ConcolicDOM: Concolic Generation of HTML for Testing JavaScript

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

As Web applications become more prevalent in our daily lives, quality assurance of Web applications has also become more important. Considerable JavaScript code is written to access and update a Web application's user interface through the Document Object Model API. The DOM models the UI in a tree structure. In this paper we present our generic and browser indepdendent approach for concolic generation of DOM trees for testing JavaScript Web applications. Testing Web applications remains a challenge because executing different parts of JavaScript code requires different yet specifically precise DOM tree structures. If there is any mismatch between the code and the DOM, e.g. when a DOM operation fails, entire code execution would eventually halt and the test would terminate prematurely. To overcome these challenges, we apply concolic techniques to generate HTML. We designed a DOM solver to support the 2D structure of the DOM tree, to infer implicit clues from DOM operations that are partial and incomplete, and to tailor the DOM tree for targeting precise subsets of the code base. We also implemented an end to end automatic system from deducing constraints to generating HTML and driving test execution because the number of unique DOM trees can grow as exponentially as the number of execution paths. We conducted a case study on the DOMtris application in which we will show how our approach significantly improves path coverage that includes a part of JavaScript code that implements a core functionality of the application.

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

Test runnability, applied concolic execution, Document Object Model (DOM), JavaScript, HTML

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 is relatively easy to adopt, and JavaScript does not have the burden of memory management and static typing. More operating systems in both the desktop [12, 20] and mobile [2, 3, 7, 17, 26, 34] actually now support installing and running JavaScript apps on the OS similar to native apps. CSS3 and the responsive UI paradigm enable UI designers to use a single HTML and still support multiple devices of different screen sizes. 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. [18]). Emergence and scalability of Node.js also make JavaScript widely adopted on the server side. Consequently, many institutions such as the Khan Academy [15] use JavaScript for teaching programming; and JavaScript has consistently been a top 2 in the RedMonk [22] popularity rankings.

Yet, despite the language's promise and ubiquity, testing JavaScript is not easy. For example, because HTML describes the graphical user interface of a Web app, considerable 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 operations on the DOM API (DOM operations) that would subtly imply the DOM tree (and thus the webpage's HTML) to have a particular 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. Indeed, a recent empirical study [21] of bug reports has concluded that a majority of bugs in JavaScript Web applications are DOM related. Being able to fully test JavaScript code that contain DOM operations would be critical in assuring the quality of a Web application.

Motivating Example. To further illustrate the necessity of having a satisfiable DOM structure, suppose we conduct

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

```
 1 function checkRows() {
 2   var field = getElementById("field");
 3   var i, row;
 4   for (i=field.children.length; i--;) {
 5     row = getElementById("row"+i);
 6     if (row.children.length === 10) {
 7       ++score;
 8       // ... row filled, update score
 9     }
10   }
11 }
```

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()`.

concolic unit-testing on the `function checkRows()` in Sample Code 1. The function is simplified from a feature Chrome Experiment [30] that uses the DOM to implement the game Tetris.

Concolic testing [33], also known as dynamic symbolic testing, would execute the app in a way to maximize path coverage. A path is a sequential permutation of branches. For example, an `if` statement has at least a `True` branch and a `False` branch. A loop has 2 branches, `Stay` and `Break`. The condition inside the statement or loop decides which branch the execution would follow. Maximizing path coverage would be to generate inputs for executing every possible permutation of condition branches. In the `function checkRows()` in Sample Code 1, we must visit both the `True` and `False` branches of the `if` statement in `line 6`.

If we intend an execution path to visit the `True` branch of the `if` statement at `line 6` of Sample Code 1, the Web application must have a DOM tree structure satisfying many constraints:

- There is an element with ID `"field"` (`line 2`).
- `field` contains children elements, so that we can first enter the `for` loop (`line 4`).
- There are rows having ID's in the nomenclature `row0`, `row1`, etc (`line 5`).
- The number of rows must be greater than or equal to the number of children that `field` has. The reason is that the ID of each `row` is made distinct by `i` (`line 5`). According to the `for` loop, `i--` goes from `field.children.length` to 1 (`line 4`).
- At least one of the rows must have exactly 10 children (`line 6`).

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 the DOM tree does not have an element with ID `"field"`, the variable `field` would have a `null` value. Consequently, the property access `field.children` would result in a `Type Error` and the rest of the function cannot be run or tested. Thus a satisfying DOM tree must be generated so that the function's execution would not crash and would visit the specific path that we intend.

While manual generation of DOM trees is possible, the manual approach would quickly become tedious and not scalable. The reason is that a unique structure of the DOM tree is required for going through a different execution path. For example, to go to the `False` branch of the above `if` statement, rows cannot have 10 children. Therefore, to cover both the `True` and `False` branches of the `if` statement in `line 6`, we must generate 2 unique DOM trees. Generally for an `if` statement, we have to consider generating an unique DOM tree for each of the `true` branch, all `else if` branches, and the `else` branch. Loops are more difficult for achieving path coverage, because generally it is not easy to determine what the upper limit is for iterating a loop. For example, `field` can have any number of children in Sample Code 1. Thus the loop can get iterated any number of times.

Nevertheless, the number of unique DOM structures would at least double whenever we try to cover an additional DOM-dependent condition, be the condition is inside an `if`, a `switch` 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 precisely targeted.

Contributions. The following are the main contributions of the Research Proficiency Evaluation (RPE) report:

- We propose an automated, generic, transparent and browser-independent approach for systematically generating DOM tree structures to test JavaScript code that contains DOM operations.
- We describe how JavaScript code and its execution can dynamically be analyzed for deducing constraints relevant for generating DOM trees.
- We design a novel DOM solver to support the DOM's 2D tree structure, to infer implicit clues from DOM operations that are partial and incomplete, and to tailor the DOM tree for targeting precise subsets of the code base.
- We present an implementation of our end to end automatic system from deducing constraints to generating HTML and driving test execution.
- We report how ConcolicDOM and its generated DOM trees can help test suites improve coverage and reach complete execution. If a function cannot be fully executed, the test case's assertions cannot be fully run.

ConcolicDOM augments approaches that aim to generate tests automatically. Random testing (e.g. [4]), mutation testing (e.g. [19]), concolic testing (e.g. [1, 9, 28, 32, 33])... to our best knowledge, almost all of existing research on test generation focused on generating 1) function arguments for unit testing, or 2) events and UI inputs (e.g strings for text fields and forms; mouse clicks for buttons and selection boxes; and key presses) for application-level testing. However, having just the function arguments, events and UI 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.

