Atrina: Inferring Unit Oracles from GUI Test Cases

Shabnam Mirshokraie Ali Mesbah Karthik Pattabiraman
University of British Columbia
Vancouver, BC, Canada
{shabnamm, amesbah, karthikp}@ece.ubc.ca

ABSTRACT

Testing JavaScript web applications is challenging due to its complex runtime interaction with the Document Object Model (DOM). Writing unit-level assertions for JavaScript applications is even more tedious as the tester needs to precisely understand the interaction between the DOM and the JavaScript code, which is responsible for updating the DOM. In this work, we propose to leverage existing DOM-dependent assertions in a human-written DOM-based test cases as well as useful execution information inferred from the DOM-based test suite to automatically generate assertions used for unit-level testing of the JavaScript code of the application. Our approach is implemented in a tool called Atrina. We evaluate our approach to assess its effectiveness. The results indicate that Atrina maps DOM-based assertions to the corresponding JavaScript code with high accuracy (99% precision, 90% recall). In terms of fault finding capability, the assertions generated by Atrina outperform human-written DOM-based assertions by 37% on average. It also surpasses the state-of-the-art mutation-based assertion generation technique by 29% on average in detecting faults.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Design, Algorithms, Experimentation

Keywords

Test generation, oracles, JavaScript, DOM

1. INTRODUCTION

The close relation between the Document Object Model (DOM) and the underlying JavaScript code creates an interactive web application. To check the application's behaviour from an end-user's perspective, testers often use popular frameworks such as Selenium. Using these frameworks to write DOM-based tests and assertions requires little knowledge about the internal operations performed at the client side code. Rather, the tester needs only basic knowledge of common event sequences to cover important DOM elements to assert. This makes it easier for the tester to write DOM-based test suites. On the other hand, writing unit test assertions at code-level for web applications that have rich interaction with the DOM through their JavaScript code is more tedious. To write unit-level assertions, the tester needs to precisely understand the full range of interaction between the code level operations of a unit and the DOM level operations of a system, and thus may fail to assert the correctness of a particular behaviour when the unit is used as a part of a system.

Our previous findings [21] indicate that DOM-based assertions can potentially miss the related portion of code-level failure, while more fine grained unit-level assertions are capable of detecting such faults. Furthermore, finding the root cause of an error during DOM-based testing can be more expensive than during unit testing. The inherent characteristics of unit and DOM-based tests, indicate that they are complementary and that there is a trade-off in individually using each to detect faults.

Current test generation approaches either produce unit test oracles based on mutation testing techniques [21, 14], or they rely on soft oracles [7]. Mutation-based approaches suffer from high computational cost and equivalent mutants which are syntactically different but semantically are the same as the original application. Soft oracles such as HTML validation and runtime exceptions are also limited in that they fail to capture logical and computational errors. Recently, Milani Fard et al. [9] proposed using the DOM-based test suite of a web application to regenerate assertions for newly detected states through exploring alternative paths of the application. However, the new assertions generated by this technique remain at the DOM-level without considering the relation between the JavaScript code and the DOM. In this work, we propose to exploit an existing DOM-based test suite to generate unit-level assertions at the code-level for applications that highly interact with the DOM through the underlying JavaScript code. We utilize existing DOM-dependent assertions as well as useful execution information inferred from a DOM-based test suite to automatically generate assertions used for testing individual JavaScript functions.

To the best of our knowledge this work is the first to propose an approach for generating unit-level assertions by using the existing DOM-based test suite of the application. The main contributions of our work include:

- A slicing-based technique to generate unit-level assertions capable of testing JavaScript functions by utilizing existing DOM-based test assertions;
- A technique for selecting effective DOM elements in detecting code-level faults, which remain unchecked in the existing DOM-based test suite;
- An implementation of our approach in a tool, called ATRINA;
- An empirical evaluation to assess the efficacy of the approach on four open-source web applications; The results show that the assertions generated by ATRINA surpass the fault finding capability of (1) the human-written DOM-based assertions by 37% on average, and (2) the state-of-the-art mutation-based assertion generation technique by 29% on average.

2. MOTIVATION

Unlike DOM-based testing, asserting the behaviour of a JavaScript application through unit-level tests requires a tester to check the correctness of several intermediate code-level variables and object properties. The code-level operations are mainly responsible for updating the DOM throughout the application execution. Therefore, a tester needs to analyze the relation between the JavaScript code and the DOM's evolution. We believe DOM-based assertions can be utilized as a guideline to generate unit test assertions at JavaScript code level.

Figure 1 presents a snippet of a JavaScript-based shopping cart application as well as a sample DOM-based SELENIUM test case that we will use as a running example through out this paper. The application's code (a) contains two main functions as follows:

1. `addToCart` is bound to the event handler of DOM elements with class `merchandise`. When the element is clicked, `addToCart` gets the information of the selected merchandise, and sets the quantity of the current available items by updating the `availItems` object. If a valid discount coupon exists, `addToCart` calculates the discount value, and disables the selected coupon button with ID `couponButt` by removing the corresponding class. Finally, `addToCart` updates the payable amount by setting the `payable` property of the `customer` object.

2. `viewCart` is invoked by clicking on a DOM element with ID `shopCart`. The function appends a message to a `div` element with class `shopContainer` including the final payable amount of the customer. If the coupon button with ID `couponButt` is not selected and the payable amount is equal to zero, then the empty cart message is shown.

Let's assume that in line 21 of Figure 1(a) `selItem.price`, which is assigned by the original price of the merchandise is 100, and `selItem.quantity` is 1. In line 25, the discount, which is calculated based on the `data` value of the `couponButt` element is 30. The DOM-based assertion in Figure 1(b) (line 11) checks the correctness of a text appended to a `div` element with class `shopContainer` containing the final amount payable by the customer, which is equal to 70 in this example. Analyzing the assertion in line 11 of Figure 1(b) indicates that the expected value of the assertion is directly influenced by the `payable` property of `customer` object as well as the object's property `coupon.expired` in function `addToCart`. We also infer that a variable in line 16 of Figure 1(a) which directly influences the value of `customer.payable`, is also used in updating the value of `availItems.count` in line 19.

Further, by leveraging the execution information obtained from running the DOM-based test case, we can identify DOM's evolution, which are not checked by the existing assertions, but potentially influence the fault finding capability of the test suite. For instance, DOM element with ID `couponButt` is accessed several times in function `addToCart` as well as `viewCart` as the test case in Figure 1(b) runs, however it remains unchecked. Since evolution of the `couponButt` DOM element pertains to the underlying JavaScript code, it is important to assert on code statements responsible for changing the aforementioned DOM element.

