Detecting Unknown Inconsistencies in Web Applications

Frolin S. Ocariza, Jr. Karthik Pattabiraman Ali Mesbah
University of British Columbia, Vancouver, BC, Canada
{*frolino, karthikp, amesbah*}*@ece.ubc.ca*

***Abstract*—Although there has been increasing demand for more reliable web applications, JavaScript bugs abound in web applications. In response to this issue, researchers have proposed automated *fault detection* tools, which statically analyze the web application code to find bugs. While useful, these tools either only target a limited set of bugs based on predefined rules, or they do not detect bugs caused by cross-language interactions, which occur frequently in web application code. To address this problem, we present an anomaly-based inconsistency detection approach, implemented in a tool called HOLOCRON. The main novelty of our approach is that it does not look for hard-coded inconsistency classes. Instead, it applies subtree pattern matching to infer inconsistency classes and association rule mining to detect inconsistencies that occur both within a single language, and between two languages. We evaluated HOLOCRON, and it successfully detected 51 previously unreported inconsistencies – including 18 bugs and 33 code smells – in 12 web applications.**

***Index Terms*—JavaScript, fault detection, cross-language interactions**

I. INTRODUCTION

The JavaScript programming language has rapidly grown in popularity over the past decade, even topping the "Most Popular Technologies" category of the two most recent StackOverflow Developer Surveys [1]. Although JavaScript programming has extended to the full web stack, its most frequent usage remains at the client-side. Unfortunately, despite its popularity, JavaScript is still notoriously error-prone [2], and these errors often lead to high-impact consequences such as data loss and security flaws [3], [4]. To mitigate this problem, web developers rely heavily on testing, and many tools have been developed for testing [5], [6], [7], [8], [9].

To complement testing, developers use static code analysis tools, which find bugs by *reasoning* about the program, without having to execute it. Several techniques have been proposed to automatically detect JavaScript bugs through static analysis. For example, JSLint [10] detects syntactic errors in JavaScript programs; Jensen et al. [11], [12] analyze JavaScript code to find type inconsistencies; Ocariza et al. propose Aurebesh [13] for automatically detecting inconsistencies in AngularJS web applications. A common issue with the above techniques is that they detect bugs based on a *predefined* list of inconsistency rules or bug patterns. As a result, the issues they detect will be limited to those encompassed by these hardcoded rules. This is especially problematic for web applications which use a wide variety of frameworks (e.g., AngularJS, BackboneJS, Ember) and libraries, each with its own coding rules and conventions. Moreover, web frameworks typically evolve fast, and hence hardcoded rules may become obsolete quickly, thereby necessitating expensive updates.

In this paper, we propose an anomaly-based inconsistency detection approach for JavaScript-based web applications. Anomaly-based approaches [14] learn the rules based on code samples, and can hence find unknown inconsistencies without hardcoded rules or patterns. Our approach differs from prior anomaly-based approaches in that it is cross-language and can hence find inconsistencies within and between two different languages, namely HTML and JavaScript. Prior work has shown that many cross-language interactions are highly error-prone in web applications [3], and hence it is important to consider such cross-language inconsistencies. In addition to bugs, JavaScript is also prone to *code smells* [15], which are pieces of code that are difficult to maintain, and are therefore prone to becoming an error when the code is modified. These code smells often also manifest as inconsistencies, particularly when there is a deviation in the code style. Therefore, by applying inconsistency detection, we can also detect (many) code smells, because our approach looks for code deviations.

We focus on detecting inconsistencies in *MVC applications* – that is, web applications implemented using JavaScript Model-View-Controller (MVC) frameworks such as AngularJS, BackboneJS, and Ember.js. We target these MVC frameworks due to their rising popularity [16], and because they do not interact directly with the DOM (Document Object Model). This makes them more amenable to static analysis than non-MVC applications. We make the following contributions:

(1) We demonstrate that there are many inconsistency classes in MVC applications, and that there is no single inconsistency class that dominates over the others. Further, many of these inconsistencies span multiple programming languages, thereby motivating approaches such as ours;

(2) We propose a technique for automatically detecting inconsistencies in JavaScript MVC applications. Unlike prior work, our approach does not look for hard-coded inconsistency classes, but instead uses subtree pattern matching to infer these classes. Further, it uses association rule mining to find the cross-language links;

(3) We implement our technique in a tool called HOLOCRON and we evaluate it on 12 JavaScript applications, from three different MVC frameworks. We find that HOLOCRON can find a total of 18 unknown bugs in these applications, five of which are cross-language. Further, HOLOCRON finds 33 code smells in these applications.

II. Background and Motivation

We target a general class of bugs that we call *inconsistencies*, in JavaScript MVC applications. As the name suggests, an MVC application consists of a model, which defines the application data; a controller, which defines the functions that manipulate the values of the application data; and a view, which uses the data and functions defined in the model and controller to define a user interface. Static analysis is sufficient for MVC applications, as they rely primarily on JavaScript bindings instead of DOM interactions; hence, even though the DOM still changes, the JavaScript code interacts primarily with these static bindings instead of directly with the DOM.

A. Definitions

We define a *code component* to be any contiguous piece of JavaScript or HTML code that could span a single line (e.g., function call, HTML text, etc.) or multiple lines (e.g., function definition, view definition, etc.). These code components can be represented by subtrees of the JavaScript code's Abstract Syntax Tree (AST) or the HTML code's DOM representation; we use these subtrees in our design (Section III).

Definition 1 (Inconsistency): Two code components C_A and C_B are inconsistent if C_A makes an erroneous assumption about C_B, where the erroneous assumption can be implicitly inferred from the code (e.g., without having to rely on specifications). The pair (C_A, C_B) is an inconsistency.

Therefore, an inconsistency is a bug that can be discovered without the help of external specifications, and hence can be detected through an automated analysis of the web application code. An inconsistency is considered *cross-language* if C_A and C_B belong to different programming languages, i.e., in our work, HTML and JavaScript.

Ideally, we would like to be able to label a web application as inconsistent by using a search approach that finds *all* inconsistent code components as described above. However, no such approach currently exists, nor do we aim to propose such an approach in this paper. Our goal, rather, is to find as many of these inconsistencies as possible by detecting anomalies in the AST and the DOM. The approach is described in Section III, and its differences compared to other anomaly detection techniques are outlined in Section VII.

A recent study provides some evidence that four classes of these inconsistencies occur in MVC applications [13]. For example, the view components in the HTML code use variables that are erroneously assumed to be defined in the model components in the JavaScript code. In Section V-D, we find through a study of bug reports that these inconsistencies abound in MVC applications, and often go much beyond the classes found in this prior study. Thus, this prior approach will not work for these other classes.

B. Motivating Examples

To illustrate the problem, we introduce examples of two real bugs and one code-smell that result from inconsistencies.

AngularJS Example. In this application [17], the JavaScript code closes a modal instance by calling the `close()` method:

```
$modalInstance.close('close');
```

However, this leads to incorrect application behaviour (i.e., a dialog box becomes broken), as the `$modalInstance` service has been replaced in the newer version of AngularJS being used by the application by `$uibModalInstance`. In this case, the function call above incorrectly assumes that the service object being dereferenced is valid, thereby leading to the inconsistency. This example demonstrates the potential usefulness of a learning-based approach for finding these inconsistencies, as the evolution of framework APIs often modifies or introduces new coding rules.

