World Applied Sciences Journal 29 (Computer Sciences, Engineering and Its Applications): 259-263, 2014
ISSN 1818-4952
© IDOSI Publications, 2014
DOI: 10.5829/idosi.wasj.2014.29.csea.2249

# Tracing the Bugs in Dynamic Web Application Using Failure Detection Algorithmand Explicit-State Model Checking

*K.P.Thooyamani, R. Udayakumar and V. Khanaa*

Professor School of Computing Science, Bharath University, Chennai-73 India

**Abtract:** Bugs are errors, failures and malformed web pages that affect the usability of web applications. The present technique is used to validate the static web pages and not used for validating dynamic web page.we introduce a technique for dynamic web application.This technique utilizes both failure detection algorithm and explicit state model checking.This technique test automatically,runs the test capturing logical constraints on inputs and minimizes the conditions on input to failing test so that resulting bug reports are small and useful in finding and fixing the underlying faults. We use apollo architecture for finding fault and validates the output conforms to html specification.

**Key words:** Markup Language · Static · Dynamic · DART · CUTC

## INTRODUCTION

Dynamic test generation tools such as DART,CUTE 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, which pose special challenges due to the dynamism of the programming languages, the use of implicit input parameters, their use of persistent state and their complex patterns of user interaction. This paper extends dynamic test generation to domain of web applications that dynamically create web (HTML) user in the browser.

Our goal is to find two kinds of failures in web applications: execution failures that are manifested as crashes or warnings during program execution and HTML failures that occur when the application generates mal-formed HTML. Execution failures may occur, for example, when a web application calls an undefined function or reads a nonexistent file. In such cases, the HTML output contains an error message and execution of the application may be halted, depending on the severity of the failure. HTML failures occur when output is generated that is not syntactically well-formed HTML (e.g., when an opening tag is not accompanied by a matching closing tag). HTML failures are generally because Web browsers are designed to tolerate some degree of malformedness in HTML, but they are undesirable for several reasons. First and most serious is that browsers' attempts to compensate for malformed webpages may lead to crashes and security vulnerabilities.2 Second, standard HTML renders faster.3 Third, malformed HTML is less portable across browsers and is vulnerable to breaking or looking strange when displayed by browser versions on which it is not tested. Fourth, a browser might succeed in displaying only part of a malformed webpage, while silently discarding important information. Fifth, search engines may have trouble indexing malformed pages Web developers widely recognize the importance of creating legal HTML. Many websites are checked using HTML validators. However, HTML validators can only point out problems in HTML pages and are by themselves incapable of finding faults in applications that generate HTML pages. Checking dynamic Web applications (i.e., applications that generate pages during execution) requires checking that the application creates a valid HTML page on every possible execution path. In practice, even professionally developed and thoroughly tested applications often contain multiple faults There are two general approaches to finding faults in web applications: static analysis and dynamic analysis (testing). In the context of Web applications, static approaches have limited potential because Web

**Corresponding Author:** K.P.Thooyamani, School of Computing Science, Bharath University, Chennai-73 India.

applications are often written in dynamic scripting languages that enable on-the-fly creation of code and 2) control in a Web application typically flows via the generated HTML text (e.g., buttons and menus that require user interaction to execute), rather than solely via the analyzed code. Both of these issues pose significant challenges to approaches based on static analysis. Testing of dynamic Web applications is also challenging because the input space is large and applications typically require multiple user interactions. The state of the practice in validation for Web-standard compliance of real Web applications involves the use of programs such as HTML Kit that validate each generated page, but require manual generation of inputs that lead to displaying different pages. We know of no automated tool that automatically generates inputs that exercise different control-flow paths in a Web application and validates the dynamically generated HTML pages that the Web application generates when those executed.This paper presents an automated technique for finding failures in HTML-generating web applications. Our technique is based on dynamic test generation, using combined concrete and symbolic (concolic) execution and constraint solving We created a tool, Apollo, that implements our technique in the context of the publicly available interpreter.Apollo first executes the Web application under test with an empty input. During each execution, Apollo program to record path constraints that reflect how input values affect control flow. Additionally, for each execution, Apollo determines whether execution failures or HTML failures occur (for HTML failures, an HTML validator is used as an oracle). Apollo automatically and iteratively creates new inputs using the recorded path constraints to create inputs that exercise different control flow. Most previous approaches for concolic execution only detect "standard errors" such as crashes and assertion failures. Our approach detects such standard errors as well, but also uses an oracle to detect specification violations in the application's output. Another novelty in our work is the inference of input parameters, which are not manifested in the source code.

