Automated Testing of PHP Application Functionality

Anonymous
Anonymous@cis.udel.edu
Computer and Information Sciences
University of Delaware
Newark, DE 19716

December 9, 2008

Abstract

Generally, testing any application should be both effective, in that the testing covers a wide range of functionality, and efficient, in that the testing process is as short as possible. As dynamic web applications increase in size, the developer's test input size also increases; therefore, testing by hand is neither effective nor efficient. The developer must manually check all inputs for all textboxes, radio buttons, etc. and click all available links and buttons throughout the application in order to fully test application. Previous work done by Artzi et. al. [1] explores avenues by which the testing process can be automated and the input size into a PHP application is reduced; however, only the HTML output of the application is verified, meaning that the structure and validity of the application output is tested. Their tool does not determine that the application is performing, functionally, as intended, and this is a critical testing avenue when considering applications that are computationally intensive and where precision is mandatory. We propose to build upon Artzi et. al.'s work by first implementing their work, then, extending it to test that the functionality of the PHP application is correct. In addition to this, we will ensure full functionality of the PHP Web application is tested by developing a process of "exploding" the JavaScript to expose any functionality hidden by previous work.

1 Introduction

Testing an application can fall into two categories: testing for syntactic correctness and testing for semantic correctness. (Here we extend sytnax errors as statements that are not only malformed or produce malformed HTML, but also statements which produce faults). That is an application can be tested to ensure that the actual written code is both legal and does not produce faults. There has been significant work in developing tools and methods to automatically perform this task with dynamic web applications. These are effective at finding bugs on a syntactic level and generating inputs automatically, and now the following question needs to be addressed: "'Is the application both correct syntactically AND is the application producing the intended results?"' A dynamic web application the does not crash but does not produce the intended results based on user input will still pass tests performed by previous tools. However, the application may not be correct, functionally, because it is not behaving as the application developer intended. For instance, a web application that contains a PHP function which prints out student grades and statistics in an HTML table may produce valid HTML. Additionally, the application may execute without faults and crashes. However, if this validly-formed HTML table is not populated with any relevant information, the application can be labeled as being misbehaved, and there is something wrong with the `print()` function in the PHP script.

A tool that solves this problem would be a solution to the second of the two categories described above.

We would like to develop an effective and efficient solution to this problem by building upon the exising foundation of finding bugs of the syntactic nature. Previous work done by Artzi et. al. [1] explores avenues by which the testing process can be automated and the input size into a PHP application is reduced. In this work, only the HTML output of the application is verified, with the structure and validity of the application output being the only testing focal points. Their tool does not confirm that the application is performing as intended in a functional manner, and this is a critical testing avenue when considering applications that are computationally intensive and where precision is mandatory. We propose to implement previous work, then, extend it to test that the functionality of the PHP application is correct. In addition to this, we will ensure full functionality of the PHP Web application is tested by developing a process of "exploding" the JavaScript to expose any functionality hidden, which is a limitation of some previous works.

2 PHP Web Applications

2.1 The PHP Language

PHP is a web application scripting language that is used in developing many applications. PHP has large library support for network interactions, database interactions, and HTTP processing. This popular scripting language is also instrumental in putting the "dynamic" aspect into the dymanic functionality of a web application. A web application being dynamic allows it to respond to user interaction; thus, the application can respond to each input from the user.

PHP is also considered an object oriented language. It can provide functionality to define classes, write interfaces, and dispatch methods dynamically. Unique to PHP, apart from object-orientation, is its scripting features. Variables are dynamically typeset at runtime, which can make it easy for the developer by removing the need to declare the variables as a certain, defined type. The side effect is that the variable's type is not immediately apparent. Another scripting feature of PHP is 'eval', which executes a value that was computed at runtime as a code fragment. For example,

```
$code = "$x = 3;";
$x = 7;
eval($code);
echo $x;
```

evaluates the expression in quotes and prints the value 3. As a scripting language, PHP also can use a predicate function to check whether a variable is defined or set

```
if(!isset($_GET[page]))...
```

and declare functions and classes anywhere in the program.

```
<?php
function validateLogin() {
 // function objectives executed
}

class MyValidationClass {
 // class definitions and methods
}
?>
```

2.2 State of the Art

Testing dynamic web applications comes with a variety of challenges, and these challenges increase as the application grows. If tested by hand, the application must be checked on all execution paths. An execution path starts at the very beginning of the application, and branches (like a tree) down through all possible ways in which a user can interact with the application. These branches define the input size for the application, which can be very large, and, as a result, the input that creates any type of fault can be difficult to find. Furthermore, generating the inputs for an application is an equally difficult problem.

