Generating Input HTML for testing JavaScript Applications

RPE proposal James Lo Oct 7, 2013

Introduction

While increasingly popular, JavaScript Web applications are also increasingly complex. Because HTML describes the front-end GUI, a Web application usually has abundant JavaScript code that access and mutate HTML. Indeed, a recent study [8] reports that the majority of bugs in JavaScript code relates to interactions with the Document Object Model (`DOM`). The `DOM` is the W3C standard API that uses operations such as `parentElement`, `firstChild`, `lastChild`, `children`, `nextSibling`, etc. to traverse the XML and HTML tree data structure for access and mutation.

To unit test a JavaScript function `F` that interacts with the DOM, we need to provide a HTML input in the state expected by F. However, generating the right input HTML to test JavaScript code is not easy since the HTML tree structure can be involved in conditions that influence branch execution decisions. For example, `F` might contain an `IF` statement as follows:

```
if (node0.parentElement.parentElement === node2.firstChild.nextSibling) {}
```

To go to the `TRUE` branch of the above `IF` statement, the input HTML must have `node0`'s grandparent (`parentElement.parentElement`) equal to `node2`'s second child (`firstChild.nextSibling`). The same principle applies to loop conditions, in which the branch decision is to stay within the loop or to break from it.

Therefore, going through a specific branch requires a specific HTML structure, and random HTML generation is often insufficient to efficiently cover all possible branch permutations (execution paths) systematically across multiple conditions. Increasing the number of conditions can also exponentially grow the number of possible execution paths, and therefore, manual HTML generation would be too cumbersome.

To test JavaScript code effectively, an automatic way of generating HTML geared towards systematic path coverage maximization is desired. In this research we employ techniques in concolic analysis [12] to generate input HTML, in which there is at least one input HTML that covers each permutation of branches (`T`/`F`, `Stay`/`Break`) across multiple conditions in a JavaScript function.

Related Work

Concolic Testing [12] is an exhaustive testing method for maximizing path coverage. Each IF statement has 2 branches: True and False. Each iteration within a loop also has 2 branches: Stay and Break. Having the constraints generated from a dynamic backward slice, concolic testing uses a constraint solver to generate input that would drive the execution of each condition towards a specific branch. Kudzu [10] is the only work in the literature that uses a constraint solver to conduct constraint-based testing for JavaScript Web applications. While our work focuses on generating HTML input to achieve path coverage, Kudzu focuses on generating string input to detect security vulnerabilities in JavaScript applications. Our work is also designed to run on multiple browsers, while Kudzu runs on only the browser that supports its backward slicing component [11].

Dynamic Backward Slice: Given a variable (e.g. an integer) at a point in time during a software's execution (e.g. at a line of code), a dynamic backward slice traces how the variable has arrived at its current value (e.g. what operations or calculations had been done and at what time), all the way back to the input before execution started. For example, if integer A equals to X + Y in line 5, integer A's backward slice would be X's and Y's backward slices, as well as the operation "+". Dynamic backward slicing first requires logging the runtime execution and our logging approach is similar to Jalangi [13]'s shadow system, in which we encapsulate each data value into an object; the object contains the log (backward trace, in our case) in addition to the data's current value. While it can also be used for concolic testing, Jalangi's shadow system is mainly aimed at record and replay.

Constraint Solvers such as SAT solvers solve for parameters that satisfy a set of predefined constraints. For example, Genevès et al. developed an XML solver [4] that takes limited XPaths as inputs; then it outputs XML that

would satisfy those XPath conditions. However the XPaths that the XML solver supports are severely limited. Their solver does not support DOM node attributes and is not scalable to more than 5 unique nodes. CVC [2, 3] is more scalable and the constraints can be more expressive. However, while used by [12] and [13], CVC is a general SMT solver and does not natively support the tree structure defined in the DOM API. Nevertheless, both solvers give only a description of the desired DOM tree (e.g. node A is child of node B) at most, rather than the actual XML/HTML. In our case for testing JavaScript applications, we have to take an additional step to transform the solver's output into HTML.