Moreover, it should be noted that the `function checkRows()` does not take any input arguments; and functions without input arguments are common in JavaScript. Yet, these functions depend on major dependencies such as the DOM. Thus even when we have a test suite that is very well defined and can potentially yield a very high coverage, whether through manual or automatic generation, considerable JavaScript code cannot get properly tested or covered unless the corresponding DOM structure is properly defined. ConcolicDOM provides an automated and systematic solution for generating satisfiable DOM trees in HTML form.

2. CHALLENGES

While trying to generate satisfiable DOM trees, we encountered challenges that motivate us to pursue a concolic approach. In the concolic approach, we also had to resolve challenges that are associated with supporting the 2D tree structure of the DOM.

1. Single Constraints are Partial and Incomplete. Each DOM operation provides a single constraint or clue to a subset of the overall DOM tree structure. An intuitive approach would be to generate DOM elements "just in time". However, such naïve approach does not always work because each single partial clue is insufficient.

A DOM operation is an operation on the DOM API. In the JavaScript language, a DOM operation is a property access or a method call to a JavaScript object that is associated with the DOM. For example, the method call `document.getElementById("field")` is a DOM operation to get the element having ID `"field"` from the DOM tree.

The actual constraint or clue of the DOM operation would depend on the branch that we intend the execution to follow. For example, if there is a condition checking `if (field===null)`, and we intend the execution to follow the `true` branch of this condition, then the constraint is that the DOM tree must not have a DOM element having ID `field`.

More often, the code would have other DOM operations that passively imply the requirement of `field`. For example, when the code has a property access (`field.children`) somewhere in the code after (`var field = document.getElementById("field")`), the code is indirectly implying a clue that it expects `field` to be an actual DOM element rather than `null`. Otherwise, e.g. when `field` is not a DOM element, executing the DOM operation `field.children` would return a `Type Error` and would terminate the program prematurely. Therefore, given a path that we intend the execution to follow, each DOM operation would directly or indirectly yield a subset clue about the structure of the overall DOM tree.

Just in time generation is to greedily create whatever DOM elements necessary for satisfying the current single DOM operation and intended execution path. For example, any time we encounter `document.getElementById()`, we could just create and return an ad-hoc DOM element satisfying the corresponding ID. When we see `(row.children === 10)`, we could additionally create 10 ad-hoc children for `row`.

The problem is the ad-hoc DOM tree may contradict DOM operations in other parts of the code. A counter example we discovered very early was by just loading Wikipedia [35]. While loading the webpage, the execution calls the jQuery function `$("#B13_120517_dwrNode_enYY")`. Calling the `$()` function with the `#` sign is to get an element by ID. In this case, the ID is `"B13_120517_dwrNode_enYY"`. Thus `document.getElementById("B13_120517_dwrNode_enYY")` would eventually get called.

To satisfy this DOM operation, we can simply create a DOM element with ID `"B13_120517_dwrNode_enYY"`. However, as the webpage continues to load, some time later the execution would call `$("div#B13_120517_dwrNode_enYY")`. This time the output is expected to be a <div> element by the exact same ID.

The second DOM operation can be satisfied *only* if we created a <div> element at the first place. There are many tag types to choose when trying to create a DOM element; any tag type other than <div> would make the ad-hoc DOM tree contradict the second DOM operation. The reason is that the tag type cannot be changed once a DOM element has been created. In this case, the two different calls to the `$()` jQuery function come from different parts of the code and may be written by different developers at different times. Thus the "just in time" greedy approach does not work because it does not consider DOM operations in other parts of the code.

Similarly, there can be many different possibilities when trying to satisfy a single DOM clue, and multiple DOM operations have to be collectively traced for complete analysis and overall tree generation.

2. Indirect Influence from Intermediate Variables. Assigning execution results to variables is a common practice in programming. In the code, it is possible that a condition may appear as a simple `if(a)`; yet the variable `a` can be the result of `(row.children.length === 10)`, or a more complex composition of prior executions. Adding to the complexity, these prior executions can happen anywhere at multiple lines scattered throughout the code. They can come from within the same function, or another function somewhere up in the runtime stack.

While the dependence on the DOM is directly obvious in the conditions of Sample Code 1, very often DOM operations may come from intermediate variables rather than directly appearing inside a condition. We have to backward slice variables so that we can accurately trace and precisely determine which DOM operations a condition may depend upon.

3. Dynamic Typing. JavaScript variables are dynamically typed. Thus given a variable, we would not exactly know what type the variable 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. Therefore, static analysis by itself is insufficient to detect which lines of code are DOM related. Indeed, authors of existing static techniques [13, 36] of analyzing JavaScript code 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 do not.

4. Logic Constraints can be Interdependent. Given a dynamic trace and a dynamic backward slice, 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 from other conditions. To compose an obvious example, we made 2 of the `if` conditions in Sample Code 2 inter-depend on each other. Specifically, the 2 sub-

```
1  // Interdependent Logic
2  // ...
3  if (d === elem.firstElementChild
4   || d === b.lastElementChild) {}
5  // ...
6  if (d === elem.parentElement
7   || d === b.parentElement) {}
8  // ...
9  if (b.previousElementSibling ===
10      c.firstElementChild) {}
11 // ...
12 if (elem.parentElement.parentElement
13     === c.lastElementChild.
           previousElementSibling) {}
14 // ...
```

Sample Code 2: Example code showing how DOM operations can have logical constraints that are interdependent with each other: `line 3` and `line 6`. To make all these `if` statements true the sub conditions in `line 3` and `line 6` become mutually exclusive: they cannot be true at the same time because `d` cannot be both a parent and a child of the same DOM element `elem`. A logic solver is required to generate a satisfiable DOM tree. Note that the final 2 conditions (`line 9` and `line 12`) would collectively influence the DOM solver to decide which sub condition (`line 3` vs. `line 6`) to become `true`.

conditions in **`line 3`** and **`line 6`** must be mutually exclusive because of the DOM policy that a DOM element (e.g. `d`) cannot be both a child and parent of the same DOM element (e.g. `a`). Therefore, when we intend an execution path to follow the **`true`** branch of both of these **`if`** conditions, we have to deploy a logic solver. The logic solver must know which of **`line 3`** and **`line 6`** to make **`true`**. To do so, the solver must be able to understand the unique policies of the DOM API and to make logic decisions accordingly for properly generating a satisfiable DOM tree.