Techniques based on combined concrete and symbolic executions may create multiple inputs expose the same fault. In contrast to previous techniques, to avoid overwhelming the developer, our technique automatically identifies the minimal part of the input that is responsible for triggering the failure. This step is similar in spirit to Delta Debugging [3]. However, since Delta Debugging is a general, black box input minimization technique, it is oblivious to the properties of inputs. In contrast, our technique is white box: It uses the information that certain inputs induce partially overlapping control-flow paths. By intersecting these paths, our technique significantly minimizes the constraints on the inputs.The contributions of this paper are the following:We adapt the established technique of dynamic test generation, based on combined concrete and symbolic execution [1, 2] to the domain of Web applications. This involves:

- Using an HTML verifier as an oracle to find errors in dynamically generated HTML
- Dynamically discovering possible input parameters
- Dealing with data types and operations specific
- Tracking the use of persistent state and how input flows through it and
- Automatically discovering input values based on the examination of branch conditions on execution paths.

**Finding Failures in Web Applications:** Our technique for finding failures in PHP applications is a variation on an established dynamic test generation technique [1-3], sometimes referred to as concolic testing. For expository purposes, we will present the algorithm in two steps. First, this section presents a simplified version of the algorithm that does not simulate user inputs or keep track of persistent session state [4-6]. Then, Section 4 presents a generalized version of the algorithm that handles user-input simulation and stateful executions and illustrates it on a more complex example. The basic idea behind the technique is to execute an application on some initial input (e.g., an arbitrarily or randomly chosen input) and then on additional inputs obtained by solving constraints derived from exercised control-flow paths. We adapted this technique to Web applications as follows [7, 8].

**Algorithm:** Figure 1 shows pseudocode for our algorithm. The inputs to the algorithm are: a program P, an oracle for the output O and an initial state of the environment S0 . The output of the algorithm is a set of bug reports B for the program P, according to O. Each report consists of a single failure, defined by the error message and the set of statements that is related to the failure [9, 10]. In addition, the report contains 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 configurations. Each configuration is a pair of a path constraint and an input. A path constraint is a conjunction of conditions on the program's input parameters. The queue is initialized with the empty path constraint and the empty input [11].

```
parameters: Program $\mathcal{P}$, oracle $O$, Initial state $S_0$
result      : Bug reports $\mathcal{B}$;
$\mathcal{B}$ : setOf(⟨failure, setOf(pathConstraint), setOf(input)⟩)
$\mathcal{B} := \emptyset$;
toExplore := emptyQueue();
enqueue(toExplore, ⟨emptyPathConstraint(), emptyInput⟩);
while not empty(toExplore) and not timeExpired() do
    ⟨pathConstraint, input⟩ := dequeue(toExplore);
    output := executeConcrete($S_0$, $\mathcal{P}$, input);
    foreach f in getFailures(O, output) do
        merge ⟨f, pathConstraint, input⟩ into $\mathcal{B}$;
    newConfigs := getConfigs(input);
    foreach ⟨pathConstraint$_i$, input$_i$⟩ $\in$ newConfigs do
        enqueue(toExplore, ⟨pathConstraint$_i$, input$_i$⟩);
return $\mathcal{B}$;

Subroutine getConfigs(input):
configs := $\emptyset$;
$c_1 \wedge \ldots \wedge c_n$ := executeSymbolic($S_0$, $\mathcal{P}$, input);
foreach $i = 1, \ldots, n$ do
    newPC := $c_1 \wedge \ldots \wedge c_{i-1} \wedge \neg c_i$;
    input := solve(newPC);
    if input $\neq \perp$ then
        enqueue(configs, ⟨newPC, input⟩);
return configs;
```

Fig. 1: The failure detection algorithm. The output of the algorithm is a set of bug reports.