Algorithm 1: Oracle Generation

input : Test suite T; The set of test cases $tc_i \in T$
output: The ordered set of oracles $oracles$

begin
1 $trace \leftarrow \text{EXEC}(T)$
2 $domAccss \leftarrow \text{GETDOMACC}(trace)$
3 $freqAccdDOM \leftarrow \emptyset$
4 $\alpha = \frac{1}{\text{READPROPERTIES}(T)}$
5 **for** $dom \in domAccss$ **do**
6 **if** $\text{ACC}(prop_{dom}) \geq \alpha$ **then**
7 $freqAccdDOM \leftarrow dom \cup freqAccdDOM$
8 **for** $tc_i \in T$ **do**
9 $trace \leftarrow \text{EXEC}(tc_i)$
10 $domAccss \leftarrow \text{GETDOMACC}(trace)$
11 **for** $asstn \in assertions_{tc_i}$ **do**
12 $asserDOMAcc \leftarrow \text{GETDOMACC}(asstn)$
13 $asserDOMMuts \leftarrow$ $\text{GETDOMMUTS}(asserDOMAcc)$
14 **for** $domMut \in asserDOMMuts$ **do**
15 $bwSts \leftarrow \text{GETBWSLICE}(domMut, trace)$
16 $expAsstnRel \leftarrow \text{GETWRVARS}(bwSts)$
17 $fwSts \leftarrow \text{GETFWSLICE}(bwSts, trace)$
18 $impAsstnRel \leftarrow \text{GETWRVARS}(fwSts)$
19 $cndDOMMuts \leftarrow \text{GETDOMMUTS}(freqAccdDOM)$
20 **for** $domMut \in cndDOMMuts$ **do**
21 $bwSts \leftarrow \text{GETBWSLICE}(domMut, trace)$
22 $cndAsstn \leftarrow \text{GETWRVARS}(bwSts)$
23 $explicitAsstn[func]_{f=1}^{n} \leftarrow$ $\text{ACCESSIBLES}([func]_{f=1}^{n}, [expAsstnRel])$
24 $implicitAsstn[func]_{f=1}^{n} \leftarrow$ $\text{ACCESSIBLES}([func]_{f=1}^{n}, [impAsstnRel])$
25 $candidateAsstn[func]_{f=1}^{n} \leftarrow$ $\text{ACCESSIBLES}([func]_{f=1}^{n}, [cndAsstn])$
26 $oracles[func]_{f=1}^{n} \leftarrow$ $\{explicitAsstn \cup implicitAsstn \cup candidiateAsstn\}$
27 return $(oracles[func]_{f=1}^{n})$

3. OVERVIEW OF APPROACH

An overview of our unit-level assertion generation technique is depicted in Figure 2. At a high level, our approach generates unit-level assertions by utilizing human written DOM-based tests and assertions. Our code level assertions fall in the following three categories: (1) explicit

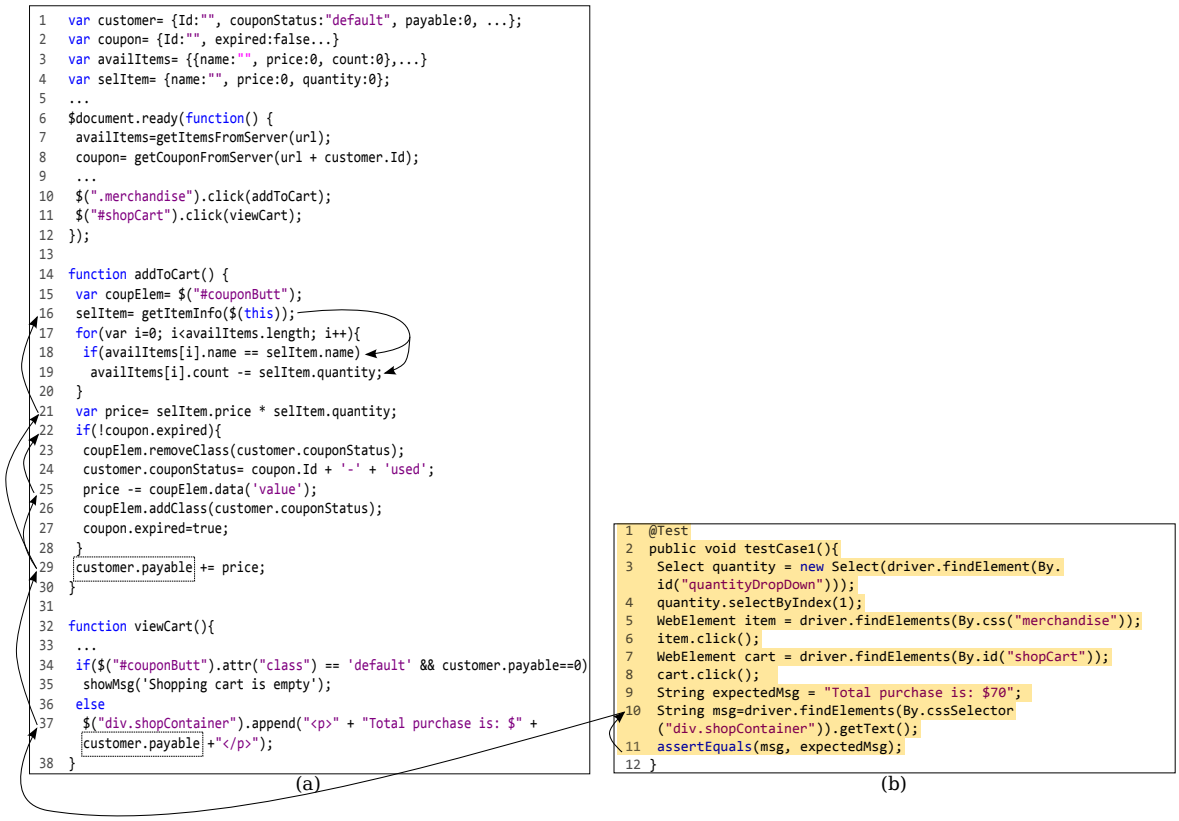

Figure 1: Running example (a) JavaScript code, and (b) DOM-based test case. The line from (b) to (a) shows the point of contact between the DOM assertion and the code. The arrow lines in (a) show the backward as well as forward slices between JavaScript statements.

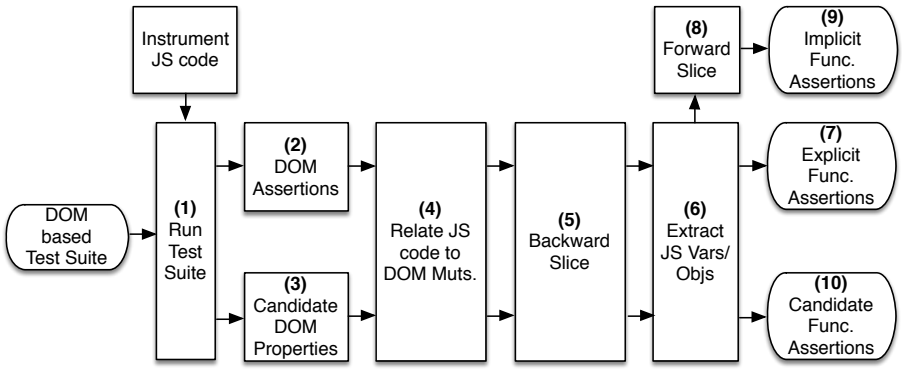

Figure 2: Overview of our assertion generation approach.

assertions, which are directly inferred from analyzing the manually written DOM-based assertions, (2) implicit assertions, which are indirectly affected by the human written DOM-based assertions, and (3) candidate assertions, which are not considered in the written DOM-based assertions, yet are potentially useful to be checked by the function level test suite. We describe our approach below. The numbers below in parentheses correspond to those in the boxes of Figure 2.

