ConcolicDOM: Generating HTML to concolic test JavaScript Web applications

James Lo
Department of Computer Science
University of British Columbia
Vancouver, Canada
tklo@cs.ubc.ca

Eric Wohlstadter
Department of Computer Science
University of British Columbia
Vancouver, Canada
wohlstad@cs.ubc.ca

Ali Mesbah
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
amesbah@ece.ubc.ca

ABSTRACT

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution, Test coverage of code, Test execution*; D.3.2 [**Software**]: Programming Languages—*JavaScript*

Keywords

JavaScript, test runnability, HTML, DOM

1. INTRODUCTION

JavaScript is increasingly a popular language for software implementation: For end users, HTML5 and its standardization enable Web apps to have an interactivity and feature-richness comparable to those implemented for traditional desktops. The latest round of browser wars makes executing JavaScript more efficient, robust, secure and consistent. For programmers, JavaScript does not have the burden of memory management and static typing; and more desktop and mobile operating systems actually now support implementing native apps using the combination of JavaScript, HTML and CSS [11]. The Bring Your Own Device (BYOD) movement in Enterprise IT increases hardware heterogeneity, which also makes JavaScript apps[1] a conveniently portable solution for delivering the application front end (e.g. [6]). Emergence and scalability of Node.js also make JavaScript widely adopted on the server side. Consequently, many institutions such as the Khan Academy [5] use JavaScript for teaching programming; and JavaScript has consistently been a top 2 in the RedMonk [8] popularity rankings.

Yet, despite the language's ubiquity, testing JavaScript is not easy. For example, because HTML describes the graphical user interface of a Web app, a lot of JavaScript code is written to access and mutate HTML through the Document Object Model (DOM) API. When JavaScript code runs, its runtime execution would encounter DOM operations that would subtly imply the DOM (and thus the webpage's HTML) having a particular tree structure. In other words, when trying to run a test case, if the DOM structure does not satisfy what the code expects it to be, execution would fail and the test case would terminate prematurely.

[1]JavaScript apps are preferred in Web browsers because they are lighter weight than Java applets and they don't require installation of any proprietary plugins such as Flash and Silverlight

```javascript
function checkRows() {
  var field = getElementById("tetris");
  var i, row;
  for (i=field.children.length; i--;) {
    row = getElementById("row"+i);
    if (row.children.length === 10) {
      // ... row filled, update score
    }
  }
}
```

Sample Code 1: Example code whose tests and execution depend on the Document Object Model having a precise tree structure. `getElementById()` is equivalent to `document.getElementById()`.

1.1 Motivating Example

To further illustrate the necessity of having a satisfiable DOM structure, suppose the `function checkRows()` in Sample Code 1 is being unit tested concolic-ly. The function is simplified from a feature Chrome Experiment [10] that uses the DOM to implement the game Tetris. Concolic testing [12] executes the app in a way to maximize path coverage; to do so, we must visit both the `True` and `False` branches of each `if` statement in Sample Code 1.

To guide the `if` statement going to the `True` branch, the web page's HTML must yield a DOM structure satisfying many constraints:

- There is an element with id `tetris`.
- `tetris` contains children elements, so that we can first enter the `for` loop.
- There are rows having id's in the nomenclature `row0`, `row1`, etc.
- The number of rows must be greater than or equal to the number of children that `tetris` has.
- At least one of the rows must have exactly 10 children.

Until all of the above constraints are satisfied, the function's execution would likely lean towards an unintended path or would even halt. For example, when `field` is `null`, the property access `field.children` would result in a `Type Error` and consequently the rest of the function cannot be run or tested. Therefore, a satisfying HTML must be generated so that execution of the function and test case would not crash and can be guided towards the intended path.

While manual generation of HTML is possible, the manual approach would quickly become tedious and not scalable. The reason is that a unique DOM structure is required for going through a different execution path. For example, to go to the `False` branch of the above `if`, rows cannot have 10 children. Therefore, to cover both the `True` and `False` branches of an `if`, we must generate 2 unique DOM structures. Generally in an `if` block, the exact number of unique DOM structures per condition is 2 plus the number of `else if`'s in the `if` block. Loops are more difficult for achieving path coverage, because it's not easy to determine the max upper limit of loop iterations. For example, in Sample Code 1, `field` can have any number of children.