5. 2D Tree Structure & Implicit Clues. While many logic solvers natively support single dimensional data types (integers, real numbers and strings), most do not natively support the Document Object Model's 2D tree structure. A challenge for designing a DOM solver is that the DOM solver must infer indirect or implicit clues before constructing the overall DOM tree. For example, when **`b.previousElementSibling === c.firstElementChild`**, the solver must infer that **`c`** is the parent of **`b`**.

This type of aliasing can get more complex because DOM operations can be chained. In JavaScript, a DOM operation on a DOM element (e.g. **`elem.parentElement`**) returns another DOM element. Thus chaining occurs when more DOM operations build on an existing DOM chain. For example, when we extend the chain **`elem.parentElement`** with another **`.parentElement`** DOM operation, **`elem.parentElement.parentElement`** returns the grandparent of **`elem`**.

In Sample Code 2, **`elem.parentElement.parentElement === c.lastElementChild.previousElementSibling`** (**`line 12`**). Therefore, the solver has to infer that **`b.nextElementSibling`** is also **`c.lastElementChild`** (Figure 1 Solved DOM), because the code also stated that (**`d === b.lastElementChild`**) and (**`d === elem.parentElement`**).

Once intermediate variables are resolved, the actual length of DOM chains can be observed and chains can span in both parental and sibling dimensions: e.g. **`elem.parentElement.parentElement.nextElementSibling.children[2].previousElementSibling`**.

A dynamic backward slice can contain multiple long chains. Aliasing can become complex because multiple subchains of varying length can refer or alias to the same DOM element. Thus the DOM tree's 2D structure and implicit clues are another reason for requiring a DOM specific solver. A non-solver approach would simply not work when it tries to generate a naïve tree (Figure 1 Naïve DOM) by grouping individual DOM chains and connecting them naïvely to a master root (the dashed lines in Figure 1).

```
1  // Before Instrumentation
2  var row = getElementById("row"+i);
3  var a = row.children.length === b;
4  if (a) {}
5
6  // After Instrumentation(i)
7  var row = _CALL(getElementById, _ADD(
      _CONST("STRING filename.js 0", "row"
      ), i));
8  var a = _SHEQ(_GET(_GET(row, "children"
      ), "length"), b);
9  /* a = {val: true
10        , op: _SHEQ
11        , 0:  {val: 10, op:_GET, ...}
12        , 1:  {val: 10, ...}}; */
13 if (__cond("IF_NAME filename.js 1", a))
      {}
```

Sample Code 3: Example showing how code is instrumented for dynamic analysis. The comment at line 9 shows the decorated object `a` and its nested tree data structure. `a`'s actual value is `true` because both left and right hand side have the same value 10: `line 11` and `line 12`

3. DOM CONSTRAINTS

Given a piece of JavaScript code, and a path that we intend the code execution to follow, our goal is to generate a DOM tree to guide and support the code execution.

As an overview, our approach is to first instrument the JavaScript code so that we can log the code's execution for producing a dynamic trace and a dynamic backward slice. Next we analyze the trace and the slice to extract DOM-specific operations and to deduce constraints, which the DOM solver would take as input for generating a satisfying DOM tree. The resulting DOM tree would be transformed into HTML, which we will insert into the webpage for guiding code execution.

Decorated Execution. Decorated execution is where we instrument the JavaScript code so that the execution of each JavaScript operator can be captured and decorated with additional data for producing a dynamic trace and a dynamic backward slice. Sample code 3 illustrates the semantics of decorated execution. A general rule of thumb is that we transform each use of a JavaScript operator (e.g. `.`) into a call to a corresponding operator function (e.g. `_GET()`). For

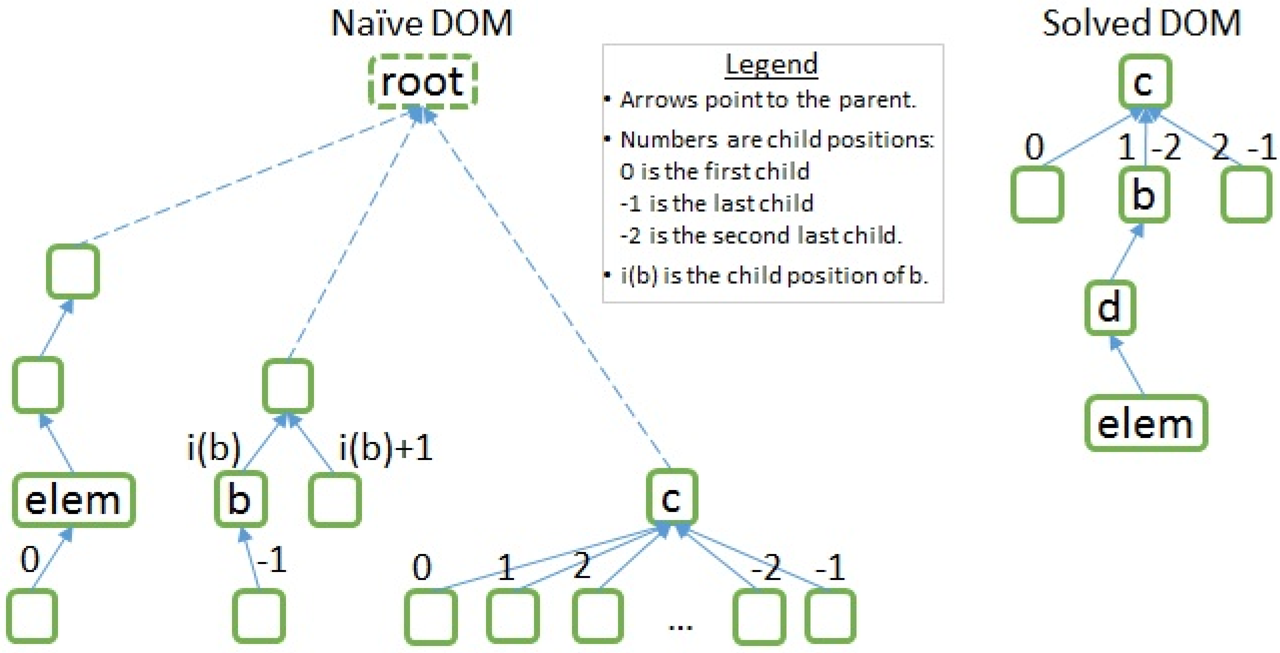

Naïve DOM
root
Legend
Arrows point to the parent.
Numbers are child positions:
0 is the first child
-1 is the last child
-2 is the second last child.
i(b) is the child position of b.
elem
0
i(b)
b
i(b)+1
-1
c
0
1
2
...
-2
-1
Solved DOM
c
0
1
-2
2
-1
b
d
elem