BackboneJS Example. In this application [18], the JavaScript code is attempting to bind an element from an HTML view template to a layout view object by assigning the `el` property with an element selector, as shown below.

```
Marionette.LayoutView.extend({
  el: '.some-view',
  ...
});
```

In this case, the selector `'.some-view'` does not correspond to any element in the HTML template, which causes the binding to fail. In other words, the view incorrectly assumes that a particular element with the class "some-view" is defined in the HTML template. This shows the difficulty of reasoning about consistency across languages.

Code Smell. Code smells are important because if ignored, they are prone to turning into bugs, and can therefore lead the program to a faulty state. These code smells often manifest as inconsistencies in the JavaScript code, particularly when a piece of code is 'smelly' because a coding style is applied to it that *deviates from the coding style adhered to in the rest of the code*. For example, consider a function `func()` that takes a number as a parameter. Suppose that in most calls to `func()`, the argument is passed to `func()` as a named constant `NUM`; however, in one sole call, the argument is passed to `func()` as a hardcoded number literal, with the same value as `NUM`. When the developer updates the value of `NUM`, they may forget that the number literal in the latter call also needs to be updated, since it is an additional portion of the code that needs to be kept track of; in this case, a functional regression will be introduced. This example, which we discover to exist in several applications as we report later in Section V-E, demonstrates that *it is not only functional bugs that manifest as inconsistencies, but code smells as well.*

C. Challenges

One of the main challenges is that we need to infer programmer intent in order to label code components as inconsistent. For example, in the AngularJS example above, how do we know that `$modalInstance` is an incorrect service name in the absence of specifications? One approach is to leverage *repeated* appearances of the same code pattern to infer intent. Any deviations from this pattern are likely to be inconsistencies. Further, the more the examples of the same pattern,

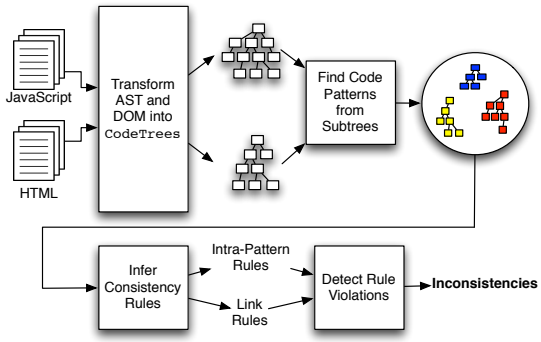

Fig. 1. Overview of our inconsistency detection approach

and the fewer the counterexamples, the more likely it is to be an actual pattern. In the AngularJS example, there are many instances of `$uibModalInstance.close(...)`, which is a near-match, though the service name is different, indicating that the service name `$modalInstance` is incorrect.

Another challenge is that we have to deal with cross-language inconsistencies, as this forces our design to infer "links" between code components coming from different programming languages. For instance, in the BackboneJS example above, our design needs to infer that the value of the `el` property needs to be a selector for an element in the HTML template. We can decide to simply hardcode this relationship in our detector, but the problem is that this link is specific to the BackboneJS framework.

III. Approach

The block diagram in Figure 1 presents an overview of our approach. As the diagram shows, our approach takes the web application's[1] JavaScript and HTML code as input, and transforms these pieces of code into their corresponding AST and DOM representations, respectively. As explained in Section III-A, the AST and the DOM trees are transformed into another tree object called a `CodeTree`, which allows the approach to perform standardized operations on those trees. In addition to the trees generated from the input web application, our technique also retrieves the AST and DOM of other web applications that use the same framework; these web applications are retrieved from the web (Section III-C).

Once the `CodeTrees` are generated for the input and sample code, the approach analyzes the trees to find commonly repeated patterns in the trees (Section III-B). To do so, it looks for *subtree repeats*, which by definition are subtrees that appear multiple times in the `CodeTrees`; these subtree repeats represent common *code patterns* in the web application.

After finding the subtree repeats, the approach examines each code pattern found in the previous module and formulates consistency rules based on them. There are two levels of consistency rules, (1) *intra-pattern consistency rules*, which are defined by the individual code patterns themselves, and (2) *inter-pattern consistency rules* (i.e., *link rules*), which are inferred based on *pairs* of code patterns. These link rules allow our approach to find consistency rules that span code written in different languages (Section III-D), unlike prior work.

Finally, our approach finds inconsistencies, based on a comparison between the `CodeTree` objects and the inferred consistency rules (Section III-E). These represent both code smells and bugs. Later, in Section V, we demonstrate the usefulness of our approach in detecting bugs and code smells.

A. Transforming Code into Trees

The first module of our approach transforms the JavaScript and the HTML code of the input web application into their corresponding AST and DOM representations. More specifically, an AST is constructed for each JavaScript file (or JavaScript code within the same `script` tag), and a DOM representation is created for each HTML file. These transformations are done to simplify analysis, as trees are a well-studied data structure for which many search and comparison algorithms have been proposed. It also makes our approach easier to extend to other languages, as it does not need complicated parsing algorithms that rely on knowledge of the syntax of specific languages.

In order to standardize the way that our approach operates on the ASTs and the DOMs, we transform them both into a data structure called the `CodeTree`. A `CodeTree` is defined as a tree $T(V,E)$, where V is the set of nodes in the tree, and E is the set of edges. For every node $v \in V$, we define:

v.type: Set to "ast" ("dom") if v is an AST (DOM) node;

v.label: Set to the node label. If v is an AST node, the label is set to either the node type (e.g., ExpressionStatement, Identifier, etc.), or the corresponding identifier or literal value. If v is a DOM node, the label is set to a tag name (`Element` node), attribute name (`Attribute` node), or text (`Text` node, or an attribute value);

In addition to the above properties, for each `CodeTree` node, we also keep track of its *parent* and *childNodes*, as well as the *lineNumber*, *columnNumber*, and *sourceFile*.

B. Finding Common Patterns

The goal of our next module is to find patterns of repeating subtrees in the `CodeTrees`. These patterns will form the basis of the consistency rules. We first define the following:

Definition 2 (Subtree Repeats): Let $T_1, T_2, ..., T_N$ be `CodeTrees`, and let $R(V_r, E_r)$ and $S(V_s, E_s)$ be two different subtrees of any of these `CodeTrees`. Then, R and S are defined to be ***subtree repeats*** of each other if R and S are isomorphic, where two nodes are considered equal iff they have the same type and label. Hence, each node $v_r \in V_r$ has a corresponding node $v_s \in V_s$ such that $v_r.type = v_s.type$ and $v_r.label = v_s.label$.

Definition 3 (Code Pattern): A ***code pattern*** C is defined as a set of subtrees, such that for every pair of subtrees $R, S \in C$, R and S are subtree repeats.

