Vejovis: Suggesting Fixes for JavaScript Faults

Frolin S. Ocariza, Jr. Karthik Pattabiraman Ali Mesbah

Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada
{frolino, karthikp, amesbah}@ece.ubc.ca

ABSTRACT

JavaScript is used in web applications for achieving rich user interfaces and implementing core functionality. Unfortunately, JavaScript code is known to be prone to faults. In an earlier study, we found that over 65% of such faults are caused by the interaction of JavaScript code with the DOM at runtime (DOM-related faults). In this paper, we first perform an analysis of 190 bug reports to understand fixes commonly applied by programmers to these DOM-related faults; we observe that parameter replacements and DOM element validations are common fix categories. Based on these findings, we propose an automated technique and tool, called VEJOVIS, for suggesting repairs for DOM-based JavaScript faults. To evaluate VEJOVIS, we conduct a case study in which we subject VEJOVIS to 22 real-world bugs across 11 applications. We find that VEJOVIS accurately suggests repairs for 20 out of the 22 bugs, and in 13 of the 20 cases, the correct fix was the top ranked one.

Categories and Subject Descriptors

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

General Terms

Design, Algorithms, Experimentation

Keywords

JavaScript, fault repair, Document Object Model (DOM)

1. INTRODUCTION

JavaScript is used extensively in modern web applications to manipulate the contents of the webpage displayed on the browser and to retrieve information from the server by using HTTP requests. To alter the contents of the webpage, JavaScript code manipulates a data structure known as the *Document Object Model (DOM)*. The DOM contains all the webpage elements (e.g., `div`, `table`, etc.) hierarchically ordered as a tree, where each element contains specific properties, such as styling information. Through the use of DOM API methods, the JavaScript code is capable of retrieving elements from the DOM, as well as changing the DOM by modifying element properties, adding elements, and deleting elements.

Due to the dynamic nature of the JavaScript language and the interaction with the DOM, JavaScript-based web applications are prone to errors. In a recent empirical study we conducted on JavaScript bug reports [13], we found that around 65% of JavaScript faults are *DOM-related*, meaning the error leading to the fault eventually propagates into a parameter value of a DOM API method call (or to the assignment value of a DOM element property assignment) before causing the failure. In addition, we found that 80% of the highest impact JavaScript faults i.e., those that cause data loss, hangs, and/or security vulnerabilities, are DOM-related. Finally, we found that DOM-related faults require more time to fix than other JavaScript faults. These results point to the prevalence, impact, and complexity of DOM-related JavaScript faults in web applications.

In this paper, our goal is to *facilitate the process of fixing DOM-related JavaScript faults*, by providing suggestions to the programmer during web application testing and debugging tasks. To this end, we first perform a study of real-world JavaScript faults to understand the common patterns in how programmers fix such faults. Then, based on these common fix patterns, we propose an automatic approach for suggesting repairs. In this paper, we use the term *repair* to encompass both fixes and workarounds for the fault, similar to other related work [11, 16].

Our approach starts from the wrong DOM API method/property, and uses a combination of static and dynamic analysis to identify the lines of code in the backward slice of the parameters or assignment values of DOM methods/properties. Once these lines are localized, it uses a string solver to find candidate replacement DOM elements, and propagates the candidate values along the backward slice to find the fix.

We implement our approach in an open-source tool called VEJOVIS.[1] VEJOVIS is deployed on a web application after the occurrence and subsequent localization of a JavaScript fault [14]. It requires neither specifications/annotations from the programmer nor any changes to the JavaScript/DOM interaction model, and can hence be deployed on unmodified web applications.

Prior work on suggesting repairs for web application faults has focused on server-side code (e.g., PHP) [11, 16], including workarounds for web API calls [4]. Other work [7] automatically transforms unsafe `eval` calls in JavaScript code to safe alternatives. None of these techniques, however, deal with DOM-related JavaScript faults. *To the best of our knowledge,* VEJOVIS *is the first technique to automatically suggest repairs for DOM-related JavaScript faults in web applications*.

[1] VEJOVIS is the Roman god of healing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/05 ...$15.00.

We make the following contributions in this paper:

- We categorize *common fixes* applied by web developers to DOM-related JavaScript faults, based on an analysis of 190 online bug reports. We find that fixes involving modifications of DOM method/property parameters or assignment values into valid replacement values (i.e., values consistent with the DOM) are the most common, followed by those involving validation of DOM elements before their use;
- Based on the above study, we present an algorithm for finding *valid replacement parameters* passed to DOM API methods, which can potentially be used to replace the original (and possibly erroneous) parameter used to retrieve elements from the DOM. The replacements are found based on the CSS selector grammar, and using information on the DOM state at the time the JavaScript code executed. The aim is to suggest replacements that are *valid* in the current DOM, and to suggest as few replacements as possible;
- We present an algorithm for suggesting repairs in the form of *actionable messages* to the programmer based on code-context. The actionable messages contain detailed directions prompting the programmer to make modifications to the code so as to eliminate the "symptoms" observed during the program's failure run;
- We describe the implementation, called VEJOVIS, which integrates the previous two contributions. We evaluate our technique on 22 real-world JavaScript bugs from 11 web applications. In our case study, VEJOVIS was able to provide correct repair suggestions for 20 of the 22 bugs. Further, the correct fix was ranked first in the list of repairs for 13 of the 20 bugs. We also found that limiting the suggestions to those that are within an edit distance of 5 relative to the original selector can decrease the number of suggestions in the other 7 bugs to 3 per bug, while reducing the number of correct fixes to 16 (from 20).

2. BACKGROUND AND CHALLENGES

The DOM is a standard object model, used internally by web browsers to represent the HTML state of a webpage at runtime. Changes made to the structure, contents or styles of the DOM elements are directly manifested in the browser's display. Client-side JavaScript[2] is used primarily to interact with i.e., to access, traverse, or manipulate the DOM. In most modern web applications, these interactions are used to incrementally update the browser display with client-side state changes without initiating a page load. Note that this is different from what happens during URL-based page transitions where the entire DOM is repopulated with a new HTML page from the server.

JavaScript provides DOM API methods and properties that allow *direct* and easy retrieval of DOM elements. For instance, elements can be accessed based on their tag name, ID, and class names, and the DOM can be traversed using the `parentNode` and `childNodes` properties. In addition, modern browsers provide APIs such as `querySelector` for retrieving DOM elements using patterns called *CSS selectors*. CSS selectors follow a well-defined grammar [17], and serve as a unified way of retrieving DOM elements; for example, retrieving a `DIV` DOM element with ID "news" translates to `"DIV#news"` in CSS selector syntax. Table 1 shows some of the commonly used components that make up a CSS selector.

Once an element is retrieved using the CSS selector, JavaScript code can use the reference to that element to access its properties,

[2] For simplicity, we refer to client-side JavaScript as JavaScript.

Table 1: List of commonly used CSS selector components.

