IJDCST @November Issue- V-1, I-7, SW-13
ISSN-2320-7884 (Online)
ISSN-2321-0257 (Print)

Automatic Test Generation in Dynamic Web Applications

[1] G.Navyasree, [2] P.Radha Krishna
[1] Student, Nova College of Engineering & Technology, Jupudi, Vijayawada ANDHRAPRADESH.
[3] Associate Prof, HOD, Dept of CSE, Nova College of Engineering & Technology, Jupudi, Vijayawada, ANDHRAPRADESH.

Abstract: Developing of dynamic software applications in basic web oriented applications is the main progression in present days. For developing these type of applications in two ways static analysis and dynamic analysis. In this requirement we develop above developed applications for generating test cases for finding faults in HTML and PHP applications in web oriented process. For this development of the dynamic applications traditionally development tool was Ochiai. In an auto test generation strategy, this approach is quite a boon to validate quality applications in time. Combined with source mapping and using an extended domain for conditional and function-call statements earlier an enhanced Ochiai, fault localization algorithms was proposed that has the ability to handle web applications as well. Currently the variant Ochiai driven oracles offers rigid support by offering static analysis services to only PHP applications. So we propose to extend existing approach for supporting dynamic dataset presentation in static code analysis. In this paper we propose to develop our proposed work for improving metamorphic relations for dynamic web applications development present in the present software applications. Our experimental results show efficient and dynamic test generation in dynamic web applications development.

Index Terms: Apollo, Ochiai, Tarantula, dynamic Web applications, fault localization, fault localization effectiveness, open-source PHP applications, path constraint, source mapping, statistical analysis, test generation strategie, Metamorphic Relations.

I. INTRODUCTION

Web applications are typically written by several combinational languages such as JAVA SCRIPT, Client Side application development like HTML. Furthermore application development can be done by the PHP application development features for

dynamic application development. The failures in the HTML code may be difficult to localize because HTML code is often dynamically generated by server- side code. For example in java and PHP really there is no html file or line number of point developer to detect the when and where the failure occurs. We present Apollo, the first fully automatic tool that efficiently finds and localizes malformed HTML and execution failures in web applications that execute PHP code on the server side. In previous work we created a tool called Apollo that implements our technique for PHP.

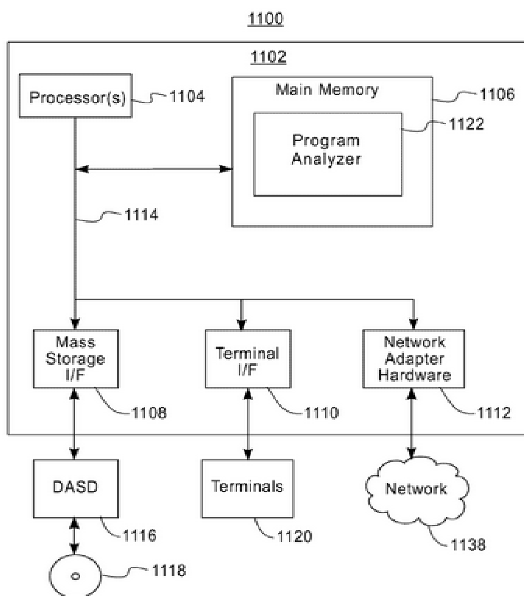

Figure 1: Faults Localization in Dynamic Applications.

By exercising the control flow in the each execution, faults that are observed during the execution are recorded. Until to achieve the sufficient coverage of the statements in the application the process is repeated. But the time budget is exhausted or sufficient number of faults has been detected. In particular,

1) It integrates an HTML validator to check for failures that manifest themselves by the generation of malformed HTML.

2) It automatically simulates interactive user input.

3) It keeps track of the interactive session state that is shared between multiple PHP scripts.

The output of a Web application is typically an HTML page that can be displayed in a browser. To find faults that are manifested as Web application crashes or as malformed HTML. The application may terminate some times, such as when a Web application calls an undefined function or reads a nonexistent file. However, our previous work focused exclusively on finding failures by identifying inputs that cause an application to crash or produce malformed HTML. We cannot address the problem of the pinpointing the web application failure. Hence, fixing the underlying faults can be very difficult and time consuming if no information is

available about where they are located. In our scheme we addressing the problem of determining where in the source code changes need to be made in order to fix the detected failures we commonly referred as the *fault localization.* The fault-localization algorithms explored in this paper attempt to predict the location of a fault based on a statistical analysis of the correlation between passing and failing tests and the program constructs executed by these tests. We investigate variations on three popular statistical fault-localization algorithms, known as Tarantula, Ochiai, and Jaccard. A suspiciousness rating is computed for each executed statement, the percentages of passing and failing tests that execute. Programmers are encouraged to examine the executed statements in order of decreasing suspiciousness. In Ochiai tool we have to implementing metamorphic relations present in the dynamic application development.