Hence, the goal of this module is to find all the code patterns in the `CodeTrees` generated earlier in the previous module. Our technique for finding these code patterns is similar to the approach used by Baxter et al. [19] to detect clones in the source code of a program using the AST. More specifically, our

[1]From here on, when we say web application, we mean MVC applications.

design looks for all *full subtrees* in each `CodeTree`, and assigns a hash value to each of these subtrees; note that a full subtree pertains to a subtree that contains *all* the descendant nodes from the subtree's root. All subtrees that hash to the same value are placed in their own hash bin. The subtrees in each hash bin are then compared to detect any collisions; if there are collisions, the hash bin is split to resolve the collisions. These hash bins represent the code patterns.

The difference with Baxter et al.'s technique is that when comparing subtrees, our design *abstracts out* the labels of nodes pertaining to variable and function identifiers, as well as attribute values. Our design also abstracts out any labels that identify the data type of a literal node (e.g., StringLiteral, NumberLiteral, etc.). Doing so enables our design to find intra-pattern consistency rules (see Section III-D).

C. Using Code Examples from the Web

In addition to the target application, our design also looks for patterns that are found in example web applications downloaded from web. The purpose of using these example applications is to allow code patterns to appear more frequently, thereby giving our design greater confidence about the validity of the pattern found. Further, using these examples will also allow "non-patterns" to appear less frequently, percentage-wise, thereby decreasing the rate of false positives.

The example web applications retrieved – chosen based on a GitHub search result of the corresponding framework's name – must use the same framework and framework version as the target web application, determined via the `script` tag of the target web application. In addition, each application must include at least one file with both a `.html` extension and a `.js` extension, not including library code included in the `lib` folder. In our experiments, we choose a total of five sample applications, as we find that this number allows our technique to find enough inconsistencies while keeping the runtime low (see Section V-G).

D. Establishing Rules from Patterns

After finding the patterns, our design then analyzes these patterns to infer consistency rules. In this case, the design looks for both intra-pattern and inter-pattern consistency rules.

1) Intra-Pattern Consistency Rules: As mentioned earlier, intra-pattern consistency rules are defined by individual code patterns. Algorithm 1 shows the pseudocode for finding these rules, and reporting violations. The main idea is to *concretize* the nodes that were abstracted out in the previous module.

The algorithm first stores each code pattern in a queue (line 2). For each code pattern C in the queue, the design determines the earliest node – in depth-first, pre-order – that is still abstracted out among the subtrees in C. It achieves this by calling the `getNextNodeToConretize()` function, which returns the pre-order number of the earliest node (line 5). Once the pre-order number of the earliest node is determined, the actual nodes in the subtrees in C that correspond to this pre-order number are compared and marked as concretized (lines 11-12), and the subtrees are partitioned according to the label of the concretized node (lines 13-18). The partitions are included in an associative array called *subPatterns* (line 9).

Algorithm 1: FindIntraPatternInconsistencies

```
   Input: C_set: The set of code patterns
   Input: t: The threshold for dominant subpatterns
   Output: PI: Set of intra-pattern inconsistencies
 1 PI ← ∅, remaining ← ∅;
 2 codePatternQueue ← {C | C ∈ C_set};
 3 while codePatternQueue is not empty do
 4     C ← codePatternQueue.dequeue();
 5     preorderNum ← getNextNodeToConcretize(C);
 6     if preorderNum < 1 then
 7         remaining ← remaining ∪ {C}; continue;
 8     end
 9     subPatterns ← ∅;
10     foreach subtree S ∈ C do
11         node ← getPreOrderNode(S, preorderNum);
12         markAsConcretized(node);
13         if subPatterns.hasKey(node.label) then
14             subPatterns[node.label].add(S);
15         end
16         else
17             subPatterns[node.label] = {S};
18         end
19     end
20     D ← getDominantPattern(subPatterns);
21     if 100 |D|/|C| >= t then
22         expected ← getPreOrderNode(D[0], preorderNum);
23         foreach code pattern CP ∈ subPatterns do
24             if CP ≠ D then
25                 foreach subtree S ∈ CP do
26                     inc ← getPreOrderNode(S, preorderNum);
27                     PI ← PI ∪ {(inc, expected)};
28                 end
29             end
30         end
31         codePatternQueue.enqueue(D);
32     end
33     else
34         codePatternQueue ← codePatternQueue ∪ subPatterns;
35     end
36 end
37 C_set ← mergeRemaining(remaining);
```

Once the partitions are found, the algorithm looks for the *dominant pattern*, which represents the largest partition (line 20). If the number of subtrees in the dominant pattern constitutes greater than $t\%$ of all the subtrees in the original code pattern C, where t is a user-set threshold, all the subtrees belonging to the *non-dominant* patterns are considered intra-pattern inconsistencies (lines 22-32) and are discarded; here, an intra-pattern inconsistency is represented by a tuple of the inconsistent node – i.e., the node that was just concretized in the inconsistent subtree – and the expected node – i.e., the node that was just concretized in any subtree belonging to the dominant pattern (line 27). This process is repeated until there are no further nodes to concretize, after which all remaining partitions belonging to the same original code pattern at the start of the algorithm are merged (line 37).

As an example, consider the subtrees in Figure 2, which form a code pattern; this code pattern is found in the AngularJS example introduced in Section II-B. Here, the current node being concretized is the left-most leaf node of each subtree, which, in this case, represents the name of the service being dereferenced. The subtrees are then partitioned according to the label of this concretized node. In this case, there are two partitions – one containing the left-most subtree, with the

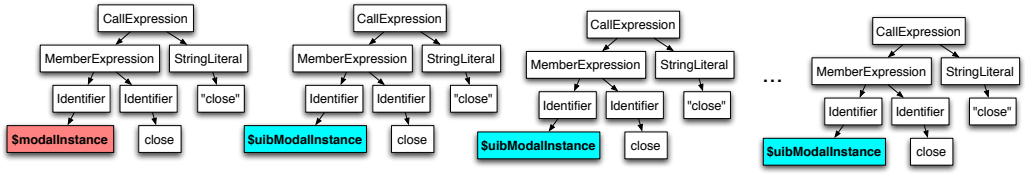

Fig. 2. Example of an intra-pattern consistency rule violation

Algorithm 2: FindLinkRules

Input: (C_{from}, C_{to}): Pair of code patterns
Output: L: Set of link rules

1 $L \leftarrow \emptyset$;
2 **foreach** $(S_{from}, S_{to}) \in C_{from} \times C_{to}$ **do**
3 $i \leftarrow 1$;
4 $node_{from} \leftarrow$ getPreOrderNode(S_{from}, i);
5 **while** $node_{from} \neq null$ **do**
6 $j \leftarrow 1$;
7 $node_{to} \leftarrow$ getPreOrderNode(S_{to}, j);
8 **while** $node_{to} \neq null$ **do**
9 **if** $node_{from} \neq node_{to}$ *and* $node_{from}.label = node_{to}.label$ **then**
10 $lr \leftarrow (i, S_{from}, j, S_{to})$;
11 $L \leftarrow L \cup \{lr\}$
12 **end**
13 $node_{to} \leftarrow$ getPreOrderNode(S_{to}, $++j$);
14 **end**
15 $node_{from} \leftarrow$ getPreOrderNode(S_{from}, $++i$);
16 **end**
17 **end**

concretized node coloured red, and another containing the rest of the subtrees, with the concretized node coloured blue. The latter partition is deemed to be dominant, so the subtree in the other partition is labeled as inconsistent.