Component	Description
Tag Name	The name of the tag associated with an element. *Examples*: `div`, `span`, `table`, etc.
ID	The id associated with an element. This is prefixed with the # symbol. *Example*: If a `div` element has an id of "myID", a CSS selector that can retrieve this element is `div#myID`.
Class Name	The name of a class to which an element belongs. This is prefixed with a period. *Example*: If a `span` element belongs to the "myClass" class, a CSS selector that can retrieve this element is `span.myClass`.
Is Descendant	A space character indicating that the element described by the right selector is a descendant of the element described by the left selector. *Example*: To find all `table` elements belonging to class "myClass" that are descendants of a `div` element, we use the selector `div table.myClass`.
Is Child	The ">" character, which indicates that the element described by the right selector is a child of the element described by the left selector. *Example*: To find all `tr` elements that are children of the `table` element with id "myID", we use the selector `table#myID > tr`.
Is Next Sibling	The "+" character, which indicates that the element described by the right selector is the next sibling of the element described by the left selector. *Example*: To find all `tr` elements that follow another `tr` element, we use the selector `tr + tr`.

```
1 function pagerSetup() {
2   var display = "catalog_view";
3   var content = "p.pages span";
4   appendTo(display, content);
5 }
6
7 function appendTo(display, content) {
8   var view_selector = "div#view-display-id-" + ↩
        display;
9   var content_selector = view_selector + " > " + ↩
        content;
10  var pageToAdd = "<div>New Content</div>";
11  var pages = $(content_selector);
12  var oldContent = pages[0].innerHTML;
13  pages[0].innerHTML = oldContent + pageToAdd;
14 }
```

Figure 1: JavaScript code of the running example.

add new or remove/modify existing properties, or add/remove elements to/from the DOM tree.

Running Example. Here, we describe the running example that we will be using throughout this paper to simplify the description of our design. The running example is based on a bug in Drupal involving jQuery's Autopager [1] extension, which automatically appends new page content to a programmer-specified DOM-element A snippet of the simplified JavaScript code is shown in Figure 1. In the `pagerSetup()` function, the programmer has set the display ID suffix to "catalog_view" (line 2), and the DOM element where the page is added as "p.pages span" (line 3). These inputs are passed to the `appendTo()` function, which sets up the full CSS selector describing where to add the new page through a series of string concatenations (lines 8-9). In this case, the full CSS selector ends up being "div#view-display-id-catalog_view > p.pages span".

The above JavaScript code runs when the DOM state is as shown in Figure 2. Note that in this case, the CSS selector will *not* match any element in the DOM state. As a result, `$()` returns an empty set in line 11; hence, when retrieving the old content of the first matching element via `innerHTML` (line 12), an "undefined" exception is thrown. The undefined exception prevents the Autopager to successfully append the contents of the new page (line 10) to the specified element in line 13.

For this particular bug, the fix applied by the programmer was to change the string literal "div#view-display-id-" to "div#view-id-" in line 8. This, in turn, changes the full CSS selector to "div#view-

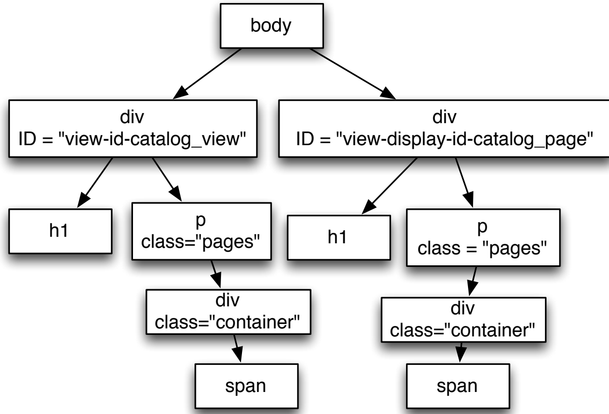

Figure 2: The DOM state during execution of the JavaScript code in the running example. For simplicity, only the elements under `body` are shown.

id-catalog_view > p.pages span", which is valid in the DOM in Figure 2.

Challenges. The interaction between two separate languages (i.e., the DOM and the JavaScript code) makes web applications highly error-prone. JavaScript errors are prominent and can lead to the impairment of major functionalities of a webpage, even in the case of popular and well-engineered production web applications [15]. In a recent empirical study of over 300 JavaScript bug reports [13], we found that approximately 65% of JavaScript faults are *DOM-related*. This means that the majority of JavaScript faults originate from errors that eventually propagate into the parameter or assignment value of a DOM method/property. The running example provides an example of a DOM-related fault, as the error present in the example eventually propagates into the DOM method `$()`, which retrieves DOM elements through CSS selectors.

DOM-related faults, by definition, involve the propagation of the error to a DOM method/property's parameter or assignment value. Therefore, the fix likely involves altering the code responsible for setting up this parameter or assignment value. The challenge, of course, is in answering the following question: *how should the code be modified to repair the fault?* Answering this question requires knowledge of (1) the location in the code that needs to be altered; and (2) the specific modification that needs to be applied to that location. For the first task, the origins (i.e., the backward slice) of the parameter or assignment value must be traced. For the second task, the specific replacement parameter or replacement assignment value must be inferred, and the way in which this replacement should be incorporated into the code must be determined.

While these challenges are difficult and sometimes impossible to tackle with arbitrary method parameters or assignment values (because they require programmer *intent* to be inferred), parameters and assignment values to DOM methods/properties – as well as the DOM itself – are more structured. Hence, for these kinds of parameters or assignment values, the problem of inferring programmer intent reduces to finding replacements that satisfy this structure. Therefore, we study common patterns in how programmers fix DOM-related faults, to prioritize the fix suggestions we infer. This study is presented in the next section.

3. COMMON DEVELOPER FIXES

To better understand how programmers implement fixes to DOM-related JavaScript faults, we analyze 190 *fixed* bug reports representing DOM-related faults and analyze the developer fixes applied to them. Our goal is to answer the following research questions:

Table 2: List of applications used in our study of common fixes.

Application	Description
Moodle	Learning Management System
Joomla	Content Management System
WordPress	Blogging
Drupal	Content Management System
Roundcube	Webmail
WikiMedia	Wiki Software
TYPO3	Content Management System
TaskFreak	Task Organizer
jQuery	JavaScript Library
Prototype.js	JavaScript Library
MooTools	JavaScript Library
Ember.js	JavaScript Library

RQ1 (Fix Categories): What are the common fix types applied by programmers to fix DOM-related JavaScript faults?

RQ2 (Application of Fixes): What modifications do programmers make to JavaScript code to implement a fix and eliminate DOM-related faults?

The above questions will help us determine how programmers typically deal with DOM-related faults. This understanding will guide us to design our automated repair algorithm.

3.1 Methodology

We perform our analysis on 190 fixed JavaScript bug reports from eight web applications and four JavaScript libraries (see Table 2). Note that these bug reports are a subset of the bug reports we explored in a recent empirical study [13]. However, the analysis conducted here is new and is not part of our previous study.

In our earlier study, we analyzed a total of 317 fixed JavaScript bug reports. Of these, about 65%, or 206 of the bugs were DOM-related JavaScript faults. Further, we found that in 92% of these DOM-related JavaScript faults (or 190 bugs), the fix involved a modification of the client-side JavaScript code. We consider only these 190 bugs in this study, as our goal is to find repairs for DOM-related JavaScript faults that involve the JavaScript code.

To answer the research questions, we perform a qualitative analysis of the fixes applied by the programmers to each bug report. To do so, we manually read the portions of each bug report documenting the fix applied (e.g., developer comments, discussions, initial report descriptions, fix descriptions, patches, etc.). Based on this analysis, we devise a classification scheme for the bug report fixes so that we can group the fixes into different, well-defined categories, to answer RQ1. Our analysis of the code patches and/or fix descriptions helps us answer RQ2.

3.2 Results

Fix Categories. We found that the fixes that programmers apply to DOM-related JavaScript faults fall into the following categories.