**Path Constraint Minimization:** The failure detection algorithm (Fig. 1) returns bug reports. Each bug report contains a set of path constraints and a set of inputs exposing the failure [12, 13]. 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 minimization algorithm attempts to find a shorter path constraint for a given bug report (Fig. 2). This eliminates irrelevant constraints and a solution for a shorter path constraint is often a smaller input.

Figure 2 the path constraint minimization algorithm. The method intersect returns the set of conjuncts that are present in all given path constraints and the method shortest returns the path constraint with fewest conjuncts.

**Combined Concrete and Symbolic Execution with Explicit-state Model Checking:** Apollo implements a form of explicit-state software model checking. Apollo systematically explores the state space of the system, *i.e.*, the program under test. The algorithm in Section 3 always restarts the execution from the same initial state and discards the state reached at the end of each execution.

```
parameters: Program $\mathcal{P}$, oracle $O$, bug report $b$
result      : Short path constraint that exposes b.failure
$c_1 \wedge \ldots \wedge c_n$ := intersect(b.pathConstraints);
pc := true;
foreach $i = 1, \ldots, n$ do
    $pc_i := c_1 \wedge \ldots c_{i-1} \wedge c_{i+1} \wedge \ldots c_n$;
    if !exposesFailures($pc_i$) then
        pc := pc $\wedge$ $c_i$;
if exposesFailures(pc) then
    return pc;
return shortest(b.pathConstraints);

Subroutine exposesFailure(pc):
$input_{pc}$ := solve(pc);
if $input_{pc} \neq \perp$ then
    $output_{pc}$ := executeConcrete($\mathcal{P}$, $input_{pc}$);
    $failures_{pc}$ := getFailures(O, $output_{pc}$);
    return b.failure $\in$ $failures_{pc}$;
return false;
```

Fig. 2: The path constraint minimization algorithm.

Thus, the algorithm reaches only one-level deep into the application, where each level corresponds to a cycle of: a script that generates an HTML form that the user interacts with to invoke the next script. In contrast, the algorithm presented in this section remembers and restores the state between executions of scripts. This technique, known as state matching, is widely known in model checking [7] and implemented in tools such as SPIN [13] and JavaPath-Finder [21]. To our knowledge, we are the first to implement state matching in the context of Web applications.

Figure 3 the failure detection algorithm: the output of algorithm is set of bug reports; each reports a failure and the set of tests exposing that failure.

**Implementation:** We created a tool called Apollo that implements our technique for my application. Apollo consists of three major components, Executor, Bug Finder and Input Generator illu-strated in Fig. 4. This section first provides a high-level overview of the components and then discusses the pragmatics of the implementation.

The inputs to Apollo are the program under test and an initial value for the environment. The initial environment usually consists of a database populated with some values and usersupplied information about username/password pairs to be used for database authentication.

The Executor is responsible for executing a script with a given input in a given state. The executor contains two subcomponents:

