

# Dompletion: DOM-Aware JavaScript Code Completion

Kartik Bajaj

Karthik Pattabiraman

Ali Mesbah

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

## ABSTRACT

JavaScript is a scripting language that plays a prominent role in modern web applications. It is dynamic in nature and interacts heavily with the Document Object Model (DOM) at runtime. These characteristics make providing code completion support to JavaScript programmers particularly challenging. We propose an automated technique that reasons about existing DOM structures, dynamically analyzes the JavaScript code, and provides code completion suggestions for JavaScript code that interacts with the DOM through its APIs. Our automated code completion scheme is implemented in an open source tool called `DOMPLETION`. The results of our empirical evaluation indicate that (1) DOM structures exhibit patterns, which can be extracted and reasoned about in the context of code completion suggestions; (2) `DOMPLETION` can provide code completion suggestions with a recall of 89%, precision of 90%, and an average time of 2.8 seconds.

## Categories and Subject Descriptors

D.2.3 [Software Engineering]: Coding Tools and Techniques

## General Terms

Algorithms, Design, Experimentation

## Keywords

Code completion, JavaScript, DOM, Web applications

## 1. INTRODUCTION

Web applications are growing fast in popularity. JavaScript is nowadays heavily used to provide user interactivity on the client-side. However, integrated development environment (IDE) support for web development still lags largely behind, which makes writing JavaScript code challenging for many developers. One of the major challenges faced by programmers is handling the interactions between JavaScript and the Document Object Model (DOM) [2, 32]. The DOM is a standard object model representing HTML at runtime. JavaScript uses DOM APIs for dynamically accessing,

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. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ASE'14, September 15-19, 2014, Vasteras, Sweden.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-3013-8/14/09 ...\$15.00.

<http://dx.doi.org/10.1145/2642937.2642981>.

traversing, and updating the content, structure, and style of HTML documents. In a prior empirical study we found that such JavaScript-DOM interactions are responsible for the majority of programmer errors in web applications [32].

The DOM of a web application evolves at runtime due to (1) client-side mutations through JavaScript, and/or (2) server-side dynamic HTML code generation. Due to the inherent difficulties of reasoning about this dynamic DOM evolution, there is a lack of IDE support for writing JavaScript code that performs DOM operations. As a result, when writing JavaScript code, the developer needs to consider all possible states the DOM could potentially be in when it is accessed by the JavaScript code, which is an error-prone [32] and time-consuming task.

Prior studies have revealed that developers make heavy use of code completion features provided within their IDE [31], and that they require code completion support for XML-based languages [34] such as XHTML and XPath. Although code completion for traditional languages such as Java has been an active research topic [36, 24, 16, 8, 7, 30, 17], code completion for web applications and JavaScript in particular, has received very limited attention from the research community. To the best of our knowledge, DOM-based code completion support for JavaScript has not been addressed in the literature yet, which is the focus of our work.

To assist web developers in writing JavaScript code that interacts correctly with the DOM, we propose an automated technique based on static and dynamic analysis of the web application's DOM structure and JavaScript code under development. Our approach (1) extracts various DOM states from the application and infers patterns from the observed DOM tree structure; (2) captures and analyzes all JavaScript code that interacts with the DOM; (3) reasons about the consequences of these interactions on the DOM state; and (4) provides code completion suggestions in terms of possible DOM elements and properties that are accessible at that specific line of JavaScript code. These suggestions help developers write valid JavaScript code that correctly interacts with the DOM at runtime. Our work makes the following main contributions:

- A discussion of the importance and main challenges behind DOM-based code completion in JavaScript.
- A fully automated code completion technique, based on a combination of static and dynamic analysis of JavaScript code and the DOM.
- An implementation of our approach in an open source tool called `DOMPLETION`.
- An empirical evaluation to assess `DOMPLETION`, demonstrating its efficacy, real-world relevance and helpfulness. Our examination of five real-world web applications indicates that DOM states do exhibit patterns in their structure, which converge and can hence be learned. The results of our case study on three

```

1 ...
2 var nav = document.getElementById('site-navigation');
3 var button, menu;
4 button = nav.getElementsByTagName('h3')[0];
5 if (!button)
6   return;
7 menu = nav.getElementsByTagName('ul')[0];
8 ...

```

**Figure 1: Example JavaScript code fragment based on *WordPress*.**

open source web applications show that DOMPLETION is capable of providing code completion suggestions with a recall of 89% and a precision of 90%. The results of our user study indicate that DOMPLETION can reduce the development time, while improving the overall accuracy of developers in code completion tasks.

## 2. CHALLENGES AND MOTIVATION

In this section, we discuss some of the challenges faced by developers when writing JavaScript code as well as challenges involved in providing automated code completion support for DOM.

### 2.1 Running Example

Figure 1 presents a JavaScript code fragment, based on the WordPress application [41], to illustrate some of the challenges involved in providing auto-complete features for JavaScript. Figure 2 represents the DOM structure of a subset of the webpage that the JavaScript code in Figure 1 is interacting with. The webpage pertaining to Figure 2 consists of a navigation menu with ID `site-navigation` at the top of the webpage. The navigation menu contains one `h3` element (element 8) and one `ul` element (element 11) that contains a number of `li` elements (elements 12 to 15), constituting the menu items.

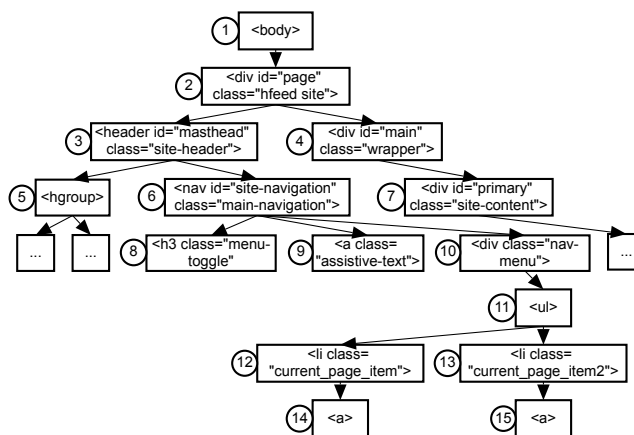
The JavaScript code obtains references of various DOM elements (Lines 2, 4, 7). While writing JavaScript code that interacts with the DOM, the programmer needs to make an educated guess for the IDs, class names and tag names of the elements; i.e., the programmer is only aware of a few possible DOM states for the particular web application. This problem is exacerbated as the size of the application increases and the number of DOM states also correspondingly increase. In the worst case, the programmer can manually inspect the DOM and gather the relevant information, but this is highly time consuming.