The latter of the two problems has been addressed by previous work done by Benedikt et. al [2], who created Veriweb. Veriweb is a tool that automatically

discovers and systematically explores web-site execution paths comparable to how a developer would completely hand-test a dynamic application. However, this tool only checks for errors at the HTTP level. DART [8], Cute [6] , and EXE [3] all find failures by starting tests on applications with concrete values as input. To derive the subsequent inputs, their tools solve a set of path constraints found from exercising the control flow paths. However, their results were not practical in the web application domain, and they only used static input to into their tool.

The problem of minimizing the size of input derived from path constraints (such as the work aforementioned) has been addressed by Arzi et. al. [1]. These authors took the work above, combined it with checking the output of PHP applications for malformed HTML and crashes due to exceptions, and took the intersection of the solutions to path constraints in order to minimize the size of the input set that could have generated a fault. They were able to defend that their tool, Apollo, was both effective and efficient at finding faults in dynmaic web applications. They based their fault finding heavily on the HTML that was output from running applications in their tool, and used this as an indicator of a faulty web application. Apollo was able to find 214 faults across 4 web applications. The Apollo tool also contained construct, internal/external, and reliability concerns.

WebKing2, proposed by Copee et. al. [4] takes the debugging to the developer level by providing options for white-box, black-box, and regression testing across the entire site, including JavaScripts, CGI scripts, and CSS errors. WebKing correctly identifies numerous types of errors, including deliberate errors made in several JavaScripts, testing the critical paths an application takes through the site. However, this work may not cover all paths, does not attempt to minimize the input set that may have possible created the fault, and the authors seeded errors into the application for thier tool to find (makeing their tool hard to generalize for real faults).

Wu et. al. [13] has presented a newly-developed theoretical model of new couplings and the dynamic flow of control of Web applications, which captures dynamic aspects of Web applications and develops and defines test criteria. Related to this concept is work done on dynamic test case generation [12] and analyzing the data flow of dynamic, JSP-based web applications to compute intraprocedural, interprocedural, and sessional data flow test paths for uncovering the data anomalies of JSP pages [7]. We would like to explore these works in building our tool.

Although Artzi et. al.'s work did combine previous work and make it applicable to the PHP scripting language, their work did not check the functionality of the web application for correctness and intended results. Our work would build upon the foundation they have laid and add this notion of functionality checking. In order to do that, we will need to create an oracle to determine if the PHP web application behaved as it should have during the execution. Another aspect that we are looking to incorporate will solve a general problem in testing PHP applications which is PHP code "hidden" behind JavaScript. In addition, we would like to consider using more than just static inputs as initial concrete inputs for testing applications with our tool. We would like to explore session-awareness [10, 11, 5] to produce inputs to our target applications (since we are testing functional correctness of the applications).

2.3 Limitations

Since we plan to make significant contributions and additions to existing work, we will be heavily monitoring the runtime of our tool. We will implement all of our features as optional, meaning we can turn them on and off. Doing this will allow us to monitor all combinanations of features for cost effectiveness. Then we will perform cost-benefit analysis to determine whether the additional overhead of adding a feature is outweighed by the benefit. If our features prove to have a high-runtime and be inefficient, additional limitations in our tool may arise. If the feature is effective, even in the light of its inefficiency, we will re-work the feature to be more efficient.

3 Challenges and Goals

The following are some of the challenges we may face and goals that we have for this work:

1. Is there a way besides HTML checking to validate correct PHP output? HTML validation checks for syntax more than intended output semantics. Part of web appliction testing isn't only, "'Does the output 'look' correct"'? It's also, "'Is the ouput correct?"'

2. Is there a way to perform testing on output resulting from running the JavaScript produced by the PHP script? JavaScript may be written as a result of certain PHP code, but may not be reflected in the testing. (e.g. JavaScript my produce a button or action that is not tested/executed but only "'written"' to the output). Not taking the execution of processes further embedded into the execution path of the real, running application is not truly testing the entire application.

