2015 30th IEEE/ACM International Conference on Automated Software Engineering

Generating Fixtures for JavaScript Unit Testing

Amin Milani Fard
University of British Columbia
Vancouver, BC, Canada
aminmf@ece.ubc.ca

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

Eric Wohlstadter
University of British Columbia
Vancouver, BC, Canada
wohlstad@cs.ubc.ca

***Abstract*—In today's web applications, JavaScript code interacts with the Document Object Model (DOM) at runtime. This runtime interaction between JavaScript and the DOM is error-prone and challenging to test. In order to unit test a JavaScript function that has read/write DOM operations, a DOM instance has to be provided as a test fixture. This DOM fixture needs to be in the exact structure expected by the function under test. Otherwise, the test case can terminate prematurely due to a null exception. Generating these fixtures is challenging due to the dynamic nature of JavaScript and the hierarchical structure of the DOM. We present an automated technique, based on dynamic symbolic execution, which generates test fixtures for unit testing JavaScript functions. Our approach is implemented in a tool called ConFix. Our empirical evaluation shows that ConFix can effectively generate tests that cover DOM-dependent paths. We also find that ConFix yields considerably higher coverage compared to an existing JavaScript input generation technique.**

Keywords*—*Test fixture, test generation, dynamic symbolic execution, concolic execution, DOM, JavaScript, web applications

I. Introduction

JavaScript has grown to be among the most popular programming languages. For instance, JavaScript is now the most prevalent language in GitHub repositories [18] and a recent survey of more than 26K developers conducted by Stack Overflow found that JavaScript is the most-used programming language [38]. To create responsive web applications, developers write JavaScript code that dynamically interacts with the Document Object Model (DOM). The DOM is a tree-like structure that provides APIs [40] for accessing, traversing, and mutating the content and structure of HTML elements at runtime. As such, changes made through JavaScript code via these DOM API calls become directly visible in the browser.

This complex interplay between two separate languages, namely JavaScript and the HTML, makes it hard to analyze statically [22], [26], and particularly challenging for developers to understand [12] and test [14], [31] effectively. As revealed in a recent empirical study [33], the majority (65%) of reported JavaScript bugs are *DOM-related*, meaning the fault pertains to a DOM API call in JavaScript code. Moreover, 80% of the highest impact JavaScript faults are DOM-related.

In order to unit test a JavaScript function that has DOM read/write operations, a DOM instance needs to be provided in the exact structure as expected by the function. Otherwise, a DOM API method (e.g., `var n = getElementById("news")`) returns `null` because the expected DOM node is not available; any operations on the variable pointing to this non-existent DOM node (e.g., `n.firstChild`) causes a null exception and the test case calling the function terminates prematurely. To mitigate this problem, testers have to write *test fixtures* for the expected DOM structure before calling the JavaScript function in their unit tests. The manual construction of proper DOM fixtures is, however, tedious and costly.

Despite the problematic nature of JavaScript-DOM interactions, most current automated testing techniques ignore the DOM and focus on generating events and function arguments [14], [36]. JSeft [31] applies an approach in which the application is executed and the DOM tree is captured just before executing the function under test. This DOM tree is used as a test fixture in generated test cases. This heuristic-based approach, however, assumes that the DOM captured at runtime contains all the DOM elements, values, and relations as expected by the function, which is not always the case. Thus, the code coverage achieved with such a DOM can be quite limited. Moreover, the DOM captured this way, can be too large and difficult to read as a fixture in a test case. SymJS [25] applies symbolic execution to increase JavaScript code coverage, with limited support for the DOM, i.e., it considers DOM element variables as integer or string values and ignores the DOM structure. However, there exist complex DOM structures and element relations expected by the JavaScript code in practice, which this simplification cannot handle.

In this paper, we provide a technique for automatically generating DOM-based fixtures and function arguments. Our technique, called ConFix, is focused on covering DOM-dependent paths inside JavaScript functions. It operates through an iterative process that (1) dynamically analyses JavaScript code to deduce DOM-dependent statements and conditions, (2) collects path constraints in the form of symbolic DOM constraints, (3) translates the symbolic path constraints into XPath expressions, (4) feeds the generated XPath expressions into an existing structural constraint solver [17] to produce a satisfiable XML structure, (5) generates a test case with the solved structure as a test fixture or function argument, runs the generated test case, and continues recursively until all DOM-dependent paths are covered.

To the best of our knowledge, our work is the first to consider the DOM as a test input entity, and to automatically generate test fixtures to cover DOM-dependent JavaScript functions.

Our work makes the following main contributions:

- A novel dynamic symbolic execution engine to generate DOM-based test fixtures and inputs for unit testing JavaScript functions;

978-1-5090-0025-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ASE.2015.26

```
1 function dg(x){
2   return document.getElementById(x);
3 }

5 function sumTotalPrice(){
6   sum = 0;
7   itemList = dg("items");
8   if (itemList.children.length == 0)
9     dg("message").innerHTML = "List is empty!";
10  else {
11    for (i = 0; i < itemList.children.length; i++){
12      p = parseInt(itemList.children[i].value);
13      if (p > 0)
14        sum += p;
15      else
16        dg("message").innerHTML += "Wrong value for the
              price of item " + i;
17    }
18    dg("total").innerHTML = sum;
19  }
20  return sum;
21 }
```

Fig. 1: A JavaScript function to compute the items total price.

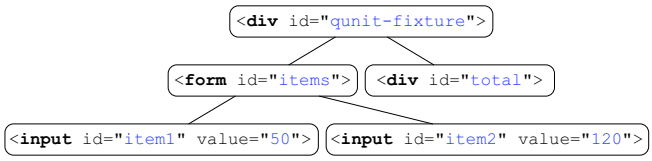

Fig. 2: A DOM subtree for covering a path (lines 6-8, 10-14, and 18-20) of `sumTotalPrice` in Figure 1.

```
1 test("A test for sumTotalPrice", function() {
2   $("#qunit-fixture").append('<form id="items"><input type
        ="text" id="item1" value=50><input type="text" id="
        item2" value=120></form><div id="total"/>');
3   sum = sumTotalPrice();
4   equal(sum, 170, "Function sums correctly.");
5 });
```

Fig. 3: A QUint test case for the `sumTotalPrice` function. The DOM subtree of Figure 2 is provided as a fixture before calling the function. This test with this fixture covers the path going through lines 6-8, 10-14, and 18 in `sumTotalPrice`.

- A technique for deducing DOM structural constraints and translating those to XPath expressions, which can be fed into existing structural constraint solvers;
- An implementation of our approach in a tool, called ConFix, which is publicly available [11];
- An empirical evaluation to assess the coverage of ConFix on real-world JavaScript applications. We also compare ConFix's coverage with that of other JavaScript test generation techniques.