The developer’s lack of knowledge about valid JavaScript-DOM interactions can lead to scenarios in which the JavaScript code behaves incorrectly and results in an exception. These exceptions can either cause the JavaScript code to terminate abnormally, or may result in a faulty DOM state. Thus, the developer needs automated tool support that provides suggestions about such valid interactions while she is writing the code.

### 2.2 Challenges

Although JavaScript is syntactically similar to other languages such as Java, there are a number of differences that makes JavaScript challenging for providing code completion features.

**Dynamically typed.** To provide code completion, we need to analyze the code and infer variable types. For strongly typed languages, static analysis is sufficient to analyze variable types, as variables once attached to a particular type remain attached throughout the code execution. Therefore, tools can provide auto-complete



**Figure 2: Overview of the DOM structure for the running example.**

**Table 1: Commonly used CSS selectors**

Selector	Example
.class	.intro
#id	#firstname
element	p
element,element	div, p
element element	div p
element>element	div>p
[attribute]	target
[attribute=value]	target=_blank

features by statically inferring the variable types. However, in a dynamically typed language such as JavaScript, every variable name is bound to an object only at execution time (unless it is null); and it is possible to bind a name to objects of different types during the execution of the program (loose typing). For example, in Figure 1 at Line 3, the variable `button` is declared. However, the type of the variable `button` is determined only when the assignment of variable `button` is executed at Line 4. Further, it is possible to assign this variable to a different type by executing a different assignment statement within the same JavaScript code. The dynamic typed feature of JavaScript makes it difficult to analyze the variable types statically, as these types can be modified during program execution [15, 23].

**DOM Interactions.** JavaScript code frequently interacts with and updates the DOM, using the DOM API calls to mutate the DOM structure. Therefore, when statically analyzing the JavaScript code, the value of variables can either be null if the target element is not present in DOM, or can be a value returned from the DOM. For example, in Figure 1 at Line 2, the variable `nav` is pointing to DOM element with ID `site-navigation`. However, without any knowledge of DOM state, it is not clear what type of DOM element it is, or even if it exists in the current DOM state. Therefore, to effectively provide code completion for JavaScript code, we need to have knowledge about the target DOM state(s). This is challenging because the number of target DOM states can be unknown, and statically inferring this information is difficult.

### 2.3 Scope of the paper

Our prior empirical studies has shown the consistent prevalence of DOM-related issues within JavaScript code in web applications [32, 4]. Therefore we provide code completion support for the DOM interactions within JavaScript code in this work.

In general, DOM interactions can be divided into three categories:

**Table 2: Overview of elements present in DOM tree of Figure 2**

Element	Tag	ID	Classes
1	body	-	-
2	div	page	hfeed, site
3	header	masthead	site-header
4	div	main	wrapper
5	hgroup	-	-
6	nav	site-navigation	main-navigation
7	div	primary	site-content
8	h3	-	menu-toggle
9	a	-	assistive-touch
10	div	-	nav-menu
11	ul	-	-
12	li	-	current_page_item
13	li	-	current_page_item2
14	a	-	-
15	a	-	-

- Obtaining references to DOM elements:** To interact with the DOM from JavaScript code, the developer first needs to obtain references to DOM elements in the appropriate DOM states. This means that she needs to have knowledge of the DOM elements in the target DOM states.
- Modifying existing DOM elements:** JavaScript developers can use the DOM API to modify DOM elements. This requires that the appropriate references have been obtained within JavaScript code. The developer also needs to know the type of the DOM element and its properties, to write meaningful code.
- Adding/Deleting DOM elements:** The DOM API can also be used to add or delete DOM elements within the DOM. To that end, the developer first needs to obtain references to a proper element in the DOM tree.

In this paper, we focus on performing code completion while obtaining references to DOM elements, as this is the most basic step for all three DOM interactions. This is also challenging for the developer as it requires understanding of the DOM states and their elements, and mapping these back to the code. We also consider the effects of programmatic additions, deletions, and modifications to the DOM, in the code completion suggestions provided to the programmer.

### 3. APPROACH

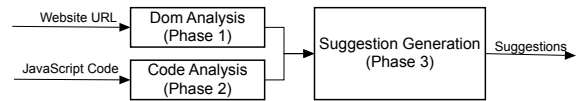
Our DOM based code completion approach consists of 3 main phases as outlined in Figure 3: (1) DOM analysis, (2) JavaScript Code analysis, and (3) Suggestion generation.

The *DOM analysis* phase involves crawling the web application and generating a list of element locators within the DOM states. The *code analysis* phase involves analyzing the JavaScript code, creating a list of in-scope variables and intercepting the DOM API calls that include reading and writing to the DOM. The results from the above two phases are passed to the *suggestion generation* phase and combined to generate a list of auto-complete suggestions that are presented to the developer in their IDE.

The DOM analysis phase is conducted once when the IDE is opened and periodically in the background to explore new DOM states by executing the in-progress JavaScript code. The code analysis and suggestion generation phases are executed every time the developer requests code completion support using a hot-key (for example).

#### 3.1 Our Design

**DOM interactions.** Our goal is to provide code completion support to developers based on an analysis of the JavaScript code as well as the DOM structure and its mutations. When the developer

**Figure 3: Approach overview**

invokes the code completion functionality, we need to extract a potential list of valid DOM elements, i.e., elements that have been spotted on relevant DOM states, and present that list to the developer writing the JavaScript code. The developer can then select the desired element’s ID, tag name or class name from the presented list.

DOM access methods such as `querySelectorAll(selector)`, let developers enter a CSS selector as a parameter and return the matching DOM elements. CSS selectors use pattern matching rules to determine which style rules apply to elements in the DOM tree. These patterns, called selectors, may range from simple element names to rich contextual patterns. We use CSS selectors to identify the range of possibilities for an element. As CSS selectors can model DOM access methods such as `getElementById` and `getElementsByTagName` universally, our technique can work with any DOM access method or library, including the *jQuery* library. Table 1 provides the list of commonly used CSS selectors. A complete list of CSS selectors can be found here [9] In the following sections, the term *DOM Element Locators* refer to the parameters passed to the DOM access methods.

