ConcolicDOM: Generating HTML for concolic testing 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 operating systems in both the desktop [11, 19] and mobile [5, 16, 24, 31] actually now support installing and running JavaScript apps on the OS similar to native apps. 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. [17]). 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 [21] popularity rankings.

Yet, despite the language's promise and ubiquity, testing JavaScript is not easy. For example, because HTML

---

[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

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 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 of bug reports has concluded that a major of bugs in JavaScript Web applications are DOM related [20]. Being able to fully test JavaScript code that contain DOM operations would be critical in assuring the quality of a Web application.

1.1 Motivating Example

To further illustrate the necessity of having a satisfiable DOM structure, suppose we conduct concolic unit-testing on the `function checkRows()` in Sample Code 1. The function is simplified from a feature Chrome Experiment [27] that uses the DOM to implement the game Tetris.

Concolic testing [30], 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, each IF statement has 2 branches: True and False; each iteration within a loop also has 2 branches: Stay and Break. Maximizing path coverage would be to generate and execute every possible permutation of available branches. Therefore, we must visit both the `True` and `False` branches of each `if` statement in `function checkRows()` in Sample Code1.

If we intend an execution path to visit the `True` branch of the `if` statement in `line 6` of Sample Code1, the web page's HTML must yield a DOM structure satisfying many constraints:

- There is an element with id `"field"`.
- `field` 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 `field` has. The reason is that the ID of each `row` is made distinct by `i`. According to the `for` loop, `i--` goes from `field.children.length` to 1.
- 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

```
 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()`.

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 to yield a proper DOM structure so that the function's execution would not crash and would visit the path that we intend.

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 the main contributions of our paper:

- We propose an automated, generic, transparent and browser-independent approach for systematically generating HTML 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 HTML.
- We design a novel DOM solver for generating satisfiable a DOM tree that would yield an HTML for going into a specific execution path.
- We present the implementation of our approach in a tool called CONCOLICDOM.

- 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. [2]), feedback directed testing (e.g. [8]), mutation testing (e.g. [18]), concolic testing (e.g. [1, 25, 29, 30])... 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 check-Rows()` does not take any function 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.

2. CHALLENGES

Our preference is always to aim for the simplest solution. Yet, we encountered and had to resolve several challenges while trying to generate a satisfiable DOM:

1. Single DOM Clues are Incomplete. Each DOM operation provides a clue to the overall DOM tree. An intuitive approach would be to generate DOM elements "just in time". However, such naïve approach does not always work because each clue by itself is insufficient.

Just in time generation is to greedily create whatever DOM elements necessary for satisfying the current single DOM clue. 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 was by just loading Wikipedia [34]. While loading the webpage, it executes the jQuery code `$("#B13_120517_dwrNode_enYY")` to get an element by a specific id. Then, some time later, the webpage calls `$("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.

While each DOM clue is incomplete, there can also be many different possibilities for satisfying a single DOM clue. When trying to satisfy `$("#B13_120517_dwrNode_enYY")`, the DOM has so many tag types, how do we know <div> is the correct tag in the first place? The consequence for an incorrect tag type is that the 2nd jQuery would not return

any DOM element, leading to an error. Thus DOM operations have to be collectively traced for complete analysis and for generating a correct answer.

2. Indirect Influence & Intermediate Variables. While executing JavaScript code would subtly or passively imply the DOM having a particular tree structure, 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 compositions of more complex operations, containing the result of multiple statements executed throughout the code. We have to backward slice variables so that we can accurately determine what 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 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 [12, 35] 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. Interdependent Logic. 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. In an obvious example, 2 of the conditions in Sample Code 2, the 2 `if` statements, 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, the 2 sub-conditions in `line 3` and `line 6` 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.

5. 2D Tree Structure & Implicit Clues. While many solvers support single dimensional data types such as numbers and strings, most do not natively solve the DOM's 2D tree structure. A reason why solving for a 2D structure is challenging is because the DOM solver must be able to resolve policies and operations defined in the W3C DOM API [33]. An example is inferring indirect clues. Here, when `b.previousElementSibling === c.firstElementChild`, the

```
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`.

solver has to understand the implication that c is the parent of b.