Nevertheless, the number of unique DOM structures would at least double whenever we try to cover an additional DOM-dependent condition, be it an `if` or a `loop`. Moreover, manual generation can become complex as DOM-dependent conditions can get scattered across multiple files in the code, making it labor intensive to accurately trace all of the DOM elements and relevant constraints. Random generation is simply not desirable because the required DOM tree may have a structure too precise for a random tree to match by chance. Thus the desired approach has to be automated, systematic and precise.

1.2 Contributions

The following are main contributions of our paper:

- We propose our automated, generic, transparent and browser-independent approach for systematically generating HTML to test JavaScript code that contain DOM operations.
- We describe how JavaScript code and its execution can dynamically be analyzed for deducing constraints relevant for generating HTML.
- We illustrate how extracted constraints can be solved into a satisfying DOM tree.
- We present the implementation of our approach in a tool called ConcolicDOM; an online video provides a demonstration.
- We report how ConcolicDOM and its generated DOM trees can help test suites improve coverage, reach complete execution, and have all their assertions done.

ConcolicDOM augments approaches that aim to generate tests automatically. Random testing [], feedback directed testing [], mutation testing [], concolic testing []... to our best knowledge, almost all of existing research focused on generating input parameters for testing functions or HTML inputs[2] for testing apps. However, having just function parameters and HTML inputs is often insufficient. For example, in a Web app, a properly satisfied dependency such as the DOM is often necessary for test cases and assertions to reach complete execution.[3] Moreover, it should be noted that the `function checkRows()` does not take any input arguments, and many functions are like that in JavaScript. Yet, these functions depend on, and would sometimes mutate, their dependencies such as the DOM.

ConcolicDOM has integration with QUnit [4] so that existing test suites can automatically take advantage of ConcolicDOM without additional manual effort. ConcolicDOM is also extensible to be integrated with other test frameworks [3]. Thus given a test case, be its inputs were generated manually or automatically, ConcolicDOM can be used to help the test case and its assertions get fully utilized.

2. CHALLENGES

While trying to generate satisfiable DOM trees, we had to resolve the following challenges:

Greedy does not always work. An intuitive approach would be to generate DOM elements "just in time"; however, such naive approach does not always work. Just in time generation is to greedily create whatever DOM elements necessary for satisfying the current single DOM operation. For example, in Sample Code 1, whenever `getElementById()` is called, we could just create and return an ad-hoc DOM element having the corresponding id. When we see `(row.children.length === 10)`, we could additionally create 10 ad-hoc children for the `row`.

The problem is that other DOM operations may contradict the ad-hoc DOM tree. A counter example we discovered very early is by just loading Wikipedia [13]. While loading the webpage, it executes the jQuery `$("#B13_120517_dwrNode_enYY")`, which is to get an element by a specific id. Then, some time later, the webpage calls `$("div#B13_120517_dwrNode_enYY")`, which is to get a <div> element by the exact same id. While the two jQueries may be written by different developers, we can easily see that the greedy approach does not work because it does not consider DOM operations in other parts of the code. when trying to satisfy `$("#B13_120517_dwrNode_enYY")`, how do we know a <div> is the correct tag type to generate in the first place? If the generated DOM element is not a <div>, then the 2nd jQuery would return `null`. There can be many different possibilities for satisfying a single current DOM operation; and picking the correct answer out of the many possible ones requires collectively tracing relevant DOM operations.

Indirect Influence. While executing JavaScript code would subtly or passively imply the DOM having a particular tree structure (as in the jQuery example), sometimes the structure of the DOM tree would actively determine which branch a condition would go into. For example, in Sample Code 1, we see that whether a `row` has 10 children would actively determine whether the `if` condition would go into the `true` or `false` branch. While the DOM dependence is obvious in Sample Code 1, very often DOM operations may not directly appear inside a condition: the result of different DOM operations may get assigned to multiple variables at various execution stages prior to the condition, either within the same function or up in the runtime stack. For example, a condition may appear as a simple `if(a)` in the code; yet the variable `a` can be `(row.children.length === 10)` or something more complex, such as the result of multiple statements executed throughout the code. We have to backward

[2]by HTML inputs we include inputs for HTML text fields, forms, buttons mouse clicks and key presses.
[3]Another category of dependencies is closure variables.

```
function DOMlogicExample() {
  // ...
  if (d === a.firstElementChild // i)
   || d === b.lastElementChild) {}
  // ...
  if (d === a.parentElement // ii)
   || d === b.parentElement) {}
  // ...
}
```

Sample Code 2: Example code showing conditions that have logical constraints interdependent with each other.

slice variables so that we can accurately determine what DOM operations a condition may depend upon.