Parameter Modification, where a value that is eventually used in the concatenation of a DOM method/property parameter or assignment value is modified. This is done either by directly modifying the value in the code, or by adding calls to modifier methods (e.g., adding a call to `replace()` so that the string value of a variable gets modified). This category makes up 27.2% of the fixes.

DOM Element Validation, where a check is added so that the value of a DOM element or its property is compared with an expected value before being used. This category makes up 25.7% of the fixes.

Method/Property Modification, where a call to a DOM API method (or property) is either added, removed, or modified in the JavaScript code. Here, modification refers to changing the *method* (or property) originally called, not the parameter (e.g., instead of calling `getElementsByClassName`, the method `getElementsByTagName` is called instead). This category makes up 24.6% of the fixes.

Major Refactoring, where significantly large portions of the JavaScript code are modified and restructured to implement the fix. This category makes up 10.5% of the fixes.

Other/Uncategorized, which make up 12% of the fixes.

As seen in the above fix categories, the most prominent categories are Parameter Modification and DOM Element Validation, which make up over half (52.9%) of the fixes. Therefore, we focus on these categories in our work. Although we do not consider *Method/Property Modifications* in our repair approach, our algorithm can be adapted to include this class of errors, at the cost of increasing its complexity (see Section 7).

Application of Fixes. We next describe how programmers modify the JavaScript code to apply the fixes. We discuss our findings for the three most prominent fix categories – Parameter Modification, DOM Element Validation, and Method/Property Modification.

Parameter Modification: We found that 67.3% of fixes belonging to the Parameter Modification fix category involve the modification of string values. The vast majority (around 70%) of these string value modifications were direct modifications of string literals in the JavaScript code. However, we also found cases where the string value modification was applied by adding a call to string modification methods such as `replace()`.

We also analyzed the DOM methods/properties whose parameters are affected by the modified values. For string value modifications, the methods/properties involved in multiple bug report fixes are `getElementById()`, `$()` and `jQuery()`; together, fixes involving these methods comprise 51.4% of all string value modifications. For non-string value modifications, fixes involved modification of the numerical values assigned to elements' style properties, particularly their alignment and scroll position.

DOM Element Validation: 75.5% of fixes belonging to this category are applied by simply wrapping the code using the pertinent DOM element within an `if` statement that performs the necessary validation (so that the code only executes if the check passes). Other modifications include (1) adding a check before the DOM element is used so that the method returns if the check fails; (2) adding a check before the DOM element is used such that the value of the DOM element or its property is updated if the check fails; (3) encapsulating the code using the DOM element in an `if-else` statement so that a backup value can be used in case the check fails; and finally (4) encapsulating the code in a `try-catch` statement. The most prevalent checks are `null`/`undefined` checks, i.e., the code has been modified to check if the DOM element is `null` or `undefined` before it is used, which constitutes 38.8% of the fixes in the DOM Element Validation category.

Method/Property Modification: 53.2% of these fixes involve changing the DOM method or property being called/assigned; the rest involve either the removal of the method call or the property assignment (e.g., remove a `setAttribute` call that changes the class to which an element belongs), or the inclusion of such a call or assignment (e.g., add a call to `blur()` to unfocus a particular DOM element). Of the fixes where the DOM method/property was changed, around 44% involve changing the event handler to which a function is being assigned (e.g., instead of assigning a particular method to `onsubmit`, it is assigned to `onclick` instead).

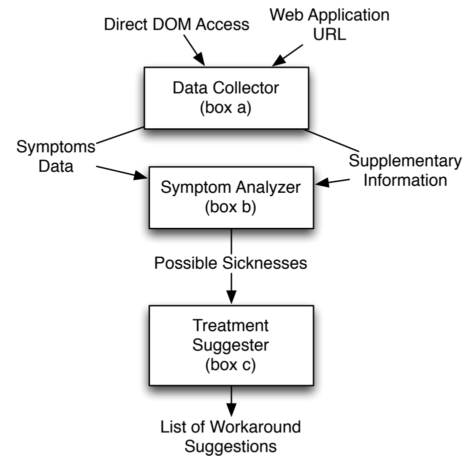

Figure 3: High-level block diagram of our design.

Summary of Findings. Our study shows that the most prominent fix categories are Parameter Modification and DOM Element Validation. Our analysis also shows the prevalence of string value modifications and `null`/`undefined` checks when applying fixes. In addition, most parameter modifications are for values eventually used in DOM methods that *retrieve* elements from the DOM, particularly the `$()`, `jQuery()` and `getElementById()` methods. These results motivate our fault model choice in Section 4 as well as our choice of *possible sickness classes* in Section 5.2.

4. FAULT MODEL

In this work, we focus on DOM API methods that retrieve an element from the DOM using CSS selectors, IDs, tag names, or class names, as we found that these were the common sources of mistakes made by programmers (Section 3). These DOM API methods include `getElementById()`, `getElementsByTagName()`, `getElementsByClassName()`, `querySelector()`, and `querySelectorAll()`. We also support DOM API wrapper methods made available by commonly used JavaScript libraries including those in jQuery (e.g., `$()` and `jQuery()`); Prototype (e.g., `$$()` and `$()`); and tinyMCE (e.g., `get()`), among others. For simplicity, we will refer to all these DOM API methods as the *direct DOM access*.

We further focus on *code-terminating* DOM-related faults, which means the DOM API method returns `null`, `undefined`, or an empty set of elements, eventually leading to a null or an undefined exception (thereby terminating JavaScript execution). However, our design can also be extended to apply to *output-related* DOM-related faults, i.e., those that lead to incorrect output manifested on the DOM. Such faults would require the programmer to manually specify the direct DOM access. In contrast, with code-terminating DOM-related faults, the direct DOM access can be determined automatically using the AutoFLox tool proposed in our prior work [14]. Thus we focus on this category of faults in this work.

The running example introduced in Section 2 is an example of a fault that is encompassed by the fault model described above. Here, the direct DOM access is the call to the `$()` method in line 11, which returns an empty set of elements. It is code-terminating because the fault leads to an undefined exception in line 12.

5. APPROACH

In this section, we describe our approach for assisting web developers in repairing DOM-related faults satisfying the fault model described in the previous section. Figure 3 shows a block diagram

of our design, which consists of three main components: (1) the *data collector*; (2) the *symptom analyzer*; and (3) the *treatment suggester*. These components are described in Sections 5.1–5.3.

Our approach assumes the parameter (or the array index) of the direct DOM access is incorrect. This is inspired by the results presented in Section 3, which demonstrated the prevalence of Parameter Modification fixes. As such, our approach attempts to find *valid replacements* for the original parameter or array index, where a valid replacement is a parameter that matches at least one element in the DOM. Once the valid replacements are found, our approach analyzes the code context to determine what *actionable message* to suggest as a potential repair to the programmer.

5.1 Data Collector

The main purpose of the data collector module is to gather dynamic data that may reveal the *symptoms* present in the web application. In general, symptoms are defined as any indications of abnormalities in the intended behaviour of the program with regard to DOM accesses. We consider the following as symptoms in our design based on our fault model:

- **Symptom 1**: The direct DOM access is returning `null`, `undefined`, or an empty set of elements. This leads to a "null" or "undefined" exception eventually.
- **Symptom 2**: The index used to access an element in the list of elements returned by the direct DOM access is out of bounds. This is only applicable to DOM methods that retrieve a list of elements (e.g., `getElementsByTagName()`, `$()`, etc.). This eventually leads to an "undefined" exception.