The results of our empirical evaluation show that ConFix yields considerably higher coverage — up to 40 and 31 percentage point increase on the branch, and the statement coverage, respectively — compared to tests generated without DOM fixtures/inputs.

II. Background and Motivation

The majority of reported JavaScript bugs are caused by faulty interactions with the DOM [33]. Therefore, it is important to properly test DOM-dependent JavaScript code. This work is motivated by the fact that the execution of some paths in a JavaScript function, i.e., unique sequences of branches from the function entry to the exit, depends on the existence of specific DOM structures. Such DOM structures have to be provided as test fixtures to effectively test such DOM-dependent paths.

A. DOM Fixtures for JavaScript Unit Testing

A test fixture is a fixed state of the system under test used for executing tests [28]. It pertains to the code that initializes the system, brings it into the right state, prepares input data, or creates mock objects, to ensure tests can run and produce repeatable results. In JavaScript unit testing, a fixture can be a partial HTML that the JavaScript function under test expects and can operate on (read or write to), or a fragment of JSON/XML to mock the server responses in case of XMLHttpRequest (XHR) calls in the code.

Running Example. Figure 1 depicts a simple JavaScript code for calculating the total price of online items. We use this as a running example to illustrate the need for providing proper DOM structures as test input for unit testing JavaScript code.

Lets assume that a tester writes a unit test for the `sumTotalPrice` function without any test fixture. In this case, when the function is executed, it throws a `null` exception at line 8 (Figure 1) when the variable `itemList` is accessed for its `children` property. The reason is that the DOM API method `getElementById` (line 2) returns `null` since there is no DOM tree available when running the unit test case. Consequently, `dg` (line 2) returns `null`, and hence the exception at line 8. Thus, in order for the function to be called from a test case, a DOM tree needs to be present. Otherwise, the function will terminate prematurely. What is interesting is that the mere presence of the DOM does not suffice in this case. The DOM tree needs to be in a particular structure and contain attributes and values as expected by the different statements and conditions of the JavaScript function.

For instance, in the case of `sumTotalPrice`, in order to test the calculation, a DOM tree needs to be present that meets the following constraints:

1) A DOM element with id `"items"` must exist (line 7)
2) That element needs to have one or more child nodes (lines 8, 10)
3) The child nodes must be of a DOM element type that can hold *values* (line 12), e.g., `<input value="..."/>`
4) The values of the child nodes need to be positive (line 13) integers (lines 12)
5) A DOM element with id `"total"` must exist (line 18).

A DOM subtree that satisfies these constraints is depicted in Figure 2, which can be used as a test fixture for unit testing the JavaScript function. There are currently many JavaScript unit testing frameworks available, such as QUnit [7], JSUnit [4], and Jasmine [2]. We use the popular QUnit framework to illustrate the running example in this paper. QUnit provides a `$"qunit-fixture"` variable to which DOM test fixtures

can be appended in a test case. A QUnit unit test is shown in Figure 3. The execution of the test case with this particular fixture results in covering a path (lines 6-8, 10-14, and 18). If the fixture lacks any of the provided DOM elements that is required in the execution path, the test case fails and terminates before reaching the assertion.

Other DOM fixtures are required to achieve branch coverage. For example, to cover the true branch of the `if` condition in line 8, the DOM must satisfy the following constraints:

1) A DOM element with id `"items"` must exist (line 7)
2) That element must have no child nodes (line 8)
3) A DOM element with id `"message"` must exist (line 9).

Yet another DOM fixture is needed for covering the `else` branch in line 16:

1) A DOM element with id `"items"` must exist (line 7)
2) That element needs to have one or more child nodes (lines 8, 10)
3) The child nodes must be of a DOM element type that can hold *values* (line 12) e.g., `input`
4) The child nodes values need to be integers (lines 12)
5) The value of a child node needs to be zero or negative (line 15)
6) A DOM element with id `"message"` must exist (line 16).
7) A DOM element with id `"total"` must exist (line 18).

B. Challenges

As illustrated in this simple example, different DOM fixtures are required for maximizing JavaScript code coverage. Writing these DOM fixtures manually is a daunting task for testers. Generating these fixtures is not an easy task either. There are two main challenges in generating proper DOM-based fixtures that we address in our proposed approach.

Challenge 1: DOM-related variables. JavaScript is a weakly-typed and highly-dynamic language, which makes static code analysis quite challenging [34], [36], [41]. Moreover, its interactions with the DOM can become difficult to follow [33], [12]. For instance, in the condition of line 13 in Figure 1, the value of the variable `p` is checked. A fixture generator needs to determine that `p` is DOM-dependent and it refers to the value of a property of a DOM element, i.e., `itemList.children[i].value`.

Challenge 2: Hierarchical DOM relations. Unlike most test fixtures that deal only with primitive data types, DOM-based test fixtures require a tree structure. In fact, DOM fixtures not only contain proper DOM elements with attributes and their values, but also hierarchical parent-child relations that can be difficult to reconstruct. For example the DOM fixture in Figure 2 encompasses the parent-child relation between `<form>` and `<input>` elements, which is required to evaluate `itemList.children.length` and `itemList.children[i].value` in the code (lines 8, 11, and 12 in Figure 1).

C. Dynamic Symbolic Execution

Our insight in this work is that the problem of generating expected DOM fixtures can be formulated as a constraint solving problem, to achieve branch coverage.

Symbolic execution [24] is a static analysis technique that uses symbolic values as input values instead of concrete data, to determine what values cause each branch of a program to execute. For each decision point in the program, it infers a set of symbolic constraints. Satisfiability of the conjunction of these symbolic constraints is then checked through a constraint solver. *Concolic execution* [19], [37], also known as *dynamic symbolic execution* [39], performs symbolic execution while systematically executing all feasible program paths of a program on some concrete inputs. It starts by executing a program with random inputs, gathers symbolic constraints at conditional statements during execution, and then uses a constraint solver to generate a new input. Each new input forces the execution of the program through a new uncovered path; thus repeating this process results in exploring all feasible execution paths of the program.

III. Approach

We propose a DOM-based test fixture generation technique, called ConFix. To address the highly dynamic nature of JavaScript, ConFix is based on a dynamic symbolic execution approach.

Scope. Since primitive data constraints can be solved using existing input generators for JavaScript [35], [36], [25], in this paper, we focus on collecting and solving DOM constraints that enable achieving coverage for DOM dependent statements and conditions in JavaScript code. Thus, ConFix is designed to generate DOM-based test fixtures and function arguments for JavaScript functions that are *DOM-dependent*.

Definition 1 (DOM-Dependent Function) *A DOM dependent function is a JavaScript function, which directly or indirectly accesses DOM elements, attributes, or attribute values at runtime using DOM APIs.* □

An instance of a direct access to a DOM element is `document.getElementById("items")` in the function `dg` (Line 2, Figure 1). An indirect access is a call to another JavaScript function that accesses the DOM. For instance, the statement at line 7 of Figure 1 is an indirect DOM access through function `dg`.