In the first part of our approach we (1) execute the instrumented application by running the existing DOM-based test suite. In this step we gather a detailed execution trace of the application. We then extract (2) DOM-based assertions, and (3) candidate DOM element properties, which are useful DOM properties that can potentially be utilized for the purpose of assertion generation. We (4) identify the initial point of connection between the application's source code and checked DOM element. We (5) calculate the backward slice of the DOM mutating statements to find the entire code blocks that update the checked DOM element. We then (6) extract accessible entities from the obtained statements. Accessible entities form our explicit assertions (7). We further (8) perform a forward slice on the extracted entities to identify statements, that are implicitly affected by such entities. The accessible entities associated with the collected

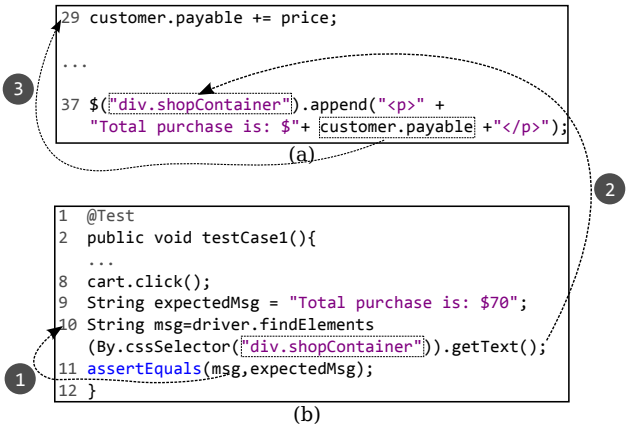

Figure 3: Finding (1) intra DOM assertion dependency within the test case (b), (2) inter DOM assertion dependency between (b) DOM-based assertion and (a) the JavaScript code, and (3) the initial point of contact between (b) DOM-based assertion and (a) the JavaScript code.

statements form our implicit assertions (9). In addition to explicit and implicit assertions, we also generate candidate assertions (10). Candidate assertions are involved with updating potentially useful DOM element properties, which are not checked in the existing DOM-based assertions. To obtain candidate assertions, we perform step (4), (5), and (6) on the inferred candidate DOM element properties (3).

Our overall unit-level assertion generation is presented in Algorithm 1. In the following sections we describe our technique for extracting DOM related information from the execution (Section 3.1), relating DOM mutations to the JavaScript code (Section 3.2), and generating unit test assertions (Section 3.3).

3.1 Extracting DOM-Related Characteristics

The DOM connects a test case to the web application's code. Therefore, we first need to analyze the DOM-based test suite and extract the following pieces of information: (1) DOM-related operations of the existing test suite that may have tight connection with the JavaScript code, and (2) frequently accessed DOM properties, which are potentially influential in improving the fault finding capability of the test suite, but left unchecked in the manually-written test suite.

DOM-Related Operations. Any written test case needs to check the correctness of the application's behaviour. In a DOM-based test case the expected behaviour is checked through DOM-based assertions. DOM-based assertion is defined as $< domProps, expVal >$, where $domProps$ consists of one or more DOM element features (e.g. attribute, and/or textual value), and $expVal$ is the correct value expected by the assertion. Through the rest of the paper, we call the DOM element feature as a DOM property. DOM-based assertions play a significant role in our approach as they can guide us towards important portions of the underlying JavaScript code that need to be checked in unit-level assertions. For each DOM-based assertion we find *intra DOM assertion dependency* within the test case.

Definition 1 (Intra DOM Assertion Dependency). *An intra DOM assertion dependency is defined as a three tuple of $< assert, domElems, domProps >$, where assert is the intended DOM-based assertion, domElems is the accessed DOM elements in the test case pertaining to the assertion, and domProps is the accessed DOM properties within the assertion.*

GetDomAcc in line 10 of Algorithm 1 retrieves DOM dependencies of the assertion in the test case. Going back to our example in Figure 3(b), tracking the assertion in line 11 shows that it has a DOM dependency to a `div` element with class `shopContainer`, which is accessed in line 10. The intra DOM assertion dependency of the example further shows that the `text` value of the DOM element is compared with the `expectedMsg` in line 11.

We further need to correlate the inferred intra DOM assertion dependency with the application's code. We call the correlation between the DOM-based assertion and the application's code as *inter DOM assertion dependency.*

Definition 2 (Inter DOM Assertion Dependency). *An inter DOM assertion dependency is defined as $< assert, initPoint >$, where assert is the intended DOM-based assertion, and initPoint is the initial line of code in the application that is responsible for mutating the property of a DOM element extracted from the intra DOM assertion dependency.*

In order to find the initial point of contact between the application's code and a mutated DOM property in the DOM-based test case, we track evolution of the accessed DOM elements (GetDomMuts in line 13 of the algorithm) as well as invoked event handlers as the test case runs. We consider additions and removals of child nodes, changes to attributes, and updates to child text nodes as DOM mutations. For instance, running the sample test case in Figure 1(b) results in mutating (1) the textual value of `div` element with class `shopContainer`, and (2) the `class` attribute of DOM element with ID `couponButt`.

In Section 3.2, we explain inferring the initial point of contact between the source code and a mutated DOM element in a DOM-based test suite in details.

Frequently Accessed DOM Properties. In addition to DOM-based assertions, we further consider DOM element properties, that are frequently accessed within the application as the test case runs (lines 1 to 7 of Algorithm 1). Acc in line 6 of the algorithm computes the access frequency of a DOM property, $freqAccdDOM$ in line 7 contains the inferred candidate DOM properties, and GetDomMuts in line 19 records DOM mutations occur on candidate DOM properties. The intuition is that frequent use of a given DOM property can point to the extent of application's behaviour dependency on the DOM property. Thus, if changes happen to that property through the JavaScript code, it is important to assert the correctness of such mutations. We define the access frequency of a DOM element property as the number of times that the element's property has been read during the execution of a test case. DOM properties include attributes as well as textual value of the elements. In order to record DOM property accesses within the application, we rewrite native function calls used by programmers to access DOM element such as `getElementById`, `getElementsByClassName`, and/or `getElementsByTagName`. The returned object from these functions is later used to access

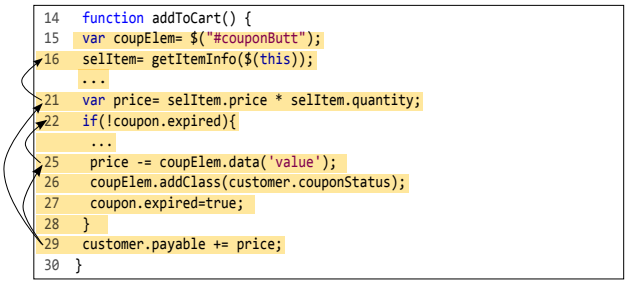

Figure 4: Intra code dependency through backward slicing.

attributes or textual values of the element. Thus, we apply a forward slice on the returned object to find instances of element's property access in the code. For example in function `addToCart` of Figure 1(a), DOM element with ID `couponButt` is assigned to `coupElem` variable. The assigned variable is later used to access the `class` attribute as well as the `value` of the DOM element in lines 23, 25, and 26.