2) Inter-Pattern Consistency Rules (i.e., Link Rules): In addition to finding the intra-pattern consistency rules, our design also looks for consistency rules that describe the relationship *between* code patterns - we call these link rules. This process allows our design to find relationships between pieces of code in the same programming language and across languages, i.e., *cross-language* relationships. All link rules are of the following form: *The i^{th} pre-order node in Subtree S_1 is equal to the j^{th} pre-order node in Subtree S_2*. Our design finds the link rules for each pair of code patterns (C_{from}, C_{to}), as shown in Algorithm 2. In this case, the algorithm iterates through every pair of subtrees between the two code patterns (line 2). For each of these pairs of subtrees, the algorithm goes through every pair of nodes between the two subtrees (lines 3-16), and compares the two nodes to see if they have the same label. If they have the same label, a new link rule is added to the list, uniquely identified by the subtree pair S_{from} and S_{to}, and their respective pre-order indices i and j.

E. Detecting Violations

Violations to the intra-pattern consistency rules are detected in conjunction with finding those rules, as described in Section III-D1. For the link rules, we make a distinction between unconditional and conditional link rule violations.

1) Unconditional Link Rule Violations: A link rule violation is unconditional if the link rule is violated by a code component regardless of where the component is located in the code. The BackboneJS example (Section II-B) is an unconditional link rule violation. To determine whether a link rule lr is violated, our design examines each pair of code patterns C_{from} and C_{to}, as before. It then determines which pairs of subtrees between C_{from} and C_{to} satisfy lr. There are two ways in which a subtree can be an inconsistency.

First, if a subtree $S_{from} \in C_{from}$ does not satisfy the link rule lr when paired with any subtree $S_{to} \in C_{to}$, *and* a large percentage $pv\%$ (a parameter chosen by the user) of the other subtrees in C_{from} satisfy lr at least once, then S_{from} will be considered an inconsistency. For instance, the left box in Figure 3 shows the code pattern to which the inconsistent code in the BackboneJS example (Section II-B) belongs. As indicated by the arrows in this figure, almost each subtree in this code pattern corresponds to a class attribute definition in the HTML code (right box in Figure 3); the only exception is the subtree with the node highlighted in red (`''some-view''`). This subtree is labeled an inconsistency, assuming $pv \leq 75\%$.

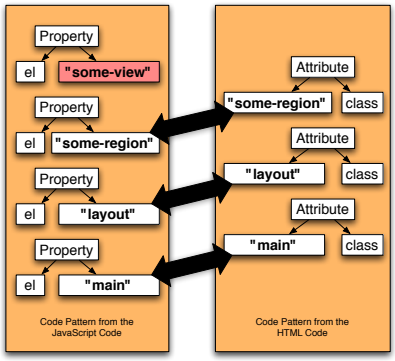

Fig. 3. Example of an unconditional link rule violation. The subtrees are slightly altered for simplicity.

Second, if a subtree $S_{from} \in C_{from}$ does not satisfy the link rule lr when paired with a *specific* subtree $S_{to} \in C_{to}$, and a large percentage of the other subtrees in C_{from} satisfy the link rule lr with S_{to}, then S_{from} will also be an inconsistency.

2) Conditional Link Rule Violations: A link rule violation is conditional if the link rule is violated *given that* the code component is located in a specific area in the code. For example, suppose a view V in the HTML code is associated with a model M in the JavaScript code. Further, suppose that the following link rule has been found: *The identifier <x> in the subtree with pattern* ***ng-model='<x>'*** *is equal to the identifier <y> in the subtree with pattern* ***$scope.<y>***. In this case, if there exists no subtrees *in the model M* with pattern **$scope.<y>** that matches a certain subtree *in the view V* with pattern **ng-model='<x>'**, then this latter subtree is

considered a violation of the link rule (i.e., V is using an identifier that is undefined in the corresponding model M). Because this link rule violation only occurs *given that* the subtrees being compared are located in M and V, this is a conditional link rule violation.

To find the conditional link rule violations, we use a well-known data mining technique called *association rule learning* [20]. This technique takes a set of *transactions* as input, where each transaction contains a set of *items* that apply to that transaction. Based on an analysis of these transactions, the technique looks for rules of the form $\{a_1, a_2, ..., a_n\} \Rightarrow \{b_1, b_2, ..., b_m\}$, where both the left and right side of the implication are subsets of all the items. In addition, the technique only reports rules that exceed a particular confidence value, i.e., the percentage of transactions that follow the rule.

Hence, when finding the conditional link rule violations between pairs of code patterns C_{from} and C_{to}, we create a transaction for each subtree pair (S_{from}, S_{to}). The items included in each transaction include all the link rules satisfied by the subtree pair, as well as the ancestor nodes of the root of each subtree; these ancestor nodes dictate which areas in the source code the subtrees are located. We use the apriori algorithm [21] to infer association rules with a confidence value greater than a user-set parameter $cv\%$; we are particularly interested in association rules of the form $\{an_{from}, an_{to}\} \Rightarrow \{lr\}$, where an_{from} and an_{to} are ancestor nodes of the subtrees S_{from} and S_{to}, respectively, and lr is a link rule. These rules are compared against each subtree pair; non-satisfying subtree pairs are reported as inconsistencies.

IV. Implementation

We implement our technique in an open-source tool called HOLOCRON[2] HOLOCRON is implemented in JavaScript as a plugin for Brackets, which is an Integrated Development Environment (IDE) for web development developed by Adobe [22]. To use HOLOCRON, the user only needs to specify the top folder of the target web application. The output of the tool is a list of the inconsistencies found; each inconsistency is shown to the user as a message identifying the inconsistent line of code, and an example of what is expected based on the consistency rule. The JavaScript code is parsed into an AST using Esprima [23], and the HTML code is parsed into its DOM representation using XMLDOM [24]. For finding the association rules, we adopt an existing implementation of the apriori algorithm [25].

V. Evaluation

A. Research Questions (RQs)

RQ1 (Prevalence of Inconsistencies): Do inconsistencies occur in MVC applications and if so, what are the characteristics of these inconsistencies?

RQ2 (Real Bugs and Code Smells): Can HOLOCRON be used by developers to detect bugs and code smells in real-world MVC applications?

[2]http://ece.ubc.ca/~frolino/projects/holocron/

RQ3 (Thresholds): How generalizable are the user-defined thresholds across applications?

RQ4 (Performance): How quickly can HOLOCRON detect inconsistencies?

B. Subject Systems

For our experiments which answer RQ2 to RQ4, we consider four open-source applications from each of the three main MVC frameworks (AngularJS, BackboneJS, and Ember.js), for a total of 12 applications. These three frameworks are the most widely used JavaScript MVC frameworks, experiencing a 538% growth in popularity from January 2013 to April 2016 [26]. The applications are listed in Table I, with the sizes ranging from 6 to 43 KB (185-1659 LOC). These sizes are representative of popular MVC applications; for example, a sample of 10 GitHub projects with at least 50 stars that use AngularJS (retrieved from the top 10 GitHub 'issue' search results) has an average of 1689 lines of code, and a median of 1104 LOC. In addition, half of these projects have fewer than 1000 LOC, not including libraries. These applications were taken from various lists of MVC applications available on GitHub [27], [28], [29]. In particular, we chose the first four applications from each framework found from these lists, filtering out the applications that did not have a GitHub repository and a working demo, as this simplified the task of reproducing the functional bugs found by our tool.