Overview. Figure 4 depicts an overview of ConFix. At a high level, ConFix instruments the JavaScript code (block 1), and executes the function under test to collect a trace (blocks 2 and 3). Using the execution trace, it deduces DOM-dependent path constraints (block 4), translates those constraints into XPath expressions (block 5), which are fed into an XML constraint solver (block 6). The solved XML tree is then used to generate a DOM-based fixture (block 7), which subsequently helps in covering unexplored paths (block 8). Finally, it generates a test suite by adding generated test fixtures into a JavaScript unit testing template such as QUnit (block 9). In the following subsections we discuss each of these steps in more details.

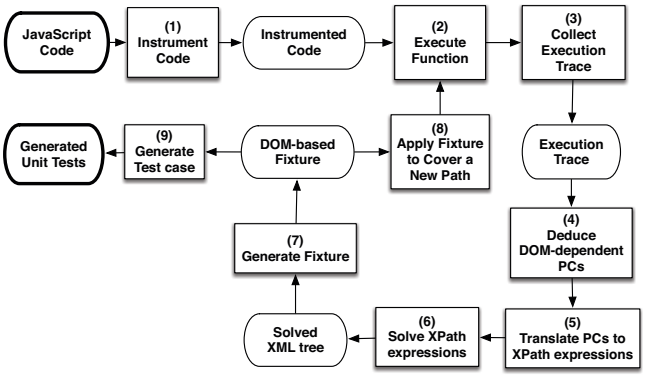

Fig. 4: Processing view of our approach.

Algorithm. Algorithm 1 demonstrates our DOM fixture generation technique. The input to our algorithm is the JavaScript code, the function under test f, and optionally its function arguments (provided by a tester or a tool [35], [36]). The algorithm concolically generates DOM fixtures required for exploring all DOM-dependent paths.

A. Collecting DOM-based Traces

We instrument the JavaScript code under test (algorithm 1 line 3) with wrapper functions to collect information pertaining to DOM interactions, which include statements or conditional constructs that depend on the existence of a DOM structure, element, attribute, or value. The instrumentation is non-intrusive meaning that it does not affect the normal functionality of the original program.

The instrumentation augments function calls, conditionals (in `if` and loop constructs), infix expressions, variable initializations and assignments, and return statements with inline wrapper functions to store an execution trace (see subsection III-E for more details). Such a trace includes the actual statement, type of statement (e.g. infix, condition, or function call), list of variables and their values in the statement, the enclosing function name, and the actual concrete values of the statement at runtime. ConFix currently supports DOM element retrieval patterns based on tag names, IDs, and class names, such as `getElementById`, `getElementsByTagName`, `children`, `innerHTML`, `parentNode`, and `$()` for jQuery-based code.

After instrumenting the source code, the modified JavaScript file is used in a runner HTML page that is loaded inside a browser (line 7, algorithm 1) and then a JavaScript driver (e.g., WebDriver [8]) executes the JavaScript function under test (line 8). This execution results in an execution trace, which is collected from the browser for further analysis (line 9).

B. Deducing DOM Constraints

An important phase of dynamic symbolic execution is gathering *path constraints* (PCs). In our work, path constraints are conjunctions of constraints on symbolic DOM elements.

Algorithm 1: Test Fixture Generation

```
   input : JavaScript code JS, the function under test f, function arguments
           for f
   output: A set of DOM fixtures fixtureSet for f
1  negatedConstraints ← ∅
2  DOMRefTrackList ← ∅
   Procedure GenerateFixture(JS, f)
   begin
3      JS_inst ← Instrument(JS)
4      fixtureSet ← ∅
5      fixture ← ∅
6      repeat
7          browser.Load(JS_inst, fixture)
8          browser.Execute(f)
9          t ← browser.GetExecutionTrace()
10         fixture ← SolveConstraints(t)
11         fixtureSet ← fixtureSet ∪ fixture
       until fixture ≠ ∅;
12     return fixtureSet

   Procedure SolveConstraints(t)
   begin
13     DOMRefTrackList ← GetDOMReferenceTracks(t)
14     pc ← GetPathConstraint(t, DOMRefTrackList)
15     fixture ← UNSAT
16     while fixture = UNSAT do
17         fixture ← ∅
18         c ← GetLastNonNegConst(pc, negatedConstraint)
19         if c ≠ null then
20             negatedConstraint ← negatedConstraint ∪ c
21             pc ← NegateConstraint(pc, c)
22             xp ← GenerateXPath(pc, DOMRefTrackList)
               /* SolveXPathConstraint returns UNSAT if
                  xp is not solvable.                    */
23             fixture ← SolveXPathConstraint(xp)
24     return fixture
```

Definition 2 (Symbolic DOM Element) *A symbolic DOM element d is a data structure representing a DOM element in terms of its symbolic properties and values. d is denoted by a 4 tuple $< \mathcal{P}, \mathcal{C}, \mathcal{A}, \mathcal{T} >$ where:*

1) *$\mathcal{P}$ is d's parent symbolic DOM element.*
2) *$\mathcal{C}$ is a set of child symbolic DOM elements of d.*
3) *$\mathcal{A}$ is a set of $\langle$`att, val`$\rangle$ pairs; each pair stores an attribute `att` of d with a symbolic value `val`.*
4) *$\mathcal{T}$ is the element type of d.* □

Note that keeping the parent-children relation for DOM elements is sufficient to recursively generate the DOM tree.

DOM constraints in the code can be conditional or non-conditional. A *non-conditional* DOM constraint is a constraint on the DOM tree required by a DOM accessing JavaScript statement. A *conditional* DOM constraint is a constraint on the DOM tree used in a conditional construct.

Example 1. Consider the JavaScript code in Figure 1. In line 2, `document.getElementById(x)` is a non-conditional DOM constraint, i.e., an element with a particular ID is targeted. On the other hand, `itemList.children.length == 0` (line 8) is a conditional DOM constraint, i.e., the number of child nodes of a DOM element is checked to be zero. □

Non-conditional DOM constraints evaluate to `null` or a not-null object, while, conditional DOM constraints evaluate to `true` or `false`. For example, if we execute `sumTotalPrice()` with a DOM subtree void of an element

with `id="items"`, the value of `itemList` at line 7 will be `null` and the execution will terminate at line 8 when evaluating `itemList.children.length == 0`. On the other hand, if that element exists and has a child, the condition at line 8 evaluates to `false`.

