-input generation based on web-crawlers and a technique proposed by Deng, Frankl, and Wang [5]. To evaluate usefulness, I will compare the generated test suites in terms of the time to generate them and their size. To evaluate effectiveness, I will compare the structural coverage achieved by the test suites using traditional criteria such as block, branch, and data-flow coverage.

Testing Criteria: Test suite adequacy is an important metric in the measurement of the quality of a test suite. Traditional testing criteria, such as statement and branch coverage, have been widely used to measure test suite adequacy. However, certain characteristics of web applications can reduce the meaningfulness of these criteria. One of these characteristics is the generation of an object-program, such as an HTML page or SQL query, by a web component. The generated object-program can have characteristics that do not correlate with the structural characteristics of the generating meta-program. Therefore, testing criteria focused on the object-program may provide a better indicator of testing quality. In particular, the web application model contains several types of elements, whose coverage may represent effective testing requirements.

In my work I will define and evaluate new testing criteria for web applications. These new criteria will be based on elements represented in the web application model. For example, new testing criteria could include coverage of accepted interfaces, interface invocations, or input parameters. To evaluate these criteria, I will compare each proposed criterion in terms of its usefulness and feasibility. This evaluation will include several metrics: 1) correlation between coverage of the test requirements and fault-detection ability, 2) accuracy in calculating the test requirements, and 3) overhead of monitoring coverage. In my evaluation, I will use these metrics to compare my criteria against traditional testing criteria, such as block, branch, and data-flow coverage.

4. PRELIMINARY RESULTS

My preliminary work on the web application model focused on identifying the accepted interfaces and using that information to improve test-input generation for web applications [10]. In previous work, I proposed an approach that was based on a two-phase static analysis. One phase identified IPs and grouped them into accepted interfaces, and the other annotated each IP with domain information. The empirical evaluation showed that the accepted interfaces information was useful for test-input generation. The approach was able to achieve a 30% increase in block coverage, a 48% increase in branch coverage, and a 94% increase in command-form coverage as compared to using interface information from a traditional technique – web crawling. A drawback of this approach is that the technique is conservative, which can lead to the identification of spurious interfaces that correspond to infeasible paths. In ongoing work, we are using symbolic execution to more accurately model the accepted interfaces and eliminate the identification of these spurious interfaces. Preliminary results indicate that the precision is useful and we are able to achieve higher coverage with significantly fewer test cases. Recent work focuses on the identification of a component's interface invocations and the development of an analysis technique for identifying erroneous invocations [11].

My preliminary work also includes an investigation of a potential test adequacy criterion for web applications. Many web applications dedicate a significant portion of their code to building SQL queries that interact with the application's underlying database. To address this, I proposed command-form coverage [9], which measures a test suite's coverage by

tracking the number of distinct database queries generated by a component under test. Higher coverage with this criterion indicates that the application is exhibiting a wider range of behaviors with respect to its interaction with the database. The empirical evaluation of this criterion shows that many test suites for web applications achieve a reasonable level of statement and branch coverage, but have a low level of command form coverage. These results indicate that different testing techniques, designed specifically for web applications, are needed to provide more adequate testing.

5. RELATED WORK

Related work includes several approaches that build models of web applications. An early technique by Ricca and Tonella allows developers to use UML to model the links and interface elements of each page in a web application [15]. Work by Betin-Can and Bultan provides a more expressive modeling languages that allows developers to represent dynamic interactions between components in a web application [2, 4]. A drawback of these techniques is that they do not provide for an automatic analysis of a web application and require developer-provided specifications.

Recent work has made use of both static and dynamic analysis to analyze a web application and automatically build models. Deng, Frankl, and Wang model a web application by scanning its code and identifying links and names of input parameters [5]. However, their approach is context and flow insensitive, so it cannot capture groupings along different paths of execution. Licata and Krishnamurthi use static analysis to model and verify the control flow of a web application, but do not model other properties [13]. Elbaum and colleagues dynamically build a model of a web-application interface by submitting a large number of requests to the application, observing the results of the requests, and dynamically inferring constraints on the domain of the input parameters [7]. However, this approach is not able to provide any guarantees of completeness.

