PYTHIA: Generating Test Cases with Oracles for JavaScript Applications

Shabnam Mirshokraie Ali Mesbah Karthik Pattabiraman
University of British Columbia
Vancouver, BC, Canada
{*shabnamm, amesbah, karthikp*} *@ece.ubc.ca*

***Abstract*—Web developers often write test cases manually using testing frameworks such as Selenium. Testing JavaScript-based applications is challenging as manually exploring various execution paths of the application is difficult. Also JavaScript's highly dynamic nature as well as its complex interaction with the DOM make it difficult for the tester to achieve high coverage. We present a framework to automatically generate unit test cases for individual JavaScript functions. These test cases are strengthened by automatically generated test oracles capable of detecting faults in JavaScript code. Our approach is implemented in a tool called PYTHIA. Our preliminary evaluation results point to the efficacy of the approach in achieving high coverage and detecting faults.**

***Index Terms*—test generation, oracles, JavaScript, DOM**

I. INTRODUCTION

JavaScript plays a prominent role in modern web applications nowadays. To test their JavaScript applications, developers often write test cases using web testing frameworks such as SELENIUM (GUI tests) and QUNIT (JavaScript unit tests). Although such frameworks help to automate test execution, the test cases need to be written manually, which can be tedious and inefficient. The event-driven and highly dynamic nature of JavaScript, as well as its complex runtime interaction with the Document Object Model (DOM) make it challenging to effectively write stable test suites that achieve high coverage.

Researchers have recently started exploring test generation for JavaScript-based applications [1], [9], [10], [12], [16]. However, current web test generation techniques suffer from two main shortcomings:

1) They all target the generation of *event sequences*, which operate at the DOM-level, to change the state of the application. These techniques fail to capture faults that do not propagate to an observable DOM state; As such, they potentially miss a large portion of code-level JavaScript faults. In oder to capture such faults, effective test generation techniques need to target the code at the JavaScript unit-level, in addition to the event-level.
2) Current techniques simplify the test oracle problem in the generated test cases by simply using generic *soft oracles*, such as HTML validation [12], [1], and runtime exceptions [1]. However, these soft oracles do not capture many kinds of errors. To be practically useful, unit testing requires strong oracles to determine whether the application under test executes correctly at the JavaScript code unit-level and at the DOM-level. While there has been some work on the generation of test inputs [16], not much attention has been paid to the automatic creation of strong test oracles i.e., assertions. A generated test case without assertions is not useful since coverage alone is not the goal of software testing. For such generated test cases, the tester still needs to manually write many assertions, which is time and effort intensive.

In this paper, we propose an automated test and oracle generation technique for JavaScript applications to address these two shortcomings. Our approach operates through a three step process. First, it dynamically explores and guides the application, using a function coverage maximization greedy method, to infer a test model. Then, it generates test cases at JavaScript function levels. Finally, it automatically generates test oracles using mutation testing. Mutation testing is typically used to evaluate the quality of a test suite [2], or to generate test cases that kill mutants [5]. In our work, we adopt mutation testing to generate an oracle that is able to detect a seeded fault in the JavaScript application.

To the best of our knowledge, our work is the first to automatically (1) generate unit tests at the function-level for JavaScript code, and (2) employ mutation testing for test oracle generation in JavaScript-based applications.

Our main contributions in this work are:

- A method that dynamically guides the exploration of a web application to maximize function coverage;
- A generic technique to generate JavaScript function-level unit tests;
- A mutation-based method to create test oracles that can detect faults in the JavaScript code;
- The implementation of our approach in an open-source tool called PYTHIA, which requires no browser modifications, and is hence portable;

II. RELATED WORK

Test generation. Different constraint-solving approaches have been proposed for test generation. For instance, pathCrawler [20] combines static and dynamic analysis and performs on-the-fly exploration of the input space of the application. Concolic testing has been employed in DART [6] and further extended in CUTE [17] and PEX [19]. It still remains to be seen if these approaches scale when applied to dynamic web applications.