C. Experimental Methodology

Prevalence of Inconsistencies (RQ1). To answer RQ1, we manually analyze bug reports that have been filed for MVC applications on GitHub. More specifically, we examine 30 bug reports for applications implemented in each of the three main MVC frameworks – AngularJS, BackboneJS, and Ember.js – for a total of 90 bug reports. We only consider fixed or closed bugs to prevent spurious reports. To find the bug reports, we use GitHub's advanced search feature, searching in particular for GitHub issues that are given the label "bug", and whose status is "closed". We perform the same search for each of the three MVC frameworks, using the keywords "angularjs", "backbone", and "emberjs", respectively. We discard any search results that correspond to applications *not* written in any of these three frameworks, as well as results that do not pertain to the web application's client-side code. We then take the first 30 bug reports that satisfy the conditions described from each of the three search results, and use those bug reports for our analysis. *Note that we did not confine ourselves to the 12 subject systems listed in Section V-B.* Further, note that even though HOLOCRON could be applied to the web applications we encountered in RQ1, we opted not to use these web applications as the subjects for RQ2-RQ4. This is because the bugs are dispersed across many different web applications, and it would be infeasible for us to perform tests on each one.

For each of the bug reports, we first determine whether the bug corresponds to an inconsistency, as defined in Section II. If so, we determine the bug's *inconsistency category*, which is defined in Section II-A; some categories may possibly contain

only one inconsistency. We also determine whether the bug is a cross-language one.

Real Bugs and Code Smells (RQ2). For RQ2, we run HOLOCRON on the subject systems described in Section V-B and record all the inconsistencies reported by the tool. We examine each of these reported inconsistencies to determine if it corresponds to a real bug (i.e., it represents an error that leads the application to a failure state) or code smell. To determine if a reported inconsistency represents a code smell, we compare the reported inconsistency with the examples of 'correct lines' provided by HOLOCRON (Section IV), and we qualitatively assess whether this deviation has any negative impact on maintenance.

We set the intra-pattern violation threshold t to 90%, the unconditional link rule violation thresholds pv to 95%, and the conditional link rule violation threshold cv to 85%; in Section V-F, we investigate the thresholds' generalizability.

Finally, using the criteria outlined in Section III-C, we use example code from five open-source web applications to train the analysis with more samples. These five applications include the other three subject systems using the same framework (e.g., if the target web application is `angular-puzzle`, we include the three other subject systems that also use AngularJS, namely `projector`, `cryptography`, and `twittersearch` as example code), as well as two additional applications – using the same MVC framework – found on GitHub [27], [28], [29]. We report the number of bugs and code smells found by HOLOCRON, as well as its precision (i.e., number of bugs and code smells per inconsistency).

Thresholds (RQ3). To answer RQ3, we perform a 4-fold cross-validation to test if the applications tested in RQ2 have similar accuracy at the given thresholds (i.e., t = 90%, pv = 95%, and cv = 85%). If they have similar accuracy, it would demonstrate the generalizability of the chosen thresholds, which would strongly indicate they can safely be used when running HOLOCRON for other applications. When performing the cross-validation, the applications are partitioned into four groups, with each group containing three applications (one from each framework). At each iteration of the cross-validation, we use one partition as the training set, and we calculate the aggregated precision value for the applications in that partition; we then calculate the mean-squared error (MSE) of the remaining applications' precision values, with respect to the aggregated value of the training set. Note that we only consider the reported violations corresponding to each parameter; for example, when varying the intra-pattern violation threshold t, we only consider the intra-pattern violations when calculating precision.

Performance (RQ4). We measure the time taken by HOLOCRON to run on each subject application. We run our experiments on a Mac OS/X 10.6.6 machine (2.66 GHz Intel Core 2 Duo, with 4 GB RAM).

D. Prevalence of Inconsistencies (RQ1)

Of the 90 bug reports we studied, we found that *70% of these bug reports correspond to an inconsistency*. These did not need the application's specifications to detect, pointing to the promise of a tool such as ours which finds inconsistencies. For example, one of the Ember.js applications passed a modal object to the `buildUrl()` method, even though this method, which is part of the Ember.js API, expects a string as its first parameter. This inconsistency could be inferred based on other usages of the method which were correct. The remaining 30% of the bugs, however, required prior knowledge of the application's specifications. For example, one of the bugs was caused by the fact that the programmer did not update the `display` style of an element to "block". Prior specification was needed to establish that the programmer intended to modify the style to "block", and hence this bug would not be detected by our approach.

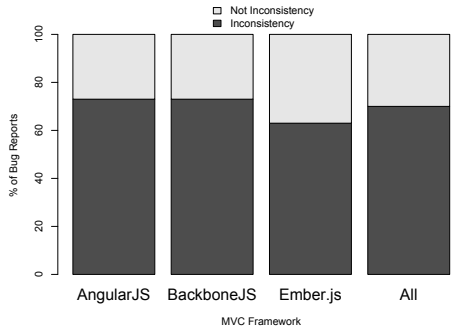

Fig. 4. Percentage of bug reports classified as an inconsistency for each MVC framework

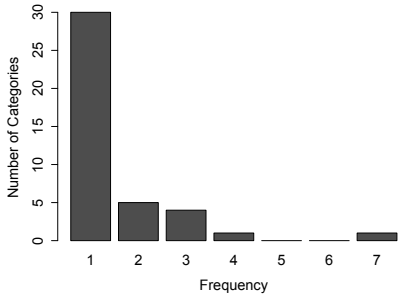

Fig. 5. Number of inconsistency categories with a particular frequency. Most inconsistency categories have just 1-2 inconsistencies in them. This shows why we need an automated approach to infer the inconsistency categories.

The per-framework results are summarized in Figure 4. As this figure shows, 73% of bug reports correspond to inconsistencies for the AngularJS and Ember.js applications, and 63% of bug reports correspond to inconsistencies for BackboneJS applications. These results suggest that inconsistencies are prevalent in web applications created using JavaScript MVC frameworks. *We further found that 35% of the inconsistencies are cross-language*. For example, one of the bugs resulted from the programmer erroneously using the `data-src` attribute instead of the `src` attribute in the HTML code, which led to incorrect bindings with the JavaScript code. Therefore, existing tools that are based on a single language will not be able to detect a significant percentage of these inconsistencies.

Figure 5 shows the distribution of inconsistency categories we found; again, an inconsistency category is uniquely iden-

TABLE I
NUMBER OF REAL BUGS FOUND PER APPLICATION, WITH CORRESPONDING ISSUE NUMBERS IN PARENTHESES (SOME ISSUES ARE AGGREGATED INTO ONE BUG REPORT). THE SIZE IS SHOWN IN KB, WITH LINES OF CODE (LOC) IN PARENTHESES.