Dynamic Typing. JavaScript variables are dynamically typed. Thus given a variable, we won't know exactly what type its value represents until we actually run the code. In our previous example, `a` can be anything: an integer, a string, a boolean, or an object. Static analysis by itself is insufficient to detect which lines of code are DOM related. Indeed, authors of existing JavaScript static techniques [2, 14] reveal substantial gaps and false positives in their own work. Thus the only way to discover whether a condition contains DOM operations is to run the code. Both tracing and backward slicing have to be dynamic to accurately determine which conditions depend on the DOM, and which don't.

Interdependent Logical Constraints. Given a dynamic trace and a dynamic backward slice deduced from the logs of decorated execution, we would have a clear mapping between DOM operations and conditions; thus a logical approach would be to generate a DOM tree directly from the trace and backward slice. However, an heuristic approach may not always work because a condition may have logical constraints that are interdependent on logical constraints in other conditions. For example, 2 of the conditions in Sample Code 2 inter-depend on each other because of the DOM policy that a DOM element cannot be both a child and a parent of another DOM element. Specifically, sub-conditions `i` and `ii` must be mutually exclusive because `d` cannot be both a child and parent of `a`. Therefore, when we want both of these `if` conditions to be `true`, a DOM specific solver is required to understand the unique policies of the DOM and make decisions accordingly for generating a proper satisfying HTML.

DOM mutations. Mutations to the DOM tree structure must be accounted for in both the backward slicing and the solver because changes to the HTML can happen any time during execution. Example mutations include adding or deleting a DOM node (e.g. in the use case of refreshing an email Inbox or deleting a message), or modifying the content or attributes within a DOM node. Expressing DOM mutations can be more challenging than expressing numerical operations (e.g. additions and subtractions), because DOM mutations are more diverse, and the DOM is a tree structure.

3. APPROACH

Each DOM operation in any part of the code is like a piece of a puzzle describing a subset clue of the overall DOM tree. ConcolicDOM has to systematically extract these puzzle pieces and analyse them collectively for generating a satisfiable DOM.

```
1  function DecoratedExecution() {
2    // 2 lines of original code
3    var a = row.children.length === b;
4    if (a) {}
5    // ...
6    // decorated version of original code
7    var a = _SHEQ(_GET(_GET(row, "
        children"), "length"), b);
8    if (__condStart()) {}
9    // ...
10 }
```

Sample Code 3: Example showing how code is decorated for logging execution and using the trace to construct a dynamic backward slice

Dynamic Backward Slicing. In a condition, dynamic backward slicing [] is required to discover what DOM operations a condition has. During execution, given a variable at a point in time, a dynamic backward slice traces how the variable has arrived at its current value: what operations or calculations had been previously done. For example, if the variable `a` equals to `row.children.length === b` at line 6 during execution, `a`'s backward slice would be backward slices of `row.children.length` and `b`, linked by the strict equal `===` operator. Eventually a dynamic backward slice would lead us to original inputs, constant values, and environmental dependencies such as the DOM. For example, a dynamic backward slice of `row` would lead us to the DOM element with ID `"row"+i`, where `"row"` is a constant string, and `i` has a backward slice leading to `field.children.length`, which would lead us to the DOM element with ID `"field"`.

Decorated Execution. To generate a dynamic backward slice, we must accurately capture the complexity and dynamic precedence of conditions in execution; and decorated execution is a simple and efficient way of doing the capture. Sample code3 illustrates the semantics of decorated execution. A condition is made up of sub-conditions, and a sub-condition can be seen as variables being compared to one another.