In this work, the input to be generated is a DOM subtree that is accessed via DOM APIs in the JavaScript code. As we explained in subsection II-B, due to the dynamic nature of JavaScript, its interaction with the DOM can be difficult to follow (challenge 1). Tracking DOM interactions in the code is needed for extracting DOM constraints. Aliases in the code add an extra layer of complexity. We need to find variables in the code that refer to DOM elements or element attributes. To address this challenge, we apply an approach similar to dynamic backward slicing, except that instead of slices of the code, we are interested in relevant DOM-referring variables. To that end, we use dynamic analysis to extract DOM referring variables from the execution trace, by first searching for DOM API calls, their arguments, and their actual values at runtime. The process of collecting DOM referring variables (Algorithm 1 line 13) is outlined further in subsection III-E.

Once DOM referring variables are extracted, constraints on their corresponding DOM elements are collected and used to generate constraints on symbolic DOM elements (see Definition 2). DOM constraints can be either *attribute-wise* or *structure-wise*. *Attribute-wise* constraints are satisfied when special values are provided for element attributes. For example, `parseInt(itemList.children[i].value) > 0` (line 13 of Figure 1) requires the value of the `i`-th child node to be an integer greater than zero. The *value* is an attribute of the child node in this example. *Structure-wise* constraints are applied to the element type and its parent-children relations. For example, in `itemList.children.length == 0` (line 8 of Figure 1 to cover the `else` branch (lines 10–19) an element with id `"items"` is needed with at least one child node.

The conjunction of these symbolic DOM constraints in an iteration forms a path constraint. For instance, the structure-wise constraint in `parseInt(itemList-.children[i].value) > 0` (line 13 of Figure 1) requires the child nodes to be of an element type that can hold values, e.g., `input` type, and the attribute-wise constraint requires the value to be a positive integer.

Our technique reasons about a collected path constraint and constructs symbolic DOM elements needed. For each symbolic DOM element, ConFix (1) infers the type of the parent node, (2) determines the type and number of child nodes, and (3) generates, through a constraint solver, satisfied values that are used to assign attributes and values (i.e., ⟨`att, val`⟩ pairs). The default element type for a symbolic DOM element is `div` — the `div` is a placeholder element that defines a division or a section in an HTML document. It satisfies most of the element type constraints and can be parent/child of many elements — unless specific attributes/values are accessed from the element, which would imply that a different element type is needed. For instance, if the `value` is read/set for an element, the type of that element needs to change to, for instance `input`, because per definition, the `div` type does not carry a `value` attribute (more detail in subsection III-E). These path constraints with satisfied symbolic DOM elements are used to generate a corresponding XPath expression, as presented in the next subsection.

C. Translating Constraints to XPath

The problem of DOM fixture generation can be formulated as a decision problem for the emptiness test [15] of an XPath expression, in the presence of XHTML meta-models, such as Document Type Definitions (DTD) or XML Schemas. These meta-models define the hierarchical tree-like structure of XHTML documents with the type, order, and multiplicity of valid elements and attributes. XPath [16] is a query language for selecting nodes from an XML document. An example is the expression `/child::store/child::item-/child::price` which navigates the root through the top-level "store" node to its "item" child nodes and on to its "price" child nodes. The result of the evaluation of the entire expression is the set of all the "price" nodes that can be reached in this manner. XPath is a very expressive language. We propose a restricted XPath expression grammar, shown in Figure 5, which we use to model our DOM constraints in.

⟨*XPath*⟩ ::= ⟨*Path*⟩ | /⟨*Path*⟩

⟨*Path*⟩ ::= ⟨*Path*⟩/⟨*Path*⟩ | ⟨*Path*⟩[⟨*Qualifier*⟩] | child::⟨*Name*⟩ | ⟨*Name*⟩

⟨*Qualifier*⟩ ::= ⟨*Qualifier*⟩ and ⟨*Qualifier*⟩ | ⟨*Path*⟩ | @⟨*Name*⟩

⟨*Name*⟩ ::= ⟨*HTMLTag*⟩ | ⟨*Attribute*⟩=⟨*Value*⟩

⟨*HTMLTag*⟩ ::= a | b | button | div | form | frame | h1 - h6 | iframe | img | input | i | li | link | menu | option | ol | p | select | span | td | tr | ul

⟨*Attribute*⟩ ::= id | type | name | class | value | src | innerHTML | title | selected | checked | href | size | width | height

Fig. 5: Restricted XPath grammar for modeling DOM constraints.

We transform the deduced path constraints, with the symbolic DOM elements, into their equivalent XPath expressions conforming to this specified grammar. These XPath expressions systematically capture the hierarchical relations between elements. Types of common constraints translated to expressions include specifying the existence of a DOM element/attribute/value, properties of style attributes, type and number of child nodes or descendants of an element, or binary properties such as selected/not selected.

Example 2. Table I shows examples of collected DOM constraints that are translated to XPath expressions, for the running example. For example, in the first row, the DOM constraint `document.getElementById("items")` ≠ `null` is translated to the XPath expression `div[@id="qunit-fixture"][div[@id="items"]]`, which expresses the desire for the existence of a `div` element with id "items" in the fixture. The second row shows a more complex example, including six DOM constraints in a path constraint, which are translated into a corresponding XPath expression. □

D. Constructing DOM Fixtures

Next, the XPath expressions are fed into a structural XML solver [17]. The constraint solver parses the XPath expressions and compiles them into a logical representation, which is tested

TABLE I: Examples of DOM constraints, translated XPath expressions, and solved XHTML instances for the running example.

DOM constraints	Corresponding XPath expressions	Solved XHTML
document.getElementById(“items”) ≠ null	div[@id=“qunit-fixture”][div[@id=“items”]]	
document.getElementById(“items”) ≠ null ∧ itemList.children.length ≠ 0 ∧ 0 < itemList.children.length ∧ parseInt(itemList.children[0].value) > 0 ∧ document.getElementById(“message”) ≠ null ∧ document.getElementById(“total”) ≠ null	div[@id=“qunit-fixture”][div[@id=“items” and child::input[@id=“Confix1” and @value=”1”]] and child::div[@id=“message”] and child::div[@id=“total”]]	 <input id=“Confix1” value=“1”/>

for satisfiability. If satisfiable, the solver generates a solution in the XML language. Since an XHTML meta-model (i.e., DTD) is fed into the solver along with the XPath expressions, the actual XML output is an instance of valid XHTML. The last column of Table I shows solved XHTML instances that satisfy the XPath expressions, for the running example. These solved XHTML instances are subsequently used to construct test fixtures.

Each newly generated fixture forces the execution of the JavaScript function under test along a new uncovered path. This is achieved by systematically negating the last non-negated conjunct in the path constraint and solving it to obtain a new fixture, in a depth first manner. If a path constraint is unsatisfiable, the technique chooses a different path constraint and this process repeats until all constraints are negated.