Framework	Application	Size (loc)	# of Bugs	Code Smells	Total
AngularJS	angular-puzzle	20 (608)	3 (4,5)	3	8
	projector	19 (569)	3 (1)	0	7
	cryptography	20 (582)	1 (2)	0	2
	twittersearch	10 (357)	1 (1)	0	1
BackboneJS	cocktail-search	10 (396)	2 (13)	6	13
	contact-manager	19 (701)	2 (1,2)	2	9
	webaudiosequencer	43 (1659)	1 (1)	7	15
	backbone-mobile	9 (240)	1 (3)	0	2
Ember.js	todomvc	8 (299)	0	2	3
	emberpress	21 (610)	2 (5,6)	11	22
	giddyup	12 (386)	1 (189)	2	9
	bloggr	6 (185)	1 (17)	0	4
OVERALL			18	33	95

tified by the two components that are inconsistent, as well as the incorrect assumption made by one of the components. As this figure illustrates, most inconsistency categories – which is defined in Section V-C – appear only once in the bug reports we studied; for example, 30 categories had only a single inconsistency each. Further, we found a total of 41 different inconsistency categories in our experiment. The large number of categories suggests that there are many different rules that are used by programmers in writing JavaScript MVC based applications, and hence motivates an approach that discovers the rules automatically (such as ours).

E. Real Bugs and Code Smells (RQ2)

Table I shows the result of running HOLOCRON on the subject systems. In total, HOLOCRON *was able to detect **18 unreported bugs** from 12 MVC applications*. We have reported these bugs to the developers, with two of the bugs' descriptions acknowledged (i.e., those from `cocktail-search`), and the rest still pending[3]; further, we were able to manually reproduce the bugs and confirm them ourselves. As seen in this table, HOLOCRON was able to find a bug in *all* of the applications tested, except for one (`todomvc`). Further, HOLOCRON found 12 unconditional link rule violations, 4 conditional link rule violations and 2 intra-pattern consistency rule violations. Thus, HOLOCRON can be used by web developers to find bugs representing various types of consistency rule violations.[2]

Further, out of 18 real bugs found, 5 were cross-language inconsistencies. The bug in `cryptography`, for instance, results from an assignment in the JavaScript code incorrectly assuming that an element in the HTML code has a numerical value, even though it is a string. In addition, the bug in `twittersearch` resulted from a controller in the JavaScript code assuming that an input element in the HTML code has its type attribute defined, which is not the case. Detection of these cross-language inconsistencies is made possible by HOLOCRON looking for link rules in the applications.

[3]Some of these applications are maintained sporadically, which could explain the delay in response from the developers

In all, HOLOCRON was able to find bugs spanning fifteen inconsistency categories. For example, the bug in `giddyup` is caused by a property assignment erroneously assuming that it has a corresponding route assignment in the Ember.js router. In contrast, Aurebesh [13], which also targets MVC applications, will not be able to detect this bug because (1) it only considers four pre-determined inconsistency categories, to which this inconsistency found in `giddyup` does not belong, and (2) it only works for AngularJS applications, whereas `giddyup` is developed with Ember.js. In total, Aurebesh only detected two of the inconsistencies that HOLOCRON identified, both from `angular-puzzle`, across all applications.

Finally, many of the bugs found have potentially severe consequences on the application. For example, the bugs in `projector` – which are caused by an incorrect assumption about the type of value being assigned to an object property – all cause the application to hang. Further, the bug in `webaudiosequencer`, an audio player, makes an audio clip unplayable after a sequence of input events. Thus, HOLOCRON finds bugs that potentially have high impact on the application.

Code Smells. We also found that 33 of the reported inconsistencies correspond to code smells, the detection of which could help the developer improve the quality of the code. The code smells found in our evaluation belong to three categories:

An inconsistency is labeled as a **"Hardcoded Constants" (HC)** code smell if (1) the inconsistent code uses a hardcoded literal value in calculations and method calls; (2) other similar pieces of code use a non-hardcoded value (e.g., variables, named constants, etc.); and (3) the hardcoded literal value could safely be replaced with a non-hardcoded value to make it more maintainable. We found 15 instances of HC code smells.

Further, an inconsistency is labeled as an **"Unsafe Value Usage" (UVU)** code smell if an object is dereferenced without accounting for the possibility of it being `null`, even though a null check is applied to similar objects in other parts of the code, or a "safer" object is used to call the same method in other parts of the code. For instance, in `emberpress`, one of the inconsistencies occurs as a result of the `get()` method being called directly through a model object repeatedly; this is potentially unsafe if the object is `null`, and it is good practice to call the method via the `Ember` object instead (i.e., `Ember.get()`), as is done in other lines of code. We found 14 instances of UVU code smells.

Finally, an inconsistency is labeled as a **"Multi-Purpose Identifiers" (MPI)** code smell if the same identifier is being used for multiple unrelated objects. This code smell manifests as an inconsistency when the identifier is used in a specific way in some parts of the code, but used in a different way in another part of the code. For example, in `angular-puzzle`, the identifier "src" is used both as a class name for a `div` element in the HTML code and as a name for a puzzle object in the JavaScript code. We found 4 instances of such smells.

Precision. Like most static analysis tools, HOLOCRON incurs false positives (i.e., inconsistencies that are neither bugs nor code smells). False positives occurred in all but one of the

applications (twittersearch). In total, HOLOCRON reported 95 inconsistencies, 51 of which were either real bugs or code smells. Hence, *approximately half the reports are potentially useful in improving the web applications' quality*.

The remaining 44 reports are false positives. While the false positive rate may seem high to users of static analysis tools [30], [31], we believe that they will not deter significantly the usability of HOLOCRON, based on our own analysis of the false positives that appeared in our experiment. In particular, we found that these false positives fall under two very specific categories. First, we found that 22 of these 44 (i.e., half) false positives occur because of frequent usage of certain kinds of literals, contrasted with infrequent usage of another kind of literal. For example, in projector, most of the object method calls take string literals as parameters, but there are three such calls that take number literals as parameters, which though correct, are reported as inconsistencies. In our experiments, it took us only an average of 1–2 minutes to discard each false positive of this kind, without prior knowledge of the code base.

Additionally, the other 22 false positives occur because of frequent usage of a certain identifier contrasted with infrequent usage of another identifier. For instance, in angular-puzzle, there are two main arrays, both of which are accessed through the this identifier – words and grid. While grid is used almost 20 times in the code, words is used only twice, leading to the false positive. We found these false positives to be trickier to discard – again, given our limited knowledge of the code base – since it was not immediately clear to us what certain identifiers are being used for; on average, it took us about 5 minutes before deciding to discard false positives of this kind. We strongly suspect, however, that those familiar with the code base would also be able to discard many of these false positives very quickly. For instance, in the angular-puzzle example, it was evident that words should *not* be replaced by grid, since words is being used in a function whose purpose is to iterate through previously found words in the puzzle (represented by words). Further, for each inconsistency, HOLOCRON includes examples of the correct lines of code that the reported inconsistency deviates from, which would help the developers gain more context about why the inconsistency is reported and determine if it is valid.

