International Journal of Computer and Communication Technology

Volume 8 | Issue 1 Article 7

January 2017

Failure Detection Algorithm for Testing Dynamic Web Applications

J. Vijaya Sagar Reddy
Department of CSE, JNTUA College of Engineering, Anantapur, Andhra Pradesh, India,
vsreddyj5@gmail.com

G. Ramesh
Department of CSE, JNTUA College of Engineering, Anantapur, Andhra Pradesh, India,
ramesh680@gmail.com

Failure Detection Algorithm for Testing Dynamic Web Applications

J. Vijaya Sagar Reddy & G. Ramesh

Department of CSE, JNTUA College of Engineering, Anantapur, Andhra Pradesh, India
E-mail: vsreddyj5@gmail.com, ramesh680@gmail.com

Abstract **-** Web applications are the most widely used software in the internet. When a web application is developed and deployed in the real environment, It is very severe if any bug found by the attacker or the customer or the owner of the web application. It is the very important to do the proper pre-analysis testing before the release. It is very costly thing if the proper testing of web application is not done at the development location and any bug found at the customer location. For web application testing the existing systems such as DART, Cute and EXE are available. These tools generate test cases by executing the web application on concrete user inputs. These tools are best suitable for testing static web sites and are not suitable for dynamic web applications. The existing systems needs user inputs for generating the test cases. It is most difficult thing for the human being to provide dynamic inputs for all the possible cases. This paper presents algorithms and implementation, and an experimental evaluation that revealed HTML Failures, Execution Failures, Includes in PHP Web applications.

Keywords **-** *Reliability, Verification, Software Testing, Web Applications, Dynamic Analysis, PHP.*

I. INTRODUCTION

Dynamic test-generation tools, such as DART, Cute and EXE, find failures by executing an application on concrete input values, and then creating additional input values by solving symbolic constraints derived from exercised control flow paths. To date, such approaches have not been practical in the domain of Web applications. This paper extends dynamic test generation to scripting languages, uses an oracle to determine whether the output of theWeb application is syntactically correct, and automatically sorts and minimizes the inputs that expose failures. Our Apollo system applies these techniques in the context of PHP, one of the most popular languages for Web programming. According to Netcraft, PHP powered 21 million domains as of April 2007, including large, well-known websites such as Wikipedia and WordPress.

The output of a Web application is typically an HTML page that can be displayed in a browser. Our goal is to find faults that are manifested as Web application crashes or as malformed HTML. Some faults may terminate the application, such as when a Web application calls an undefined function or reads a nonexistent file. In such cases, the HTML output presents an error message and the application execution is halted.

More commonly in deployed applications, a Web application creates output that is not syntactically well-formed HTML, for example by generating an opening tag without a matching closing tag. Web browsers are designed to tolerate some degree of malformedness in HTML, but this merely masks underlying failures. Malformed HTML is less portable across browsers and is vulnerable to breaking on new browser releases. An application that creates invalid (but displayable) HTML during testing may create undisplayable HTML on different executions. More seriously, browsers' attempts to compensate for malformed Web pages may lead to crashes and security vulnerabilities1. A browser might also succeed in displaying only part of a malformed webpage, silently discarding important information. Search engines may have trouble indexing incorrect pages. Standard HTML renders on more browsers, and valid pages are more likely to look as expected, including on future versions of Web-browsers. Standard HTML renders faster. For example, in Mozilla, "improper tag nesting triggers residual style handling to try to produce the expected visual result, which can be very expensive"

Web developers widely recognize the importance of creating legal HTML. Many websites are validated using HTML validators. However, HTML validators are used only for static pages.

Validating *dynamic* Web applications (i.e., application that generate pages during the execution) is hard. Even professionally developed applications often contain multiple faults. To prevent faults, programmers must make sure that the application creates a valid HTML page on *every* possible execution path. There are

two general approaches to this problem: static and dynamic checking (testing).