Assume that the developer invoked the code completion function when writing the code in Line 2 in Figure 1, just after typing the `getElementById`. From the DOM tree in Figure 2, the DOM state contains 5 different DOM elements (See Table 2) with IDs. Therefore, our approach would show all the five elements as potential candidates for the `getElementById` in line 2. This way, the developer would know exactly which elements exist in the DOM and can choose the appropriate element for the argument to the `getElementById`. However, in more complex cases, there may be too many suggestions to be useful to the developer. Below, we consider two methods for filtering the list of DOM elements to present to the developer.

**1. Hierarchical nature of DOM elements.** The hierarchical positioning of DOM elements can be used for narrowing down the scope of the suggestions presented to the developer. For example, in Line 4 of Figure 1, the developer uses the `nav` parent object to call the `getElementsByTagName()` function. As seen from the code, the JavaScript object `nav` points to the DOM element with ID as `site-navigation`, i.e., Element 6 (Figure 2); we narrow down the search space to the child elements of element 6. In this case, we only need to look through the child elements of Element 6, namely Elements 8–15. However, in Line 2, the developer uses the `document` object as the parent object and it points to the root node of the tree, i.e., Element 1. In this case, we will need to scan the complete tree, i.e., Elements 2 to 15.

**2. JavaScript objects in scope.** At any point in the code, some DOM elements may not be accessible from the JavaScript code. At Line 2, the object `nav` points to the DOM element with ID as `site-navigation` (Element 6) and is then used as a parent object in Lines 4 and 7. This means only child elements of Element 6 (i.e., Elements 8–15) are accessible at lines 4 and 7. Therefore, to provide meaningful code-completion suggestions at Lines 4 and 7, we need to collect information about the declared objects in their scope and only consider those when providing suggestions.

**Usage Model and Assumptions.** We make three assumptions in our work. First, we assume that the developer has access to the en-

**Table 3: Optimization techniques used to minimize the list of DOM element locators.**

Optimization Type	Constraints		Example	
	Input	Output	Input	Output
Removing Duplicates	$m_1 = m_2$ $W_{1j} = W_{2j}$	$m_1 = m_2 = m_3$ $W_{1j} = W_{2j} = W_{3j}$	ul.nav-menu.toggled-on li.item ul.nav-menu.toggled-on li.item	ul.nav-menu.toggled-on li.item
Combining Similar IDs	$m_1 = m_2$ $T_{1j} = T_{2j}$ $C_{1j} = C_{2j}$ $I_{1j} \neq I_{2j}$	$m_1 = m_2 = m_3$ $T_{1j} = T_{2j} = T_{3j}$ $C_{1j} = C_{2j} = C_{3j}$ $I_{3j} = I_{1j} + I_{2j}$	ul.nav-menu.toggled-on li#item1 ul.nav-menu.toggled-on li#item2	ul.nav-menu.toggled-on li#item1#item2
Combining Similar Classes	$m_1 = m_2$ $T_{1j} = T_{2j}$ $I_{1j} = I_{2j}$ $C_{1j} \neq C_{2j}$	$m_1 = m_2 = m_3$ $T_{1j} = T_{2j} = T_{3j}$ $I_{1j} = I_{2j} = I_{3j}$ $C_{3j} = C_{1j} + C_{2j}$	ul.nav-menu.toggled-on li.item1 ul.nav-menu.toggled-on li.item2	ul.nav-menu.toggled-on li.item1.item2

tire JavaScript code of the web application, so that we can analyze it. Second, we assume that the developer has access to the HTML template of the website, and can hence obtain the initial DOM. This is needed as we crawl the partially complete application to obtain the dynamic DOM state(s). Finally, we assume that the JavaScript code within the editor is partially complete. By partially complete, we mean that (1) JavaScript code that is syntactically correct, (2) all the variables in use are defined, and (3) it uses no global variables (global variables are considered harmful [42]). The assumptions listed above align with the JavaScript development guidelines enforced by various organizations [39, 19, 20, 14].

There is no special input required from the developer, other than pressing the code completion short key while writing the code. However, our approach can use inputs provided by the developer, typically the first one or two letters of the desired DOM identifier, to enhance the accuracy of code completion. Code completion is provided when any of the functions in Table 1 or similar functions (libraries and custom functions) are written by the developer. The only restriction is that the library function should attempt to access a DOM element through a list of supported DOM element locators.

The output of our approach is a list of DOM element locators which are shown to the developer as a ranked list. The generated list is dependent on (1) JavaScript variables in the scope, (2) the scope of the parent element within the DOM, and (3) additions/deletions made to the DOM on the available JavaScript code paths.

### 3.2 DOM Analysis

In order to provide DOM based code completion suggestions, we first need to get information about the elements present in the DOM. To that end we crawl the web application. Further, for each DOM element on the obtained DOM trees, we extract its locator information. Each leaf node in the DOM is expressed as a sequence of tag names, IDs and class names found in the hierarchy. The list of DOM element locators is generated separately for each available DOM state crawled. Finally, the DOM element locators generated from all the DOM states obtained are combined to generate a superset that are then used in the later phase.

**Converting DOM to a List of DOM Element Locators.** Every element within the DOM hierarchy can be represented as a space separated sequence of nodes beginning from the root node. Each node is itself a DOM element and can be represented as a combination of tag name, id and a list of classes attached to the element. For every leaf node in the DOM, we reverse engineer a unique DOM element locator capable of locating that element on the DOM; thereby covering all the DOM elements present within the DOM tree. The list of DOM element locators for each DOM state is then accumulated to generate a superset of all locator strings.

More formally, we represent the list of DOM element locators for each state as follows:

$$R = \{R_i \mid 0 \leq i < n\}$$

$$R_i = \{W_{ij} \mid 0 \leq j \leq m_i\}$$

$$W_{ij} = T_{ij} \cup I_{ij} \cup C_{ij}$$

**Table 4: DOM element locators for Element 9 ( $R_3$ )**

Element ( $W_{ij}$ )	Depth	Tag name ( $T_{ij}$ )	Id ( $I_{ij}$ )	Class name ( $C_{ij}$ )
1	0	body	-	.home.blog.custom-font-enabled.single-author
2	1	div	#page	.hfeed.site
3	2	header	#masthead	.site
6	3	nav	#site-navigation	.main-navigation
9	4	a	-	.assistive-touch

where,

- $R$  is the list of DOM element locators for each DOM state.
- $R_i$  is the set of nodes in the  $i^{th}$  row in the list of DOM element locators.
- $n$  is the total number of leaf nodes in the DOM state.
- $W_{ij}$  represents the element in the  $i^{th}$  row at  $j^{th}$  depth.
- $m_i$  is the depth if  $i^{th}$  leaf node.
- $T_{ij}$ ,  $I_{ij}$ , and  $C_{ij}$  represent the tag name, id and class names attached to each node respectively.