Figure 1: An naïve and the solver approach to generating a satisfiable DOM. The naïve DOM attempts and fails to satisfy the DOM operations by simply grouping individual DOM clues and connecting them with a single root (dashed lines). In the Solved DOM `b` is both child 1 (seocnd child) and child -2 (last child's previous sibling). `elem` does not have any child because `elem.firstElementChild` is actually in an `or` clause (Sample Code 2), in which the solver has decided to make the other sub clause `true`.

example, `row.children` becomes `_GET(row, "children")`. `_SHEQ` represents the strict equal operator (`===`). Each operator function wraps (thus decorates) the actual computed value inside a decorated object that also contains other data for tracing and slicing.

A special case happens when we transform the `&&` and `||` (`or`) operators, in which we have to consider the precedence of the operator's left hand side. For example, if the code is (`a && a.b`), the transformed version becomes `_AND(a, a.b)`; yet we do not want to execute `a.b` when `a` is `null` or `undefined`. A possible solution is to reuse `a`: `_AND(a, a && _GET(a, "b")`. However, the left hand side can be a call to a function that may change the internal state of the application: e.g. `appendLog() && update()`. Thus our solution is to assign the left hand side into a temporary variable, and then check the value of the temporary variable before executing the right hand side: `_AND(t = a, t && _GET(a, "b"))` and `_AND(t = _CALL(appendLog), t && _CALL(update))`.

Another special case is the `++` and `--` operators. For example, with `i++` we have to first assign the original value of `i` to a temporary variable before incrementing `i`, then we return the temporary variable.

Backward Slicer & Post Order Traversal. The data structure of the decorated objects can be seen as a nested or tree structure because the calls to the operator functions are nested inside one another. For example, in Sample Code 3, the call to `_GET(..., "length")` is nested inside the call to `_SHEQ()`. Therefore, if we simply put the name of the operator function (e.g. `"_GET"`, `"_SHEQ"`, ...), inside the trace data, we can easily generate a backward slice via a tree traversal.

In Sample Code 3, the variable `a` equals to (`row.children.length === b`). Thus `a`'s backward slice must contain the backward slice of `b` and the backward slice of `row.children.length`, linked by the strict equal (`===`) operator. At line 8, the decorated object returned by `_SHEQ()`, assigned to the variable `a`, is the tree parent of 2 decorated objects: `b`, and the decorated object returned by the `_GET()` call.

The tree children of a decorated object always come from earlier executions, e.g. `_GET(..., "children")` is executed before `_GET(..., "length")` before `_SHEQ(..., b)`. Thus the tree's hierarchical structure is reversely proportional to the temporal order in which the operator functions are executed.

During the traversal, we identify conditions that contain DOM operations and extract these DOM operations accordingly. In a chain of DOM operations, the operations closer to the chain head always come from earlier executions, thus the tree's hierarchy is also reversely proportional to the chaining order of DOM operations. The backward slicer traverses the decorated objects in post order, which is bottom up from leaf to root. This way, the dynamic backward slice not only yields a temporal history of the code's runtime execution, it also conveniently traces the DOM operation chains in the order from head to tail.

Each tree leaf represents an input or a constant. 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"`. Because variables can be used

```
% document.getElementById("field");
% document.getElementById("row"+0);
ASSERT DISTINCT(field, row0);

% (field.children.length)--;
ASSERT childrenLength(field) > 0;

% row.children.length === 10;
ASSERT childrenLength(row0) = 10;
```

Sample Code 4: Constraints for generating a DOM tree that would satisfy for going the `True` branch in the `if` statement of Sample Code 1. The constraints are shown in the input format for the CVC [6] implementation of the SMT solver. `%` is the comment operator in CVC.

```
<span id="field"><span></span></span>
<span id="row0">
  <span></span><span></span>
  <span></span><span></span>
  <span></span><span></span>
  <span></span><span></span>
  <span></span><span></span>
</span>
```

Sample Code 5: Example HTML generated from the results of the DOM solver based on the constraints defined in Sample Code 4. Note that `row0` is not a child of `field` because the source code in Sample Code 1 did not require the rows to be children of `field`.

multiple times, each variable can belong to more than 1 tree and can have more than 1 parent. Thus the data structure would appear more like a directed acyclic graph than a tree, even though a variable would never be a tree ancestor of any of its own ancestors.

Trace Mapper & Constraints Deducer. For each instance that a condition is executed, the backward slicer would yield what DOM operations the instance has and how these DOM operations are related or interdependent on one another. Because each condition can get executed more than once at different time points, the MapDeducer would aggregate all executed conditions, map them according to their ID, and deduce constraints for the DOM solver to generate a satisfiable DOM tree. So far everything is code-oriented in which we focus on each condition and its dynamic backward slice. The MapDeducer would transition the focus to be DOM-oriented in which we assemble clues about the same part of the DOM tree that are scattered across multiple lines of code. The MapDeducer would put together the processed clues across multiple parts of the DOM tree back together, into a single set of constraints for the DOM solver to generate a satisfiable DOM tree.

Sample Code 4 illustrates constraints for going to the `true` branch of the `if` statement in Sample Code 1, resulting in Sample Code 5. If we want to go to the `false` branch, e.g. `ASSERT NOT(childrenLength(row0) = 10)`, then the solver would generate a number of children not equal to 10 for `row0`. The exact number of children has not been deterministic based on our experiments: sometimes `row0` has 2 children, sometimes `row0` has none.

```
<span id="c">
  <span/>
  <span id="b">
    <span id="d">
      <span id="elem"/>
    </span>
  </span>
  <span/>
</span>
```

Sample Code 6: HTML generated for guiding the execution to follow the `true` branch in the `if` statement in Sample Code1.label

4. DOM SOLVER

The DOM solver takes the constraints defined by the MapDeducer and attempts to generate a satisfiable DOM structure. The solver is implemented as an extension of a SMT solver [6] and would report anything not satisfiable.