Feedback Directed Testing is an adaptive testing approach that uses the outcome of executing an input, to determine what the next input should be for achieving a goal, mostly maximizing coverage. Concolic testing is a form of feedback directed testing, in which it conducts dynamic backward slicing to generate inputs, and then uses the resulting executed path as feedback. In contrast, Artemis [1]'s approach generates initial inputs randomly and uses the output of functions as feedback. Pythia [9] also generates initial inputs randomly, their feedback is changes to a state flow graph model, and their goal is to maximize the number of functions being called. In contrast to Artemis and our tool, Pythia is for regression testing; it requires a previous version of bug free software, and also mandates that the current version has zero change in both behavior and interface such that the same input always yields identical output. When a software requires regression testing, it either has a bug fixed (violates the bug-free requirement) or has an improvement or a new feature implemented (may violate the zero change requirement). An occasion when both external behavior and interface don't change, is when a function's internal implementation has changed for improving only performance. Then, JavaScript applications are known to lack determinism [7], meaning the same source code is known to yield different outputs even for the same inputs. Moreover, while we aim to infer an HTML input, Pythia uses the application's existing HTML to unit test JavaScript functions.

Challenges

The following are challenges that we would encounter when generating an input HTML appropriate for driving the software through a specific execution path:

Extracting constraints: Accesses and mutations to HTML can happen any time during execution. Because JavaScript as a programming language is highly dynamic, static analysis is insufficient to detect and resolve DOM operations [7]. Indeed, authors of static techniques such as [6] and [15] report substantial gaps and false positives in their own work. Thus dynamic backward slicing is required to extract the data flow of variables during runtime and determine what types of DOM operations are involved in branching a condition. For example, a JavaScript program can have

```
1. nodeY = nodeX.parentElement // only dynamic backward slicing can tell if nodeX is a DOM
2. …                           // element, or an ordinary object
3. if (nodeZ === nodeY.parentElement)
```

Thus in the input HTML, node Z has to be node X's grandparent. Node Y is only an intermediate node.

Solving interdependent constraints across multiple conditions: A software system usually has multiple conditional statements. The constraints in branching one conditional statement may be interdependent with the constraints in branching another. For example, a JavaScript program can have two conditional statements

```
1. if (node0 === nodeA.firstChild || node0 === nodeB.lastChild)
2. if (node0 === nodeA.parentElement || node0 === nodeB.parentElement)
```

In the DOM, each node can have only one parent. Therefore, in the 1st If statement, node 0's parent can be either node A or node B. Similarly if node 0's parent is node A, then node 0 cannot be node A's parent: a node cannot be both child and parent of the same node. Consequently, to make both IF statements true, node 0's parent has to be node B. Indeed, conditions can get more complex than having just an `OR`. Increasing complexity within a condition, and increasing the number of conditions, can together increase the overall complexity of constraints. As

a result, a constraint solver is often required to generate input HTML that is appropriate for branching multiple conditions in the same execution path.

Solving logic for multiple data types: This is required because `DOM` operations can interleave with operations on other data types. In the following example, the constraint solver must support constraints for at least `DOM` and numbers, to solve and generate a valid input HTML, for driving the execution path through the `TRUE` branch of the `IF` statement in line 3:

```
1. var x = a-d*f(10);
2. var n = nodeY.children.length + x/3;
3. if (node0.children[n].parentElement === nodeY.lastChild)
```

DOM Mutations: Changes to the HTML can also happen any time during execution, and must also be accounted for in both the backward slicing and the solver. Example mutations include adding or deleting a DOM node, and modifying the content or attributes within a DOM node. Expressing DOM mutations can be more challenging than expressing numerical operations such as additions and subtractions, because DOM mutations are more diverse, and the DOM is a tree structure.