II. RELATED WORK

2.1. PHP WEB APPLICATIONS

PHP is a widely used scripting language for implementing web applications, in part due to its rich library support for network interaction. A typical PHP web application is a client/server program in which data and control flow interactively between a server, which runs PHP scripts, and a client, which is a web browser.

2.2. The PHP Scripting Language

PHP is object oriented, in the sense that it has classes, interfaces, and dynamically dispatched methods. PHP also has features of scripting languages, such as dynamic typing. For example, the following code fragment: $code =" $x = 3 ;";$x= 7; eval ($code); echo $x; prints the value 3.

The PHP code is delimited by <?php and ?> tokens. The use of HTML in the middle of PHP indicates that HTML is generated as if it occurred in a print statement. The require statements resemble the C #include directive by causing the inclusion of code from another source file. The degree of flexibility is prized by PHP developers for enabling rapid application prototyping and development. The flexibility can make the overall structure of program hard to discern and render programs prone to code-quality problems that are difficult to localize.

ISSN-2320-7884 (Online)
ISSN-2321-0257 (Print)

2.3 Failures in PHP Programs

Generally we target on the two types of the failures that may occur in the time of the execution. Execution failures : These are caused by missing included files and uncaught exceptions. Less serious execution failures, such as those caused by the use of deprecated language constructs and incorrect SQL queries, produce obtrusive error messages but do not halt execution. HTML failures. These involve situations in which generated HTML code is not syntactically correct, causing them to be rendered incorrectly in certain browsers. This may not only lead to portability problems.

2.4 Fault Localization

Early work on fault localization relied on the use of program slicing. Lyle and Weiser introduce program dicing, a method for combining the information of different program slices. Detecting failures only demonstrates that a fault exists. The next step is to find the location of the fault that causes each failure. Some information might help for the further use. For HTML failures, HTML validators provide the problematic locations in the HTML code. Malformed HTML fragments can then be correlated with the portions of the PHP scripts that produced them. Comparing that set of statements with the set of statements executed by the failing runs can then provide clues that can help localizing the fault.

III. BACK GROUND WORK

In Apollo, we implemented a shadow interpreter based on the Zend PHP interpreter that simultaneously performs concrete program execution using concrete values. Apollo uses the choco constraint solver to solve path constraints during the combined concrete and symbolic test generation. We implemented the following extensions to the shadow Interpreter to support fault localization:

- ***Statement Coverage :***
  Shadow interpreter records the set of executed statements for each execution by hooking into the zend_execute and compile file methods.

- ***Html validator :***
  HTML validators as an oracle for checking HTML output: the Web Design Group (WDG) HTML validator and the CSE HTML Validator V9.0.

- ***Source mapping:***
  The source-mapping correlates a fault found in the HTML output with the statements producing the erroneous output fragments.

- ***Condition modeling :***
  Our shadow interpreter records the results of all comparisons in the executed PHP script for the conditional modeling technique. For each comparison it records a pair consisting of the statement's line number and the relevant Boolean result.

- ***Return-value modeling :***
  The shadow interpreter stores the line number of the call and an abstract model of the value. The fault localization technique to distinguish between null and non-null values.

IV. FAULT LOCALIZATION

First we can see the basic fault localization algorithms and there implementations and there extensions. We then present an alternative technique that is based on source mapping and positional information obtained from an oracle. The se of an extended domain for conditional and return-value expressions can help improve the basic algorithm's effectiveness.

3.1 Fault Localization Algorithms

Tarantula presents a fault-localization technique that associates with each statement a suspiciousness rating that indicates the likelihood for that statement to contribute to a failure. suspiciousness rating $S_{tar}(l)$ for a statement that occurs.

$S_{tar}(l)=$

$$\frac{Failed\ (l)/Total\ Failed}{Passed\ (l)/Total\ passed+Failed\ (l)/Total\ Failed}$$

where Passed is the number of passing executions that execute statement l, Failed(l) is the number of failing executions that execute statement l, TotalP assed is the total number of passing test cases, and Total Failed is the total number of failing test cases. The coefficients have been proposed that can also be used to calculate Suspiciousness ratings. The jacard Coefficient has been used

$$S_{jac}(l) = \frac{Failed\ (l)}{Total\ Failed+Passed\ (l)}$$

the Ochiai coefficient used in the molecular biology domain:

$$S_{och}(l) = \frac{Failed\ (l)}{\sqrt{Failed\ (l) * (Passed\ (l) + Failed(l))}}$$

After suspiciousness ratings have been computed Ranks need not be unique: The rank of each statement is the number of statements with greater than or equal suspiciousness:

$$rank(l) = |l' : S(l') >= S(l)|.$$

In this we computed for each failing test run a score that indicates the percentage of the application's executable statements that the programmer need not examine in order to find the fault. The score is now computed by dividing the number of statements in the set by the total number of executed statements. Using this 13.9 percent of the failing test runs were scored in the 99-100 percent range.