DOM Tree & DOM Operations. Recall a major part of the DOM is its single parent, multi-children tree structure. When generating a satisfiable DOM, we use the execution of DOM operations to infer the overall DOM tree. Each DOM operation in any line of code is like a piece of a puzzle describing a subset clue of the overall DOM tree. For example `a = elem.parentElement.nextElementSibling` implies 2 subset clues: `elem` has a parent element, and the parent has a sibling. Note that when the condition is `a = elem.parentElement.nextElementSibling === null`, then the clues become `elem` has a parent element, yet the parent has no next sibling and thus is the last child.

That said, questions remain unanswered about exactly where does `elem` fit in or belong in the overall DOM tree; and other DOM operations would provide clues for that. The DOM solver would take all the clues and generate a satisfiable tree structure.

DOM Operations into SMT Quantifiers. In the solver we transform each DOM operation into a SMT function. We then use quantifiers (e.g. `EXISTS`, `FORALL`) to define how the SMT functions relate to each other. Sample Code 7 shows the boolean functions and integer functions we defined for supporting the `elem.children.length` operation. We first quantify the parent-child relationship:

- a node cannot be a child of itself, see `line 1` and `line 4` in Sample Code 7.
- a child of a node cannot be the node's parent at the same time: `line 8`.
- a child can have only 1 parent: `line 13`.

Next, we define how children are ordered and quantify `children.length`:

- first child starts at position or index 0: `line 18` and `line 24`.
- last child has the largest child index: `line 30`.
- `children.length` equals to one plus the child index of the last child, because the first child starts at position 0: `line 40`.

The parent SMT functions are quantified as the inverse of the child SMT functions (e.g. `line 48`). Similar to `firstChild()` and `lastChild()`, the sibling SMT functions are defined by extending `children(x, y, j)`. For example, the next sibling of a node has the same parent and child index `j+1`, when the node has child index `j`.

```
% child(x, y): x is a child of y.
child: (Node , Node) -> BOOLEAN;

% x cannot be a child of itself.
ASSERT FORALL (x: Node):
  NOT(child(x,x));

% when y is the parent of x,
% then y cannot be a child of x.
ASSERT FORALL (x, y: Node):
  child(x,y) => NOT(child(y,x));

% a child has only 1 parent.
ASSERT FORALL (x, y, z: Node):
  (child(x,y) AND DISTINCT(y,z))
    => NOT(child(x,z));

% x is the j-th child of y.
children: (Node , Node , INT) -> BOOLEAN;
ASSERT FORALL (x,y:Node , j:INT):
  children(x, y, j) =>
    child(x, y) AND j >= 0;

% child position/index starts at 0.
firstChild:   (Node , Node) -> BOOLEAN;
ASSERT FORALL (x, y:Node):
  firstChild(x, y) <=>
    children(x, y, 0);

% every other child must have an index
% smaller than that of the last child.
lastChild: (Node , Node) -> BOOLEAN;
ASSERT FORALL (x, y:Node):
  lastChild(x, y) => EXISTS(j:INT):
    children(x, y, j) AND
    (FORCALL(z:Node , k:INT):
      (children(z, y, k) AND
      DISTINCT(z, x)) => k < j);

% children.length equals to 1 plus
% the child index of the last child.
childrenLength: (Node) -> INT;
ASSERT FORALL (y:Node , j:INT):
  childrenLength(y) = j <=>
    EXISTS(x:Node): (lastChild(x, y)
      AND children(x, y, j-1));

% example of inversion
% y is the parent of x, is the same as
% x is a child of y.
parent: (Node , Node) -> BOOLEAN;
ASSERT FORALL (x, y: Node):
  parent(y, x) <=> child(y, x);
```

Sample Code 7: SMT functions for defining the children.length DOM operation. We start with defining the parent-child relationships; then move on to the ordering of children; then use the child index of the last child to define and quantify the `childrenLength()` boolean function

Negation. Negations are supported by SMT solvers by default. To negate a constraint we wrap the constraint with `NOT()`.

Solver Output into HTML. Because we used Boolean functions and Integer functions, CVC is only going to solve for the interdependent logic constraints and it does not directly yield a concrete DOM tree. Instead, CVC only expands the quantifiers and it outputs more `ASSERT` statements for making the constraints satisfiable.

In Sample Code 2, the condition inside the first `if` statement has 2 sub conditions:

- `d === elem.firstElementChild` (`line 3`), and
- `d === b.lastElementChild` (`line 4`).

When the DOM solver solves this `if` statement for going to the `true` branch, it would output `"ASSERT lastChild(d, b);"` because it has decided on making the second sub condition (`line 4`) `true`.

To build the DOM tree, we built an API that parses the CVC output text and builds a model for the DOM. In the CVC output, CVC creates many temporary variable names, thus each DOM element can have more than one alias. To consolidate the aliases, we start with DOM operations expressing the parent child relationship because each child can have only 1 parent. We also used other deterministic DOM operations such as `firstChild()` and `lastChild()` to group aliases together. For example, if `x` is the first child of `y` and `z` is also the first child of `y`, then `x` and `z` are two aliases of the same DOM element.

Once the parent child relationship is established, our next step is to organize the ordering of children. Some DOM children have their positions explicit (e.g. `firstChild(x, y)`, `lastChild(x, y)` and `children(x, y, j)`), yet very often the child parent relationship is implied. For example, `nextSibling(x, z)` implies `x` and `z` share the same parent. To calculate the ordering of children, we use the explicitly positioned children as anchors and we relate the anchors to other children by the sibling operations.

5. IMPLEMENTATION

Concolic Driver. The concolic driver automates and coordinates the concolic process end to end from deducing constraints to generating HTML and driving execution. Specifically, the concolic driver would repeatedly

- Open the target URL in the Web browser
- Load the generated html (initially empty html)
- Execute the target JavaScript code
- Measure coverage, and decide which path to go next
- Call the MapDeducer, which returns the DOM constraints in text
- Send the constraints to the DOM solver, which returns a satisfiable DOM tree
- Go back to the first step with the newly generated HTML.

The concolic driver would iterate the cycle until it has reached a certain point, which can be configured to be a specific number of iterations (e.g. 1,000 DOM trees generated), a certain level of coverage (e.g. 100% branch coverage), or both.