3. Is there a way to combine client session data, data path analysis, and predetermined seeded input into one input suite? An ideal situation for input data into a web application is to mimic the actions of a real user behind a real web browser while he/she is pointing, clicking, and entering data (correct or incorrect) into the web application. The challenge here is integrating web browser, on-line functionality, session-awareness, and server integration into our tool. Additionally, do we take Artzi et. al.'s [1] approach in mimicking user interaction with switch statements or take an approach where we deterimine the appropriate interaction by doiong a "webcrawl" through the application's interface.

4. Dynamic applications are verified via malformed HMTL. This is great for syntax, but what about semantics? The tool will incorporate HMTL verification methods used in the other related works, but can we also take existing tools for unit testing PHP and incorporate them into our tool? In this way PHPUnit would be used as an initial oracle. The challenge is that we need to decide whether the application developer would write the PHPUnit tests (which would make the developer the oracle) or find a way we can generate these PHPUnit tests automatically?

4 Proposed Research

4.1 Tool Building Goals

We propose to build our tool with a foundation that implements previous work. More specifically, we will implement the tool with input generation, input minimization, and static HTML result verification of correctness of dynamic PHP applications by basing the foundation of our tool on work done by Artzi et. al. [1]. This is straighforward, per se. This reproduces the results of previous work in PHP web application testing. Further, we will create an extension for automatic PHPunit test case generation and execution. This will examine approaches to test the functional correctness of the PHP/HTML output (asserts, PHPunit). This will also help with the semantic checking of the output of some user's session with an application. The approach would examine the actual code and determine what "'should"' happen functionally. This PHPunit generation may add overhead to the tool, but will be a feature that is optional on each execution of the tool. (We will include the overhead vs. efficiency gain analysis as part of our evaluation.) Finally, we will implement an extension of the tool to run any JavaScript and expose any hidden PHP that is not evaluated in other tools. Doing this will reveal any other faulty code and increase the coverage of our tool. Optimally, we will get greater coverage than previous PHP testing tools.

4.2 Testing Goals

Optimally, we would like to test our tools by developing input data with the goals of achieving the highest-fidelity and minimizing the threats to validty during our testing. We will aspire to eliminate internal and external threats to validity by incorporating multiple views. As mentioned before, we will use Artzi et. al.

as a foundation. However, our testing strategy will attempt to completely eliminate the use of seeded-input into the application during initial phases of testing by implementing the work on generating test suites derived by using user session data [5, 11, 10]. We will use a suite of user session data as our initial inputs.

Our testing method will also incorporate UML-level test-input generation for high level testing [9]. This can be a "'tier 1"' form of early-and-often testing. Next, we will seek to incorporate data flow testing techniques of Liu et. al. [7] in a PHP setting. This is a more fine grained strategy that can be combined with the first input generation technique (using user-session data as input) to produce high quality input.

We will perform proof of concept testing by matching the results of running our tool against the same PHP applications as did Artzi et. al. [1] in their work (faqforge, webchess, schoolmate, and phpsysinfo). This is to make certain that our tool can perform just as good as these authors' tool. Then we turn on the new features - PHPunit generation, JavaScript Explosion, UML-level testing, Data-flow testing - with the goal of verifying that our results are more finely-grained and more valid without too much overhead. Lastly, to demonstrate practicality, we will follow the proof of concept testing with industry-testing of our tool against industry-level applications. We will accomplish this by gathering a group of programmers from novice to experienced, freshmen undergraduate to veteran web developers, and have them use our tool when building and testing their web applications.

As one could observe, our tool-building phase innovatively interleaves natively autonomous testing strategies with solutions to overcome limitations experienced in previous work. Our goal is to use experience as a teaching strategy to help us implement a more effective soluion, not only in verifying the syntactic correctness of code, but also in verifying that the semantics of the program are correct in that application functionally performs as intended.

Another observation is the interconnection of multiple test-input-generation strategies into our input generation scheme. The goal is to achieve an optimal suite of inputs that eliminates as many internal threats to validity as possible. This will enable us to more concretely verify the validity of our approach and our tool.

5 Evaluation Approach

We will ask the following questions in evaluating the performance, efficiency, and practicality of our tool:

- Does our tool reproduce the results of equivalent Java technology based tools? We will evaluate this by comparing our results against tools that perform automated testing of Java-based web applications.