Let $Acc(prop_{el})$ be the access frequency computed for property *prop* of DOM element *el*, then:

$Acc(prop_{el}) = \frac{Read(prop_{el})}{\sum_{e=1}^{n} Read(domElem_e)}$, where $Read(domElem_e)$ is the number of times that DOM element *domElem* is read, given that the total number of DOM elements during the execution of a test case is n. Note that reading a DOM element refers to accessing the element to read the corresponding property. In Figure 1(a), the `class` attribute of DOM element `couponButt` is read in lines 23 and 34, and thus the access frequency computed for the `class` attribute of the element is equal to $\frac{2}{3}$.

We choose element's property with access frequencies above a threshold α as potential candidates, which are later used for the purpose of unit-level assertion generation. We automatically compute this threshold for each test case as:

$\alpha = \frac{1}{ReadProperties(T)}$, where $ReadProperties(T)$ is the total number of properties which have been read during the execution of test suite T.

Going back to our running example and the sample DOM-based test case in Figure 1, `class` attribute of the `coupon-Butt` is selected as a potential candidate since its access frequency ($\frac{2}{3}$) is greater than the computed threshold, which is equal to $\frac{1}{2}$ in this example.

3.2 Relating DOM Changes to the Application's Code

To determine the initial point of contact between DOM and the underlying application's code, we first cross reference the DOM element as well as the property we are interested in with a set of DOM mutations obtained from the execution trace. The desired DOM element and its property are inferred from either the intra DOM assertion dependency or the candidate DOM properties as described in Section 3.1. Recall that our execution trace contains information about triggered events, event handlers, and DOM mutations caused by the events. Therefore, we can identify relevant events and invoked functions corresponding to a given DOM mutation. For example, the collected execution trace in Figure 3 contains information about the mutations of a `div` element with class `shopContainer`, which pertains to the DOM-based assertion.

To figure out where the mutation originated in our execution trace, we keep record of DOM accesses within the invoked functions. For each DOM access, we track JavaScript lines of code that are responsible for updating the corresponding DOM element. Going back to our example in Figure 3, given that the textual property of the `div` element is extracted from the intra DOM assertion dependency, we identify line 37 in function `viewCart` as the initial point of contact responsible for changing the `text` of DOM element.

After inferring DOM mutant statements, we identify the *intra code dependency* within the application's code.

Definition 3 (Intra Code Dependency). *An intra code dependency is defined as $< criterion, codeSts >$, where criterion is a variable at the initial point of contact, and codeSts is the set of statements that are either affected by or influence the criterion.*

To find the intra code dependency, we perform backward as well as forward slicing techniques by using *criterion* as the slicing criterion. GetBWSlice in lines 15 and 21 of Algorithm 1 computes a backward slice with respect to assertion related DOM mutations, and candidate DOM property mutations respectively. We use dynamic slicing to capture run-time dependencies accurately. Note that instrumenting the entire application's code to perform dynamic slicing incurs high performance overheads. To avoid this, we first intercept the code sent from the server to the client, and then statically instrument only those statements that may affect a given DOM element. To extract the subset of the code statements, we first find the JavaScript closure scope which contains the definition of the variable in the initial slicing criteria. Then all references to the variable within the closure scope are found. Therefore, we can identify all locations in the code where the variable is updated, read, or a new alias is created. For each variable update/read related to the variable of the slicing criteria, we track the data dependencies for such an operation. The aforementioned steps are performed iteratively for each dependencies to collect the subset of code statements, which are instrumented for a given initial slicing criteria. The instrumented code keeps track of all updates and accesses to all relevant data and control dependencies. Once the test case runs, we collect traces from the instrumented code. This trace is used to dynamically extract backward slicing as well as forward slicing statements. Note that in addition to backwards slicing which is later used to generate explicit assertions, we also use forwards slicing to generate our implicit assertions (Section 3.3.2).

The backwards slicing technique starts by extracting instances of the initial slicing criteria from the trace. For each *read* operations, the trace is traversed backwards to find the nearest related *write* operation. Once found, the *write* operation is added to the slice under construction. This process is repeated for all the data dependencies related to that write operation. A similar approach is taken for including control dependencies in the slice. Our slicing technique supports inter procedural slicing. For example, if a variable is assigned by the return value of a called function, the slicer recursively tracks the function and performs a backward slice on the statement returned by the called function.

To address aliasing when computing the slice of a variable that has been set by a non-primitive value, we need to consider possible aliases that may refer to the same object. Specifically in JavaScript *dot notation* and *bracket notation* are frequently used to modify objects at run time. Since static analysis techniques often ignore this issue [10], we use dynamic slicing. If a reference to an object of interest is saved to a second object's property, e.g. through the use of the *dot notation*, the object of interest may also be altered via aliases of the second object. For example, after executing statement `a.b.c = objOfInterest;`, updates to `objOfInterest` may be possible through `a`, `a.b`, or `a.b.c`. To deal with such scenarios, our slicing technique searches through the collected trace and adds the forward slice for each detected alias to the current slice for our variable of interest (e.g. `objOfInterest`).

Given `customer.payable` as the initial slicing criteria in our example, Figure 4 shows the relevant backward slice statements (lines 29, 25, 22, 21, and 16), where `customer.payable`, variable `price`, as well as properties of the object `selItem` are assigned, and the value of `coupon.expired` is checked in the conditional statement. By the end of backward slicing step, we collect all the relevant statements to a given DOM element. These are later used to derive test assertions.

3.3 Generating Unit-Level Assertions

Our approach targets postcondition assertions which are used to examine the expected behaviour of a given function after it is executed in a unit test case. By analyzing a given DOM-based test case, we generate unit-level assertions in the following three categories: (1) explicit assertions, (2) implicit assertions, and (3) candidate assertions.

3.3.1 Explicit Assertions

After collecting all the statements, that are relevant to a given DOM-based assertion, we extract accessible entities from these statements (ACCESSIBLES in line 23 of the algorithm). Types of accessible entities include (1) the function's returned value, (2) the used global variables in that function, (3) the object's property where the object is accessible in the outer scope of the function, and/or (4) the accessed DOM element in that function. Dynamic backward slice of a DOM-based assertion helps to (1) track all statements that contribute to the checked result and as such identify those entities that might have influenced the checked property value of the DOM element, and (2) eliminate unrelated entities that are not involved in the computation that leads to the update performed on the checked DOM element.

Since our dynamic slice is extracted from the program run, we can track all concrete values associated with accessible entities. During the run of a test case, there might be different instances where a given statement is executed. Different execution instances can lead to different behaviour. Since we are using dynamic slicing, an instance that leads to the required behaviour, which is checked through the DOM-based assertion, is on the backward slice. Given that the manually-written expected value, that is checked against the DOM's property is valid, the concrete values of related entities in the backward slice are potentially correct. Therefore, concrete value of an entity in the backward slice can be used as the expected value of the entity in unit-level assertions to test the current version of the application (discussed in Section 4.4). *explicitAsstn* in line 23 of Algorithm 1 contains the inferred explicit assertions.

```
14  function addToCart() {
15    var coupElem= $("#couponButt");
16    selItem = getItemInfo($(this));
17    for(var i=0; i<availItems.length; i++){
18      if(availItems[i].name == selItem.name)
19        availItems[i].count -= selItem.quantity;
20    }
      ...
    }
```

Figure 5: Intra code dependency through forward slicing.