V. PROPOSED APPROACH

We propose to extend the Ochiai algorithm with Metamorphic testing strategies to develop an integrated framework that can offer support beyond PHP such as Java/HTML/JavaScript. Instead of employing any oracles for initiating testing we propose to implement metamorphic testing. Metamorphic testing is a technique for the verification of software output without a complete testing oracle. Metamorphic testing observes that even if the executions do not result in failures, they still bear useful information. Follow-up test cases should be constructed from the original set of test cases with reference to selected necessary properties of the specified function. Such necessary properties of the function are called metamorphic relations. The subject program is verified through metamorphic relations (MR). It is unlikely for a single MR to detect all possible faults. Therefore, four MRs that are quite different from one another with a view to detecting various faults were used here. Finding good MRs requires knowledge of the problem domain, understanding of user requirements, as well as some creativity. These MRs are identified according to equivalence and nonequivalence relations among regular expressions. So this kind of testing facilitates in an automated addressing of all possible forms of failures in most web technologies. Our work differs from most previous research on fault localization in that it does not assume the existence of a test suite with passing and failing test cases. Our previous work focused exclusively on finding failures by identifying inputs that

ISSN-2320-7884 (Online)
ISSN-2321-0257 (Print)

cause an application to crash or produce malformed HTML. This paper addresses the problem of determining where in the source code changes need to be made in order to fix the detected failures. We introduce program dicing, a method for combining the information of different program slices. The idea behind the schema is when a program computes a correct value for variable x and an incorrect value for variable y, the fault is likely to be found in statements that are in the slice w.r.t. y, but not in the slice w.r.t. x. Variations. Use of set-union, set-intersection, and nearest neighbor methods for fault localization; these all work by comparing execution traces of passing and failing program runs.

VI. EXPERIMENTAL RESULTS

Our approach is based on the concept of metamorphic testing (Chen et al., 1998), summarized below. To facilitate that approach, we must identify the relations that the algorithms are expected to exhibit between sets of inputs and sets of outputs.

6.1. Metamorphic Relations

We define the MRs that we anticipate classification algorithms to exhibit, and define them more formally as follows.

MR-0: Consistence with affine transformation. The result should be the same if we apply the same arbitrary affine transformation function, f(x) = kx + b, (k 6= 0) to every value x to any subset of features in the training data set S and the test case ts.

MR-1.1: Permutation of class labels. Assume that we have a class-label permutation function Perm() to perform one-to-one mapping between a class label in the set of labels L to another label in L. If the source case result is li, applying the permutation function to the set of corresponding class labels C for the follow-up case, the result of the follow-up case should be Perm(li).

MR-1.2: Permutation of the attribute. If we permute the m attributes of all the samples and the test data, the result should remain unchanged.

VII. CONCLUSION

Currently the variant Ochiai driven oracles offers rigid support by offering static analysis services to only PHP applications. So we propose to extend existing approach

for supporting dynamic dataset presentation in static code analysis. In this paper we propose to develop our proposed work for improving metamorphic relations for dynamic web applications development present in the present software applications. Our experimental results show efficient and dynamic test generation in dynamic web applications development. As further improvement of our proposed work can efficiently accessing metamorphic relations present in testing strategies.

VIII. REFERENCES

[1] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia ," Fault Localization for Dynamic Web Applications" IEEE Transactions on Ssoftware Engineering, vol. 38, no. 2, pp. 314-334, march/april 2012.

[2] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M.D.Ernst, "Finding Bugs in Dynamic Web Applications," Proc. Int'lSymp. Software Testing and Analysis, pp. 261-272, 2008.

[3] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M.D.Ernst, "Finding Bugs in Web Applications Using Dynamic TestGeneration and Explicit State Model Checking," IEEE Trans. Software Eng., vol. 36, no. 4 pp. 474-494, July/Aug. 2010.

[4]F. Tip, "A Survey of Program Slicing Techniques," J. Programming Languages, vol. 3, no. 3 pp. 121-189, 1995.

[5] X. Ren and B.G. Ryder, "Heuristic Ranking of Java Program Edits for Fault Localization," Proc. ACM/SIGSOFT Int'l Symp. Software Testing and Analysis, D.S. Rosenblum and S.G. Elbaum, eds., pp. 239-249, 2007.

[6] J. Lyle and M. Weiser, "Automatic Bug Location by Program Slicing," Proc. Second Int'l Conf. Computers and Applications, pp. 877-883, 1987.

[7] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," Proc. European Software Eng. Conf. Held Jointly with 13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng., 2005.

[8] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler, "EXE: Automatically Generating Inputs of Death," Proc. Conf. Computer and Comm. Security, 2006.

[9] P. Godefroid, M.Y. Levin, and D. Molnar, "Automated Whitebox Fuzz Testing," Proc. Symp. Network and Distributed System Security, 2008

[10] J.A. Jones, M.J. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization," Proc. Int'l Conf. Software Eng., pp. 467-477, 2002.