Dynamic backward slicing first requires logging the runtime execution and our logging approach is similar to Jalangi [11]'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 dataâĂŹs current value. While it can also be used for concolic testing, JalangiâĂŹs shadow system is mainly aimed at record and replay. Each condition is composed of 1 or more sub-conditions nested inside or linked beside other sub-conditions. Each sub-condition is composed of 1 or more variables being compared to other variables.

DOM Solver.

Conditional Slicing.

Integration with QUnit and Selenium.

Limited Path Coverage.

4. IMPLEMENTATION

```
function DecoratedExecution() {
  // 2 lines of original code
  var a = row.children.length === b;
  if (a) {}
  // ...
  // decorated version of original code
  var a = _SHEQ(_GET(_GET(row, "
      children"), "length"), b);
  if (__condStart()) {}
  // ...
}
```

Sample Code 4: Example showing how code is decorated for logging execution and using the trace to construct a dynamic backward slice

Architecture & Workflow End to end Dynamic, automatic generation of DOM

Pre condition

Post condition:

Components Proxy: WebScarab, Java Abstract Syntax Tree: Google Closure compiler Backward Slicing: JavaScript library, analyze execution tree DOM Solver: extended version of SMT (CVC3), Java API to translate SMT output into XML. Selenium: runs on multiple browsers, including headless browsers such as PhantomJS. QUnit: ConcolicDOM is designed to be easily extensible to other testing frameworks including Jasmine and Mocha.

Indexing Functions.

eval(), inline and native code.

5. FUTURE WORK

Indeed, the majority of JavaScript bugs are DOM related [7]. Ultimately, our higher level goal is to foster closer collaboration among designers and developers.[4] For example, because ConcolicDOM generates reference HTML for satisfying code execution, it can be used to detect DOM mismatches between the designer's HTML vs. what is expected in the developer's JavaScript code. A mismatch may not always be the developer's fault. Sometimes it is possible that the designer may have made a mistake when updating the HTML or may be just too busy and have forgotten to notify the developer about a change.

6. RELATED WORK

Whether the test inputs are manually, randomly (e.g. [1]) or symbolically (e.g. [9, 11]) defined.

Concolic Execution Concolic execution [12], also known as dynamic symbolic execution, is a method of exhaustively executing the source code for maximizing path coverage. A path is a sequential permutation of branches. For example, each IF statement has 2 branches: True and False; each iteration within a loop also has 2 branches: Stay and Break. Execution of branches is mutually exclusive: going to True implies not going to False. Having the constraints generated from a dynamic backward slice, concolic execution uses a constraint solver to generate input that would drive the execution of each condition towards a specific branch.

[4]As part of separating concerns, design and development are often done by distinct individuals having very different backgrounds.

Kudzu [9] 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].

7. CONCLUSION AND FUTURE WORK

element0.children.length - 3

8. REFERENCES

[1] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 571–580, New York, NY, USA, 2011. ACM.

[2] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for ajax intrusion detection. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 561–570, New York, NY, USA, 2009. ACM.

[3] A. Hidayat. Test frameworks in javascript. `https://github.com/ariya/phantomjs/wiki/Headless-Testing`.

[4] jQuery Foundation. Qunit: A javascript unit testing framework. `http://qunitjs.com/`.

[5] S. Khan. Khan academy. `http://www.khanacademy.org/`.

[6] Microsoft. Bnsf railway co. moves its mobile workforce to the cloud. `http://www.microsoft.com/en-us/news/press/2013/nov13/11-06bnsfcustomerspotlightpr.aspx`.

[7] F. S. Ocariza Jr, 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*, 2013.

[8] S. O'Grady. The redmonk programming language rankings: June 2013. `http://redmonk.com/sogrady/2013/07/25/language-rankings-6-13/`.

[9] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[10] J. Seidelin. Domtris: A tetris clone made with dom & javascript. `http://www.chromeexperiments.com/detail/domtris/`.

[11] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A tool framework for concolic testing, selective record-replay, and dynamic analysis of javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 615–618, New York, NY, USA, 2013. ACM.

[12] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held*

Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[13] Wikimedia. Wikipedia, the free encyclopedia. `http://en.wikipedia.org/`.

[14] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 805–814, New York, NY, USA, 2011. ACM.