In our running example (Figure 4), explicit assertions check the correctness of `customer.payable`, `coupon.expired`, as well as `price` and `quantity` properties which belong to `selItem` object. Assuming that the original price of the item is 100, the number of selected item is 1, and the calculated discount according to the `value` attribute of a DOM element with ID `couponButt` is 30, then the expected values included in the assertions for each of the entities are 70, boolean value `true`, 100, and 1, respectively.

3.3.2 Implicit Assertions

We gather all the statements that explicitly affect the computations relevant to a given DOM-based assertion. While assertions inferred from such statements are inherently important, we further need to consider entities that are implicitly influenced by the checked DOM element in the manually-written test suite. For this purpose we apply a dynamic forward slice on the statements collected from a backward slice of a DOM-based assertion. A forward slice with respect to a statement *st*, indicates how subsequently an operand at *st* is being used. This can help the tester to ensure that *st* establishes the expected outcome of the computations assumed by later statements.

GETFWSLICE in line 17 of the algorithm computes forward slice on the variable operands of a statement in the backward slice. The process of forward slicing is similar to the backward slicing technique discussed earlier (Section 3.2). The slicing criterion of the forward slice module is either a variable, object's property, or an accessed DOM property extracted from the statements in a backward slice segments of the code. The accessible entities (ACCESSIBLES in line 24), which have been set within the collected forward slice statements establish the implicit assertions. *implicitAsstn* in line 24 of Algorithm 1 contains the inferred implicit assertions. Figure 5 shows the intra code dependency obtained by performing forward slicing on the running example. As shown in the figure, the properties of object `selItem` are set in line 16, that is recorded during the backward slice process. Given line 16 as the forward slice criteria, we mark `availItems.count` (line 19) as an implicit assertion.

3.3.3 Candidate Assertions

In addition to explicit and implicit assertions, we also verify the correctness of code-level entities pertaining to DOM updates, which are essentially important but not checked in the existing DOM-based test cases. We derive such unit-

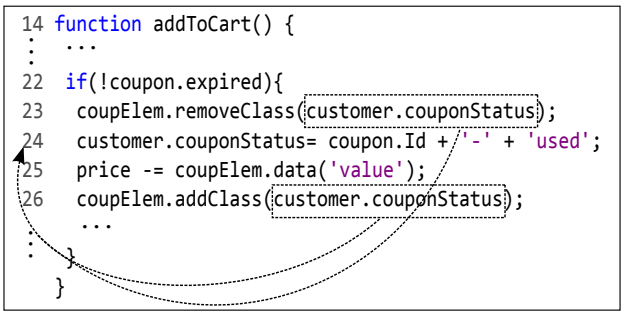

Figure 6: Relating candidate DOM element to JavaScript code.

level assertions, namely candidate assertions, from the candidate DOM element properties previously obtained from the test case execution (box 3 in Figure 2). As the test case runs, we monitor the DOM's evolution and match the list of mutated DOM elements and their properties with property updates of the candidate DOM elements. Once a match is found, we infer backwards slice statements pertaining to the mutation of DOM element's property (GETBWSLICE in line 21 of the algorithm). Therefore, in this case the slicing criteria which is given as input to the backwards slicing module is an update to the property of the candidate DOM element. After gathering the related JavaScript statements within the application, we extract accessible entities of these statements (ACCESSIBLES in line 25) which form our candidate assertions. *candidateAsstn* in line 25 contains our candidate assertions.

Recall from the running example, one such potential DOM property which we record as part of Section 3.1, is `class` attribute associated with DOM element with ID `couponButt`. As shown in Figure 6 monitoring DOM changes reveal that line 26, where the `class` attribute of the element is set, is the initial point of contact between DOM mutation and the JavaScript code. Given line 26 as the slicing criteria, `customer.couponStatus` (line 24) is marked as the candidate assertion.

3.4 Tool Implementation: Atrina

We have implemented our JavaScript unit test assertion generation in an automated tool called ATRINA. The tool is written in Java and is publicly available for download [1]. We use proxy server to intercept HTTP responses which contain JavaScript code. JavaScript Mutation Summary library [3] is used to track DOM changes during the execution of the test suite. Trace information is collected by the proxy once received from the browser. To instrument Selenium test cases we convert them into an abstract syntax tree (AST) by employing Eclipse Java development tools (JDT). Once the transformation is done, we run the Java code of the changed AST on the application under test.

4. EMPIRICAL EVALUATION

To quantitatively assess the efficacy of our test generation approach, we have conducted a case study, in which we address the following research questions:

RQ1 How accurate is ATRINA in mapping DOM-based assertions to the corresponding JavaScript code?

Table 1: Characteristics of the experimental objects.

ID	Name	LOC (JS)	# Test Cases	# Assertions
1	Phormer	1.5K	7	18
2	EnterpriseStore	57K	19	17
3	WolfCMS	1.3K	12	42
4	Claroline	36K	23	35

RQ2 How effective is ATRINA in generating unit test assertions that detect faults?

RQ3 Is our approach more effective than DOM-based assertions written manually by the tester in terms of fault finding capability?

RQ4 How does our approach compare to the existing mutation-based technique for identifying unit test assertions?

ATRINA and the experimental data are available for download [1].

4.1 Objects

Our study includes four open source JavaScript web applications that have SELENIUM test cases. Table 1 presents the experimental objects and their properties. Phormer [4] is a photo gallery web application. EnterpriseStore [6] is an asset management web application. WolfCMS [5] is a content management system, and Claroline [2] is a collaborative online learning and course management system.

4.2 Setup

To address our research questions, we provide the URL as well as the available manually written DOM-based test suite of each experimental object to ATRINA. Unit level test assertions are then automatically generated by the tool.

Accuracy (RQ1). To evaluate the accuracy of ATRINA, we measure precision and recall. We manually compare the slices generated by ATRINA with the JavaScript code that is relevant to each assertion. Precision and recall are defined as follows:

Precision is the fraction of lines in a slice produced by ATRINA, that are actually related to the human-written DOM-based assertion: $\frac{TP}{TP+FP}$

Recall is the fraction of the correct set of related lines of code to each assertion, which is actually present in the slice produced by ATRINA: $\frac{TP}{TP+FN}$

where TP (true positives), FP (false positives), and FN (false negatives) respectively represent the number of lines of code that are correctly reported, falsely reported, and missed to report as related to the DOM-based assertion.

Effectiveness (RQ2). To assess the effectiveness of ATRINA, we measure the fault finding capability of the assertions generated by the tool. Moreover, to understand the effect of each type of assertion produced by ATRINA in detecting faults, we compare the fault detection rate of using (1) exclusively explicit assertions, (2) explicit assertions and implicit assertions, and (3) explicit assertions and candidate assertions. Since explicit assertions compose the core body of our assertions, we consider implicit and candidate assertions in conjunction with explicit ones rather than separate them.