In the main loop of Algorithm 1 (lines 6–11), fixtures are iteratively generated and added to the `fixtureSet`. In the `SolveConstraints` procedure, a fixture is initialized to `UNSAT`; the loop (lines 16–23) continues until the fixture is set either to a solved DOM subtree[1] (line 23), or to ∅ (line 17) if there exist no non-negated constraints in the PCs. When a ∅ fixture returns from `SolveConstraints`, the loop in the main procedure terminates, and the `fixtureSet` is returned.

EXAMPLE 3. Table II shows the extracted path constraints and their values, as well as the current and next iteration fixtures at each iteration of the concolic execution for the running example. Since there is no fixture available in the first iteration, the constraint obtained is `document.getElementById("items")=null`. This means the execution terminates at line 8 of the JavaScript code (Figure 1) due to a null exception. Our algorithm negates the last non-negated constraint (`document.getElementById("items")≠null`), and generates the corresponding fixture (`<div id="items">`) to satisfy this negated constraint. This process continues until the solver fails at producing a satisfiable fixture in the sixth iteration. `UNSAT` is returned because the constraints `itemList.children.length≠0` and the newly negated one (`0≮itemList.children.length`) require that the number of child nodes be negative, which is not feasible in the DOM structure. It then tries to generate a fixture by negating the last non-negated constraint without applying any fixtures, i.e., the path constraint extracted in the sixth iteration. However, in this case there are no non-negated constraints left since the last one (`0< itemList.children.length`) had already been negated and the result was not satisfiable. Consequently the algorithm terminates with an empty fixture in the last iteration. The table also shows which paths of the running example are covered in terms of lines of JavaScript code. □

[1]Note that `SolveXpathConstraint` returns `UNSAT` when it fails to solve the given path constraint.

E. Implementation Details

CONFIX currently generates QUnit test cases with fixtures, however, it can be easily adapted to generate test suites in other JavaScript testing frameworks. To parse the JavaScript code into an abstract syntax tree for instrumentation, we build up on Mozilla Rhino [5]. To collect execution traces while executing a function in the browser, we use WebDriver [8].

XML Solver. To solve *structure-wise* DOM constraints using XPath expressions, we use an existing XML reasoning solver [17]. A limitation with this solver is that it cannot generate XML structures with valued attributes (i.e., attributes are supported but not their values). To mitigate this, we developed a transformation technique that takes our generic XPath syntax (Figure 5) and produces an XPath format understandable by the solver. More specifically, we merge attributes and their values together and feed them to the solver as single attributes to be generated. Once the satisfied XML is generated, we parse and decompose it to the proper attribute-value format as expected in a valid XHTML instance. Another limitation is that it merges instances of the same tag elements at the same tree level when declared as children of a common parent. We resolved this by appending an auto-increment number to the end of each tag and remove it back once the XML produced.

Handling asynchronous calls. Another challenge we encountered pertains to handling asynchronous HTTP requests that send/retrieve data (e.g., in JSON/XML format) to/from a server performed by using the `XMLHttpRequest` (XHR) object. This feature makes unit-level testing more challenging since the server-side generated data should also be considered in a test fixture as an XHR response if the function under test (in)directly uses the XHR and expects a response from the server. Existing techniques [21] address this issue by mocking the server responses, but they require multiple concrete executions of the application to learn the response. This is, however, not feasible in our case because we generate JavaScript unit tests in isolation from other dependencies such as the server-side code. As a solution, our instrumentation replaces the XHR object of the browser with a new object and redefines the XHR `open()` method in a way that it always uses the `GET` request method to synchronously retrieve data from a local URL referring to our mocked server. This helps us to avoid null exceptions and continue the execution of the function under test. However, if the execution depends on the actual value of the retrieved data (and not merely their existence), our current approach can not handle it. In such cases, a string solver [23] may be helpful.

TABLE II: Constraints table for the running example. The “Next to negate" field refers to the last non-negated constraint.

Iteration	Current fixture	Current DOM constraints	Negated	Next to negate	Fixture for the next iteration	Paths covered
1	∅	document.getElementById(“items”) = null	-	✓		Lines 1–8
2		document.getElementById(“items”) ≠ null ∧ itemList.children.length = 0 ∧ document.getElementById(“message”) = null	✓ - -	- - ✓	 	Lines 1–9
3	 	document.getElementById(“items”) ≠ null ∧ itemList.children.length = 0 ∧ document.getElementById(“message”) ≠ null	✓ - ✓	- ✓ -	 	Lines 1–9 and 20
4	 	document.getElementById(“items”) ≠ null ∧ itemList.children.length ≠ 0 ∧ 0 < itemList.children.length ∧ parseInt(itemList.children[0].value) ≯ 0 ∧ document.getElementById(“message”) ≠ null ∧ document.getElementById(“total”) = null	✓ ✓ - - ✓ -	- - - - - ✓	 	Lines 1–8, 10–13, and 15–18
5	 	document.getElementById(“items”) ≠ null ∧ itemList.children.length ≠ 0 ∧ 0 < itemList.children.length ∧ parseInt(itemList.children[0].value) ≯ 0 ∧ document.getElementById(“message”) ≠ null ∧ document.getElementById(“total”) ≠ null	✓ ✓ - - ✓ ✓	- - - ✓ - -	 <input id=“Confix1” value=“1”/> 	Lines 1–8, 10–13, and 15–20
6	 <input id=“Confix1” value=“1”/> 	document.getElementById(“items”) ≠ null ∧ itemList.children.length ≠ 0 ∧ 0 < itemList.children.length ∧ parseInt(itemList.children[0].value) > 0 ∧ document.getElementById(“message”) ≠ null ∧ document.getElementById(“total”) ≠ null	✓ ✓ - ✓ ✓ ✓	- - ✓ - - -	UNSAT ⇒ Negate last non-negated constraint	Lines 1–8, 10–14, and 18–20
	 <input id=“Confix1” value=“1”/> 	document.getElementById(“items”) ≠ null ∧ itemList.children.length ≠ 0 ∧ 0 < itemList.children.length ∧ parseInt(itemList.children[0].value) > 0 ∧ document.getElementById(“message”) ≠ null ∧ document.getElementById(“total”) ≠ null	✓ ✓ ✓ ✓ ✓ ✓	- - - - - -	No non-negated constraint exists ⇒ Fixture = ∅	

Tracking DOM-referring variables. To detect DOM-referring variables — used to generate constraints on symbolic DOM elements (Definition 2) — we automatically search for DOM API calls, their arguments, and their actual values at runtime, in the execution trace. Algorithm 1 keeps track of DOM references (line 13) by storing information units, called DOM Reference Track (DRT), in a data structure.