Meta-heuristic search approaches have been used as an alternative to constraint-based techniques. Examples of such approaches are Ribeiro et al. [15], Fraser et al. [3] and Harman et al. [7]. The success of such methods depends on the availability of appropriate fitness functions that can properly guide the solution towards an optimal one [8]. Devising such a fitness function is a significant challenge.

To overcome the drawbacks of search-based approaches, Malburg et al. [8] propose to combine constraint-based and search-based testing. The focus of these techniques is mostly on input generation to efficiently cover every possible program flow. In our work, we focus on maximizing the function coverage of the program instead of state space coverage, as the number of states can become prohibitively large in web applications. Further, instead of generating inputs to uncover errors, we create oracles capable of detecting faults.

Oracle generation. There has been limited work on test oracle generation for testing. Fraser et al. [5] propose μTEST, which employs a mutant-based oracle generation technique. It automatically generates unit tests for Java object-oriented classes by using a genetic algorithm to target mutations with high impact on the application's behaviour. In a follow up paper [4], they extend μTEST to simplify the human understanding of the generated tests by identifying relevant pre-conditions on the test inputs and post-conditions on the outputs. At a high level, our work is similar to μTEST in that we also apply mutation analysis to create effective test oracles. However, our test cases and oracles are generated using the dynamic event-driven model of JavaScript, which has its own unique challenges and is different from traditional programming languages such as Java.

Staats et al. [18] address the problem of selecting oracle data, which is formed as a subset of internal state variables as well as outputs for which the expected values are determined. They apply mutation testing to produce oracles and rank the inferred oracles in terms of their fault finding capability. This work is different from our approach in that they merely focus on supporting the creation of test oracles by the programmer, rather than fully automating the process of test case generation. Further, they do not support JavaScript.

Web application tests and oracles. Mesbah et al. [12] use dynamic analysis to construct a model of the application's state space, from which event-based test cases are automatically generated. They use generic and application-specific invariants as a form of soft oracles. Marchetto and Tonella [9] propose a search-based algorithm for generating event-based sequences to test Ajax applications. Our earlier work, JSART [13], automatically infers program invariants from JavaScript execution traces and uses them as regression assertions in the code.

Saxena et al. [16] combine random test generation with the use of symbolic execution for systematically exploring a JavaScript application's event space as well as its value space. Closely related to our work is ARTEMIS [1], which supports automated testing of JavaScript applications. ARTEMIS considers the event-driven execution model as well as the JavaScript code's interaction with the DOM to generate test cases. Our work is different in two main aspects from these works: (1) they all target the generation of event sequences at the DOM level, while we generate unit tests at the JavaScript code level, and (2) they do not address the problem of test oracle generation and only check against soft oracles (e.g., invalid HTML). Our work is able to capture errors that occur at the JavaScript code level by generating effective test cases with strong oracles.

```
var currentDim=20;
function cellClicked() {
  var divTag = '<div id='divElem' />';
  if($(this).attr('id') == 'cell0'){
    $(#cell0).after(divTag);
    $('div #divElem').click(setup);
  }
  else if($(this).attr('id') == 'cell1'){
    $(#cell1).after(divTag);
    $('div #divElem').click(function(){setDim(20)});
  }
}

function setup() {
  setDim(10);
  $(#startCell).click(start);
}

function setDim(dimension) {
  var dim=($('body').width + $('body').height))/dimension;
  currentDim+=dim;
  $(#endCell).css('height', dim+'px');
  return dim;
}

function start() {
  if(currentDim>20)
    $(this).width('height', currentDim+'px');
  else
    $(this).remove();
}
$document.ready(function() {
  ...
  $(#cell0).click(cellClicked);
  $(#cell1).click(cellClicked);
});
```

Fig. 1. JavaScript code of the running example.

III. Motivation and Challenges

In this section, we illustrate the challenges associated with generating tests in JavaScript applications.