The experimental objects do not come with a rich version history to apply ATRINA on real regression changes. Therefore we mimic regression faults by automatically injecting mutations to the application, and evaluate the tool's

ability in detecting the seeded faults. Using our recently developed mutation testing tool, MUTANDIS [19], we automatically inject 50 random first-order mutations into the JavaScript code of the applications. The mutation operators are chosen from a list of common operators such as changing the value of a variable, modifying a conditional statement, altering unary operations, as well as common mistakes made by developers when developing a given web application [20], e.g., changing the ID/tag name passed into DOM access functions such as `getElementById` or `getElementsByTagName`, and modifying the attribute name/value in `setAttribute`. The fault is considered detected if an assertion generated by ATRINA fails when run on the mutated code, and our manual examination confirms that the failed assertion is detecting the seeded fault.

Comparison with human-written DOM-based Assertions (RQ3). To assess the usefulness of ATRINA, we compare the human written DOM-based assertions with the unit-level test assertions generated by our approach in terms of fault finding capability. Similar to RQ2, we perform fault injection on both. The faults injected into our experimental objects in response to RQ3 are the same as the ones that we seed in applications to answer RQ2.

Comparison with Mutation-based Assertion Generation (RQ4). To assess how ATRINA performs with respect to the current state-of-the-art oracle generation technique, we compare our tool's fault finding capability with the mutation-based assertion generation approach [21, 14]. To generate mutation-based assertions for the JavaScript code, we use human-written DOM-based test suite as a means to execute the application and infer the execution traces required for the purpose of mutation analysis. We perform the following steps to generate test assertions using mutation analysis.

1. Remove assertions from the human-written DOM-based test suite.
2. Execute the test suite on the original version of the application to obtain execution traces.
3. Inject mutations for the purpose of oracle generation.
4. Execute the human-written test suite on the generated mutants, and produce test oracles by comparing execution traces obtained from the mutants and the original version of the application.

The number of mutants, that are generated to produce test assertions is 50 for each application. Note that the implementation and evaluation of the mutation analysis technique both use mutation operators from our prior work [20]. Therefore, our evaluation is biased in favour of mutation-based assertion generation approach over our technique.

4.3 Results

Accuracy (RQ1). Table 2 shows the number of correctly reported (true positive), the number of incorrect reported (false positive), and the number of missed (false negative) JavaScript lines of code, which are related to human-written DOM-based assertions. The table also shows the percentage of precision and recall achieved by ATRINA. The Recall calculated for Phormer (ID 1) and WolfCMS (ID 3) is 100%. For EnterpriseStore (ID 2) and Claroline (ID 4), the recall rate is 84% and 79% respectively. The precision computed for the Phormer and WolfCMS is 100%. For EnterpriseStore application as well as Claroline, the precision rate is 98%.

Table 2: Accuracy achieved by Atrina.

ID	# TP	# FP	# FN	Precision (%)	Recall (%)
1	174	0	0	100	100
2	861	18	162	98	84
3	193	0	0	100	100
4	1446	29	385	98	79
AVG	-	-	-	99	90

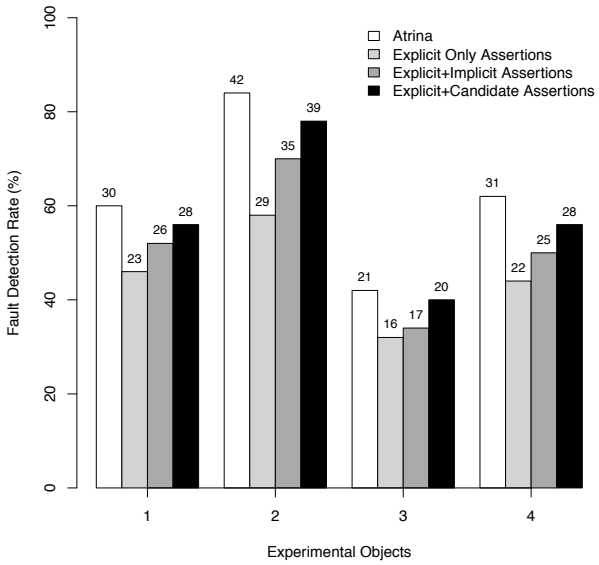

Figure 7: Fault detection rate using different types of generated assertions.

We noticed that the lower recall rate obtained by ATRINA is mainly due to the use of third party libraries. Currently, we focus only on the application source code and do not consider libraries in our slicing technique. The underlying assumption is that faults mainly originate from the application's code. The small drop observed in precision is due to the functions, that are called but are not instrumented due to limitations in our current implementation. If the definition of a called function is not instrumented, we assume that the function call is related to our slice, while it may not be. We also observed that in rare cases a variable is seemingly assigned by a return value of a function, though the `return` statement is not found in the actual code of the called function. Our current implementation includes such variable assignments in the pertaining slices. Note that both recall and precision can be improved to 100% with a more robust implementation of our technique.

Effectiveness (RQ2). Figure 7 depicts the fault detection rate (percentages) achieved by (1) ATRINA, (2) explicit assertions when included individually, and (3) explicit assertions in conjunction with either implicit assertions or (4) candidate assertions. The number on each bar represent the number of faults detected by the corresponding assertion

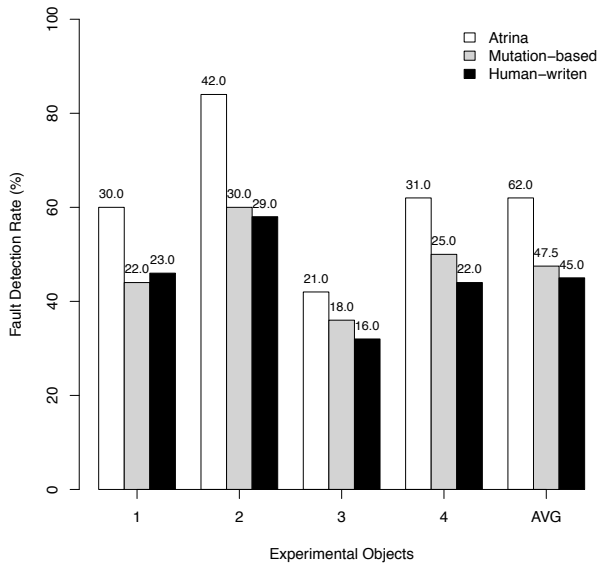

Figure 8: Fault finding capability.

types. As shown in Figure 7, ATRINA detects on average 62% of the total faults (ranges from 42-84%). The percentage of faults detected by including only explicit assertions is less than that detected through the combination of explicit with either implicit assertions or candidate assertions. This indicates that implicit as well as candidate assertions are essential entities in improving the fault finding capability of ATRINA. By eliminating implicit and candidate assertions, fault detection rate drops 27% on average and up to 31% for the EnterpriseStore application (ID 2).

Figure 7 shows that the improvement contributed by implicit assertions is 8% on average, while the improvement due to candidate assertions is 21% on average. This indicates that candidate assertions play a more prominent role in increasing the number of faults detected by ATRINA in comparison with implicit assertions. Not surprisingly, explicit assertions contribute the most among the other assertion types generated by ATRINA. Explicit assertions detect 73% of the total faults on average (ranges from 69-77%). These assertions are derived directly from the DOM-based oracles written by the developer of the application who has a deep knowledge of the application's behaviour. Therefore, it is expected that explicit assertions derived directly from such oracles have the highest impact on fault finding capability of our tool.