For example, for Figure 2 the size of  $R$  i.e., total number of leaf nodes ( $n$ ) is 7. For the 3<sup>rd</sup> leaf node in Figure 2 i.e., Element 9, the depth ( $m_3$ ) is 4. The DOM element locators at each level for Element 9 ( $i = 3, 0 \leq j \leq 4$ ) are presented in Table 4.

As the number of DOM states increases, the number of rows in the list of DOM element locators exponentially, thereby increasing the space and time complexity of the approach. To effectively provide code completion suggestions, we need to compress the list of DOM element locators of the superset that is generated at the end of this step. We use three compression techniques: (1) eliminating duplicate DOM element locators, (2) combining DOM element locators with similar IDs, and (3) combining DOM element locators with similar classes.

Table 3 summarizes the techniques used to compress the representation. The first column shows the compression technique, the second column shows the constraints that must be satisfied by the DOM element locators in order to be compressed, and the third column shows the output produced by the compression after the compression. The remaining columns illustrate the compression techniques with examples. As seen from the examples, the DOM hierarchical information is preserved during compression, therefore preserving the quality of suggestions provided by DOMPLETION.

### 3.3 Code Analysis

This phase analyzes how a particular piece of JavaScript code interacts with the DOM by executing different code paths in an isolated environment. To do so, we need to instrument the control flow of the program and analyze each code path separately. The overall approach is presented in Algorithm 1. The JavaScript code in the

## Algorithm 1: JavaScript Code Analysis

```
Data: (JavaScript Code)  $\mathbb{C}$ 
Result: DOM element locators used in the program
1  $n = \text{countNumFunctionDefinitions}(\mathbb{C});$ 
2 (Function Definitions)  $\mathbb{F} = \text{extractFunctionDefinitions}(\mathbb{C});$ 
3 (Executing Code)  $\mathbb{E} = \mathbb{C} - \mathbb{F};$ 
4  $m = \text{countNumIfConditions}(\mathbb{E});$ 
5 (Code Paths)  $\mathbb{P} = [\mathbb{E}];$ 
6 while  $\text{containsIfStatement}(\forall \text{ elements in } \mathbb{P})$  do
7    $\text{numElements} = \text{size}(\mathbb{P});$ 
8   for  $i = 0$  to  $\text{numElements}$  do
9      $\lambda = \mathbb{P} \rightarrow \text{pop}();$ 
10    if  $\text{containsIfStatement}(\lambda)$  then
11       $[\alpha, \beta] = \text{generateCodePaths}(\lambda);$ 
12       $\text{insertLogStatements}(\alpha, \beta);$ 
13       $\mathbb{P} \rightarrow \text{push}(\alpha, \beta)$ 
14    else
15       $\mathbb{P} \rightarrow \text{push}(\lambda)$ 
16    end
17  end
18  $\mathbb{R} = \emptyset;$ 
19 while  $\lambda = \mathbb{P} \rightarrow \text{pop}()$  do
20    $\text{codePath} = \mathbb{F} \cup \lambda;$ 
21    $\text{logs} = \text{execute}(\text{codePath});$ 
22    $\mathbb{R} \rightarrow \text{push}(\text{analyze}(\text{logs}));$ 
23 end
24 return  $\mathbb{R}$ 
```

```
1 ...
2 var nav = document.getElementById('site-navigation');
3 var button, menu;
4 button = nav.getElementsByTagName('h3')[0];
5 ! button;
6 completeLog("func1 T", "");
7 return;
8 menu = nav.getElementsByTagName('ul')[0];
9 ...
```

**Figure 4:** One possible code path within the executing code. Statements highlighted in green are inserted by DOMPLETION

application is parsed to extract both function definitions (Line 2), and the corresponding code segments (Line 3). The code segments are then modified to replace the `if` conditions (Lines 6–18) in order to track their control-flow. This operation can generate up to  $2^m$  different versions, where  $m$  is the number of `if` conditions within the code segments. The `if` conditions are replaced recursively starting from the innermost segments and proceeding outwards. For each newly generated code path, relevant log statements are inserted within the code to keep track of the code path (Line 12). The executing code segment is then combined with function definitions extracted in Line 2, therefore covering as many code paths as possible (Lines 20–24).

For the example given in Figure 1, two different code paths are generated and the results for the code completion will reflect all these paths. Figure 4 represents one possible code path executed by the code. The first `if` condition is removed from the code and the code path with `true` block is executed. Note that the code after Line 7 will not be executed due to the execution of `return` statement, and will not be considered. The logs generated (Line 22) after the code execution are then analyzed (Line 23) and a list of DOM elements referred in the code is generated. The results of this analysis are used as an input for the next phase.

This execution of JavaScript code is performed in an isolated environment, by intercepting the calls to DOM API and global objects. JavaScript code contains references to global objects available within the browser. These objects can be classified into two

```
1 create #site-navigation,
2 local nav|#site-navigation,
3 create #site-navigation h3,
4 local button|#site-navigation h3,
5 func1 T
```

**Figure 5:** Logs generated after path execution. First 6 statements are generated by the execution environment and the last statement is executed from the JavaScript code under analysis.

## Algorithm 2: Generating Regular Expression from given DOM element locator path

```
Data: List of DOM element locators from DOM  $\mathbb{S}_1$ , DOM element locator from JavaScript code  $\mathbb{S}_2$ 
Result: Matching set of DOM element locators
1 (Hierarchy)  $\mathbb{H} = \text{splitWithSpace}(\mathbb{S}_2);$ 
2 for  $i$  in  $\mathbb{H}$  do
3    $r = \text{createRegex}(\mathbb{H}[i]);$ 
4   (Matching Set)  $\mathbb{M} = r \rightarrow \text{match}(\mathbb{S}_1);$ 
5   if  $\mathbb{M} \neq \text{null}$  then
6      $\mathbb{M} = \mathbb{R} \rightarrow \text{removeMatchingPart}(\mathbb{M});$ 
7      $\mathbb{S}_1 = \mathbb{M};$ 
8      $\text{result} = \mathbb{S}_1;$ 
9   else
10    return  $\emptyset;$ 
11    break;
12  end
13 end
14 return  $\mathbb{S}_1;$ 
```

main categories [18]: (1) Browser Objects and (2) HTML DOM Objects. Browser objects include window, navigator, screen, history, and location. HTML DOM objects include document, and element objects. These objects expose an API that can be used to interact with the browser and modify the HTML contents. Therefore, we need to execute these APIs in an isolated environment.