The data collector collects the direct DOM access' line number, and the name of the function containing it. This data is provided by the user (manually) or gathered automatically using a tool such as AUTOFLOX [14]. The data collector module also collects the following *supplementary information*, which can help infer the context under which a particular symptom is appearing:

- The dynamic execution trace of the JavaScript program, with each trace item containing the line number of the executed line, the name of the function containing the line, and the names and values of all in-scope variables at that line. It also includes the lines in the body of a loop, and a list of `for` loop iterator variables (if any). The data describing which lines are part of a loop are used by the treatment suggester to infer code context, to determine what actionable repair message to provide to the programmer; more details are in Section 5.3.
- The state of the DOM when the direct DOM access line is executed. For instance, in the running example, the DOM state in Figure 2 is retrieved; The DOM state will be used by the symptom analyzer to determine possible replacements for the direct DOM access parameter (if any); in particular, if the direct DOM access is returning `null` or `undefined` (i.e., Symptom 1), this means that the parameter to the direct DOM access does not correspond to any element in the current DOM state, so our technique can look at the current DOM state to see if there are any reasonable replacements that *do* match an element (or a set of elements) in the DOM.

5.2 Analyzing Symptoms

The *symptom analyzer* (Figure 3, box b) uses the data gathered by the data collector to come up with a list of *possible sicknesses* that the web application may have. Each possible sickness belongs to one of the following classes:

- **String**: A [variable | expression | string literal] has a string value of `X`, but it should probably have string value `Y`. This sickness triggers Symptom 1.
- **Index**: An array index has a numerical value of `X`, but it should fall within the allowed range `[Y-Z]`. This sickness triggers Symptom 2.
- **Null/Undefined**: A line of code `X` accessing a property/method of the DOM element returned by the direct DOM access should not execute if the DOM element is [`null` | `undefined`]. This sickness can trigger both Symptoms 1 and 2.

These classes are based on the results of our study of common bug report fixes. In particular, the "String" and "Null/Undefined" classes account for Parameter Modification and DOM Element Validation fixes, respectively. The "Index" class is included because in some cases, an undefined exception occurs not because of retrieving the incorrect element, but because of using an out-of-bounds index on the returned array of DOM elements.

The symptom analyzer takes different actions depending on the symptom in Section 5.1 as follows:

1. **String Replacement**: Assume that the program suffers from Symptom 1. This implies that the string parameter being passed to the direct DOM access does *not* match any element in the DOM – i.e., the program may be suffering from the "String" sickness class, as described above. Our design will look for potential replacements for these parameters, where the replacements are determined based on the current DOM state. Each potential replacement represents a possible sickness belonging to the "String" class.
2. **Index Replacement**: Assume that the program suffers from Symptom 2. This implies that the program may be suffering from the "Index" sickness class, as described above. This step is only taken if the direct DOM access corresponds to a method that returns a *set* of DOM elements. Our approach will determine the allowed range of indices, representing a possible sickness belonging to the "Index" class.
3. **Null/Undefined Checks**: By default, our design additionally assumes a possible sickness belonging to the "Null/Undefined" class.

Each of the above cases will be described in detail. Because CSS selectors provide a unified way of retrieving elements from the DOM, we will only describe how the possible sicknesses are determined for the case where the parameter to the direct DOM access is a CSS selector (as in the case of the running example).

Case 1: String Replacement. The main assumption here is that the string parameter passed to the direct DOM access is incorrect; we call this parameter the ***erroneous selector***. Hence, the goal is to (1) look for potential replacement parameters that *match* an element (or a set of elements) in the current DOM state (i.e., are *valid replacements*), and (2) suggest only the most viable replacements so as to not overwhelm the programmer; therefore our approach assumes that the replacement will be relatively close to the original, erroneous selector (i.e., only one component of the original selector is assumed incorrect by any given replacement). Algorithm 1 shows the pseudocode for this step. The sub-steps are described below in more detail.

Dividing Components: The first step is to divide the erroneous selector into its constituent components, represented by **C** (line 1). In essence, **C** is an ordered set, where each element c_i corresponds to a selector component ($c_i.comp$); its matching component type ($c_i.type$; see Table 1); and its level in the selector, where each level is separated by a white space or a > character ($c_i.level$). The

Algorithm 1: Parameter Replacement

```
   Input: trace: The dynamic execution trace
   Input: dda: The direct DOM access
   Input: dom: The current DOM state
   Output: listOfpossibleSicknesses: A list of possible sicknesses
 1 C ← {c_1, c_2, ..., c_N};
 2 GSS ← {(s_1, l_1), (s_2, l_2), ..., (s_K, l_K)};
 3 foreach c_i ∈ C do
 4 |   LSS_i ← match(c_i, GSS);
 5 end
 6 VS ← ∅;
 7 foreach c_i ∈ C do
 8 |   PVE ← {dom.root};
 9 |   for j ← 0 to c_i.level do
10 |   |   nextElems ← ∅;
11 |   |   foreach e ∈ PVE do
12 |   |   |   all ← e.getElementsByTagName("*");
13 |   |   |   foreach f ∈ all do
14 |   |   |   |   if f matches level j of erroneous selector then
15 |   |   |   |   |   nextElems.add(f);
16 |   |   |   |   end
17 |   |   |   end
18 |   |   end
19 |   |   PVE ← nextElems;
20 |   end
21 |   foreach e ∈ PVE do
22 |   |   newElems ← ∅;
23 |   |   if level after c_i.level is the "descendant" then
24 |   |   |   newElems ← getAllDescendants(e);
25 |   |   end
26 |   |   else if level after c_i.level is the "child" then
27 |   |   |   newElems ← getAllChildren(e);
28 |   |   end
29 |   |   else if level after c_i.level is the "next sibling" then
30 |   |   |   newElems ← getNextSibling(e);
31 |   |   end
32 |   |   foreach f ∈ newElems do
33 |   |   |   if f has c_i.type then
34 |   |   |   |   newSelector ← dda.erroneousSelector.replace(c_i.comp,
   |   |   |   |     c_i.type of f);
35 |   |   |   |   VS.add(newSelector);
36 |   |   |   end
37 |   |   end
38 |   end
39 |   foreach selector ∈ VS do
40 |   |   if e ← selector(dom) ∉ dom then
41 |   |   |   VS.remove(selector);
42 |   |   end
43 |   end
44 end
45 PR ← replacementsFinder(VS, LSS_1, LSS_2, ..., LSS_N);
46 foreach rep ∈ PR do
47 |   possibleSickness ← craftPossibleSickness(rep);
48 |   listOfPossibleSicknesses.add(possibleSickness);
49 end
```

erroneous selector itself is retrieved from the direct DOM access (*dda*) which is input to the algorithm. For example, consider the erroneous selector in the running example: "div#view-display-id-catalog_view > p.pages span". This selector contains the following components: (1) the tag name "`div`"; (2) the "has ID" identifier "`#`"; (3) the ID name "`view-display-id-catalog_view`"; (4) a ">" character indicating that the next component is a child of the previous; (5) the tag name "`p`"; (6) the "has class" identifier "`.`" (i.e., a dot character); (7) the class name "`pages`"; (8) whitespace indicating that the next component is a descendant of the previous one; and (9) the tag name "`span`".