Comparison with human-written DOM-based Assertions (RQ3). Figure 8 compares the fault detection rate achieved by the code-level assertions generated by ATRINA with the human-written DOM-based assertions. The numbers shown on each bar represent the actual number of faults detected by the corresponding assertion generation technique. As shown in the figure, the percentage of faults found by ATRINA is higher than manually written DOM-based assertions for all applications. Overall, our approach outperforms manual assertions in terms of fault finding capability by 37% on average (ranges from 30-45%). We observed that on average, 52% of the candidate DOM properties that we select to construct our candidate assertions were ignored in human-written DOM assertions, although their values are updated through the JavaScript code. We further noticed that for each failed manual DOM assertion as a result of an injected fault, at least one explicit assertion fails in ATRINA (three failed explicit assertion on average). We observed that most often DOM assertions written by the tester are too generic in nature. Therefore even when a DOM assertion detects a JavaScript fault, pinpointing the root cause of the error can be quite challenging. However, code-level assertions make it easier for the tester to localize the fault, as they directly correlate with the code.

We observed in several cases that the value of a DOM element property that is checked in the human-written test suite is later used in a JavaScript code that involves internal computations only. If the seeded fault falls in the corresponding computational statements, the resulting error is not captured through the manually written DOM assertions. In such cases, implicit assertions are capable of detecting the error, which points to the importance of incorporating these types of assertions in our approach. We also noticed that around 33% of the faults found by implicit assertions are undetected by the human-written ones. This is because they require executing a more complex sequence of events to propagate to the observable DOM (e.g., when an object's property is assigned in a function to be later used in updating a value of a DOM element after a specific event is triggered).

Comparison with Mutation-based Assertion Generation (RQ4). Figure 8 presents the results of comparing fault finding capability of ATRINA with mutation-based assertion generation technique. As shown in the figure our approach produces unit assertions that are more effective than those produced by mutation-based technique. ATRINA surpasses mutation-based approach by 29% on average (ranges from 17-40%), although both implementation and evaluation of the mutation-based technique use common mutation operators, which favours mutation-based assertion generation as we discussed in Section 4.2. This points to the importance of incorporating the information that exists in human-written DOM-based test cases.

4.4 Discussion

Time Efficiency. While the results demonstrate that ATRINA is more effective than mutation-based approach in terms of fault detection, we further investigate efficiency of our approach in terms of time overhead. We compute overhead of ATRINA as the summation of time required for (1) instrumenting the application, and (2) analyzing the collected trace to compute JavaScript slices. To calculate time overhead of the mutation-based approach, we consider the total time required for running the test suite multiple times (once per mutation), generating mutants, as well as the time needed to compare the original and the mutated version of the application to generate assertions. Figure 9 shows the results of time overhead computed for each approach. Our results show that the time overhead for ATRINA is 52 seconds on average, while the overhead computed for mutation-based technique is 123 seconds on average. As shown in the figure, for the EnterpriseStore application (ID 2), which is the largest application we considered (57K LOC), time efficiency is increased by 58% using ATRINA. This indicates

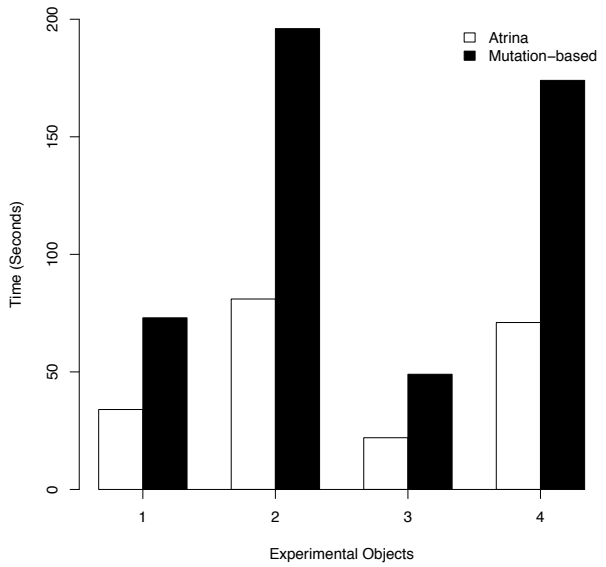

Figure 9: Time overhead for each approach.

that our approach significantly outperforms mutation-based assertion generation as far as time efficiency is concerned.

Fault Masking. As we mentioned in Section 3.3.1, the concrete value of an entity in the computed backward slice can potentially be used as the expected value of the entity in explicit assertions to test the current version of the application. The actual values of the related entities in the backward slice are correct unless there exists a masked fault which is concealed in the chain of computations and thus does not propagate to the checked state of the DOM element. However, we conjecture that fault masking rarely happens in JavaScript web applications as it is more prevalent in programs with many small expressions whose results are stored in several intermediate values. We also observed no fault masking occurrence during the evaluation of Atrina on four JavaScript applications used in this study.

Limitations. The effectiveness of the generated assertions by Atrina in terms of fault finding capability depends on the quality of human-written DOM-based test cases. If the DOM assertions contained in the DOM-based test suite check useless information, the explicit assertions obtained by our tool point to entities that may not be important from the tester's point of view. This can also negatively affect the fault finding capability of implicit assertions as they are indirectly inferred from the DOM-based assertions. Moreover, if the human-written test suite does not execute application's state with effective DOM elements, our tool is not able to infer effective candidate assertions.

4.5 Threats to Validity

An external threat to the validity of our evaluation is the limited number of JavaScript applications used to measure the effectiveness of our approach. We mitigated this threat by using web applications from various domains, code size, and functionality. Another threat concerns validating failed assertions through manual inspection that can be error-prone. To mitigate this threat, we carefully examine the code in which the assertion failed to make sure that the injected fault was indeed responsible for the assertion failure. Moreover, manual computation of the JavaScript slices to measure precision and recall is a time intensive task done by the authors of the paper, and thus could be error-prone. However, we made every effort to mitigate this threat by precisely examining the application's code.

The regression faults we inject to evaluate the effectiveness of Atrina may not be realistic. We mitigate this threat by injecting mutations that represent common JavaScript applications faults, as well as using real-world web applications, and Selenium test cases written by developers.

5. RELATED WORK

While automated test generation has significantly addressed in the literature, there has been limited work on supporting the construction of test oracles. Recently, Harman et al. [15] have conducted a comprehensive survey of current techniques used to address the oracle problem. Mesbah et al. [17] automatically produce generic invariants in a form of soft oracles to test AJAX applications. JSART [18] automatically infers JavaScript invariants from the execution traces for the purpose of regression testing. Jalangi [25] is a framework to support writing of heavy-weight dynamic analyses. The framework detects generic JavaScript faults such as null, undefined values, and type inconsistencies. Jensen et al. [16] incorporate server interface descriptions to test the correctness of communication patterns between client and server through learning the communication patterns from sample data in AJAX applications. Xie et al. explore test oracle generation for GUI systems [29]. Eclat [22], and DiffGen [27] are used for automatically generating invariant-based oracles. Our work is different from these approaches in that we use the available DOM-related information in a human written test suite to infer unit-level assertions at the JavaScript code-level. Moreover, we generate assertions that capture application's behaviour, rather than generic and soft oracles.