F. Thresholds (RQ3)

As mentioned earlier, we used 4-fold cross validation to investigate the generality of the user-defined thresholds and measured the MSE. The MSE can range between 0 and 1, with lower values indicating lower error, and are hence better. When setting the intra-pattern violation threshold to the one used in RQ2 experiment (i.e., 90%), the average MSE = 0.198.

An MSE of 0.198 is non-negligible, as it indicates that the precision can vary by as much as 40%. However, closer investigation reveals that the aggregated precision for all the applications when considering only intra-pattern violations is 50%, and *only one application out of 12* fell below this average (contact-manager). Similarly, when setting the unconditional link rule violation threshold to 95%, the average MSE value is 0.126; the aggregated precision for all the applications when considering only unconditional link rule violations is 67.35%, and only three of the 12 applications (cocktail-search, giddyup, and bloggr-client) fell below this precision value. These results indicate that the error represented by the MSE values primarily represent how much the aggregated precision values are *exceeded* by the applications; for example, for the intra-pattern inconsistencies, two of the applications (todom-vc and giddyup) had perfect precision, and the remainder of the applications had precisions within the range 50% to 60%. *Hence, the precision values reported earlier are conservative ones, and the thresholds chosen work well across the applications, showing their generality.* Note that the MSE value is 0 when setting the conditional link rule violation threshold to the chosen percentage (i.e., 85%), as the applications in our experiment had perfect precision when only considering conditional link rule violations.

G. Performance (RQ4)

We find that on average, HOLOCRON ran for 1.14 minutes for each of the 12 applications. Because HOLOCRON will typically be run prior to deployment of the application, this is an acceptable overhead. The worst-case overhead occurred with webaudiosequencer – which is also the largest application – where HOLOCRON ran for almost 8 minutes; however, note that many popular MVC applications are smaller in size than this on average (Section V-B). In this case, most of the time was spent on finding the link rule violations, as all pairs of subtree classes are compared with each other, as well as all subtrees within these classes.

H. Threats to Validity

An external threat to validity is that we used a limited number of applications in our evaluation. To mitigate this threat, we chose applications coming from three popular MVC frameworks, sizes, and various application types, as seen in Table I. Further, for our study of bug reports in RQ1, we categorize the inconsistencies based on a qualitative analysis of the reports, some of which may be misleading. To mitigate this, we also look at other aspects of the bug, including patches and commits associated with the bug. Finally, the bugs that we studied for RQ1 are limited to fixed bugs with the label "bug" and with the status "closed"; however, many GitHub developers do not label bug reports, and hence we may have missed certain bugs in our analysis. Nonetheless, we choose bug reports this way to minimize spurious or invalid bugs.

VI. DISCUSSION

The main assumption behind our approach is that there are sufficient examples of a consistency rule that appear for it to both successfully learn the consistency rule and detect any violations to that rule. While this is the case for large web applications, small web applications have very few samples to learn from, and hence may incur large numbers of false positives and false negatives. To mitigate this problem, we augment our code patterns with subtrees found in code examples from

other web applications, as discussed in Section III-B. Doing so allows our design to be more confident about the validity of the consistency rules, as well as "debunk" any consistency rules that may lead to false positives. In fact, we found that without the example code, our false positive rates more than doubled. Using more and better examples can likewise bring down the false positive rate. This is a subject of future work.

Furthermore, our main focus in this paper is in using HOLOCRON to detect inconsistencies in MVC applications. Nonetheless, our design can be run on web applications using non-MVC JavaScript frameworks, such as jQuery. This may lead to a large number of inaccuracies, as the JavaScript code in these frameworks interacts *directly* with the DOM, which undergoes many changes throughout the execution of the web application. However, HOLOCRON may be able to detect inconsistencies *within* the JavaScript code, as well as inconsistencies between the JavaScript code and any component of the DOM that does not get modified. To detect the remaining inconsistencies, we may need to use dynamic analysis - this is also a subject for future work.

VII. RELATED WORK

Bug and Code Smell Detection. Considerable work has been done on software fault detection [32], [33], [34], [35], [36], [37], [38], [39], [40] and code smell detection [15], [41], [42], [43] by identifying code patterns. An alternate approach for finding bugs is anomaly detection. This technique was proposed by Engler et. al. [14] and commercialized as the Coverity [44] tool, which also supports JavaScript to some extent. Instead of hardcoding rules as the above techniques do, this approach looks for *deviant behaviours* in the input application's code, with these deviations providing an indication of potential bugs in the program. Reiss [45] has also proposed a similar tool that finds "unusual code" in programs. This approach has the advantage that it can learn rules from common patterns of behaviour, and hence the rules do not need to be updated for each framework.

The main difference with our work is that these prior techniques do not use subtree patterns as the basis for the consistency rules, and they also cannot detect cross-language inconsistencies, as they implicitly assume a single-language model. Further, static analysis techniques such as FindBugs [46] detect faults based on hardcoded rules or bug templates. Additionally, dynamic analysis techniques such as DLint [47] check consistency rules based on "bad coding practices". As shown in our evaluation (RQ1), this can lead to many missed bugs, especially for JavaScript MVC applications, as there are no specific inconsistency categories that dominate the others. Further, the frameworks used in web applications evolve fast.

Tools such as Flow [48] and TypeDevil [49] are capable of finding type-related faults. These tools analyze the static or dynamic data flow of the program to find these faults, and hence this approach will not be able to detect faults that do not stem directly from this data flow. While TypeDevil also leverages the structure of dynamically observed data to find inconsistent types, it does not consider the link rules; hence, unlike HOLOCRON, it will not be able to detect cases where a variable defined in the model is assigned a type inconsistent with how it is used in the HTML template representing the view (e.g., a variable assigned a string being used as the value for the "count" attribute in AngularJS, which expects a number). In addition, Nguyen et al. [50] propose a technique that can detect dangling (i.e., undefined) references in JavaScript code that is generated from PHP code; however, unlike our tool, this technique is only capable of finding these inconsistencies if the references are embedded in the PHP strings that generated them. In most of the applications we studied, PHP is not used (nor any other server-side scripting language), and hence this approach would not work.

Lastly, HOLOCRON shares some similarities with AUREBESH [13] in terms of its goal, namely detecting inconsistencies in JavaScript MVC applications. However, HOLOCRON conceptually differs from AUREBESH in three ways. First, AUREBESH only supports AngularJS, while HOLOCRON supports two of the most commonly used MVC frameworks in addition to AngularJS. Secondly, HOLOCRON is also able to detect more types of inconsistencies, as it infers the consistency rules *automatically* instead of hardcoding them into the design; indeed, as we pointed out in Section V-E, AUREBESH only detected 2 of the 18 bugs that HOLOCRON identified. Finally, HOLOCRON is able to also infer cross-language relations between JavaScript and HTML, while AUREBESH is not.

Cross-Language Computing. Much of the work done on cross-language computing has focused on detecting the dependencies between multiple programming languages [51], [52]. Only a few techniques perform analysis in a cross-language-aware manner, including XLL [53] and X-Develop [54], both of which perform code refactoring. In recent work, Nguyen et al. [55] proposed a tool to perform cross-language program slicing for web applications, with particular focus on PHP code and its interaction with client-side code. Unlike HOLOCRON however, none of these above techniques deal with the inconsistencies in cross-language interactions.