Finding the Global String Set: The next step is to determine the *string set* corresponding to each component (lines 2-5). The string set refers to the list of locations, in the JavaScript code, of the origins of all the parts that make up a particular string value. For instance, consider the erroneous selector in the running example, whose final string value is "div#view-display-id-catalog_view > p.pages span". This entire string is made up of a concatenation of the following strings: (1) "`div#view-display-id-`" in Figure 1, line 8; (2) "`catalog_view`" in line 2; (3) " > " in line 9; and (4) "`p.pages span`" in line 3.

The algorithm first determines the *global string set*, which refers to the string set of the entire erroneous selector; in Algorithm 1, this is represented by **GSS** (line 2). The global string set is found by recursively extracting the dynamic backward slice of each concatenated string value that makes up the erroneous selector (using the dynamic execution trace) until all the string literals that make up the erroneous selector have been included in the string set. Note that the slice extraction process is a dynamic one, and is hence precise. However, it may be unable to resolve the origin of every variable in the code e.g., because a variable gets its value from an external XML file. Unresolved portions of the erroneous selector are left as "gaps" in the string set.

The **GSS** consists of an ordered set of tuples of the form (s_i, l_i), where s_i is a string value and l_i is the location in the JavaScript code where that value originated (i.e., line number and enclosing function). Each tuple represents an element in the string set. In the running example, given the string set of the erroneous selector just described above, the ordered set of tuples will be as follows: {("`div#view-display-id-`", 8), ("`catalog_view`", 2), ("`>`", 9), ("`p.pages span`", 3)}.[3] Note that a gap in the string set is likewise represented as a tuple; the string value s_i is retained, but the location l_i is left undefined, and a special variable is used to store the *earliest* expression from which the unresolved string value originated.

Finding the Local String Sets: Once the global string set is found, the *local string set* of each component – represented by **LSS**$_i$ – is inferred (lines 3-5). In essence, this procedure matches each erroneous selector component c_i with the corresponding elements in the global string set (line 4). For example, consider the id name component "`view-display-id-catalog_view`" in the running example. If *startIndex* and *endIndex* refer to the index range of the characters from the global string set element that belong to the local string set, then the string set of this component is {(("`div#view-display-id-`", 8), *startIndex*: 4, *endIndex*: 19), (("`catalog_view`", 2), *startIndex*: 0, *endIndex*: 11)}.

Finding Valid Selectors: Lines 6-44 of Algorithm 1 looks for *valid selectors* (**VS**) in the current DOM state. This portion of the algorithm iterates through each component c_i of the erroneous selector and assumes that c_i is incorrect; it then traverses the current DOM state's tree to see if it can find new CSS selectors (i.e., those in which the component assumed to be erroneous is replaced by a different value) that match an element in the current DOM state. This procedure is carried out for each component of the erroneous selector; hence, by the end of this procedure, each component will have a corresponding set of CSS selectors (may be empty).

Precisely, to find the valid selectors, the algorithm first looks for possibly valid elements, represented by **PVE** (lines 8-20). These are the elements that match the original selector *up to and including* the the selector level c_i*.level*, neglecting the component being assumed erroneous. For instance, suppose in the running example, the tag component "`p`" of the erroneous selector is assumed as incorrect by our design. This component is found in level 2 of the erroneous selector. Hence, our design traverses the DOM to look for elements that match the selector up to level 2 neglecting "p" – i.e., elements that match "`div#view-display-id-catalog_view > .pages`".

Once **PVE** is found, the algorithm (lines 21-38) checks if the element does indeed contain a corresponding replacement for the

[3] Due to space constraints, we omit the enclosing functions.

component that was assumed to be incorrect (e.g., if an ID is being replaced, the element must have an ID) (line 32-37). In our example, "p" – which is a tag component – was assumed incorrect so the verification will pass for all elements in **PVE** because every element has a tag name. It also checks if the element contains any descendants, children, or siblings, depending on the structure of the erroneous selector (lines 22-31). Again, in the running example, the next level (level 3) of the erroneous selector must be the "descendant" of the first two levels, because of the whitespace between the level 2 components and the level 3 components; hence, the check will pass for an element if it contains any descendants. If both checks pass, the corresponding component is used to create a new selector; each new selector is stored in **VS**. Finally, for each new selector, a final verification step is carried out to ensure that the new selector is indeed valid in *dom* (lines 39-43).

In summary, for the running example, our design looks for matching selectors of the form "div#view-display-id-catalog_view **> <NEW-TAG>**.pages span". Similarly, if the ID component "view-display-id-catalog_view" were assumed incorrect, the algorithm looks for matching selectors of the form "div#**<NEW-ID>** > p.pages span". In the latter case, two matching valid selectors are found: "div#view-id-catalog_view > p.pages span" and "div#view-display-id-catalog_page > p.pages span"

Inferring Possible Replacements: To determine the possible sickness, our design determines if any element of the local string set of each component (**LSS**$_1$, **LSS**$_2$, ..., **LSS**$_N$) can be replaced to match one of the valid selectors in **VS**. This is accomplished by the `replacementsFinder()` function (line 45). The basic idea is as follows: for each component string set element, assume that this element is incorrect, then determine if any of the valid selectors can provide a replacement string value for that element. We accomplish this matching with the valid selectors through the use of a string constraint solver (see Section 5.4).

Let us go back to the running example. Suppose the design is currently considering the "`view-display-id-catalog_view`" component, whose local string set was found earlier. Also, as mentioned, two valid replacement selectors were found for this component. Our design goes through each element in the local string set to look for possible replacements. First, it assumes that the first string set element – namely (("`div#view-display-id-`", 8), *startIndex*: 4, *endIndex*: 19) – is incorrect; hence, it checks if any of the valid selectors is of the form "`div#<`**`NEW-STRING`**`>catalog_view > - p.pages span`" – i.e., the erroneous selector with the string "view-display-id-" replaced. In this case, the constraint solver will find one matching selector: "`div#view-id-catalog_view > p.pages span`". Next, our design will move on to the second local string set element and perform the same procedure to find the following matching selector: "`div#view-display-id-catalog_page > p.pages span`".

Case 2: Index Replacement. In this step, our design assumes that the index used to access the list of elements returned by the direct DOM access is incorrect. To check whether this assumption holds, our approach records the size of the array returned by the direct DOM access; this is determined based on the value of an instrumented variable added to the JavaScript code to keep track of the size. The erroneous array index used, if any, is also recorded.

The erroneous array index is compared with the size to see if it falls within the allowed range of indices (i.e., [0–size-1]). If not, our approach will package the following as a possible sickness (belonging to the "Index" sickness class), to be added to the list of possible sicknesses: "`An array index has a numerical value of X that does not fall within the range [0-Z]`"; here, `X` is the erroneous array index, and `Z` is `size-1`.

Case 3: Null/Undefined Checks. By default, our design packages a possible sickness belonging to the "Null/Undefined" class to account for cases where the repair is a DOM Element Validation. In essence, this means the line of code must be wrapped in an `if` statement that checks whether the DOM element being used is `null` or `undefined`. If code termination was caused by a null exception (or undefined exception), the following is packaged and added to the list of possible sicknesses: "`A line of code X accessing a property/method of the DOM element returned by the direct DOM access should (probably) not execute if the DOM element is null (or undefined)`".

5.3 Suggesting Treatments

Once the symptom analyzer has found a list of possible sicknesses, each of these possible sicknesses is examined by the *treatment suggester* (Figure 3, box c). The goal of the treatment suggester is as follows: given a possible sickness, create an *actionable* repair message that would prompt the programmer to modify the JavaScript code in such a way that the symptom represented by the possible sickness would disappear. In order to accomplish this, the code context, as inferred from the supplementary information retrieved by the data collector, is analyzed. This module handles each possible sickness class separately, as described below.