Figure 1 presents a snippet of a JavaScript application that we use as a running example throughout the paper. This simple example contains four main JavaScript functions:

1) `cellClicked` is bound to the click handler of the two DOM elements with IDs `cell0` or `cell1` (Lines 34-35). These two elements become available when the DOM is fully loaded. Depending on the element clicked, `cellClicked` inserts a `div` element with ID `divElem` after the clicked element and makes it clickable by attaching either `setup` or `setDim` as its event handler.
2) `setup` calls `setDim` to change the value of the `currentDim` global variable. It further makes an element with ID `startCell` clickable by setting the click handler to `start`.
3) `setDim` receives an input variable. It performs some computations to set the `height` value of the `css` property of a DOM element with ID `endCell` and the value of global variable `currentDim`. `setDim` also returns the computed dimension.
4) `start` is called when the element with ID `startCell` is clicked (Line 16), which either sets the dimension of the element from which it has ben called, or removes the element (Lines 27-30).

The first challenge in testing JavaScript applications is that an error may not become immediately apparent due to the event-driven nature of the execution. For example, if the '+' sign in Line 21 is mistakenly replaced by '−', the affected result does not immediately propagate to the observable DOM state after the function exits. While this mistakenly changes the value of a global variable, `currentDim`, which is later used in `start` (Line 27), it neither affects the returned value of the `setDim` function nor the `css` value of element `endCell`. Therefore, a simple GUI testing approach does not help to detect the fault in this case. However, a unit test case that targets individual functions, e.g., `setDim` in the example, helps a tester to spot the fault, and thus easily resolve it. This is why we focus on generating unit tests with oracles in this paper.

Another challenge in testing JavaScript applications is the event-driven dynamic nature of JavaScript, and its extensive interaction with the DOM resulting in many execution paths and states. In the initial state, clicking on `cell0` or `cell1` takes the browser to two different states as a result of the `if-else` statement in Lines 4 and 8 of the function `cellClicked`. Even in this simple example, it can be seen that expanding either of the resulting states has different consequences due to different functions that can be potentially triggered. Executing either `setup` or `setDim` in Lines 6 and 10 results in different execution paths, DOM states, and code coverage. It is this dynamic interaction of the JavaScript code with the DOM (and indirectly CSS) that makes it challenging to generate test cases for JavaScript applications. A related important challenge is that when unit testing JavaScript functions that have DOM interactions, such as `setDim`, the DOM tree in the state expected by the function has to be present during test execution, otherwise the test will fail due to a null exception.

IV. Approach

Our main goal in this work is to generate test cases coupled with effective test oracles, capable of detecting regression JavaScript faults. We assume that the amount of time available to generate test cases is finite. Consequently, we guide the test generation to maximize coverage under a given time constraint. Another goal is to make the oracles as efficient as possible, so we choose to only include those elements in the assertions that are absolutely essential in detecting potential errors.

Our approach automatically generates test cases and oracles at function-level unit tests which consist of unit tests with assertions that verify the functionality of JavaScript code at the function level.

At a high level, our approach is composed of three main steps:

1) In the first step, we dynamically explore various states of a given web application, in such a way as to maximize the number of functions that are covered throughout the program execution. The output of this initial step is a State-Flow Graph (SFG) [11], capturing the explored dynamic DOM states and event-based transitions between them.
2) In the second step, test cases are generated at JavaScript function levels by running the instrumented version of the application. Through an execution trace obtained, we extract JavaScript function states at the entry and exit points, from which function-level unit tests are generated.
3) To create effective test oracles, we automatically generate mutated versions of the application. The test oracles are then generated at the JavaScript code level by comparing the two execution traces obtained from the original and the mutated versions.

Next, we describe each of these three steps.

V. Maximizing Function Coverage

As mentioned before, our goal is to generate test cases that maximize the number of functions covered while exercising the program's event space. To that end, our approach combines dynamic and static analysis to decide which state/event(s) should be selected for expansion to maximize the probability of covering uncovered functions. The function coverage maximization algorithm chooses a next state that has the maximum value of the sum of the following two metrics:

1) The number of uncovered functions that can potentially be visited through the execution of DOM events in a given DOM state; To calculate this metric, our approach identifies all JavaScript functions that are directly or indirectly attached to a DOM element as event handlers, in a given state. When a given function f is set as the event-handler of a DOM element d, it makes the element a potential clickable element in a DOM state s.