Several related approaches use program analysis to verify the HTML output of a web application [1, 3, 14]. They use program analysis to approximate the set of generated HTML pages and then check the pages using an HTML validator. These approaches are focused exclusively on validating the correct syntactical structure of the generated HTML pages and are unable to verify other properties.

One common approach to test case generation for web applications is to use captured user-session data to guide test case generation. An early approach by Kallepalli and Tian [12] mines server logs to build a statistical model of web application usage that can be used to guide testing and measure the reliability of a web application. A subsequent approach by Sant, Souter, and Greenwald [16] focuses on how these statistical models could be used to generate better test cases. Another approach by Elbaum and colleagues [8] captures user-session data and uses the captured data directly as test inputs. Sprenkle and colleagues [17] propose an automated tool that can support the approach proposed by Elbaum and colleagues and generate additional test cases based on the captured user-session data. These approaches have been shown to be effective at finding bugs, but are inherently limited since they can model only observed behavior and cannot analyze or test behaviors they have not seen.

6. REFERENCES

[1] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding Bugs in Dynamic Web Applications. In *Proceedings of the Intl. Symp. on Software Testing and Analysis*, Jul. 2008.

[2] A. Betin-Can and T. Bultan. Verifiable Web Services with Hierarchical Interfaces. In *Proceedings of the Intl. Conf. on Web Services*, Jul. 2005.

[3] C. Brabrand, A. Møller, and M. I. Schwartzbach. Static Validation of Dynamically Generated HTML. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, Jun. 2001.

[4] T. Bultan. Modeling Interactions of Web Software. In *Intl. Workshop on Automated Specification and Verification of Web Systems*, Nov. 2006.

[5] Y. Deng, P. Frankl, and J. Wang. Testing Web Database Applications. *SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004.

[6] S. Elbaum, K.-R. Chilakamarri, B. Gopal, and G. Rothermel. Helping End-Users "Engineer" Dependable Web Applications. In *Intl. Symp. of Software Reliability Engineering*, Nov. 2005.

[7] S. Elbaum, K.-R. Chilakamarri, M. F. II, and G. Rothermel. Web Application Characterization Through Directed Requests. In *Intl. Workshop on Dynamic Analysis*, May 2006.

[8] S. Elbaum, S. Karre, and G. Rothermel. Improving Web Application Testing with User Session Data. In *Intl. Conference on Software Engineering*, Nov. 2003.

[9] W. G. Halfond and A. Orso. Command-Form Coverage for Testing Database Applications. In *Intl. Conf. on Automated Software Engineering*, Sep. 2006.

[10] W. G. Halfond and A. Orso. Improving Test Case Generation for Web Applications Using Automated Interface Discovery. In *Proceedings of the Symp. on the Foundations of Software Engineering*, Sep. 2007.

[11] W. G. Halfond and A. Orso. Automated Identification of Parameter Mismatches in Web Applications. In *Proceedings of the Symp. on the Foundations of Software Engineering*, Nov. 2008.

[12] C. Kallepalli and J. Tian. Measuring and Modeling Usage and Reliability for Statistical Web Testing. *IEEE Transactions on Software Engineering*, 27(11):1023–1036, 2001.

[13] D. Licata and S. Krishnamurthi. Verifying Interactive Web Programs. In *Proceedings of the Intl. Conf. on Automated Software Engineering*, Sep. 2004.

[14] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *Proceedings of the Intl. World Wide Web Conf.*, May 2005.

[15] F. Ricca and P. Tonella. Analysis and Testing of Web Applications. In *Intl. Conf. on Software Engineering*, May 2001.

[16] J. Sant, A. Souter, and L. Greenwald. An Exploration of Statistical Models for Automated Test Case Generation. In *Proceedings of the Intl. Workshop on Dynamic Analysis*, May 2005.

[17] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated Replay and Failure Detection for Web Applications. In *Proceedings of the Intl. Conf. on Automated Software Engineering*, Nov. 2005.