Approach

Step 1: we trace the runtime execution of the JavaScript Web app, and conduct a dynamic backward slice using the trace. To start dynamic analysis, we would use Google's Closure Compiler API [5] to AST-instrument the source code so that we can log the runtime execution of a JavaScript Web app: what variables are involved in each branch execution, and whether these variables have gone through any DOM operation.

Step 2: we analyze the backward slice and for each condition that involves DOM operations, we identify the following:

1. The types of HTML nodes: e.g. `<a>`, `<p>`, `<div>`, etc.
2. The types of DOM operations: `parentElement` access, `firstChild` access, `nextSibling` access, insertions (`appendChild`), deletions (`removeChild`), etc.
3. The types of logical relationships (`AND`, `OR`, `NOT`) between DOM operations within the same conditional statement.

From this information we would generate the constraints necessary for a constraint solver to generate input HTML, that would drive the software to take a specific branch in one single conditional statement.

Step 3: we conjunct the constraints across multiple conditional statements. Steps 2 & 3 involve developing a JavaScript library that also handles the negation of conditional statements for symbolic- and concolic-testing. The solver can be an XML solver [4] or a CVC solver [2, 3].

Step 4: Translate the output from the constraint solver into actual HTML. This step is implemented by parsing the text output of the solvers, and using a DOM API [14] to generate HTML.

Evaluation

Our goal is to generate input HTML that would drive the software execution towards a specific path of branch permutation. To determine whether the generated HTMLs are correct, we will run experiments by taking the input HTML, and comparing the expected vs. actual execution path. We set the expected execution path (e.g. `T` in the 1st `IF` statement, `F` in the 2nd, and so on), and then use the expected path to generate the input HTML. To get the actual execution path, we are going to instrument the source code so that when the software runs, it would show which branch has been taken for each condition. Then we compare if the branch actually executed would match the initial branch we've set. Next, we are going to follow the evaluation approach of Pythia's ICSE submission and compare our tool to Pythia and Artemis, in terms of coverage and ability to find bugs.

References

1. S. Artzi, J. Dolby, S. Jensen, A. Moller, 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. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. **CVC4**. In *CAV*, pages 171--177, 2011.
3. C. Barrett and C. Tinelli. **CVC3.** In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298-302. Springer, July 2007. Berlin, Germany.
4. P. Genevès, N. Layaïda and A. Schmitt. Efficient static analysis of **XML** paths and types, Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, June 10-13, 2007, San Diego, California, USA
5. Google Closure Compiler API. https://developers.google.com/closure/compiler/
6. A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In Proc. of the International World Wide Web Conference (WWW), pages 561–570. ACM, 2009.
7. J. Mickens, J. Elson, and J. Howell. **Mugshot**: deterministic capture and replay for Javascript applications. In 7th USENIX conference on Networked systems design and implementation, NSDI'10, 2010.
8. F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. An empirical study of client-side JavaScript bugs. In Proc. ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'13). IEEE Computer Society, 2013.
9. Pythia. http://salt.ece.ubc.ca/software/pythia/
10. 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.
11. P. Saxena, S. Hanna, P. Poosankam, and D. Song. **FLAX**: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *17th Annual Network & Distributed System Security Symposium, (NDSS)*, 2010.
12. 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.
13. K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. **Jalangi**: A selective record-replay and dynamic analysis framework for JavaScript. In Proc. European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'013). ACM, 2013
14. W3C DOM API for Java. http://docs.oracle.com/javase/7/docs/api/org/w3c/dom/package-summary.html
15. Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In Proc. of the International World-Wide Web Conference (WWW), pages 805–814. ACM, 2011.

Follow up	Section	Date
Given existing test cases (preferably whose that use DOM), compare coverage with our tool (hypothesized improvement) vs. without (experimental control)	Evaluation	2013/10/22
Comparisons of coverage vs. time, between existing tools	Evaluation	2013/10/22
Find apps that have test cases (preferably QUnit), see if our tool can extend these existing test cases	Evaluation	2013/10/22
Find apps that 1. use DOM 2. have test cases on DOM-using code 3. preferably QUnit	Evaluation	2013/10/22