To execute the objects in an isolated environment, we redefine the functions within these objects and insert log statements to keep track of when the particular function was called and what parameters were passed to it. Additional variables are added for the code analysis purpose. Every reference to DOM element returned by these re-defined functions contains a property named `csspath` to keep track of the DOM element locators attached to that element, which is then logged. The logs generated after executing the code path in Figure 4 are presented in Figure 5.

## 3.4 Suggestion Generation

To provide code completion suggestions, we need to match the DOM element locators we extracted from the DOM hierarchy to the DOM element locators used in the JavaScript code. However, the information available in the DOM element locator is often incomplete. For example, in Figure 1 at line 4, the developer used only the tag name to select a particular DOM element. However, there can be multiple DOM elements with the same tag name. Also, from the given JavaScript code at line 2, we can see that the parent object (`nav`) refers to the DOM element with id `site-navigation`. However, this need not be its immediate child, but can be any descendant of that element. Therefore a simple string matching algorithm cannot be used to match the DOM element locators, and we need an alternative and a flexible way to overcome this problem.

We use the information from DOM element locators to form patterns, and then use the patterns to find relevant suggestion. For example, in Figure 1 at line 4, we can see that the developer is looking for an element with tag name `h3` among the descendants of element with id `site-navigation`. Therefore, we only look for element

```

1 RE1 = /^[^#\.\ ]*(?=[ \.\#])#site-navigation↔
  (\.\.[^#\.\ ]+)*(\.([^\.\#]+\.))?(?=[ \ ])/;
2
3 RE2 = /h3([^\.\# ]+)*(?=[ \.]) (\.\.[^#\.\ ]+)*↔
  *(\.([^\.\#]+\.))?(?=[ \ ])/;

```

**Figure 6: First regular expression matches any type of DOM element with `id='site-navigation'` and any number of classes attached to it. Second regular expression matches all the `h3` tags with any id and any number of classes attached to them, within the child elements of the previously matched element.**

with tag name `h3` among the descendants of the first element. To express these constraints, we convert the DOM element locators to regular expressions. We create regular expressions satisfying the above mentioned criteria and then test these regular expressions against the DOM element locators available in DOM hierarchy.

The overall approach is presented in Algorithm 2. First we split the hierarchical information (Line 1) into separate elements. For example the hierarchical information available at Line 4 in Figure 1 is `#site-navigation h3`. We split this information into two elements: `#site-navigation` and `h3`. Then we create a regular expression (Line 3) for each element in the hierarchy. Regular expressions for the above two elements are presented in Figure 6. As the DOM element locator information from JavaScript code may be incomplete, the regular expressions are conservative to oversee additional information attached to DOM element locators available in DOM. Only the DOM element locators from DOM that satisfy the criteria specified by regular expressions are included in further analysis (Line 4) and rest of the DOM element locators are discarded. For the matching DOM element locators we then remove the matching element and its predecessors from the hierarchy (Line 6). For example, for element `h3`, the generated regular expression (RE2) will match all the elements that have tag name `h3`, but this regular expression will only be tested against the descendants of elements that match the first regular expression RE1. If no DOM element locator from the DOM matches the generated regular expression, the algorithm is terminated (Line 10) and the developer is notified about the possible anomalies in the DOM element locators used in the JavaScript code.

## 4. IMPLEMENTATION

We have implemented our approach in a tool called DOMPLETION<sup>1</sup>. DOMPLETION is itself built using JavaScript, as a code completion plugin for the open source editor IDE, Brackets [6].

For the **DOM Analysis**, we use Brackets Live Development feature [26]. Using Brackets allow us to integrate web application crawling within the development environment. Developers can either crawl the web application manually therefore covering specific DOM states or randomly crawl the web application using the JavaScript crawler available in DOMPLETION.

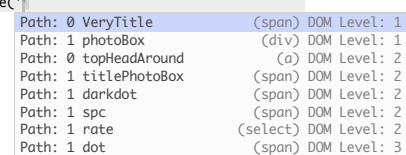
In the **Code Analysis** phase, to parse JavaScript code we use Esprima [10], which is a JavaScript parser written in JavaScript. Esprima converts the given JavaScript into an Abstract Syntax Tree (AST). We use the AST to (1) extract function definitions and code segments, and (2) generate possible code paths. For **dynamic analysis** of JavaScript code we use the `Function` [12] constructor to execute the code segments within the execution environment (dis-

<sup>1</sup><https://github.com/saltlab/ompletion>

```

1 a = document.getElementById('maincol').innerHTML;
2
3 if(a == "header") {
4   elem = document.getElementById('headerBar');
5 } else {
6   elem = document.getElementById('photoBoxes');
7 }
8 elem.getElementsByClassName('

```



Path	Element	DOM Level
Path: 0	VeryTitle (span)	DOM Level: 1
Path: 1	photoBox (div)	DOM Level: 1
Path: 0	topHeadAround (a)	DOM Level: 2
Path: 1	titlePhotoBox (span)	DOM Level: 2
Path: 1	darkdot (span)	DOM Level: 2
Path: 1	spc (span)	DOM Level: 2
Path: 1	rate (select)	DOM Level: 2
Path: 1	dot (span)	DOM Level: 3

**Figure 7: Screenshot of code-completion using DOMPLETION. Path indicates the possible code paths within the JavaScript code. DOM Level indicates the dept of elements in the DOM hierarchy.**

**Table 5: Implemented Functions**

Function	Calling Objects
<code>getElementById()</code>	<code>document</code>
<code>querySelector()</code>	<code>document, element</code>
<code>getElementsByName()</code>	<code>document, element</code>
<code>getElementsByTagName()</code>	<code>document, element</code>
<code>\$( ), jQuery()</code>	-
<code>children()</code>	<code>\$, jQuery</code>

cussed in Section 3.3) as we did not want to interfere with the original application's variables.

Figure 7 shows a screenshot of how DOMPLETION works in practice. The figure shows how DOMPLETION is integrated with the IDE to provide code completion suggestions on demand.

**Supported Functions and DOM element locators.** DOMPLETION currently supports a subset of the functions that are used to fetch DOM elements from JavaScript code. Table 5 list the functions supported by the current version of DOMPLETION. As we show in our empirical evaluation Section 5, this is sufficient for the vast majority of the web applications we studied. Currently, DOMPLETION supports selecting DOM elements based on tag names, ids and class names. Adding support for other functions that use DOM element locators and supporting more DOM element locators is straightforward. We plan to implement these in future versions of DOMPLETION.