VIII. CONCLUSIONS

We presented an automatic fault detection technique that finds inconsistencies in JavaScript MVC applications. Our technique analyzes the AST and DOM representations of the web application code, and it finds both intra-pattern consistency rules and link rules; violations to these rules are thereby reported to the user. We implemented this approach in an open-source tool called HOLOCRON, and in our evaluation of 12 open-source MVC applications, HOLOCRON was able to find 18 previously unreported bugs, and a significant number (33) of code smells. Further, it took just over a minute on average.

ACKNOWLEDGMENT

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), and a research gift from Intel Corporation. We thank the ASE 2017 reviewers for their insightful comments.

REFERENCES

[1] StackOverflow, "StackOverflow Developer Survey 2015," 2015, http://stackoverflow.com/research/developer-survey-2015#tech (Accessed: May 16, 2015).

[2] F. Ocariza, K. Pattabiraman, and B. Zorn, "JavaScript errors in the wild: an empirical study," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 2011, pp. 100–109.

[3] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "A study of causes and consequences of client-side JavaScript bugs," *IEEE Transactions on Software Engineering (TSE)*, p. 17 pages, 2017.

[4] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do: A large-scale study of the use of eval in JavaScript applications," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2011, pp. 52–78.

[5] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of Ajax web applications," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2008, pp. 121–130.

[6] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of Ajax user interfaces," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2009, pp. 210–220.

[7] K. Pattabiraman and B. Zorn, "DoDOM: leveraging DOM invariants for web 2.0 application robustness testing," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 2010, pp. 191–200.

[8] S. Artzi, J. Dolby, S. Jensen, A. Møller, and F. Tip, "A framework for automated testing of JavaScript web applications," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 571–580.

[9] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Guided mutation testing for JavaScript web applications," *Transactions on Software Engineering (TSE)*, vol. 41, no. 5, pp. 429–444, 2015.

[10] Douglas Crockford, "JSLint," 2012, http://www.jslint.com (Accessed: April 18, 2012).

[11] S. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," *Proceedings of the International Static Analysis Symposium (SAS)*, pp. 238–255, 2009.

[12] 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 Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2011, pp. 59–69.

[13] F. Ocariza, K. Pattabiraman, and A. Mesbah, "Detecting inconsistencies in JavaScript MVC applications," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2015.

[14] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2001, pp. 57–72.

[15] A. Milani Fard and A. Mesbah, "JSNose: Detecting JavaScript code smells," in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, 2013, pp. 116–125.

[16] D. Synodinos, "Top JavaScript MVC frameworks," 2013, http://www.infoq.com/research/top-javascript-mvc-frameworks (Accessed: May 16, 2015).

[17] Two Sigma, "Beaker," 2016, https://github.com/twosigma/beaker-notebook (Accessed: April 29, 2016).

[18] MarionetteJS, "Backbone Marionette," 2016, https://github.com/marionettejs/backbone.marionette (Accessed: April 29, 2016).

[19] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 1998, pp. 368–377.

[20] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 1993, pp. 207–216.

[21] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proceedings of the International Conference on Very Large Databases (VLDB)*. Morgan Kaufmann Publishers Inc., 1994, pp. 487–499.

[22] Adobe Systems, "Brackets," 2015, http://www.brackets.io (Accessed: May 16, 2015).

[23] A. Hidayat, "Esprima," 2015, http://www.esprima.org/ (Accessed: May 16, 2015).

[24] jindw, "XMLDOM," 2016, https://www.github.com/jindw/xmldom (Accessed: April 29, 2016).

[25] K. Sera, "apriori.js," 2016, https://github.com/seratch/apriori.js (Accessed: April 29, 2016).

[26] U. Shaked, "AngularJS vs. BackboneJS vs. EmberJS," 2014, http://www.airpair.com/js/javascript-framework-comparison (Accessed: May 16, 2015).

[27] Google, "Built with AngularJS," 2015, https://github.com/angular/builtwith.angularjs.org/blob/master/projects/projects.json (Accessed: May 16, 2015).

[28] J. Ashkenas, "BackboneJS: Tutorials, blog posts and example sites," 2016, https://github.com/jashkenas/backbone/wiki/Tutorials,-blog-posts-and-example-sites (Accessed: April 29, 2016).

[29] EmberSherpa, "Open source Ember apps," 2016, https://github.com/EmberSherpa/open-source-ember-apps (Accessed: April 29, 2016).

[30] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2013, pp. 672–681.

[31] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. ACM, 2007, pp. 1–8.

[32] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2002, pp. 291–301.

[33] Z. Li and Y. Zhou, "PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code," in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2005, pp. 306–315.

[34] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *Transactions on Software Engineering (TSE)*, vol. 32, no. 3, pp. 176–192, 2006.

[35] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2005, pp. 461–476.

[36] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2007, pp. 35–44.

[37] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2007, pp. 55–64.

[38] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy, "Using web corpus statistics for program analysis," in *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. ACM, 2014, pp. 49–65.

[39] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2009, pp. 283–294.

[40] N. Gruska, A. Wasylkowski, and A. Zeller, "Learning from 6,000 projects: lightweight cross-project anomaly detection," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2010, pp. 119–130.

[41] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Detection of embedded code smells in dynamic web applications," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2012, pp. 282–285.

[42] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2008, pp. 329–331.

[43] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 2002, pp. 97–106.

[44] Synopsys, "Coverity," 2016, http://www.coverity.com/ (Accessed: April 29, 2016).

[45] S. P. Reiss, "Finding unusual code," in *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 2007, pp. 34–43.

[46] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *Companion Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 2004, pp. 132–136.

[47] L. Gong, M. Pradel, M. Sridharan, and K. Sen, "DLint: Dynamically checking bad coding practices in JavaScript," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2015.

[48] Facebook, "Flow: a static type checker for JavaScript," 2016, http://flowtype.org/ (Accessed: April 29, 2016).

[49] M. Pradel, P. Schuh, and K. Sen, "TypeDevil: Dynamic type inconsistency analysis for JavaScript," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2015, pp. 314–324.

[50] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Dangling references in multi-configuration and dynamic PHP-based web applications," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society/ACM, 2013, pp. 399–409.

[51] T. Polychniatis, J. Hage, S. Jansen, E. Bouwers, and J. Visser, "Detecting cross-language dependencies generically," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2013, pp. 349–352.

[52] R.-H. Pfeiffer and A. Wasowski, "Taming the confusion of languages," in *Proceedings of the European Conference on Modelling Foundations and Applications (ECMFA)*. Springer, 2011, pp. 312–328.

[53] P. Mayer and A. Schroeder, "Cross-language code analysis and refactoring," in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, 2012, pp. 94–103.

[54] D. Strein, H. Kratz, and W. Löwe, "Cross-language program analysis and refactoring," in *Proceedings of the International Workshop on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, 2006, pp. 207–216.

[55] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Cross-language program slicing for dynamic web applications," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 369–380.