String class. Possible sicknesses belonging to the "String" class require the string value `X` at some line in the JavaScript code to be replaced by another string value `Y`. If applied correctly, this would cause the parameter at the direct DOM access to be valid, so the direct DOM access would no longer output `null`, `undefined`, nor an empty set of elements (i.e., Symptom 1 disappears). As we discovered in Section 3.2, most Parameter Replacement fixes are direct string literal replacements; hence, at first, it may seem straightforward to simply output a message prompting the programmer to directly perform the replacement. However, there are several cases that may make this suggestion invalid, for example:

1. The string value is not represented by a string literal, but rather, by a variable or an expression. Recall that when calculating the string set, gaps may exist in this string set due to string values originating from sources external to the JavaScript code, or due to values not originating from string literals. Hence, a simple "replace" message would be inappropriate to give out as a suggested repair;
2. The string value may be in a line that is part of a loop. Here, a "replace" message may also be inappropriate, since the replacement would affect other (possibly unrelated) iterations of the loop, thereby possibly causing unwanted side effects.

To account for these cases, before outputting a repair message, our approach first examines (a) the string set element type (i.e., is it a variable, expression, or string literal?), and (b) the location (i.e., inside a loop?). Through this analysis, the treatment suggester can provide a suggested repair message. The algorithm essentially performs a "background check" on the code suffering from the bug to determine what message to output. For example, if our design finds that a string set element is in a line inside a loop, and this line executed multiple times a message such as "replace at iteration" or "off by one" – will be given. The complete list of messages is presented in Table 3.

When the running example is subjected to the treatment suggester algorithm, the possible sicknesses found by the symptom analyzer will lead to two REPLACE messages being suggested, one of which is the fix described in Section 2: *Replace the string literal "div#view-display-id-" with "div#view-id-" in line 8.* The

Table 3: List of output messages.

Type	Description
REPLACE	*Replace the string literal X with Y in line L*
REPLACE AT ITERATION	*Wrap line L in an if statement so that the string literal X instead has value Y at iteration I*
OFF BY ONE AT BEGINNING	*Change the lower bound of the for loop containing line L so that the original first iteration does not execute*
OFF BY ONE AT END	*Change the upper bound of the for loop containing line L so that the original last iteration does not execute*
MODIFY UPPER BOUND	*Change the upper bound of the for loop containing line L so that the loop only iterates up to iteration I (inclusive)*
EXCLUDE ITERATION	*Skip iteration I of the for loop containing line L by adding a 'continue'*
ENSURE	*Ensure that the string value at line L has value Y instead of X.* This is a fall back message which is given if a precise modification to the code cannot be inferred by the suggester. Thus, our suggester is conservative in that it only provides a particular suggestion if it is certain that the suggestion will lead to a correct replacement.

other message is a spurious suggestion: *Replace the string literal "catalog_view" with "catalog_page" in line 2*.

Index and Null/Undefined class. For the "Index" class, the suggestion is always as follows: *Modify the array index in line L to some number within the range [0–Z]*. For the "Null/Undefined" class, the suggestion depends on whether the exception was a null exception or an undefined exception. If the exception is a null exception, the following message is given: *Wrap line L in an if statement that checks if expression E is null*. Here the expression E is inferred directly from the error message, which specifies which expression caused the null exception. An analogous message is given if the exception is "undefined".

5.4 Implementation: Vejovis

We implemented our approach in a tool called VEJOVIS, which is freely available for download [12].

The data collector is implemented by instrumenting the JavaScript code using RHINO and running the instrumented application using CRAWLJAX [10]. For the symptom analyzer, we use the string constraint solver HAMPI [8] for `replacementFinder()` (see Algorithm 1, line 45), which looks for viable replacements among the valid parameters found. The symptom analyzer treats the valid parameters found as defining the context-free grammar (CFG).

In keeping with our goal of providing as few suggestions as possible, VEJOVIS allows the users to modify a parameter called the *edit distance bound*. The edit distance bound is a cutoff value that limits the suggested replacement strings to only those whose edit distance with the original string is within the specified value. We use Berico's [3] implementation of the Damerau-Levenshtein algorithm to calculate the edit distance.

6. EVALUATION

To evaluate the efficacy of VEJOVIS in suggesting repairs for DOM-related faults, we answer the following research questions.

RQ3 (Accuracy): What is the accuracy of VEJOVIS in suggesting a correct repair?

RQ4 (Performance): How quickly can VEJOVIS determine possible replacements? What is its performance overhead?

We perform a case study in which we run VEJOVIS on real-world bugs from eleven web applications. To determine accuracy (RQ3), we measure both the precision and recall of our tool. To calculate the performance (RQ4), we compare the runtimes of VEJOVIS with and without instrumentation.

6.1 Subject Systems

The bugs to which we subject VEJOVIS come from eleven open-source web applications, studied also in Section 3; hence, these bugs represent *real-world* DOM-related faults that occurred in the subject systems. We choose two bug reports randomly from the set of bugs that satisfy our fault model, for each of the eleven web applications, for a total of 22 bugs. Note that TaskFreak is not included among the applications studied, as we only found 6 JavaScript bugs from that application, none of which fit the fault model described in Section 4. Descriptions of the bugs and their corresponding fixes (henceforth called the *actual fixes*) can be found online [12]. It took programmers an average of 47 days to repair these bugs after being triaged, indicating that they are not trivial to fix. We had to restrict the number of bugs to two per application as the process of deploying the applications and replicating the bugs was a time and effort intensive one. In particular, most of the bugs were present in older versions of the web applications. This presented difficulties in installation and deployment as some of these earlier versions are no longer supported.

6.2 Methodology

Accuracy. We measure accuracy based on both *recall* and *precision*. In the context of this experiment, recall refers to whether our tool was able to suggest the "correct fix" – that is, whether one of the suggestions provided by VEJOVIS matches the *actual* developer fix described in the corresponding bug report. Hence, in this case, recall is a binary metric (i.e., either 0% or 100%), because the actual fix either appears in the list of suggestions, or it does not. Note that in some cases, the suggested fix is not an exact match of the applied fix, but is semantically equivalent to it, and is considered a match. Precision refers to the number of suggestions that match the actual fix divided by the number of suggestions provided by VEJOVIS. Again, since there is only one matching fix, precision will be either 0 (if the correct fix is not suggested), or $\frac{1}{\#Suggestions}$. This metric is a measure of how well VEJOVIS prunes out irrelevant/incorrect fixes.

To measure the above metrics, we first replicated the bug, and ran VEJOVIS with the URL of the buggy application and the direct DOM access information (i.e., line number and enclosing function) as input; for the libraries, the bugs are replicated by using the test applications described in the bug reports. VEJOVIS outputs a list of suggestions for the bug, which we compare with the actual developer fix to see if there is a match. Based on this comparison, we calculated the recall and precision for that particular attempt. In our experimental setup, the suggestions are sorted according to the edit distance of the replacement string with respect to the original string, where replacements with smaller edit distances are ranked higher. Suggestions for "null" or "undefined" checks are placed between suggestions with edit distance 5 and those with edit distance 6. In the event of a tie, we assume the worst case i.e., the correct fix is ranked lowest among the suggestions with the same edit distance.