Another example is the fact that the DOM clues can subtly overlap with each other. DOM clues can overlap because DOM operations can be chained. In JavaScript, a DOM operation of a DOM element (e.g. `elem.parentElement`) returns another DOM element. Thus DOM operations can be chained: `elem.parentElement.parentElement` returns the grandparent of `elem`.

In Figure 2, when `elem.parentElement.parentElement === c.lastElementChild.previousElementSibling`, the subtle implication is that `b.nextElementSibling` is `c.lastElementChild`, because `d === b.parentElement` and `d === b.lastElementChild`.

Overlapping can become complex after the intermediate variables are resolved into a long dynamic backward slice. The reason is that, there can be multiple long chains in both parental and sibling dimensions: e.g. `elem.parentElement.parentElement`

The multiple long chains can overlap and interweave with each other on multiple spots. Thus the 2D structure and indirect 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 (Naïve DOM in Figure 2) by grouping the DOM operations and then connecting the individual tree pieces to a master root (the dashed lines in Figure 2).

3. DOM CONSTRAINTS

Given a piece of JavaScript code, along with the execution path that we intend the code to go into, our goal is to generate HTML so that the HTML would yield a DOM tree structure that would guide and support the execution of the code being tested.
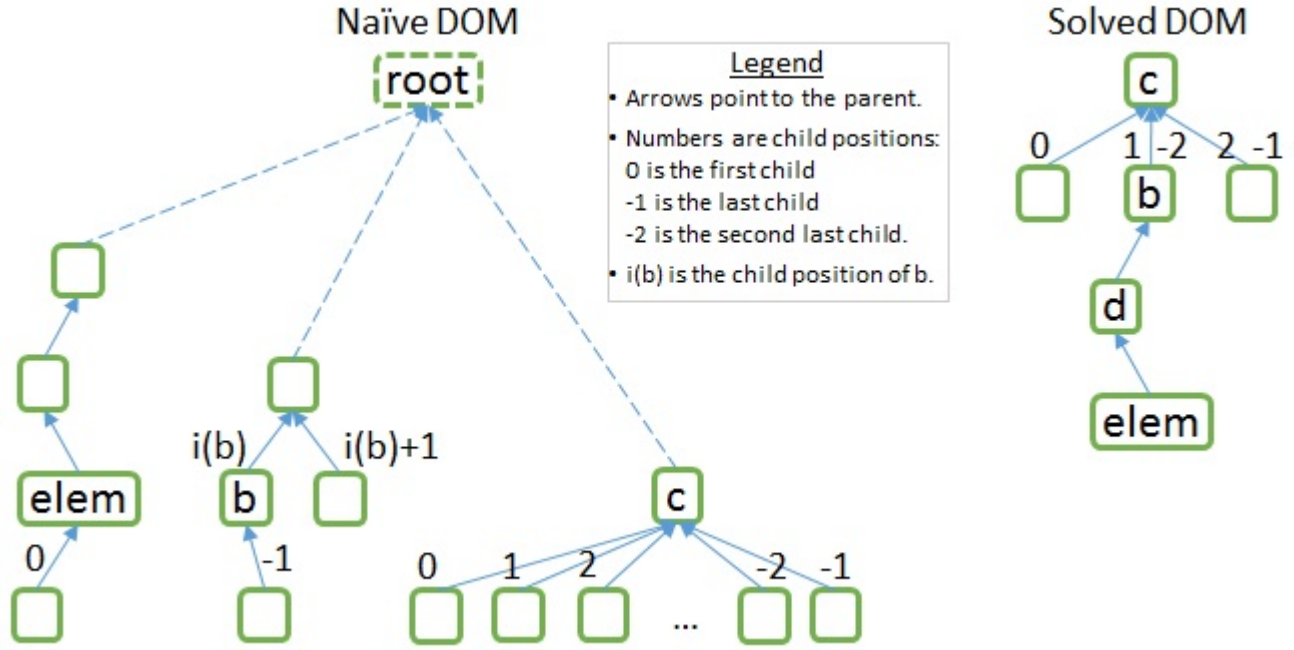
Figure 1: An naïve and the solver approach to generating a satisfiable DOM. The naïve DOM attempts and fails to satisfy the DOM operations by simply grouping individual DOM clues and connecting them with a single root (dashed lines). In the Solved DOM `b` is both child 1 (seocnd child) and child -2 (last 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`.