## 5. EVALUATION

### 5.1 Goals and Research Questions

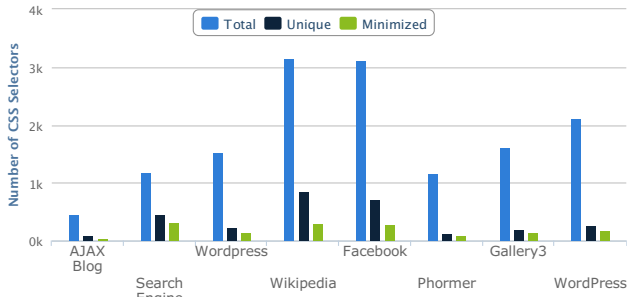
In order to evaluate how effective is DOMPLETION in code completion, we answer the following research questions in our study.

- RQ1:** Do DOM element locators for web applications converge, and if so, what is the convergence rate?
- RQ2:** How accurate are the code completion suggestions provided by DOMPLETION?
- RQ3:** What is the performance overhead incurred by DOMPLETION, and how effective is it in helping web developers with code completion tasks?

### 5.2 Methodology

We used three web applications, namely Phormer [35], Gallery3 [13], and WordPress [40] as case studies to evaluate DOMPLETION. We answer each question as follows:

**RQ1: Convergence.** In addition to the three web applications Phormer, Galery3, and WordPress considered above, we also studied five major websites including three listed in the top 100 websites on Alexa [1]. We crawled the web applications in a random



**Figure 8: Convergence of DOM element locators for different websites.**

order until the number of minimized DOM element locators stabilized. For each web application, we measured the number of total, unique and minimized DOM element locators for each website.

The websites we included were Facebook [11] (User specific content), Wikipedia [38] (User generated content), Bing [5] (Search Engine), a Wordpress Blog and an Ajax-based blog. All the applications chosen for analysis were dynamic in nature, i.e., the DOM tree generated for these websites is either user or input dependent.

**RQ2: Accuracy.** We used the Phormer, Gallery3, and WordPress applications for this RQ. For RQ2, we used DOMPLETION to generate suggestions after removing some DOM access from the JavaScript code. We then compared the list of generated suggestions with the removed code. The list of generated suggestions is considered valid if it contains the DOM access within the code where the code completion was initiated. Prior work [8, 25, 17] has used a similar approach for evaluating code completion systems.

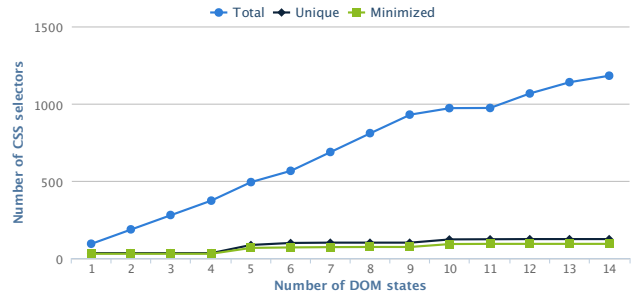
**RQ3: Performance.** For assessing performance, we used the Phormer web application. For measuring the performance overhead, we used DOMPLETION to generate code completion suggestions for Phormer similar to RQ2. We report the time taken by DOMPLETION to generate these suggestions. We also report the time taken by DOMPLETION in the initialization phase i.e., crawling the web application.

**User study.** We also conducted a small scale user study using Phormer. We compared the time taken by users to perform code-completion tasks both with and without DOMPLETION. We also measured the accuracy of the tasks completed by the participants with and without DOMPLETION.

### 5.3 Convergence of element locators (RQ1)

We first measure whether the DOM element locators in web applications converge. We then measure the convergence rate for one web application, Phormer.

Figure 8 presents the results of our analysis for the five real world websites and three selected web applications when they converge (all the websites eventually exhibited convergence). The number of DOM states in a web application is huge [29], making it difficult to cover all the DOM states using a reasonable amount of time and resources. Therefore we focus on number of DOM elements. A detailed example may be found in our companion technical report [3]. As seen from the results, the final converged values of the number of DOM element locators are a small fraction of their initial values, showing that all of the websites exhibit patterns in their DOM element locators. The patterns can be used to predict the structure of DOM states even if they were not encountered during crawling (assuming that the patterns hold). Therefore, a code completion system such as DOMPLETION can detect patterns and provide sug-



**Figure 9: Rate of Convergence of DOM element locators for Phormer.**

gestions even for DOM states that have not been encountered, but are similar to the ones that have been encountered.

We also measure the convergence rate for the Phormer web application as a function of the number of DOM states. The results are shown in Figure 9. The results for the other web applications were similar, and are hence not reported. As seen from the graph, the total number of DOM element locators increase linearly with each DOM state, while the total number of minimized DOM element locators saturates with the growing number of DOM states. Therefore, we achieve convergence within a relatively small number of DOM states. This is because DOM states in web applications exhibit recurring patterns in their DOM element locators, and these patterns tend to converge quickly with increasing number of DOM states, as more of the application is crawled.

### 5.4 Accuracy (RQ2)

We used the Phormer, Gallery3, & WordPress applications for evaluating the accuracy of DOMPLETION. We evaluated the efficacy of DOMPLETION for performing code completion on each DOM JavaScript interaction site in the applications as follows:

1. We remove a DOM element locator used for DOM interaction from the JavaScript code.
2. DOMPLETION is asked to generate a list of valid DOM element locators for the removed selector.
3. Based on the list of generated suggestions, and the corresponding accurate DOM element locator, the evaluation metrics defined in Section 5.4.1 are calculated.

#### 5.4.1 Evaluation Metrics

We use precision and recall to assess the efficacy of our tool. These measures are commonly used for evaluating information retrieval systems.