To test our assumption that the replacement parameter closely resembles the original parameter, we control the edit distance bound (defined in Section 5.4) for VEJOVIS. We first run our experiments with an edit distance bound of *infinity*, which, means the suggestions given do not have to be within any particular edit distance relative to the original string being replaced (i.e., no edit distance bound assigned). Then, to observe how this bound affects VEJO-

Table 4: Accuracy results, with edit distance bound set to infinity i.e., no bound assigned. BR1 refers to the first bug report, and BR2, the second bug report (from each application). Data in parentheses are the results for when the edit distance bound is set to 5.

Application	Accurate?		Precision	
	BR1	BR2	BR1	BR2
Drupal	✓ (✗)	✓ (✗)	3% (0%)	25% (0%)
Ember.js	✓ (✓)	✓ (✓)	50% (50%)	33% (100%)
Joomla	✓ (✓)	✓ (✓)	1% (25%)	1% (100%)
jQuery	✓ (✓)	✗ (✗)	1% (25%)	0% (0%)
Moodle	✓ (✓)	✓ (✓)	3% (33%)	3% (100%)
MooTools	✓ (✗)	✓ (✓)	50% (0%)	50% (50%)
Prototype	✓ (✓)	✓ (✓)	17% (50%)	50% (50%)
Roundcube	✓ (✓)	✗ (✗)	1% (25%)	0% (0%)
TYPO3	✓ (✓)	✓ (✓)	1% (100%)	100% (100%)
WikiMedia	✓ (✗)	✓ (✓)	4% (0%)	1% (100%)
WordPress	✓ (✓)	✓ (✓)	3% (7%)	1% (50%)

VIS' accuracy, we re-run our experiments with a smaller edit distance bound of 5. We choose the value 5 based on a pilot study.

Performance. For each bug, we measure the performance overhead introduced by VEJOVIS' instrumentation by comparing the corresponding web application with and without instrumentation. This evaluates the performance of the data collection phase of VEJOVIS. We also measure the time it takes for VEJOVIS to generate the repair suggestions. This evaluates the performance of the symptom analysis and treatment suggestion phases of VEJOVIS.

6.3 Results

Accuracy. Table 4 shows the results of our experiments when the edit distance bound is set to infinity i.e., no bound is assigned (numbers outside parentheses). The "Accurate" column of Table 4 indicates, for each bug, whether the actual fix appears among the list of repairs suggested by VEJOVIS (i.e., the recall was 100%). As the results show, assigning no bound causes VEJOVIS *to accurately suggest repairs for 20 out of the 22 bugs, for an overall recall of 91%*. The only unsuccessful cases are the second bugs in Roundcube, where, the correct replacement selector is ":focus:not(body)", and jQuery, where the correct replacement selector is "[id="nid"]"; VEJOVIS does not currently support these CSS selector syntax .

Note that in three of the successful cases, the repair suggestion does not exactly match the actual fix, but rather is equivalent to (or close to) the actual fix.

First, in the second TYPO3 bug, the actual fix documented in the bug report is to add a check to ensure that the NodeList `valueObj`, which is populated by a direct DOM access call to *getElementsByName()*, has a length greater than 0, thereby preventing the use of `valueObj[0].value` from throwing an undefined exception. VEJOVIS, in contrast, suggested an alternate but equivalent fix with no side effects, namely adding a check to see if the expression `valueObj[0]` is undefined before trying to access one of its properties.

Second, in both the first Moodle bug and the second Prototype bug, VEJOVIS provides the fallback "ENSURE" suggestion. In the Moodle bug, VEJOVIS suggests the following: *Ensure the value of variable* `itemid` *is "id_itemname" instead of "itemname"*; this is because the string literal "itemname" originated from an anonymous function, which our implementation currently does not support, leaving a gap in the string set. Nonetheless, this simplifies the debugging task for the programmer, as it points her directly to the problem – i.e., the string "itemname", located somewhere in the JavaScript code, needs to be changed to "id_itemname". Similarly, in the Prototype bug, VEJOVIS suggests the following: *Ensure the expression* `id.replace(/[\.:]/g, "\\$0")` *has value "outer.div" instead of "outer\\$0div"*. Again, while VEJOVIS is not able to provide the exact fix, it points the programmer to the relevance of the *replace()* method in the fix. These results show that even in cases when VEJOVIS is unable to fully resolve the origins of the erroneous selector's string values, it still provides meaningful suggestions that facilitate debugging and are hence useful to the programmer.

Table 5: Rank of the correct fix when suggestions are sorted by edit distance. The denominator refers to the total number of suggestions. Top ranked suggestions are in bold.

Application	Rank	
	BR1	BR2
Drupal	31 / 40	**01 / 04**
Ember.js	**01 / 02**	**01 / 03**
Joomla	**01 / 88**	**01 / 88**
jQuery	02 / 108	–
Moodle	02 / 37	**01 / 37**
Moodle	02 / 02	**01 / 02**
Prototype	**01 / 06**	**01 / 02**
Roundcube	04 / 79	–
TYPO3	**01 / 187**	**01 / 01**
WikiMedia	06 / 24	**01 / 71**
WordPress	13 / 30	**01 / 170**

Among the successful cases, the average precision ("Precision" column of Table 4) is approximately 2%; on average, this translates to VEJOVIS providing 49 suggestions for each bug, with a maximum of 187 total suggestions for the first TYPO3 bug. The high number of suggestions motivated us to implement the simple ranking scheme based on edit distance (Section 6.2).

Table 5 shows, for each bug, the rank of the actual fix among the list of suggestions provided by VEJOVIS; only the cases where the actual fix appears among the list of suggestions are considered. As shown in the table, the correct fix appears as the first suggestion in 13 out of the 20 cases, and as the second suggestion in three more cases. In fact, for the WordPress bug, the correct fix is tied for first place among 13 bugs; we listed its rank as 13 because we consider the worst case. Hence, despite providing a large number of suggestions on average when the edit distance bound is set to infinity, *our simple ranking scheme based on edit distance ranked most of the actual fixes near the top of the list of suggestions*.

As mentioned, the above results were obtained with an edit distance bound of infinity. To quantify the effects of using a finite bound, we re-ran the above accuracy experiment with an edit distance bound of 5. The results are shown in parentheses in Table 4. As the results show, assigning a bound of 5 decreases the number of successful cases from 20 to 16, where four additional cases became unsuccessful because the actual fix required replacing the original parameter with another parameter that is more than an edit distance of 5 away. However, the precision jumps dramatically to 36% with this bound, which translates to around 3 suggestions given for each bug on average. Hence, assigning a finite edit distance bound can significantly decrease the number of suggestions, which makes the list of suggestions more manageable; however, this comes at the cost of lower recall of 73% (as compared to 91%).

Performance. There are two sources of performance overhead in VEJOVIS: (1) instrumentation overhead, and (2) symptom analysis and treatment suggestion overhead. Table 6 shows the results. The time taken with and without instrumentation during the data collection phase of VEJOVIS are shown in the second and third columns of the table. The time varies per application, ranging from 16.3 to 85.0 seconds, for an average of 39.3 seconds. The time in the symptom analysis and treatment suggestion phases is shown in the last

Table 6: Performance results.