By instrumenting the application and dynamically monitoring the execution trace, we detect uncovered functions and any associated clickable elements that have those functions as event-handlers. In addition, by inferring the static function call graph of the application, we calculate the total number of functions that (indirectly) will be executed if such clickable elements are exercised.

2) The number of DOM elements that can potentially become clickable elements, i.e., if the event-handlers bounded to those clickables are triggered, new (uncovered) functions will be executed; To obtain this number, we statically analyze the previously obtained potential functions within a given state in search of such clickable elements. Of course, the functions that can potentially be attached to such elements should not have been covered before.

While exploring the application, the next state for expansion is selected by adding the two metrics and choosing the state with the highest sum. Note that if multiple clickables with the same function handler exist in the selected state, we randomly select one of those and add it to the clickable list of elements.

During the exploration, the state-flow graph, the execution trace, uncovered functions, as well as new detected clickables are updated as an event is fired and a new DOM state is detected.

Going back to our example in Figure 1, in the initial state, clicking on elements with IDs `cell0` and `cell1` results in two different states due to an `if-else` statement in Lines 4 and 8 of `cellClicked`. Let's call the state in which a `DIV` element is located after the element with ID `cell0` as s_0, and the state in which a `DIV` element is placed after the element with ID `cell1` as s_1. If state s_0, with the clickable `cell0`, is chosen for expansion, function `setup` is called. As shown in Line 15, `setup` indirectly calls `setDim`, and thus, by expanding s_0 both of the aforementioned functions get called by a single click. Moreover, a potential clickable element is also created in Line 16, with `start` as the event-handler. Therefore, expanding s_1 results only in the execution of `setDim`, expanding s_0 results in the execution of functions `setup`, `setDim`, as well as a potential execution of `start` in future states. Therefore, our greedy state selection algorithm chooses s_0 to potentially maximize the function coverage.

At the end of this step, we obtain a state-flow graph of the application that can be used in the test generation step.

VI. Generating Function-level Unit Tests

In the second step,we generate test cases at function-level, as described below.

To generate unit tests that target JavaScript functions, we need to log the state of each function at their entry and exit point during execution. To that end, we automatically intercept the JavaScript code by setting up a proxy between the server and the browser, and instrumenting the code to log various entities at the entry and exit points of functions.

The entities that we log at the entry point of a given JavaScript function include: (1) function parameters including passed variables, functions, and DOM elements, (2) global variables used in the function, and (3) the current DOM structure just before the function is executed. At the exit point of the JavaScript function and before every `return` statement, we log the state of the (1) return value of the function, (2) global variables that have been accessed in that function, and (3) DOM elements accessed in the function. At each of the above points, our instrumentation records the name, runtime type, and actual values. The dynamic type is stored because JavaScript is a dynamically typed language, meaning that the variable types cannot be determined statically.

Going back to our running example in Figure 1, at the entry point of `setDim`, we log the value and type of both the input parameter `dimension` and global variable `currentDim`, which is accessed in the function. Similarly, at the exit point, we log the values and types of the returned variable `dim` and `currentDim`.

In addition to the values logged above, we need to capture the DOM state for functions that interact with the DOM. To be able to unit test functions that have DOM API calls, the expected DOM elements need to be present for the function to proceed. Otherwise, the function may throw an exception or produce an incorrect result. To mitigate this problem, we capture the state of the DOM just before the function starts its execution, and include that as a test fixture in the generated test case.

In the running example, at the entry point of `setDim`, we log the `innerHTML` of the current DOM as the function contains several calls to the DOM, such as retrieving the element with ID `endCell` in Line 22. We further include in our execution trace the way DOM elements and their attributes are modified by the JavaScript function at runtime. Based on the pattern in which the JavaScript DOM modifications occur, we can add instrumentation code to record the accessed DOM elements. The information that we log for accessed DOM elements includes the ID attribute, the XPath position of the element and all changed attributes. We use a set collection to keep the information about DOM modifications, so that we can record the latest changes to a DOM element without any duplication within the function. For instance, we record both `width` and `height` properties of the `BODY` element as well as the ID and `height` value of the element `endCell`.