Definition 3 (DOM Reference Track (DRT)) *A DOM reference track is a data structure capturing how a DOM tree is accessed in the JavaScript code. It is denoted by a 4 tuple* $< \mathcal{D}, \mathcal{P}, \mathcal{A}, \mathcal{T} >$ *where:*

1) $\mathcal{D}$ *(DOMVariable) is a JavaScript variable* v *that is set to refer to a DOM element* d*.*
2) $\mathcal{P}$ *(ParentVariable) is a JavaScript variable (or the* `document` *object) that refers to the parent node of* d*.*
3) $\mathcal{A}$ *(AttributeVariables) is a set of* $\langle$`att:val, var`$\rangle$ *pairs; each pair stores the variable* `var` *in the code that refers to an attribute* `att` *of* d *with a value* `val`*.*
4) $\mathcal{T}$ *(ElementType) is the node type of* d*.* □

When JavaScript variables are evaluated in a condition, the DRT entries in this data structure are examined to determine whether they refer to the DOM. If the actual value of a JavaScript variable at runtime contains information regarding a DOM element object, and it does not exist in our DRT data structure, we add it as a new DOM referring variable. Table III presents an example of the DRT for the running example.

TABLE III: DRT data structure for the running example.

Iteration	DOMVariable	ParentVariable	Element Type	AttributeVariables	Exists?
1	`itemList`	`document`	`div`	⟨id:`items`, -⟩	✗
2	`itemList`	`document`	`div`	⟨id:`items`,-⟩	✓
	-	`itemList`	`div`	⟨id:`Confix1`,-⟩	✗
	-	`document`	`div`	⟨id:`message`,-⟩	✗
3	`itemList`	`document`	`div`	⟨id:`items`,-⟩	✓
	-	`itemList`	`div`	⟨id:`Confix1`,-⟩	✗
	-	`document`	`div`	⟨id:`message`,-⟩	✓
...	...	...	...	...	...
6	`itemList`	`document`	`div`	⟨id:`items`,-⟩	✓
	-	`itemList`	`input`	⟨id:`Confix1`,-⟩, ⟨value:1, `p`⟩	✓
	-	`document`	`div`	⟨id:`message`,-⟩	✓
	-	`document`	`div`	⟨id:`total`,-⟩	✓

We implemented a constraint solver that reasons about some common symbolic DOM constraints such as string/integer attribute values, and number of children nodes. Specifically the solver infers conditions on DOM referring variables by examining DRT entries. If the constraint is on an attribute of a DOM element, then the `AttributeVariables` property of the corresponding DRT will be updated with a satisfying value. In case the constraint is a structural constraint, such as number of child nodes, a satisfying number of DRT entities would be added to the table. Table III depicts the process of constructing the DRT during different iterations of the concolic

execution. The `Exists` field indicates whether the element exists in the DOM fixture.

EXAMPLE 4. Consider the running example of Figure 1. When `sumTotalPrice()` is called in the first iteration, `dg("items")` returns `null` as no DOM element with ID `items` exists. Table III would then be populated by adding the first row: `DOMVariable` is `itemList`, the `ParentVariable` points to `document`, the default element type is set to `div`, and the attribute `id` is set to `items`; and this particular element does not exist yet. The execution terminates with a null exception at line 8. In the next iteration, CONFIX updates the DOM fixture with a `div` element with id `items`. Therefore `dg("items")` returns a DOM element and line 8 evaluates the number of child nodes under `itemList`. This would then update the table with a new entry having `ParentVariable` point to `itemList` and attribute `id` set to an automatically incremented id `"Confix1"` (in the second row). This process continues as shown in Table III. □

Generating DOM-based function arguments. Current tools for JavaScript input generation (e.g., [35], [36]) only consider primitive data types for function arguments and thus cannot handle functions that take DOM elements as input. Consider the following simple function:

```
1  function foo(elem) {
2    var price = elem.firstChild.value;
3    if (price > 200) {
4        ...
5    }}
```

The `elem` function parameter is expected to be a DOM element, whose first child node's value is read in line 2 and used in line 3. The problem of generating DOM function arguments is not fundamentally different from generating DOM fixtures. Thus, we propose a solution for this issue in CONFIX. The challenge here, however, is that JavaScript is dynamically typed and since `elem` in this example does not reveal its type statically nor when `foo` is executed in isolation in a unit test (because `elem` does not exist to log its type dynamically), it is not possible to determine that `elem` is a DOM element. To address this challenge, CONFIX first computes a forward slice of the function parameters. If there is a DOM API call in the forward slice, CONFIX deduces constraints and solves them similarly to how DOM fixtures are constructed. The generated fixture is then parsed into a DOM element object and the function is called with that object as input in the test case. In the example above, there is a DOM API call present, namely `firstChild` in the forward slice of `elem`. Therefore, CONFIX would know that `elem` is a DOM element and would generate it accordingly. Then `foo` is called in the test case with that object as input.

IV. EMPIRICAL EVALUATION

To assess the efficacy of our proposed technique, we have conducted a controlled experiment to address the following research questions:

RQ1 (Coverage) Can fixtures generated by CONFIX effectively increase code coverage of DOM-dependent functions?

RQ2 (Performance) What is the performance of running CONFIX? Is it acceptable?

TABLE IV: Characteristics of experimental objects excluding blank/comment lines and external JavaScript libraries.

Name	JS LOC	# Branches	# Functions	% DOM-Dependent Functions	# DOM Constraints (DC)	% Non-Conditional DC	% Conditional DC
ToDoList	82	10	7	100	19	84	16
HotelReserve	106	88	9	56	13	69	31
Sudoku	399	344	18	78	66	67	33
Phormer	1553	464	109	71	194	70	30
Total	2140	906	143	72	292	70	30

The experimental objects and our results, along with the implementation of CONFIX are available for download [11].

A. Experimental Objects

To evaluate CONFIX, we selected four open source web applications that have many DOM-dependent JavaScript functions. Table IV shows these applications, which fall under different application domains and have different sizes. *ToDoList* [9] is a simple JavaScript-based todo list manager. *HotelReserve* [1] is a reservation management application. *Sudoku* [10] is a web-based implementation of the Sudoku game. And *Phormer* [6] is an Ajax photo gallery. As presented in Table IV, about 70% of the functions in these applications are DOM-dependent. The table also shows the lines of JavaScript code, and number of branches and functions in each application.

B. Experimental Setup

Our experiments are performed on Mac OS X, running on a 2.3GHz Intel Core i7 with 8 GB memory, and FireFox 37.

1) Independent variables: To the best of our knowledge, there exists no DOM test fixture generation tool to compare against; the closest to CONFIX is JSeft [31], which generates tests that use the entire DOM at runtime as test fixtures. However, JSeft does not *generate* DOM fixtures, and it requires a deployed web application before it can be used.

Therefore, we construct baseline test suites to compare against. We compare different types of test suites to evaluate the effectiveness of test fixtures generated by CONFIX. Table V depicts different JavaScript unit test suites. We classify test suites based on the type of test input they have support for, namely, (1) DOM fixtures, and/or (2) DOM function arguments.