Application	Crawl Time w/o Instrumentation (s)	Crawl Time with Instrumentation (s)	Average Treatment Time (s)
Drupal	28.5	49.6	1.8
Ember.js	10.8	16.3	0.8
Joomla	57.0	85.0	6.1
jQuery	12.0	19.0	4.1
Moodle	46.9	59.6	7.4
MooTools	13.3	19.1	0.9
Prototype	12.0	17.4	8.6
Roundcube	25.1	34.7	3.4
TYPO3	39.9	72.8	6.5
WikiMedia	15.9	24.8	5.4
WordPress	20.2	33.8	7.0
Average	-	39.3	4.7

column. The average time for these phases is 4.7 seconds, ranging from 0.8 to 7 seconds. Thus, on average, VEJOVIS takes less than one minute (44 seconds) to find the correct fix, with a worst-case time of 91.1 seconds for Joomla.

7. DISCUSSION

Extensions. First, VEJOVIS suggests treatments belonging to the Parameter Modification and DOM Element Validation categories as mentioned in our empirical study of common fixes in Section 3. While these together constitute more than half of the fix types we found in the study, another common fix category is Method/Property Modification, in which a DOM API method or property is added, removed or replaced with another method/property. We do not incorporate this fix category in our design; however VEJOVIS can be extended to account for this category. For instance, it is possible, in some cases, to reduce the problem of replacing DOM methods to replacing CSS selectors. As an example, replacing `getElementById("str")` with `getElementsByClassName("str")` can be thought of as replacing the CSS selector "#str" with ".str".

Second, the results of our evaluation show that while VEJOVIS accurately predicts the actual fix in almost all of the bug reports analyzed, the number of suggestions provided can be large, thereby lowering its precision. In our evaluation, we showed that ranking the fixes based on edit distance makes the actual fix rank high in many cases. We are currently exploring more intelligent ways to perform this ranking; for example, based on the textual patterns of the strings.

Threats to Validity. An *external validity threat* is that the bugs we analyzed come from only 11 web applications. However, the systems considered were developed for different purposes and hence, represent a reasonable variety. Further, the corresponding bug reports have been fixed by the developers, and are therefore representative of the issues encountered in practice.

An *internal threat* to validity is that we have assumed the fixes described in the bug reports are correct as many experienced developers are typically involved with providing patches for these bugs. Nonetheless, to mitigate this threat, we carefully analyzed the patches provided in the bug reports and have tested the fixes on our own platforms to see if they are sound.

Additionally, the bugs we considered in our evaluation were taken from the bug report study in Section 3, which may be a potential source of bias. This threat can be mitigated by considering other applications, which we plan to do in the future. As for repeatability, VEJOVIS is available [12], and the experimental subjects are open source, making our case study fully repeatable.

8. RELATED WORK

Program Repair. Program repair refers to the act of fixing bugs through automated techniques. Perhaps the best-known application of program repair is to data structures. Demsky et al. [5] use formal specifications to suggest fixes for data structures. Elkareblieh et al. [6] use programmer specified assertions for data structure repair. However, these techniques are limited to repairing data-structures, and do not fix the underlying defect that produced the erroneous structure. While the DOM can be considered a data-structure, VEJOVIS goes beyond the DOM and actually can suggest ways to modify the JavaScript code based on the defective DOM access.

Generating fixes at the source code level has gained attention recently [2, 18, 19]. Weimer et al. [19] propose the use of genetic algorithms for repairing C programs. The main idea is to copy other parts of the program to the faulty portion of the program and check if the modified program passes the existing test cases. However, it is not clear how this technique could be applied to web applications, where the code base includes different languages such as JavaScript and HTML/DOM.

In recent work, Zhang et al. [20] propose FlowFixer, a technique to repair broken workflows in Java-based GUI applications. Similar to VEJOVIS, FlowFixer attempts to find repairs for errors that arise due to a mismatch between the code and the GUI state. However, there are two main differences between VEJOVIS and FlowFixer. First, FlowFixer is concerned with correcting the *sequence of user actions* applied to the GUI; in contrast, VEJOVIS is concerned with correcting the *code* that drives the functionality of the application. Second, FlowFixer uses random testing to find replacements; VEJOVIS is different in that it performs a systematic traversal of the DOM to find valid replacement selectors.

Web Application Repair. There has been limited work on exploring fault repair for web applications. Carzaniga et al. [4] propose automatic workarounds for web applications that experience errors in using APIs by replacing the buggy API call sequence with a functionally equivalent, but correct sequence. Samimi et al. [16] have proposed a technique for PHP code to fix errors that result in the generation of webpages with malformed HTML; similar work has been done by Zheng et al. [21]. Neither of these techniques consider JavaScript code, nor do they apply to DOM-related JavaScript faults. In recent work, Jensen et al. [7] and Meawad et al. [9] introduce techniques to transform unsafe *eval* calls in JavaScript code to functionally equivalent, but safe constructs. This is more of a prevention than repair technique. However, they do not consider JavaScript errors, and in particular DOM-related errors.

9. CONCLUSION

JavaScript interacts extensively with the DOM to create responsive applications; yet, such interactions are prone to faults. In this paper, we attempt to understand common fixes applied by programmers to DOM-related faults. Based on these findings, we propose an automated technique for providing repair suggestions for DOM-related JavaScript faults. Our technique, implemented in a tool called VEJOVIS, is evaluated through a case study of 22 bugs based on real-life bug reports. We find that VEJOVIS can accurately predict the repair in 20 out of the 22 bugs, and that the correct fix appears first in the list of fix suggestions for 13 of the 20 bugs.

10. ACKNOWLEDGMENTS

This research was supported in part by an NSERC Strategic Project Grant, a Four Year Fellowship (FYF) from UBC, and a research gift from Intel Corporation. We thank the anonymous reviewers of ICSE 2014 for their feedback to improve the paper.

11. REFERENCES

[1] Autopager jQuery extension. http://code.google.com/p/jquery-autopager/.

[2] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of World Congress on Evolutionary Computation (CEC)*, pages 162 –168. IEEE Computer Society, 2008.

[3] Berico. Damerau-Levenshtein Java implementation. http://www.gettingcirrius.com/2011/06/calculating-similarity-part-3-damerau.html.

[4] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 237–246. ACM, 2010.

[5] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 176–185. ACM, 2005.

[6] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 64–73. ACM, 2007.

[7] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 34–44. ACM, 2012.

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

[9] F. Meawad, G. Richards, F. Morandat, and J. Vitek. Eval begone!: semi-automated removal of eval from JavaScript programs. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 607–620. ACM, 2012.

[10] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.

[11] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Auto-locating and fix-propagating for HTML validation errors to PHP server-side code. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 13–22. IEEE Computer Society, 2011.

[12] F. Ocariza. Vejovis. http://ece.ubc.ca/~frolino/projects/vejovis/.

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

[14] F. Ocariza, K. Pattabiraman, and A. Mesbah. AutoFLox: An automatic fault localizer for client-side JavaScript. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 31–40. IEEE Computer Society, 2012.

[15] 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)*, pages 100–109. IEEE Computer Society, 2011.

[16] H. Samimi, M. Schafer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 277–287. IEEE Computer Society, 2012.

[17] W3C. Cascading style sheets level 2 revision 1 specification: Selectors, 2011. http://www.w3.org/TR/CSS2/selector.html.

[18] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 61–72. ACM, 2010.

[19] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 364–374. IEEE Computer Society, 2009.

[20] S. Zhang, H. Lü, and M. D. Ernst. Automatically repairing broken workflows for evolving gui applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 45–55. ACM, 2013.

[21] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a z3-based string solver for web application analysis. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 114–124. ACM, 2013.