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 backward slice and a dynamic trace. 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 suitable DOM structure and corresponding HTML. Then we integrate the HTML into a test framework for running and asserting test cases.

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 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 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 appli-

```
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`

cation: 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 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 HTML. The MapDeducer works like MapReduce [7]. 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-

```
1  % document.getElementById("field");
2  % document.getElementById("row"+0);
3  ASSERT DISTINCT(field, row0);
4
5  % (field.children.length)--;
6  ASSERT childrenLength(field) > 0;
7
8  % row.children.length === 10;
9  ASSERT childrenLength(row0) = 10;
```

Sample Code 4: DOM constraints for generating an HTML 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 [4] implementation of the SMT solver. `%` is the comment operator in CVC.

```
1  <span id="field"><span></span></span>
2  <span id="row0">
3    <span></span><span></span>
4    <span></span><span></span>
5    <span></span><span></span>
6    <span></span><span></span>
7    <span></span><span></span>
8  </span>
```

Sample Code 5: Example HTML generated by 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`.

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 HTML.

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.

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 [4] and would report anything not satisfiable.

DOM Tree & DOM Operations. 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 if the condition is `a = elem.parentElement.nextElementSibling === null`, then

```
1  <span id="c">
2    <span/>
3    <span id="b">
4      <span id="d">
5        <span id="elem"/>
6      </span>
7    </span>
8    <span/>
9  </span>
```

Sample Code 6: HTML generated for going the all true path in Sample Code 2. The all true path is the execution path in which every condition goes to the true branch.

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 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`.

Negation. We use the CVC implementation of SMT, which natively supports booleans and integers by default. While it is possible to define a customized data structure to mimic the DOM tree in CVC, the boolean/integer function approach is simple and convenient for negating conditions and going into different execution paths.

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 that is valid for making the constraints satisfiable.

In Sample Code 2, to solve (`d === elem.firstElementChild || d === b.lastElementChild`), CVC would output "AS-

```
1  % child(x, y): x is a child of y.
2  child: (Node, Node) -> BOOLEAN;
3
4  % x cannot be a child of itself.
5  ASSERT FORALL (x: Node):
6    NOT(child(x,x));
7
8  % when y is the parent of x,
9  % then y cannot be a child of x.
10 ASSERT FORALL (x, y: Node):
11   child(x,y) => NOT(child(y,x));
12
13 % a child has only 1 parent.
14 ASSERT FORALL (x, y, z: Node):
15   (child(x,y) AND DISTINCT(y,z)
16     => NOT(child(x,z));
17
18 % x is the j-th child of y.
19 children: (Node, Node, INT) -> BOOLEAN;
20 ASSERT FORALL (x,y:Node, j:INT):
21   children(x, y, j) =>
22     child(x, y) AND j >= 0;
23
24 % child position/index starts at 0.
25 firstChild:   (Node, Node) -> BOOLEAN;
26 ASSERT FORALL (x, y:Node):
27   firstChild(x, y) <=>
28     children(x, y, 0);
29
30 % every other child must have an index
31 % smaller than that of the last child.
32 lastChild: (Node, Node) -> BOOLEAN;
33 ASSERT FORALL (x, y:Node):
34   lastChild(x, y) => EXISTS(j:INT):
35     children(x, y, j) AND
36     (FORCALL(z:Node, k:INT):
37       (children(z, y, k) AND
38       DISTINCT(z, x)) => k < j);
39
40 % children.length equals to the
41 % child index of the last child.
42 childrenLength: (Node) -> INT;
43 ASSERT FORALL (y:Node, j:INT):
44   childrenLength(y) = j <=>
45     EXISTS(x:Node): (lastChild(x, y)
46       AND children(x, y, j-1));
47
48 % example of inversion
49 % y is the parent of x, is the same as
50 % x is a child of y.
51 parent: (Node, Node) -> BOOLEAN;
52 ASSERT FORALL (x, y: Node):
53   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

SERT lastChild(d, b)" because it has decided on making (`d === b.lastElementChild`) `true`.

To build the DOM tree, we built an API that parses the CVC output 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. In the iterative process of concolic testing, we implemented a concolic driver that would repeatedly

1. Open the target url

2. Load the generated html (initially empty html)

3. Execute the target JavaScript code

4. Call the MapDeducer, which returns the DOM constraints in text,

5. Send the constraints to the DOM solver, which returns a satisfiable DOM tree

6. Go back to step 1 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[28] to drive Web browsers for executing JavaScript. WebDriver runs on multiple browsers, including headless browsers such as PhantomJS [14].