The Shadow Interpreter is interpreter that we have modified to propagate and record path constraints and positional information associated with output. This positional information is used to determine which failures are likely to be symptoms of the same fault.

```
parameters: Program $\mathcal{P}$, oracle $O$, Initial state $S_0$
result      : Bug reports $\mathcal{B}$;
$\mathcal{B}$ : setOf(⟨failure, setOf(pathConstraint), setOf(input)⟩)
$\mathcal{B} := \emptyset$;
toExplore := emptyQueue();
enqueue(toExplore, ⟨emptyPC(), emptyInput(), $S_0$⟩);
visited := {⟨emptyPathConstraint(), emptyInput(), $S_0$⟩};
while not empty(toExplore) and not timeExpired() do
    ⟨pathConstraint, input, $S_{start}$⟩ := dequeue(toExplore);
    ⟨output, $S_{end}$⟩ := executeConcrete($S_{start}$, $\mathcal{P}$, input);
    foreach f in getFailures(O, output) do
        merge ⟨f, pathConstraint, input⟩ into $\mathcal{B}$;
    newConfigs := getConfigs(input, output, $S_{start}$, $S_{end}$);
    newConfigs := newConfigs − visited;
    foreach ⟨$pathConstraint_i$, $input_i$, $S_i$⟩ ∈ newConfigs do
        enqueue(toExplore, ⟨$pathConstraint_i$, $input_i$, $S_i$⟩);
        visited := visited ∪ {⟨$pathConstraint_i$, $input_i$, $S_i$⟩};
return $\mathcal{B}$;

Subroutine getConfigs(input, output, $S_{start}$, $S_{end}$):
configs := $\emptyset$;
$c_1 \wedge \ldots \wedge c_n$ := executeSymbolic($S_{start}$, $\mathcal{P}$, input);
foreach $i = 1, \ldots, n$ do
    newPC := $c_1 \wedge \ldots \wedge c_{i-1} \wedge \neg c_i$;
    input := solve(pathConstraint);
    if input $\neq \perp$ then
        enqueue(configs, ⟨newPC, input, $S_{start}$⟩);
foreach $newPC_i$ ∈ analyzeOutput(output) do
    newInput := solve($newPC_i$);
    if newInput $\neq \perp$ then
        configs := configs ∪ ⟨$newPC_i$, $newInput_i$, $S_{end}$⟩;
return configs;
```

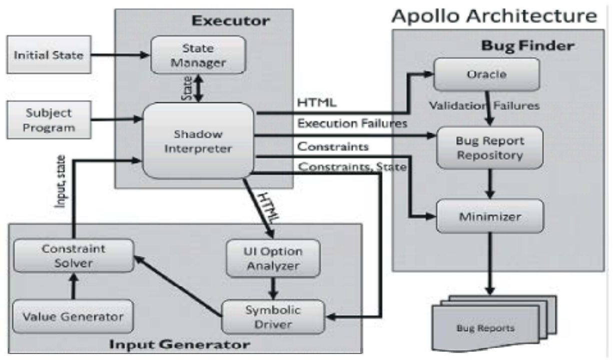

Fig. 3: The failure detection algorithm: the output of algorithm is set of bug reports; each reports a failure and the set of tests exposing that failure.

The State Manager restores the given state of the environment (database, session and cookies) before the execution and stores the new environment after the execution. The Bug Finder uses an oracle to find HTML failures, stores all bug reports and finds the minimal conditions on the input parameters for each bug report. The Bug Finder has the following subcomponents:

- The Oracle finds HTML failures in the output of the program.
- The Bug Report Repository stores all bug reports found during 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 implements the algorithm described in Fig. 4. The Input Generator contains the following subcomponents:

The UI Option Analyzer analyzes the HTML output of each execution to convert the interactive user options into new inputs to execute.

The Symbolic Driver generates new path constraints from the constraints found during execution [13].

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[14].

**Executor:** The shadow interpreter performs the regular (concrete) program execution using the concrete values and simultaneously performs symbolic

**Bug finder:** The bug finder is in charge of transforming the results of the executed inputs into bug reports. Below is a detailed description of the components of the bug finder.Bug report repository. This repository stores the bug reports found in all executions [15-17].

A failure is uniquely defined by the following set of characteristics: the type of the failure (execution failure or HTML failure), the corresponding message message (error/warning message for execution failures and validator message for HTML failure

**Input Generato:** Apollo's approach to the above challenges is to simulate user interaction by analyzing the dynamically created HTML output and tracking the symbolic parameters through the environment:

- Apollo automatically extracts the available user options from the HTML output so that it collects all HTML forms in the page and their components, e.g., buttons and text areas, through which the user can provide input. Any default values for such elements are also collected.