**Recall** is a measure of the extent to which DOMPLETION can identify relevant DOM element locators that are actually used by the developer. A list of suggestions generated by DOMPLETION may or may not contain the actual DOM element locator used by the developer. The list is considered valid when it contains the actual suggestion used by the developer. *Recall* is defined as the total number of valid suggestions versus the total number of attempts i.e., sum of valid and invalid outputs. In some cases the DOM element locator used within the JavaScript code is not supported by the current implementation of DOMPLETION. We compute the recall both with and without the unsupported selectors.

$$Recall = \frac{Valid\ Output}{Valid\ Output \cup Invalid\ Output \cup Unsupported\ Selectors}$$

**Precision** is a measure of DOMPLETION’s ability to filter out suggestions that are not useful to the developer and present relevant suggestions at the top of the list. One way to perform the filtering is

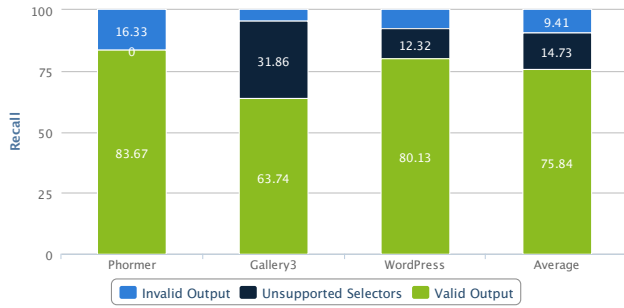


Figure 10: Recall of DOMPLETION

to rank the suggestions and present the ranked list to the developer. We rank our suggestions based on the hierarchy of DOM elements. The root element is listed first, followed by its immediate children, which are in turn followed by the elements at the next level.

The precision is inversely correlated with the rank of the correct suggestion. Therefore we measure precision based on how far a suggestion is in the list of suggestions provided by DOMPLETION, regardless of the number of suggestions provided by the tool. To normalize the evaluation results, we report precision with respect to the rank of top 100 suggestions, instead of the rank of correct suggestion. More precisely, for each increment in the rank of required suggestion, we decrement the precision by 1%. Therefore, precision is 100% when the required suggestion is at the first rank and is 0% when the required suggestions is listed after 100 suggestions. One may think that a more reasonable metric is to use the ratio of the rank of the correct suggestion divided by the total number of suggestions. However, this metric can be misleading. For example, an approach that adds unnecessary suggestions to the list of suggestion can end up having higher precision if precision was measured based on the total number of suggestions in the list. Note that the precision metric discussed above is conservative as we are looking for the exact match used in the JavaScript code. However, there is often more than one possible DOM element that can be used without impacting the correctness.

$$Precision = MAX(0, 100 - Rank_{Actual\ Suggestion})$$

To explain the metrics, we consider the code snippet in Figure 1. Assume that the developer invokes DOMPLETION when passing parameters to `getElementsByTagName()` function at Line 7. The total number of suggestions generated is 5 (unique tag names in Element 8 – 15) and the rank of the suggestion used by the programmer is 4. For this case, the recall would be 100% as the correct suggestion was provided, and precision would be 96% (100-4).

#### 5.4.2 Accuracy Results

Both precision and recall together allow us to assess the quality of the system for a given query. We average the precision and recall values over multiple queries. Figure 10 shows the recall achieved by the system for the three web applications. As seen from the results, DOMPLETION can provide code completion suggestions with a recall of about 89% of the total cases when only supported DOM element locators are considered, and a recall of about 75% overall when both supported and non-supported selectors are considered. Unsupported selectors represent the case when DOM element locators used in the code were not supported by current version of DOMPLETION (Section 4). Note that there is no fundamental reason why DOMPLETION cannot be extended to support these selectors, in which case, the recall will increase even further.

Invalid suggestions mean that DOMPLETION did not provide the correct suggestion among the list of suggestions. There are two

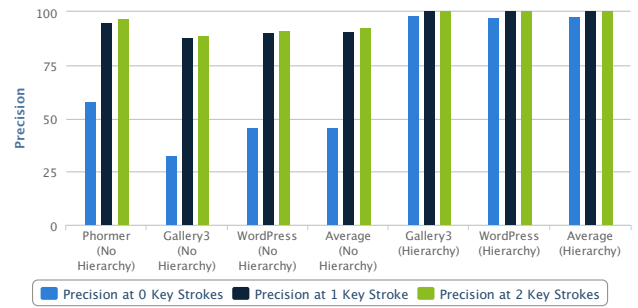


Figure 11: Precision of DOMPLETION

reasons for the invalid selectors: (1) insufficient coverage of DOM states of the website in the initial phase of DOMPLETION, because some modes of the applications required specific user input, which our automated crawler was not equipped with (e.g., logins), and (2) lack of support in DOMPLETION for some corner cases in the JavaScript code such as *alert boxes*.

Figure 11 shows the precision achieved by the system for the three web applications. When using code completion systems, developers typically provide the first one or two keystrokes, and expect the system to automatically infer the rest. These keystrokes can be used to sort and prune the suggestions, resulting in better precision. We report precision for three cases: 1) no input is provided by the developer, 2) one keystroke is provided, and 3) two keystrokes are provided.

Recall that in Section 3.1, we introduce hierarchical information about the DOM into our analysis to prune the search space. To analyze the effect of this optimization, we present results for the precision both with and without the hierarchical information.

**Case 1: No Hierarchical Information.** As seen from the results, DOMPLETION was able to provide precision of about 45%, when no inputs are provided by the developer. However, we see that when one or two keystrokes are provided by the developer, the precision achieved by DOMPLETION quickly rises up to 90%. Therefore, DOMPLETION can provide high precision when the user supplies one or two keystrokes, even without hierarchical information.

**Case 2: Hierarchical Information.** This part of analysis was only performed for two of the above web applications (Gallery3 & WordPress), as the code in Phormer did not utilize hierarchy specific information, and therefore does not benefit from this optimization (and hence its precision does not change). As seen from the results (Figure 11, Hierarchy), when hierarchical information about the DOM is utilized, the precision achieved by DOMPLETION is very high. In fact, DOMPLETION can achieve precision of about 95% even without any input from the developer. When even one or two keystrokes are provided by the developer, DOMPLETION achieved an accuracy of nearly 100%. Therefore, hierarchical information can considerably improve the precision of DOMPLETION.

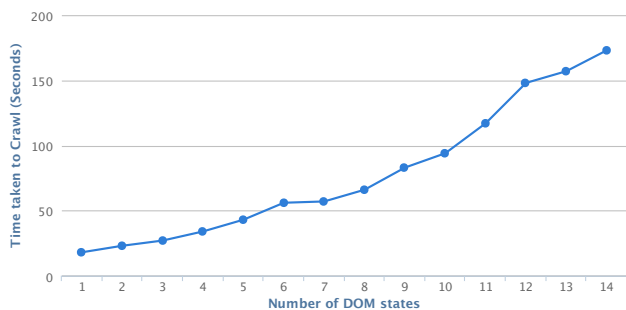
### 5.5 Performance (RQ3)

For RQ3, we focus on the web application Phormer, to measure the performance overhead of DOMPLETION.