Static checking of dynamic Web applications cannot fully capture their behavior. Such application are often written in languages such as PHP that enable on-the-fly creation of code and overriding of methods. In many Web applications, part (further pages) of the application is referenced from the generated HTML text (e.g., buttons and menus that require user interaction to execute), rather than from the analyzed code. Specialized analysis may be possible for a custom language.

Testing of dynamicWeb applications is also hard, because the input space is large, and application usually require multiple user interactions. The state-of-the-practice in validation for Web-standard compliance of real Webapplications is using programs such as HTML Kit4 that validate each generated page, but require manual generation of inputs that lead to displaying different pages. We know of no automated validator for scripting languages that dynamically generate HTML pages.

This paper presents an automated technique for finding failures in HTMLgenerating Web applications. Our technique adapts the technique of dynamic test generation, based on combined concrete and symbolic execution and constraint solving, to the domain of Web applications. In our technique, the Web application under test is first executed with an empty input. During each execution, the program is monitored to record path constraints that capture the outcome of control-flow predicates. Additionally, for each execution an oracle determines whether fatal failures or HTML well-formedness failures occur, the latter via use of an HTML validator. The system automatically and iteratively creates new inputs by negating one of the observed constraints and solving the modified constraint system. Each newly-created input explores at least one additional control flow path.

In summary, the contributions of this paper are:

- We adapt the established technique of dynamic test generation, based on combined concrete and symbolic execution, to the domain of Web applications. The challenges include inferring the input parameters, which are not indicated by the source code; using an HTML verifier as an oracle; dealing with language-specific datatypes and operations; and simulating user input for interactive applications.
- We evaluated our tool by applying it to real Web applications and comparing the results with random testing. We show that dynamic test generation is highly effective when adapted to the domain of Web applications written in PHP.

II. BACKGROUND

Finding Failures in PHPWeb Applications

Our technique for finding failures in PHP applications is a variation on an established dynamic test generation technique sometimes referred to as concolic testing. The basic idea is to execute an application on an initial input (e.g., an arbitrarily or randomly-chosen input), and then on additional inputs obtained.

```
parameters: Program P, oracle O
result      : Bug reports B;
B : setOf(⟨failure, setOf(pathConstraint), setOf(input)⟩)
1  P' := simulateUserInput(P);
2  B := ∅;
3  pcQueue := emptyQueue();
4  enqueue(pcQueue, emptyPathConstraint());
5  while not empty(pcQueue) and not timeExpired() do
6      pathConstraint := dequeue(pcQueue);
7      input := solve(pathConstraint);
8      if input ≠ ⊥ then
9          output := executeConcrete(P', input);
10         failures := getFailures(O, output);
11         foreach f in failures do
12             merge ⟨f, pathConstraint, input⟩ into B;
13         c1 ∧ ... ∧ cn := executeSymbolic(P', input);
14         foreach i = 1,...,n do
15             newPC := c1 ∧ ... ∧ ci−1 ∧ ¬ci;
16             enqueue(pcQueue, newPC);
17 return B;
```

Figure 1: The failure detection algorithm. The *solve* auxiliary function uses the constraint solver to find an input satisfying the path constraint, or returns □ if no satisfying input exists. The output of the algorithm is a set of bug reports. Each bug report contains a failure, a set of path constraints exposing the failure, and a set of input exposing the failure.

Algorithm

Figure 1 shows the pseudo-code of our algorithm. The inputs to the algorithm are: a program P and an output oracle O. The output of the algorithm is a set of bug reports B for the program P, according to O. Each bug report contains: identifying information about the failure, the set of all inputs under which the failure was exposed, and the set of all path constraints that lead to the inputs exposing the failure.