- Apollo collects static HTdocuments that can be called from the dynamic HTML output, i.e., Apollo gather all href attributes in the HTML document.
- Apollo performs a cursory analysis of JavaScript code to find other syntactic references, for instance, a window.open call with a static url as a parameter. Since additional code on the client side (for instance,JavaScript) might be executed when a button is pressed, this approach might induce false positive bug reports. In our experiments, this limitation produced no false positive bug reports.

## CONCLUSIONS

We have presented a technique for finding faults in Web applications that is based on combined concrete and symbolic execution. The work is novel in several respects.First, the technique not only detects runtime errors but also uses an HTML validator as an oracle to determine situations where malformed HTML is created.second we perform an automated analysis to minimize the size of failure-inducing inputs.

## REFERENCES

1. Anand, S., P. Godefroid and N. Tillmann, 2008. "Demand-Driven Compositional Symbolic Execution," Proc. Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems, pp: 367-381.
2. Artzi, S., A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar and M.D. Ernst, 2008. "Finding Bugs in Dynamic Web Applications," Proc. Int'lSymp. Software Testing and Analysis, pp: 261-272.
3. Benedikt, M., J. Freire and P. Godefroid, 2002. "VeriWeb: Automatically Testing Dynamic Web Sites," Proc. Int'l Conf. World Wide Web.
4. Brumley, D., J. Caballero, Z. Liang, J. Newsome and D. Song, 2007. "Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation," Proc. 16th USENIX Security Symp.
5. Cadar, C., D. Dunbar and D.R. Engler, 2008. "Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," Proc. USENIX Symp. Operating Systems Design and Implementation, pp: 209-224.
6. Cadar, C. and D.R. Engler, 2005. "Execution Generated Test Cases: How to Make Systems Code Crash Itself," Proc. Int'l SPIN Workshop Model Checking of Software, pp. 2-23, 2005.
7. Cadar, C., V. Ganesh, P.M. Pawlowski, D.L. Dill and D.R. Engler, 2006. "EXE: Automatically Generating Inputs of Death," Proc. Conf.Computer and Comm. Security, pp. 322-335.
8. Clause, J. and A. Orso, 2009. "Penumbra: Automatically IdentifyingFailure-Relevant Inputs Using Dynamic Tainting," Proc. Int'lSymp. Software Testing and Analysis.
9. Cleve, H. and A. Zeller, 2005. "Locating Causes of Program Failures,"Proc. Int'l Conf. Software Eng., pp: 342-351.
10. Cleve, H. and A. Zeller, 2005. "Locating Causes of Program Failures" Proc. Int'l Conf. Software Eng., pp: 342-351.
11. Csallner, C., N. Tillmann and Y. Smaragdakis, 2008. "DySy: DynamicSymbolic Execution for Invariant Inference," Proc. Int'l Conf. Software Eng., pp. 281-290.
12. Dean, D. and D. Wagner, 2001. "Intrusion Detection via Static Analysis,"Proc. Symp. Research in Security and Privacy, pp: 156-169.
13. Demartini, C., R. Iosif and R. Sisto, "A Deadlock Detection Toolfor Concurrent Java Programs," Software-Practice and Experience, 29(7): 577-603.
14. Emmi, M., R. Majumdar and K. Sen, 2007. "Dynamic Test InputGeneration for Database Applications," Proc. Int'l Symp. Software Testing and Analysis, pp: 151-162.
15. Godefroid, P., 2007. "Compositional Dynamic Test Generation," Proc.Ann. Symp. Principles of Programming Languages, pp: 47-54.
16. Godefroid, P., A. Kie_zun and M.Y. Levin, 2008. "Grammar-BasedWhitebox Fuzzing," Proc. ACM SIGPLAN Conf. ProgrammingLanguage Design and Implementation, pp: 206-215.
17. Godefroid, P., N. Klarlund and K. Sen, 2005. "DART: Directed Automated Random Testing," Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation, pp: 213-223.