Instrumenting and Executing JavaScript. Our approach has to be generic, transparent and browser-independent. We use Selenium's WebDriver [31] to drive Web browsers for executing JavaScript. WebDriver runs on multiple browsers, thus using WebDriver meets our design goal of being browser-independent.

When JavaScript is getting downloaded onto a Web browser, we use the WebScarab proxy [25] to intercept the download and to instrument code. The proxy passes intercepted JavaScript code to the Google Closure Compiler API [11], which transforms JavaScript into calls to the operator functions.

Both the Backward Slicer and MapDeducer are implemented as JavaScript APIs, and WebDriver natively supports calling JavaScript functions within the browser. Thus our approach is entirely transparent and can be applied to multiple brands of Web browsers.

Inline JavaScript & `eval()`. In addition to source files, JavaScript code can also be found within `eval()` and inlined as attributes of a DOM element inside the HTML declaration (e.g. `<body onload="runFunction()">`). We instrument each `eval(code)` statement into `eval(instrument(code))`. We do not override the native `eval()`, because the native `eval()` must be called inside the closure to give `code` access to the closure's local variables. The `instrument()` function would send the `code` to the proxy for instrumentation via a XMLHttpRequest.

To instrument inline JavaScript, we traverse the webpage's original existing HTML using the JSoup API [14]. Once we detect DOM attributes that contain JavaScript, we pass the JavaScript to the Google Closure Compiler API for instrumentation. For newly created elements, e.g. `elem.innerHTML = text`, we use getters and setters to detect the new elements. Once detected, we traverse the new elements, extract JavaScript and call the `instrument()` function.

DOM Solver. CVC allows writing the constraints in Java, yet we do not want to hardcode the constraints because the constraints are different for each execution path. Thus we use Java's ProcessBuilder [23] class to communicate with the executable (.exe) version of CVC. We decided to use CVC3 [6] rather than CVC4 [5] because CVC3 is generally more stable during our experimentation. The API that parses the CVC output is also implemented in Java, thus we used the W3C DOM API [24] for building a satisfiable DOM tree and generating HTML.

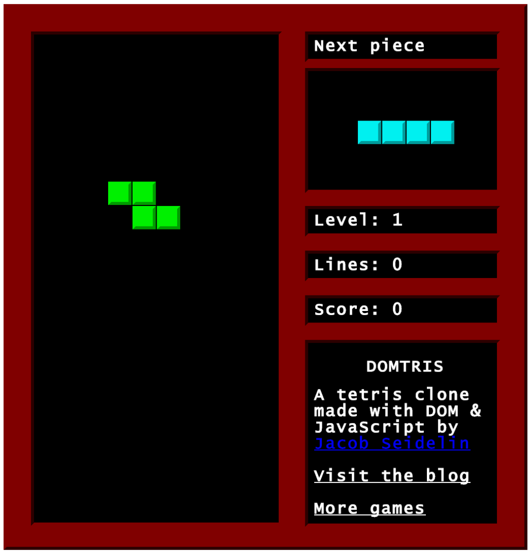
Limited Path Coverage. Very often we would not know how many times to execute a loop. For example, in Sample Code 1, there is no upper limit to the number of children that `field` can have. ConcolicDOM would execute loops zero, one, and `n` times, where `n` can be configured for a particular loop or for all loops. Thus ConcolicDOM would achieve limited path coverage rather than full path coverage.

Inserting HTML into Webpage. In HTML, certain DOM elements must be wrapped inside other DOM elements [27].

For example, table cells (`<td>` and `<th>`) have to be contained inside a table row (`<tr><td></td></tr>`). Table rows (`<tr>`) must be contained inside

- a table header: `<thead> <tr></tr>... </thead>`,
- a table body: `<tbody> <tr></tr>... </tbody>`, or
- a table footer: `<tfoot> <tr></tr>... </tfoot>`.

Similarly, every table header, table body, and table footer must be contained inside a table too: `<table> <thead>...</thead> <tbody>...</tbody> <tfoot>...</tfoot> </table>`. Each table contains at most 1 header, 1 body and 1 footer.



Level: 1

Lines: 0

Score: 0

DOMTRIS

A tetris clone made with DOM & JavaScript by Jacob Seidelin

Figure 2: The DOMtris game field has 20 rows, thus `field.children.length` is set to be 20 in our evaluation.

`<option>` and `<optgroup>` have to be inside a `<select multiple="multiple"> ... </select>`. `<legend>` has to be inside a `<fieldset> ... </fieldset>`. ConcolicDOM processes the generated HTML before inserting it into a webpage.

6. EVALUATION

Our evaluation would take the form of a case study in which we compare different approaches to testing the `function checkRows()` in Sample Code 1, and measure how much coverage each approach can achieve. Existing approaches to concolic testing JavaScript [28, 32] focus on generating input parameters. Yet the `function checkRows()` does not take any input arguments. Thus these approaches would likely execute the function by simply calling `checkRows()`. Therefore, in this case study we will call the `function checkRows()` in the following 3 scenarios and measure how much code is covered in each case.

- *Without HTML*
- *Existing HTML* from the application
- ConcolicDOM generated HTML

For each approach, we would follow the same methodology of loading and execution the function:

- Open the target URL in the Web browser
- Load the HTML (except *Without HTML*)
- Execute the target code
- Measure coverage

Recalling from the Implementation section, the 4 steps above are identical to the initial 4 steps that our concolic driver would take in each iteration. Additionally the concolic driver would take the executed paths as feedback and generate new HTML. For `function checkRows()`, we can simply call the function via `checkRows()` because it does not take any input arguments.

		Count (Number Of)		
App	Function	Statements	Branches	Paths
DOMtris	checkRows()	6	4	41

Figure 3: Number of Statements, Branches and Paths of function being evaluated.

Counting Execution Paths. Figure 3 shows the number of statements, branches and paths that the function has. When counting the number of paths in the `function checkRows()` in Sample Code 1, the original code does not pose an upper bound to the number of times the `for` loop would get iterated because `field.children.length` can be any value. Therefore, we would set `field.children.length` to a specific number for calculating the total number of possible execution paths in the function. `field` is set to have 20 children because the actual application always has 20 rows (Figure 2). The following shows how the number of statements, branches and paths are counted for the `function checkRows()`:

- *Statements*: 6, `line 2` to `line 7`, inclusive.
- *Branches*: 2 + 2. The `for` loop has 2 branches: `stay` and `break`; plus the `if` condition also has 2 branches: `true` and `false`.
- *Paths*: 20 (`stay` branch in for loop) * 2 (`true` and `false` branches in `if` statement) + 1 (`break` branch).