- How many faults does out tool find? We will use straight-forward counting methods to determine the amount of faults found.

- How effective is our tool at localizing the faults? We will evaluate this by running our tool on the same applications that Artzi. et. al. [1] ran their tool on and prove our effectiveness by comparing the sets of results. These applications include faqforge, webchess, schoolmate, and phpsysinfo (all openly available).

- How is the coverage of our test geneartion tool? We will evaluate this by calculating the ratio of the number of PHP statements experienced and total number of PHP statements in the application. We aim to prove that exploding the JavaScript will enhance the coverage of our tool.

- Does adding PHPunit test case generation improve our fault finding? We will implement this feature as an optional feature that can be turned on and off. We will run the application with this feature on and repeat the run with the feature off. Then, we will compare our results of these two runs to determine the effectiveness in terms the number of faults found in the application in either run. We will also keep track of the overhead of running the tool with the feature on and off.

- Does running the JavaScripts improve the fault finding and localization? This will also be implemented as a feature that can be turned on and off. We will evaluate this much like the PHPunit feature. We will also measure for additional coverage due to exploding the JavaScript.
- Does dataflow add overhead to the application? Is it negligable or too much for practicality? We will implement this as an optional feature and run the similar evaluation as the previous two features on JavaScript explosion and PHPunit generation. The overhead will be increased, but is it signficant enough to leave out of the tool when considering its contribution to effectively finding faults.
- Does this improve testing on industry-level applications? We will test our tool on industry applications after rigorous testing of our tool on the different test suites as mentioned above. If we are able to have industry professionals use this tool successfully, we will have proven a practical and efficient tool.
- Is our tool practical?
- Does our tool perform just as well or outperform previous work with the additional features? By implementing the features as optional, we are able to turn them on and off and test all possible combinations of features in order to completely evaluate our tool. Our goal is not to introduce significant overhead when comparing against the effecitiveness of finding faults.
- What is the coverage of our tool across each individual application? With our features, namely the JavaScript explosion, we want to see if our coverage increases and if the increase leads to more bug finding.

6 Summary of Forseen Contributions

We plan to contribute the following to the state of the art:

- A tool that checks that the application is not only outputs syntactically correct HTML but also functionally behaves in the intended manner.
- A tool that uses automated PHPUnit generation as an oracle to determine correct functionality of the application.
- A tool that is both efficient and effective in finding semantic and syntactic bugs in PHP web applications.
- A tool that is practical and usable in the industry.
- A tool that expands JavaScript in the web application to explore more PHP code and get a higher code coverage.

References

[1] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 261–272, New York, NY, USA, 2008. ACM.

[2] Michael Benedikt, Juliana Freire, and Patrice Godefroid. Veriweb: Automatically testing dynamic web sites. In *11th International WWW Conference*, 2002. Bell Laboratories, Lucent Technologies.

[3] P M Pawlowski D L Dill D R Engler C Cadar, V Ganesh. Exe: Automatically generating inputs of death. *CCS*, 2006.

[4] Todd Coopee. Put dynamic web pages to the test. *InfoWorld*, 22, 2000.

[5] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *25 International Conference on Software Engineering*, volume 2003, pages 49–59, 2003.

[6] D Marinov G Agha K Sen. Cute: A concolic unit testing engine for c. *FSE*, 2005.

[7] Chien-Hung Liu. Data flow analysis and testing of jsp-based web applications. *Information and Software Tecnology*, 48, 2006.

[8] K Sen P Godefroid, N Klarlund. Dart: Directed automated random testing. *PLDI*, 2005.

[9] Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *23 International Conference on Software Engineering*, volume 2001, page 25, 2001.

[10] Sreedevi Sampath, Sara Sprenkle, Emily Gibson, Lori Pollock, and Amie Souter Greenwald. Applying concept analysis to user-session-based testing of web appications. In *IEEE Transactions on Software Engineering*, volume 33, 2007.

[11] Sebastian, Gregg Rothermel, Srikanth Karre, and Marc Fisher II. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, 2005.

[12] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 249–260, New York, NY, USA, 2008. ACM.

[13] X Du Y Wu, J Offutt. Modeling and testing of dynamic aspects of web applications. 2004. Submitted For Publication. GM U.