After instrumenting the application, we run the application to produce an execution trace that contains:

- Information required for preparing the environment for each function to be examined in a test case, including its input parameters, used global variables, and the DOM tree in a state that is expected by the function, and
- Necessary entities that need to be assessed after the function is executed, including the function's output as well as the touched DOM elements and their attributes (The actual assessment process is explained in Section VII).

VII. Generating Test Oracles

In the third step, our approach generates test oracles for the function-level test cases generated in the previous step.

Similar to the test generation step (Section VI), the oracle generation part targets JavaScript code mutations, which are used to generate assertions for our generated function-level unit tests.

Our oracle generation technique is a mutation-based process, which iteratively executes the following steps:

1) a mutation is generated by injecting a single fault into the original version of the web application,
2) related entry/exit program states at the DOM and JavaScript function levels of the mutant and the original web application are captured,
3) relevant observed state differences at each level are detected and abstracted into test oracles,
4) the generated test oracles are injected as assertions into the corresponding test case.

To seed code level faults, we use our recently developed JavaScript mutation testing tool, MUTANDIS [14]. Mutations generated by MUTANDIS are selected through a *function rank* metric, which ranks functions in terms of their relative importance from the application's behaviour point of view. The mutation operators are chosen from a list of common operators, such as changing the value of a variable or modifying a conditional statement. Once a mutant is produced, it is automatically instrumented. We collect a new execution trace from the mutated program by executing the same sequence of events that was used on the original version of the application. We extract the state of each JavaScript function at its entry and exit points, similar to the function state extraction mechanism explained in Section VI.

After the execution traces are collected for all the generated mutants, we generate test oracles by comparing the execution trace of the original application with the traces we obtained from the modified versions. Our function-level oracle generation targets *postcondition assertions*. The postcondition is an observable state after the unit test execution. Therefore, postcondition oracle assertions can be used to examine the expected behaviour of a given function after it is executed in a unit test case.

Our technique generates postcondition assertions for functions that exhibit a *different exit-point state* but the *same entry-point state*, in the mutated execution traces. Thus, each assertion for a function contains (1) the function's returned value, and (2) the used global variables in that function.Each assertion is coupled with the expected value obtained from the execution trace of the original version.

Due to the highly dynamic, asynchronous, and non-deterministic behaviour of JavaScript applications, a function with the same entry state can exhibit different outputs when executed multiple times. In this case, we need to combine assertions to make sure that the generated test cases do not mistakenly fail. Let's consider a function f with an entry state $entry$ in the original version of the application (A), with two different exit states $exit_0$ and $exit_1$. If in the mutated version of the application (A_m), f exhibits an exit state $exit_m$ that is different from both $exit_0$ and $exit_1$, then we combine the resulting assertions as follows:

$\texttt{assert1}(exit_1, expRes_1) || \texttt{assert2}(exit_2, expRes_2)$, where the expected values $expRes_1$ and $expRes_2$ are obtained from the execution trace of A.

Going back to the running example of Figure 1, if we mutate the plus sign into a minus sign in Line 20, the affected elements are: the return value of `setDim` in Line 23, and the global variable `currentDim` in Line 21.

The generated assertions that target variables, compare the value as well as the runtime type against the expected ones. Assuming that `width` and `height` are 100 and 200 accordingly, one such assertion would be: `equal(setDim(10), 30)`. To check the type of the `currentDim` as the function exits, we generate: `equal(typeof(currentDim), 'number')`.

VIII. TOOL IMPLEMENTATION

We have implemented our JavaScript test and oracle generation approach in an automated tool called PYTHIA. The tool is written in Java and is publicly available for download.[1] Our implementation requires *no browser modifications*, and is hence portable.

Our function coverage maximization technique extends and builds on top of the dynamic event-driven AJAX crawler, CRAWLJAX [11]. As mentioned before, to mutate JavaScript code, we use our recently developed mutation testing tool, MUTANDIS [14]. For JavaScript code interception, we employ an enhanced version of WebScarab proxy. This enables us to automatically analyze the content of HTTP responses before they reach the browser. To instrument the intercepted code, Mozilla Rhino[2] is used to parse JavaScript code to an AST, and back to the source code after the instrumentation is performed. We use Rhino's APIs to search for program points where instrumentation code needs to be added. JavaScript function-level tests are generated in the QUNIT unit testing framework,[3] which is capable of testing any generic JavaScript code.