	Count (Number Of)		
Approach	Statements	Branches	Paths
Optimal	6	4	41
Without HTML	1	0	0
Existing HTML	5	3	1
ConcolicDOM	6	4	41

Figure 4: Statement, Branch and Path coverage of different approaches to testing the `function checkRows()`. The Optimal approach is stated here to reference what perfect coverage would look like.

Coverage Results. Figure 4 shows the coverage results of different approaches to testing the `function checkRows()`.

The *Without HTML* approach cannot cover any statement because the first statement in the `function checkRows()` is already a DOM operation requiring the existence of an element with ID `"field"`.

The *Existing HTML* approach is able to cover 5 statements inclusively from `line 2` to `line 6` because the original HTML already has 20 rows inside `field`. However the *Existing HTML* approach cannot cover the statement in `line 7` because the rows do not have any children at the start of the game. *Existing HTML* is able to cover 3 of the 4 possible branches: both the `stay` and `break` branches in the `for` loop, and the `false` branch in the `if` condition. Yet, the *Existing HTML* approach is able to cover only one path because going through a different path in `function checkRows()` requires another unique DOM tree structure.

ConcolicDOM is able to cover all possible paths in `function checkRows()` because we have set `field.children.length` to be 20, and all the conditions (the `for` loop and the `if` condition) inside the function are all driven by DOM operations. In case when there are conditions that are driven by other data types, our solver would have to be extended to support these data types.

Discussion. In the specific example of `function checkRows()`, using only the existing HTML is sufficient to achieve high statement coverage (5/6 or 83%) and high branch coverage (3/4 or 75%).

Concolic testing often takes additional time, in our small example the DOM solver would take about 15 to 30 seconds to generate each additional HTML on a MacBook pro with a 2.0GHz quad core Intel i7 processor and 16GB RAM. Thus whether concolic testing is worth it would depend on how much additional coverage the concolic approach would provide, and the benefits of having the additional coverage.

While ConcolicDOM covers only 1 additional statement (`line 7`) and only 1 additional branch (`true` branch of the `if` condition) compared to *Existing HTML*, the additionally covered statement is responsible for updating the game's score, and scoring is a core functionality of DOMtris. Another benefit of ConcolicDOM is that it covers all the different paths that generate the different permutations of increasing the score.

Our current concolic driver is primitive in the sense that it exhaustively tries to cover all execution paths possible. In the future, it may be worth investigating if it is possible to design a smarter concolic driver that can give priority to execution paths that would yield the highest marginal benefit of additional coverage: e.g. highest additional branch coverage, the most diverse permutation of increasing the score, etc. That way the tester can exercise concolic testing partially, combine concolic testing with other less expensive testing methods (e.g. [9, 16]), and still achieve certain desired outcomes without incurring the full costs associated with complete concolic execution. Thus having a prioritized concolic engine would give the tester greater control to manage the costs and benefits of the testing life cycle.

7. RELATED WORK

Concolic Testing. While there have been numerous publications on concolic testing (e.g. [33, 8, 1]), Kudzu [28] and Jalangi [32] are the only two for concolic testing software that are written in a dynamically typed language; and both focus on JavaScript.

A main contribution of Kudzu is a string solver that can handle select regular expressions. The solver is deployed to generate strings as UI inputs for detecting security vulnerabilities in JavaScript Web applications. A main contribution of ConcolicDOM is our DOM solver. Our solver generates HTML for running and testing JavaScript code that contains DOM operations. While DOM trees are usually represented in HTML string form, designing a DOM solver is different from designing a regex string solver. As discussed in the Challenges section, the main reasons are that the DOM solver has to support a 2D hierarchical tree structure while strings are usually single dimensional. Moreover, inferring implicit clues from DOM operations is also different from inferring regex patterns. In addition, the architecture of ConcolicDOM is implemented to run on multiple Web browsers, while Kudzu runs only on their single proprietary browser that supports Kudzu's component [29] for tracing and slicing.

A main contribution of Jalangi is their system of shadow values for selective record and replay. In the shadow system, they encapsulate each data value into an object; the encapsulated object can contain any metadata (the shadow) about the actual data value. While our system of decorated execution is similar to Jalangi's shadow system, a tester using Jalangi would manually identify which variables are inputs and would manually specify each input's type. Jalangi then generates various input values for those variables that are manually identified by the tester. In contrast, ConcolicDOM uses post-order tree traversal for automatically identifying possible inputs.

When ConcolicDOM instruments JavaScript, it would label each constant value as a constant. For example, the JavaScript statement `var x = "string";` would be instrumented into `var x = _CONST("string");`.

Therefore, during the post order traversal, if a tree leaf of the decorated objects tree is not labelled, the variable inside the leaf would be identified as a candidate input, because the data value does not come from within the code. ConcolicDOM generates a dynamic backward slice and thus executes the code, therefore the data type of candidate inputs can easily be determined from the actual data value.

Another differentiation is that for supporting the DOM we designed a Trace MapDeducer for extracting and transforming code-centric backward slice into DOM-centric constraints for the DOM solver. Both CUTE [33] and Jalangi use the CVC3 [6] solver for supporting integers and strings.

Constraint Solvers. Constraint solvers (e.g. SAT solvers) solve for parameters that satisfy a set of predefined constraints.

Genevès et al. developed an XML solver [10] that takes limited XPaths as inputs; then it outputs XML that would satisfy those XPath conditions. Initially we intended to extending the XML solver. However, after experimentations, we find it difficult to encode DOM node attributes into the XML solver and more importantly the XML solver is not scalable to more than 5 unique nodes.

CVC [6, 5] is a general purpose SMT solver and it is more scalable. However, being a general purpose solver also means that CVC does not natively support the tree structure defined in the DOM API. Our DOM solver uses quantifiers to encode and model the DOM within CVC. Nevertheless, the output of CVC yields only a description of the desired DOM tree (e.g. node A is child of node B), rather than the actual XML/HTML. Thus we have to take additional steps to transform CVC outputs into HTML.

Feedback Directed Testing. 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. Random testing and concolic testing are two major formats of feedback directed testing that is automated. Concolic testing is a form of feedback directed testing because it conducts backward slicing to generate inputs, and then it uses the resulting executed path as feedback for generating new inputs.