When JavaScript is getting downloaded onto a Web browser, we use the WebScarab proxy to intercept the download and to instrumenting code. The proxy passes intercepted JavaScript code to the Google Closure Compiler API [10] for transforming 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.

DOM Solver. CVC allows writing the constraints in Java, yet we do not want to hardcode the constraints because the constraints are different in each execution path. Thus we use Java's ProcessBuilder [22] class to communicate with the executable (.exe) version of CVC. We decided to use CVC3 rather than CVC4 [3] 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 [23] for generating HTML.

Limited Path Coverage. Very often we would not know how many times to execute a loop. For example, in Sampe 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 loops or for all loops. Thus CONCOLICDOM would achieve limited path coverage rather than full path coverage.

Indexing Functions. Most of the time JavaScript functions are defined inside closures and are not accessible [32]. When we instrument JavaScript code, we extract the functions by assigning them to an object that we have access to. Because functions can have the same name, we use the node number in the JavaScript Abstract Syntax Tree as ID.

`eval()` & inline JavaScript. 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 cannot 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 original HTML using the JSoup API [13]. Once we detect DOM attributes that contain JavaScript, we pass the JavaScript to ClosureCompiler 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.

6. EVALUATION

Our evaluation would take the form of a case study in which we would compare different approaches to testing the `function checkRows()` is Sample Code**??**, and measure how much coverage each approach can achieve.

checkRows() without HTML, see how far it goes checkRows() with original HTML, see how far it goes checkRows() with Concolic Driver, see how far it goes

Set field.children.length to 10,

7. RELATED WORK

Concolic Testing. While there have been numerous publications on concolic testing (e.g. [30, 6, 8, 1]), Kudzu [25] and Jalangi [29] 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 contain DOM operations. While HTML are usually represented in string form, designing a DOM solver is different from de-
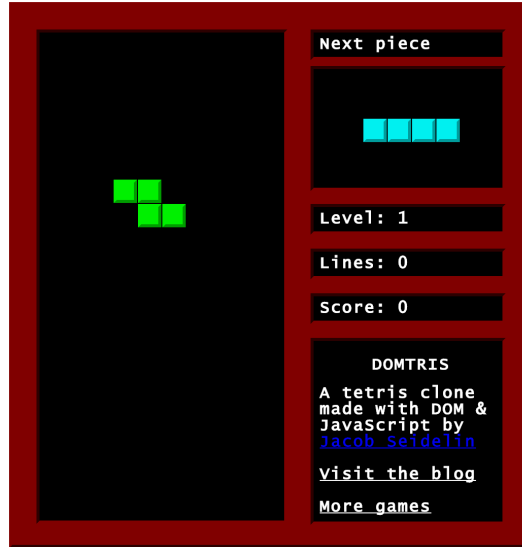
Figure 2: The DOMtris game field (left) has 21 rows; thus `field.children.length` should be 21.

signing 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 on only the single browser that supports its component [26] 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 test engineer using Jalangi would manually identify which variables are inputs and would manually specify each input's type. Jalangi then generates inputs for those variables manually identified by the test engineer. 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: e.g. 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.

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 [30] and Jalangi use the CVC3 [4] 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 [9] 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 the XML solver is not scalable to more than 5 unique nodes.

CVC [4, 3] 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 [2] randomly generates initial inputs and uses the output of functions (rather than executed paths) as feedback, for increasing coverage. Pythia [18] 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 contain 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. CONCLUSION

9. REFERENCES

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

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

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

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

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

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

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] 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.

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

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

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

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

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

[14] A. Hidayat. PhantomJS. `http://phantomjs.org/`.

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

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

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

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

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

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

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

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

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

[24] C. Raval. Develop android Application using Html,Css and JavaScript. `http://stackoverflow.com/questions/6785875/develop-android-application-using-html-css-and-javascript`.

[25] 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.

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

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

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

[29] 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.

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

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

[32] R. Valstar. Unit testing private functions in Javascript. `http://www.sjeiti.com/unit-testing-private-functions/`.

[33] W3C. Document Object Model (DOM). `http://www.w3.org/DOM/`.

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

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