The algorithm uses a queue of path constraints. A *path constraint* is a conjunction of conditions on the program's input parameters. The queue is initialized with the empty path constraint (line 4). The algorithm uses a constraint solver to find a concrete input that

satisfies a path constraint taken from the queue (lines 6–7). The program is executed concretely on the input and tested for failures (lines 9–10). The path constraint and input for each detected failure are merged into the corresponding bug report (lines 11–12). Next, the program is executed symbolically on the same input (line 13). The result of symbolic execution is a path constraint, V*ni*=1 *ci*, that is fulfilled if the given path is executed (here, the path constraint reflects the path that was just executed). The algorithm then creates new test inputs by solving modified versions of the path constraint (lines 14–16), as follows. For each prefix of the path constraint, the algorithm negates the last conjunct (line 15). A solution, if it exists, to such an alternative path constraint corresponds to an input that will execute the program along a prefix of the original execution path, and then take the opposite branch.

Path Constraint Minimization

```
   parameters: Program P, oracle O, bug report b
   result    : Short path constraint that exposes b.failure
 1 c1 ∧ ... ∧ cn := intersect(b.pathConstraints);
 2 pc := true;
 3 foreach i = 1,...,n do
 4     pci := c1 ∧ ... ci−1 ∧ ci+1 ∧ ... cn;
 5     input := solve(pci);
 6     if input ≠ ⊥ then
 7         output := executeConcrete(P, input);
 8         failures := getFailures(O, output);
 9         if b.failure ∉ failures then
10             pc := pc ∧ ci;
11 inputpc := solve(pc);
12 if inputpc ≠ ⊥ then
13     outputpc := executeConcrete(P, inputpc);
14     failurespc := getFailures(O, outputpc);
15     if b.failure ∈ failurespc then
16         return pc;
17 return shortest(b.pathConstraints);
```

Figure 2: The path constraint minimization algorithm. The method *intersect* returns a conjunction containing the conditions that are present in all given path constraints, and the method *shortest* returns the path constraint with fewest conjuncts. The other auxiliary functions are the same as in Figure 1.

The failure detection algorithm (Figure 1) returns bug reports for different failures. Each bug report contains a set of path constraints leading to inputs exposing the failure. Previous dynamic test generation tools presented the whole input to the user without an indication of the subset of the input responsible for the failure. As a postmortem phase, our minimizing algorithm attempts to find a shorter path constraint for a given bug report(Figure 3). This eliminates irrelevant constraints, and a solution for a shorter path constraint is often a smaller input.

For a given bug report *b*, the algorithm first intersects all the path constraints exposing *b.failure* (line 1). The minimizer systematically removes one condition at a time (lines 3-10). If one of these shorter path constraints does not expose *b.failure*, then the removed condition is required for exposing *b.failure*. The final path constraint is the conjunction of all such required conditions. From the minimized path constraint, the algorithm produces a concrete input that exposes the failure.

The algorithm in Figure 3 does not guarantee that the returned path constraint is the shortest possible that exposes the failure. However, the algorithm is simple, fast, and effective in practice.

Each failure might be encountered along several execution paths that might partially overlap. Without any information about the properties of the inputs, delta debugging minimizes only a *single* input at a time, while our algorithm handles *multiple* path constraints that lead to a failure.

III. IMPLIMENTATION

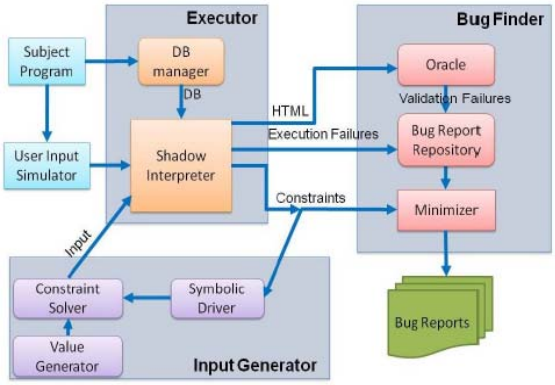

Fig. 3 : The architecture

We Implemented technique for PHP. It consists of three major components, **Executor**, **Input Generator**, and **Bug Finder**, illustrated in Figure 3. This section first provides a high-level overview of the components and then discusses the pragmatics of the implementation. The **User Input Simulator** component performs a transformation of the program that models interactive user input. The **Executor** is responsible for executing a given PHP file with a given input. Before each execution, the executor creates the appropriate database for the application.