IX. EVALUATION

Our preliminary evaluation results have been very promising. To empirically assess the efficacy of our test generation approach, we are currently in the process of conducting a large study on several real-world web applications.

X. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a framework for testing JavaScript applications, which automatically generates test cases accompanied with oracles for individual JavaScript functions. These test cases are coupled with automatically generated test oracles capable of detecting faults in JavaScript code. We implemented our approach in an open-source tool called PYTHIA. We are currently conducting an empirical evaluation of PYTHIA on real-world applications to assess the efficacy of the approach in terms of coverage and fault finding capabilities.

[1] http://salt.ece.ubc.ca/software/pythia/
[2] http://www.mozilla.org/rhino/
[3] http://qunitjs.com

Acknowledgment

This work was supported in part by a Strategic Project Grant (SPG) from the Natural Science and Engineering Research Council of Canada (NSERC) and a gift from Intel Corporation. We thank the anonymous reviewers for ASE'13 for their insightful comments.

References

[1] S. Artzi, J. Dolby, S. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *Proc. 33rd International Conference on Software Engineering (ICSE'11)*, pages 571–580, 2011.

[2] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[3] G. Fraser and A. Arcuri. Evolutionary generation of whole test suites. In *Proc. 11th International Conference on Quality Software (QSIC'11)*, pages 31–40, 2011.

[4] G. Fraser and A. Zeller. Generating parameterized unit tests. In *Proc. International Symposium on Software Testing and Analysis (ISSTA'11)*, pages 364–374, 2011.

[5] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)*, 38(2):278–292, 2012.

[6] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, pages 213–223, 2005.

[7] M. Harman, F. Islam, T. Xie, and S. Wappler. Automated test data generation for aspect-oriented programs. In *Proc. 8th ACM International Conference on Aspect-Oriented Software Development (AOSD'09)*, pages 185–196, 2009.

[8] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *Proc. 26th International Conference on Automated Software Engineering (ASE'11)*, pages 436–439, 2011.

[9] A. Marchetto and P. Tonella. Using search-based algorithms for Ajax event sequence generation during testing. *Empirical Software Engineering*, 16(1):103–140, 2011.

[10] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of Ajax web applications. In *Proc. 1st Int. Conference on Sw. Testing Verification and Validation (ICST'08)*, pages 121–130. IEEE Computer Society, 2008.

[11] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.

[12] A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering (TSE)*, 38(1):35–53, 2012.

[13] S. Mirshokraie and A. Mesbah. JSART: JavaScript assertion-based regression testing. In *Proc. International Conference on Web Engineering (ICWE'12)*, pages 238–252. Springer, 2012.

[14] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Efficient JavaScript mutation testing. In *Proc. 6th International Conference on Software Testing Verification and Validation (ICST'13)*, pages 74–83. IEEE Computer Society, 2013.

[15] J. Ribeiro, M. Zenha-Rela, and F. de Vega. A strategy for evaluating feasible and unfeasible test cases for the evolutionary testing of Object-Oriented software. In *Proc. 3rd International Workshop on Automation of Software Test (AST'08)*, pages 85–92, 2008.

[16] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *Proc. Symp. on Security and Privacy (SP'10)*, pages 513–528. IEEE Computer Society, 2010.

[17] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05)*, pages 263–272, 2005.

[18] M. Staats, G. Gay, and M. Heimdahl. Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing. In *Proc. International Conference on Software Engineering (ICSE'11)*, pages 870–880, 2011.

[19] N. Tillmann and J. de Halleux. Pex: white box test generation for .NET. In *Proc. 2nd International Conference on Tests and Proofs (TAP'08)*, pages 134–153, 2008.

[20] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: automatic generation of path tests by combining static and dynamic analysis. In *Proc. 5th European Dependable Computing Conference (EDCC'05)*, pages 281–292, 2005.