In random testing for JavaScript Web applications, Artemis [4] randomly generates initial inputs and uses the output of functions (rather than executed paths) as feedback, for increasing coverage. Pythia [19] 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. The number of functions being called is directly proportional to the number of lines covered. For covering JavaScript code that contains DOM operations, Pythia would use a Web application's existing HTML if such HTML is available. Thus Pythia does not generate new HTML for covering execution paths that conflict with the existing HTML. For example, when the `true` branch of an `if` statement requires a DOM element having 10 children, Pythia would never enter the `true` branch if the existing HTML does not contain 10 children for that DOM element.

8. FUTURE WORK & CONCLUSION

We presented a generic and browser indepedenent approach to testing and covering JavaScript code that contain DOM operations. DOM operations are abundant in the JavaScript code of Web applications and the majority of JavaScript bugs are DOM related. Our approach works in the JavaScript layer and supports the standard Document Object Model API defined by the standard body W3C the World Wide Web Consortium.

At this stage ConcolicDOM covers DOM operations that navigate the parent-child and sibling relationships of a DOM tree. In the future we would like to extend our solver to solve for mutations of the DOM tree, as well as integers and strings so that ConcolicDOM can support solving for attributes of DOM nodes. We also want to investigate if it is possible to design a smarter concolic driver so that the tester can achieve sufficient coverage without incurring the full costs of complete concolic execution.

Another direction of future work is to extend ConcolicDOM for fostering closer collaboration among designers and developers. As part of separating concerns, it is possible that designers of a development team may specialize mostly in the aesthetic aspects (HTML, CSS) of an application while the programmers specialize in the technical aspects. ConcolicDOM generates reference HTML for satisfying code execution. Therefore, when a designer wants to update the UI of a Web application, ConcolicDOM can help a designer to determine which DOM elements can be modified, which DOM elements can be renamed, which DOM elements must have a certain structure, and which lines of code to tell the programmer to pay attention to if the designer decides to change the DOM in a way that may break the original source code.

The Document Object Model API is the standard for interacting with mark up language documents. While this paper focused on testing front end JavaScript/HTML apps, our approach can be extended to cover XML operations in backend JavaScript and other programming languages.

9. REFERENCES

[1] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated Concolic Testing of Smartphone Apps. In *ESEC/FSE*, 2012.

[2] Apache. Apache cordova. `http://cordova.apache.org/`.

[3] AppleDeveloper. UIWebView Class Reference. `https://developer.apple.com/library/ios/documentation/uikit/reference/UIWebView_Class/Reference/Reference.html`.

[4] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A Framework for Automated Testing of JavaScript Web Applications. In *ICSE*, 2011.

[5] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, 2011.

[6] C. Barrett and C. Tinelli. CVC3. In *CAV*, 2007.

[7] BlackBerryDeveloper. BlackBerry WebWorks. `http://developer.blackberry.com/html5/documentation/beta/what_is_a_webworks_app.html`.

[8] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, 2008.

[9] P. Garg, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta. Feedback-directed Unit Test Generation for C/C++ Using Concolic Execution. In *ICSE*, 2013.

[10] P. Genevès, N. Layaïda, and A. Schmitt. Efficient Static Analysis of XML Paths and Types. In *PLDI*, 2007.

[11] GoogleDevelopers. Google Closure Compiler API. `https://developers.google.com/closure/compiler/`.

[12] GoogleDevelopers. Installable Web Apps. `https://developers.google.com/chrome/apps/?csw=1`.

[13] A. Guha, S. Krishnamurthi, and T. Jim. Using Static Analysis for Ajax Intrusion Detection. In *WWW*, 2009.

[14] J. Hedley. jsoup: Java HTML Parser. `http://jsoup.org/`.

[15] S. Khan. Khan Academy. `http://www.khanacademy.org/`.

[16] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, 2007.

[17] MDN. Firefox OS. `https://developer.mozilla.org/en-US/Firefox_OS`.

[18] MicrosoftPress. BNSF Railway Co. moves its mobile workforce to the cloud. `http://www.microsoft.com/en-us/news/press/2013/nov13/11-06bnsfcustomerspotlightpr.aspx`.

[19] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. PYTHIA: Generating test cases with oracles for JavaScript applications. In *ASE*, 2013.

[20] MSDN. Create your first Windows Store app using JavaScript. `http://msdn.microsoft.com/en-us/library/windows/apps/br211385.aspx`.

[21] F. S. Ocariza Jr, K. Bajaj, K. Pattabiraman, and A. Mesbah. An Empirical Study of Client-Side JavaScript Bugs. In *ESEM*, 2013.

[22] S. O'Grady. The RedMonk Programming Language Rankings: June 2013. `http://redmonk.com/sogrady/2013/07/25/language-rankings-6-13/`.

[23] Oracle. ProcessBuilder Class. `http://docs.oracle.com/javase/7/docs/api/java/lang/ProcessBuilder.html`.

[24] Oracle. W3C DOM API for Java. `http://docs.oracle.com/javase/7/docs/api/org/w3c/dom/package-summary.html`.

[25] OWASP. OWASP WebScarab Project. `https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project`.

[26] C. Raval. Develop Android Application using HTML, CSS and JavaScript. `http://stackoverflow.com/questions/6785875/develop-android-application-using-html-css-and-javascript`.

[27] J. Resig and B. Bibeault. *Secrets of the JavaScript Ninja*, chapter 14 Manipulating the DOM. Manning Publications Co., 2012.

[28] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *Proc. of IEEE Symposium on Security and Privacy*, 2010.

[29] P. Saxena, S. Hanna, P. Poosankam, and D. Song. An Empirical Study of Client-Side JavaScript Bugs. In *NDSS*, 2010.

[30] J. Seidelin. DOMTRIS: A Tetris clone made with DOM & JavaScript. `http://www.chromeexperiments.com/detail/domtris/`.

[31] SeleniumHQ. WebDriverJS User Guide. `https://code.google.com/p/selenium/wiki/WebDriverJs`.

[32] 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 *ESEC/FSE*, 2013.

[33] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *ESEC/FSE*, 2005.

[34] TizenDevelopers. Sample Web Applications. `https://developer.tizen.org/downloads/sample-web-applications`.

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

[36] Y. Zheng, T. Bao, and X. Zhang. Statically Locating Web Application Bugs Caused by Asynchronous Calls. In *WWW*, 2011.