Test suites without DOM fixtures. *NoInput* is a naive test suite that calls each function without setting any fixture or input for it. *Jalangi* produces (non-DOM) function arguments using the concolic execution engine of JALANGI [36]. *Manual* is a test suite that uses manually provided (non-DOM) inputs.

Test suites with DOM fixtures. To assess the effect of DOM fixtures and inputs generated by CONFIX, we consider different combinations: *ConFix + NoInput* has DOM fixtures generated by CONFIX but no function arguments, *ConFix + Jalangi*

TABLE V: Evaluated function-level test suites.

Test Suite	Function Arguments	DOM Fixture	DOM Input	# Test Cases
NoInput	No input	✗	✗	98
Jalangi	Generated by JALANGI	✗	✗	98+4
Manual	Manual inputs	✗	✗	98+55
ConFix + NoInput	No input	✓	✗	98+125
ConFix + Jalangi	Generated by JALANGI	✓	✗	98+129
ConFix + Manual	Manual inputs	✓	✓	98+236

has DOM fixtures generated by CONFIX but uses the inputs generated by JALANGI for non-DOM function arguments, and *ConFix + Manual* uses DOM fixtures and DOM function arguments generated by CONFIX and manual inputs for non-DOM function arguments. Table V shows all these combinations along with the number of test cases in each category.

Note that since our approach is geared toward generating DOM-based test fixtures/inputs, we only consider test generation for DOM-dependent functions and thus for all categories we consider the same set of 98 DOM-dependent functions under test, but with different inputs/fixtures. The 55 manual non-DOM function arguments were written by the authors through source code inspection.

2) Dependent variables: Our dependent variables are code coverage and generation time.

Code coverage. Code coverage is commonly used as a test suite adequacy criterion. To address RQ1, we compare the JavaScript code coverage of the different test suites, using JSCover [3]. Since our target functions in this work are DOM-dependent functions (Definition 1), code coverage is calculated by considering only DOM-dependent functions.

Fixture generation time. To answer RQ2 (performance), we measure the time (in seconds) required for CONFIX to generate test fixtures for a test suite, divide it by the number of generated tests, and report this as the average fixture generation time.

C. Results

Coverage (RQ1). Figure 6 illustrates the comparison of code coverage achieved by each test suite category. We report the total statement and branch coverage of the JavaScript code obtained from the experimental objects.

Table IV shows that in total about 14% of the code (i.e., 292 out of 2140 LOC) contains DOM constraints. However, as shown in Figure 6, this relatively small portion of the code has a remarkable impact on the code coverage when comparing test suites with and without DOM test fixtures. This is due to the fact that if a DOM constraint is not satisfied, the function terminates as a result of a null exception in most cases. Such constraints may exist at statements near the entrance of functions (as shown in Figure 1) and thus, proper DOM test fixtures are essential to achieve proper coverage. Table VI shows the coverage increase for the test suites.

Our results, depicted in Figure 6 and Table VI, show that *Manual* and *Jalangi* cannot achieve a much higher code coverage than *NoInput*. This again relates to the fact that if expected DOM elements are not available, then the execution

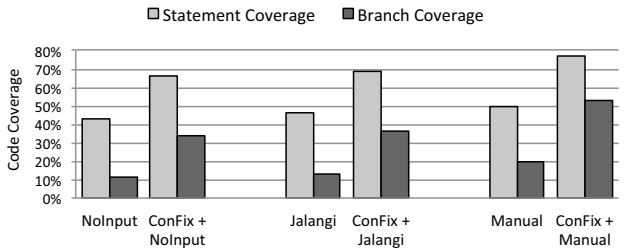

Fig. 6: Comparison of statement and branch coverage, for DOM-dependent functions, using different test suite generation methods.

TABLE VI: Coverage increase (in percentage point) of test suites on rows over test suites on columns. Statement and branch coverage are separated by a slash, respectively.

	NoInput	*Jalangi*	*Manual*	*ConFix + NoInput*	*ConFix + Jalangi*
Jalangi	3 / 2	—	—	—	—
Manual	7 / 9	4 / 7	—	—	—
ConFix + NoInp	23 / 23	20 / 21	16 / 14	—	—
ConFix + Jalangi	26 / 25	23 / 23	19 / 16	3 / 2	—
ConFix + Manual	**34 / 42**	**31 / 40**	**27 / 33**	11 / 19	8 / 17

of DOM-dependent functions terminates and consequently code coverage cannot be increased much, no matter the quality of the function arguments provided.

The considerable coverage increase for *ConFix + NoInput* vs. *Jalangi* and *Manual* indicates that test suites generated by CONFIX, even without providing any arguments, is superior over other test suites with respect to the achieved code coverage. The coverage increases even more when function arguments are provided; for example, for *ConFix + Manual* compared to *Jalangi*, there is a 40 percentage point increase (300% improvement) in the branch coverage, and a 31 percentage point increase (67% improvement) in the statement coverage.

The coverage increase for *ConFix + Manual* vs. *ConFix + NoInput* and *ConFix + Jalangi* is more substantial in comparison with the coverage increase for *Manual* vs. *NoInput* and *Jalangi*. This is mainly due to (1) the DOM fixtures generated, which are required to execute paths that depend on manually given arguments; and (2) the DOM arguments generated, which enable executing paths that depend on DOM elements provided as function arguments.

Although DOM fixtures generated by CONFIX can substantially improve the coverage compared to the current state-of-the-art techniques, we discuss why CONFIX does not achieve full coverage in section V.

Performance (RQ2). The execution time of CONFIX is mainly affected by its concolic execution, which requires iterative code execution in the browser and collecting and solving constraints. Our results show that, on average, CONFIX requires 0.7 second per test case and 1.6 second per function to generate DOM fixtures. Since the number of conditional DOM constraints in

typical JavaScript functions is not extremely large, concolic execution can be performed in a reasonable time.

V. Discussion

Applications. Given the fact that JavaScript extensively interacts with the DOM on the client-side, and these interactions are highly error-prone [33], we see many applications for our technique. ConFix can be used to automatically generate JavaScript unit tests with DOM fixtures that could otherwise be quite time consuming to write manually. It can also be used in combination with other existing test generation techniques [31], [36] to improve the code coverage. In case a DOM constraint depends on a function argument, we can perform concolic execution i.e., beginning with an arbitrary value for the argument, capturing the DOM constraint during execution, and treating the DOM referring variable and the argument as symbolic variables. In addition to DOM fixtures, ConFix can also generate DOM-based function arguments, i.e., DOM elements as inputs, as explained in subsection III-E. Currently there is no tool that supports DOM input generation for JavaScript functions.