Fraser et al. [14] propose a mutation-based oracle generation system called μTEST. μTEST automatically generates unit tests for Java object-oriented classes by employing a genetic algorithm which target mutations with high impact on the application's behaviour. They further enhance the system [13] to improve human comprehension through identifying relevant pre-conditions on the test inputs and post-conditions on the outputs. The authors assume that the tester will manually correct the generated oracles. However, the results on the effectiveness of such approaches which rely on the "generate-and-fix" assumption to construct test oracles are not conclusive [12]

Staats et al. [26] propose an oracle data selection technique, which is based on mutation testing to produce oracles and rank the inferred oracles in terms of their fault finding capability. This work suffers from the scalability issues of mutant-generation based techniques as well as the problem of estimating the proper number of mutants required for generating effective oracle data set. Similar to mutation-based techniques, differential test case generation approaches [27, 8] also target generating test cases that show the difference between two versions of a program. Pastore et al. [23] exploit crowd sourcing approach to check assertions. In this approach the developer produces tests and provides sufficient API documentation for the crowd such that crowd

workers can determine the correctness of assertions. However, recruiting qualified crowd to generate test oracles can be quite challenging.

In the context of leveraging the existing test cases to generate more complex tests, Pezzè et al. [24] propose a technique to construct integration tests which focus on class interactions by utilizing the unit test cases. The integration tests are formed by combining initialization and execution sequences of simple unit tests to form new ones. However, the proposed technique does not deal with assertions. eToc [28] and EvoSuite [11] use search based techniques to evolve the initial population of test cases. Their main goal is to increase the code coverage achieved by the test suite. However, in this work our aim is to increase the fault finding capability by focusing on test assertions rather than increasing the code coverage. Milani Fard et al. [9] propose Testilizer which utilizes DOM-based test suite of the web application to explore alternative paths and consequently regenerate assertions for new detected states. Our work is different from this approach in that we exploit DOM-related information in a human written test suite to capture the behaviour of the application at the unit-level JavaScript. Furthermore, they do not generate code-based assertions which we do.

6. CONCLUSIONS

In this paper, we present an automated technique to generate unit-level assertions for the JavaScript code. Given (1) a web application that highly interact with the DOM through the underlying JavaScript code, and (2) a DOM-based test suite, we make use of the human-written DOM-based test cases to generate effective assertions that can capture regression faults in the JavaScript code. We implemented our approach in an open-source tool called Atrina. We empirically evaluated Atrina on four web applications. The results show that our approach (1) is accurate in mapping the assertions to theJavaScript code, (2) is effective in detecting injected regression faults (62% on average), (3) outperforms human-written DOM-based assertions in terms of fault finding capability by 37% on average, and (4) generates unit assertions that are more effective (29% on average) than those produced by mutation-based technique.

7. REFERENCES

[1] Atrina. `http://www.ece.ubc.ca/~shabnamm/data/atrina.zip`. Accessed: 2015-05-30.

[2] Claroline. `http://www.claroline.net/`. Accessed: 2015-03-30.

[3] Google. Mutation Summary Library. `http://code.google.com/p/mutation-summary/`. Accessed: 2014-10-30.

[4] Phormer Photogallery. `http://sourceforge.net/projects/rephormer/`. Accessed: 2015-03-30.

[5] WolfCMS. `https://github.com/wolfcms/wolfcms`. Accessed: 2015-03-30.

[6] WSO2 EnterpriseStore. `https://github.com/wso2/enterprise-store`. Accessed: 2015-03-30.

[7] S. Artzi, J. Dolby, S. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *Proc. 33rd International Conference on Software Engineering (ICSE'11)*, pages 571–580, 2011.

[8] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE Transactions on Software Engineering (TSE)*, 35(1):29–45, 2009.

[9] A. M. Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proc. IEEE International Conference on Automated Software Engineering (ASE'14)*, pages 67–78, 2014.

[10] A. Feldthaus, M. SchaÌĹfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for javascript ide services. In *Proc. International Conference on Software Engineering (ICSE)*, pages 752–761. ACM, 2013.

[11] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *Proc. International Conference on Software Testing Verification and Validation (ICST'12)*, pages 121–130. IEEE Computer Society, 2012.

[12] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated white-box test generation really help software testers? In *Proc. International Symposium on Software Testing and Analysis (ISSTA'13)*, pages 188–198. ACM, 2013.

[13] G. Fraser and A. Zeller. Generating parameterized unit tests. In *Proc. International Symposium on Software Testing and Analysis (ISSTA'11)*, pages 364–374, 2011.

[14] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)*, 38(2):278–292, 2012.

[15] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. A comprehensive survey of trends in oracles for software testing. Technical Report CS-13-01, Department of Computer Science, University of Sheffield, 2013.

[16] C. Jensen, A. Møller, and Z. Su. Server interface descriptions for automated testing of JavaScript web applications. In *Proc. European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'013)*. ACM, 2013.

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

[18] S. Mirshokraie and A. Mesbah. JSART: JavaScript assertion-based regression testing. In *Proc. International Conference on Web Engineering (ICWE'12)*, pages 238–252. Springer, 2012.

[19] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Efficient JavaScript mutation testing. In *Proc. 6th International Conference on Software Testing Verification and Validation (ICST'13)*, pages 74–83. IEEE Computer Society, 2013.

[20] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Guided mutation testing for javascript web applications. *IEEE Transactions on Software Engineering (TSE)*, 41(5):429–444, 2015.

[21] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. JSEFT: Automated javascript unit test generation. In

Proc. 8th International Conference on Software Testing Verification and Validation (ICST'15), pages 1–10. IEEE Computer Society, 2015.

[22] C. Pacheco and M. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proc. European Conference on Object-Oriented Programming (ECOOP'05)*, pages 50–âĂŞ527, 2005.

[23] F. Pastore, L. Mariani, and G. Fraser. CrowdOracles: Can the crowd solve the oracle problem? In *Proc. International Conference on Software Testing Verification and Validation (ICST'13)*, pages 342–351. IEEE Computer Society, 2013.

[24] M. Pezzè, K. Rubinov, and J. Wuttke. Generating effective integration test cases from unit ones. In *Proc. International Conference on Software Testing Verification and Validation (ICST'13)*, pages 11–20. IEEE Computer Society, 2013.

[25] K. Sen, S. Kalasapur, T., and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proc. European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'013)*. ACM, 2013.

[26] M. Staats, G. Gay, and M. Heimdahl. Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing. In *Proc. International Conference on Software Engineering (ICSE'11)*, pages 870–880, 2011.

[27] K. Taneja and T. Xie. DiffGen: Automated regression unit-test generation. In *Proc. International Conference on Automated Software Engineering (ASE'08)*, pages 407–410. IEEE Computer Society, 2008.

[28] P. Tonella. Evolutionary testing of classes. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04)*, pages 119–128, 2004.

[29] Q. Xie and A. M. Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(1), 2007.