The executor contains two sub-components:

- The **Shadow Interpreter** is a PHP interpreter that we have modified to record path constraints and positional information associated with output.

- The **DatabaseManager** initializes the database used by a PHP application, and restores it before each execution.

The **Bug Finder** uses an oracle to find HTML failures, stores the all bug reports, and finds the minimal conditions on the input parameters for each bug report.

The Bug Finder has the following sub-components:

- The **Oracle** finds HTML failures in the output of the program.
- The **Bug Report Repository** stores all bug reports containing execution failures and HTML failures found during all executions.
- The **Input Minimizer** finds, for a given bug report, the smallest path constraint on the input parameters that results in inputs inducing the same failure as in the report.
- The **Input Generator** contains the implementation of the algorithm described. The Input Generator contains the following sub-components:
- The **Symbolic Driver** generates new path constraints, and selects the next path constraint to solve for each execution.
- The **Constraint Solver** computes an assignment of values to input parameters that satisfies a given path constraint.
- The **Value Generator** generates values for parameters that are not otherwise constrained, using a combination of random value generation, and constant values mined from the program source code.

Evaluation

We experimentally measured the effectiveness of our technique in finding faults in PHP Web applications. We designed the experiments to answer the following research questions:

Q1. How many faults can Apollo find, and of what varieties?

Q2. How effective is the fault localization technique compared to alternative approaches such as randomized testing, in terms of the number and severity of discovered faults and the line coverage achieved?

Q3. How effective is our minimization in reducing the size of inputs parameter constraints and failure-inducing inputs?

IV. METHODOLOGY

We ran each test input generation strategy for 10 minutes on each subject program. The time limitwas chosen arbitrarily, but it allows each strategy to generate hundreds of inputs and we have no reason to think the results would be much affected by a different time limit. This time budget includes all experimental tasks, i.e., program execution, harvesting of constant values from program source, test generation, constraint solving (where applicable), output validation via oracle, and line coverage measurement. To avoid bias, we ran both strategies inside the same experimental harness. This includes the Database Manager, that supplies user names and passwords for database access. For our experiments, we use the WDG offline HTML validator, version 1.2.2.

We measured line coverage, i.e., the ratio of the number of executed lines to the total number of lines with executable PHP code in the application. We statically computed the total number of executable PHP lines in the subject programs by counting, in the interpreter, the number of lines with PHP opcodes.

We classify the discovered faults into five groups based on their different failure characteristics as follows:

execution crash : PHP interpreter terminates with an exception.

execution error : PHP interpreter emits a warning visible in the generated HTML.

execution warning : PHP interpreter emits a warning invisible in the generated HTML.

HTML error: program generates HTML for which the validator produces an error report.

HTML warning : program generates HTML for which the validator produces a warning report.

V. CONCLUSION

This paper implementation describes failure detection algorithm which can work on dynamic web applications. It can generate dynamic test cases for the dynamic web applications (PHP).The approach focus on Server-Side PHP-Code and some of client-side through web forms. The approach aims to identify Includes and two kinds of failures of web applications like Execution failures and HTML failures with the three modules called executor, bug tracker and input generator.

REFERENCES

[1] “Finding Bugs in Web Applications Using Dynamic Test Generation and Explicite-State Model checking” Shay Artzi, Adam Kie _zun,

Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, Senior Member, IEEE, and Michael D. Ernst.

[2] Y. Minamide, "Static Approximation of Dynamically Generated Web Pages," Proc. Int'l Conf. World Wide Web 2005.

[3] W.G. Halfond, S. Anand, and A. Orso, "Precise Interface Identification to Improve Testing and Analysis of Web Applications," Proc. Int'l Symp. Software Testing and Analysis, 2009.

[4] W.G.J. Halfond and A. Orso, "Improving Test Case Generation for Web Applications Using Automated Interface Discovery," Proc. Joint Meeting European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng., pp. 145-154, 2007.

❑❑❑