Limitations. We investigated why ConFix does not achieve full coverage. The main reasons that we found reveal some of the current limitations of our implementation, which include: (1) we implemented a simple integer/string constraint solver to generate XPath expressions with proper structure and attribute values, which cannot handle complex constraints currently; (2) we do not support statements that require event-triggering; (3) the XML solver used in our work cannot efficiently solve long lists of constraints, (4) some paths are browser-dependent, which is out of the scope of ConFix; (5) the execution of some paths are dependent on global variables that are set via other function calls during the execution, which is also out of the scope of ConFix; and (6) ConFix does not analyze dynamically generated code using `eval()` that interacts with the DOM.

Threats to validity. A threat to the external validity of our experiment is with regard to the generalization of the results to other JavaScript applications. To mitigate this threat, we selected applications from different domains (task management, form validation, game, gallery) that exhibit variations in functionality, and we believe they are representative of JavaScript applications that use DOM APIs for manipulating the page; although we do need more and large applications to make our results more generalizable. With respect to the reproducibility of our results, ConFix, the test suites, and the experimental objects are all available [11], making the experiment repeatable.

VI. Related Work

Most current web testing techniques focus on generating sequences of events at the DOM level, while we consider unit test generation at the JavaScript code level. Event-based test generation techniques [27], [29], [30] can miss JavaScript faults that do not propagate to the DOM [31].

Unit testing. Alshraideh [13] generates unit tests for JavaScript programs through mutation analysis by applying basic mutation operators. Heidegger et al. [20] propose a test case generator for JavaScript that uses contracts (i.e., type signature annotations that the tester has to include in the program manually) to generate inputs. Artemis [14] is a framework for automated testing of JavaScript, which applies feedback-directed random test generation. None of these techniques consider DOM fixtures for JavaScript unit testing.

More related to our work, JSeft [31] applies a heuristic-based approach by capturing the full DOM tree during the execution of an application just before executing the function under test, and uses that DOM as a test fixture in a generated test case. This approach, however, cannot cover all DOM dependent branches.

Symbolic and concolic execution. Nguyen et al. [32] present a technique that applies symbolic execution for reasoning about the potential execution of client-side code embedded in server-side code. Kudzu [35] performs symbolic reasoning to analyze JavaScript security vulnerabilities, such as code injections in web applications. Jalangi [36] is a dynamic analysis framework for JavaScript that applies concolic execution to generate function arguments; however, it does not support DOM-based arguments nor DOM fixtures, as ConFix does. SymJS [25] contains a symbolic execution engine for JavaScript, as well as an automatic event explorer. It extends HTMLUnit's DOM and browser API model to support symbolic execution by introducing symbolic values for specific elements, such as text inputs and radio boxes. However, it considers substituting DOM element variables with integer or string values and using a traditional solver, rather than actually generating the hierarchical DOM structure. ConFix on the other hand has support for the full DOM tree-structure including its elements and their hierarchical relations, attributes, and attribute values.

To the best of our knowledge, ConFix is the first to address the problem of DOM test fixture construction for JavaScript unit testing. Unlike most other techniques, we consider JavaScript code in isolation from server-side code and without the need to execute the application as a whole.

VII. Conclusions

Proper test fixtures are required to cover DOM-dependent statements and conditions in unit testing JavaScript code. However, generating such fixtures is not an easy task. In this paper, we proposed a concolic technique and tool, called ConFix, to automatically generate a set of unit tests with DOM fixtures and DOM function arguments. Our empirical results show that the generated fixtures substantially improve code coverage compared to test suites without these fixtures.

For future work, we plan to enhance ConFix by addressing its current limitations, and evaluate its bug finding capability of the approach, and (3) conduct experiments on a larger set of JavaScript applications.

VIII. Acknowledgments

This work was supported by the National Science and Engineering Research Council of Canada (NSERC) through its Strategic Project Grants programme and Alexander Graham Bell Canada Graduate Scholarship.

REFERENCES

[1] Hotel Reservation. https://github.com/andyfeds/HotelReservationSystem.

[2] Jasmine. https://github.com/pivotal/jasmine.

[3] JSCover. http://tntim96.github.io/JSCover/.

[4] JsUnit. http://jsunit.net/.

[5] Mozilla Rhino. https://github.com/mozilla/rhino.

[6] Phormer Photogallery. http://sourceforge.net/projects/rephormer/.

[7] QUnit. http://qunitjs.com/.

[8] Selenium HQ. http://seleniumhq.org/.

[9] SimpleToDo. https://github.com/heyamykate/vanillaJS.

[10] Sudoku game. http://www.dhtmlgoodies.com/scripts/game_sudoku/game_sudoku.html.

[11] Generating fixtures for JavaScript unit testing. Implementation and experimental dataset. http://salt.ece.ubc.ca/software/confix/, 2015.

[12] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding JavaScript event-based interactions. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 367–377. ACM, 2014.

[13] M. Alshraideh. A complete automation of unit testing for JavaScript programs. *Journal of Computer Science*, 4(12):1012, 2008.

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

[15] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *Proceedings of the Symposium on Principles of Database Systems*, pages 25–36. ACM, 2005.

[16] J. Clark and S. DeRose. Xml path language (xpath). http://www.w3.org/TR/1999/REC-xpath-19991116, November 1999.

[17] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 342–351, New York, NY, USA, 2007. ACM.

[18] GitHut. A small place to discover languages in GitHub. http://githut.info, 2015.

[19] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[20] P. Heidegger and P. Thiemann. Contract-driven testing of Javascript code. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, TOOLS'10, pages 154–172, Berlin, Heidelberg, 2010. Springer-Verlag.

[21] C. S. Jensen, A. Møller, and Z. Su. Server interface descriptions for automated testing of JavaScript web applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 510–520. ACM, 2013.

[22] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE'11, pages 59–69. ACM, 2011.

[23] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 105–116. ACM, 2009.

[24] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[25] G. Li, E. Andreasen, and I. Ghosh. SymJS: Automatic symbolic testing of JavaScript web applications. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, page 11 pages. ACM, 2014.

[26] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *ESEC/FSE*, 2013.

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

[28] G. Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[29] A. Milani Fard and A. Mesbah. Feedback-directed exploration of web applications to derive test models. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 278–287. IEEE Computer Society, 2013.

[30] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 67–78. ACM, 2014.

[31] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. JSeft: Automated JavaScript unit test generation. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE Computer Society, 2015.

[32] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Building call graphs for embedded client-side code in dynamic web applications. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 518–529. ACM, 2014.

[33] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. An empirical study of client-side JavaScript bugs. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 55–64. IEEE Computer Society, 2013.

[34] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM, 2010.

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

[36] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 488–498. ACM, 2013.

[37] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE, pages 263–272. ACM, 2005.

[38] Stack Overflow. 2015 developer survey. http://stackoverflow.com/research/developer-survey-2015#tech, 2015.

[39] N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 134–153. Springer Verlag, April 2008.

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

[41] S. Wei and B. G. Ryder. Practical blended taint analysis for JavaScript. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 336–346. ACM, 2013.