The overhead of DOMPLETION consists of two parts, namely (1) time taken to crawl the web application when initializing DOMPLETION, and (2) time taken to generate code completion suggestions when the programmer invokes the code completion functionality.

**DOM Analysis.** When DOMPLETION is initialized, it first needs to gather data about the possible DOM states of the web application





**Figure 12: Time overhead incurred during DOM Analysis**

by crawling it. This data is then used to provide code completion suggestions to the developers. After the initialization, the crawling still continues in the background unnoticed by the developer, and updates the DOM states. The time required is proportional to the number of DOM states needed to reach convergence.

Figure 12 shows the time taken to crawl each DOM state for Phormer, until the state is reached e.g., for DOM state 10, we report the time taken to crawl DOM states 1 to 10. As shown in the table, it takes around 173 seconds (about three minutes) to analyze all 14 DOM states, i.e., crawl until the number of minimized DOM element locators converges to a constant (Section 5.3). Even though the time taken to analyze the DOM states is relatively high, the crawling process recedes in the background, and incurs minimal overhead once the tool is initialized.

**Code Analysis.** The code completion phase starts when the developer activates the tool by pressing the code completion hot key. The time required to analyze the code can vary depending upon the number of conditional statements within the code, size of code that needs to be analyzed and number of functions defined within the code. We calculated the time for all code completion tasks in the JavaScript code for the three web applications. The time required to analyze the code varies from 1 to 6 seconds with an average of about 2.8 seconds, which is quite reasonable for real time code completion. This latency is comparable to that of code completion systems for other languages [30, 16, 28].

## 5.6 User Study

To assess the usefulness of DOMPLETION in helping developers write JavaScript code, we performed a small scale user study on Phormer, with 9 student participants (6 males and 3 females) chosen from different educational levels: 1 undergraduate, 4 Master’s students, and 4 Ph.D. students. Participants were divided in two groups randomly: Group A (5) and Group B (4). Group A and B participants had an average of 1.7 and 2.4 years of web development experience respectively. Participants in Group A were asked to use DOMPLETION to complete the tasks, while participants in Group B were free to use any tool that they were familiar with. One of the Group B participants used Firebug and the other 3 used Chrome Dev Tools to complete the tasks.

Table 6 lists the tasks assigned to the participants in this study. The tasks were based on the web application *Phormer*. For each task, participants were supposed to analyze a piece of JavaScript code from Phormer, and write down the answers. Tasks 1 and 2 focused on understanding the DOM-JavaScript interactions in the code, while tasks 3 and 4 were focussed on code completion for DOM-JavaScript interactions. The JavaScript code provided for each task interacts with and modifies the DOM. Analyzing the changes in DOM is a non-trivial task that requires careful observation by the developer. We measured both the times taken by the

**Table 6: Categorization of tasks for user study**

Category	Task ID	Task
Affected Elements	Task 1	List out ids of DOM elements being removed after executing the code given for Task 1.
	Task 2	List out ids of DOM elements being removed after executing the code given for Task 2.
Available Elements	Task 3	List out the possible ids of elements in the scope of JavaScript object <code>targetElem</code>
	Task 4	List out the possible ids of immediate children of JavaScript object <code>parent</code> .

users and the correctness of the answers for each task. Details of the tasks and questionnaires used for both experiments can be found in our technical report [3].

The results of our user study show that there is a significant difference between the times taken by users in each group to complete the tasks. On average, users in Group B (without DOMPLETION) took almost double the time as compared to users in Group A (with DOMPLETION). Further, the precision and recall achieved by the users in Group A (90.83%, 97.5%) is relatively high as compared to the users in Group B (76.25%, 47.5%). The time taken by users in Group B increased with the number of available code paths. Group B users took maximum time (5 minutes and 31 seconds) for the task (Task 3) with maximum number of code paths (3). On the other hand, the time take by users in Group A was unaffected by the number of code paths. Thus, DOMPLETION significantly lowers the time and improves the accuracy for code completion tasks.

## 6. DISCUSSION

In this section, we discuss some of the limitations of DOMPLETION, and the threats to the validity of our evaluation.

### 6.1 Limitations

**Library support.** DOMPLETION needs to understand library-specific JavaScript APIs that are used for accessing DOM elements in order to provide code completion for them. So we need to wrap library function calls by converting them to DOM element locators. This is straightforward as every DOM access can be written as a DOM element locator. Further, this process needs to be done only once for each library. Currently, DOMPLETION supports calls to jQuery library and we plan to provide wrappers for other major JavaScript libraries in the future.

**Limited support for DOM API.** Currently, DOMPLETION supports a limited set of DOM API functions (Table 1). Functions such as `getElementsByName` and other DOM based functions can also be supported with minor changes in the system. Further, we discard much information about the DOM in the DOM analysis phase. By incorporating that information, we plan to extend our tool to support as many DOM API functions as possible. Note that in spite of the above implementation shortcomings, DOMPLETION was able to offer code completion with high recall for mature web applications such as Phormer, Gallery3 and Wordpress.

### 6.2 Threats to Validity

**Internal Threats.** We evaluated DOMPLETION by crawling the application using a random strategy, rather than an application-specific one. Hence, we may not explore all parts of the application uniformly. This is an internal threat to validity. Adding application specific crawling would increase the number of DOM element locators, which may affect the convergence results in Section 5.3. However, we find patterns in many popular web applications, and

hence we believe that the DOM element locators explored using application-specific crawling would also exhibit similar patterns.

Another internal threat is that `DOMPLETION` modifies the JavaScript code to insert log statements and remove conditional statements from the code. This may affect the behaviour of the application in some cases, especially if the application's behaviour is timing dependent. However, we have taken care to ensure that the modifications are relatively few, and that the execution of code by `DOMPLETION` does not interfere with the main application.

Finally, when crawling the web application, we stop after the number of minimized DOM element locators have stabilized over a period of time. However, it is possible although unlikely, that the locators have not really stabilized, and may increase in the future.

**External Threats.** We considered a limited number of applications and users to evaluate the correctness and usability of `DOMPLETION`. This is an external threat to validity. However the applications chosen for evaluation were commonly used open source applications. We plan to evaluate our tool with a larger number of users in our future work to mitigate this threat.

**Reproducibility.** The websites used for analysis in Section 5.3 may change over time, the number of DOM states and DOM element locators will differ. However, we have made our analysis dataset freely available for download<sup>2</sup> to aid the reproducibility of our results. We have also made `DOMPLETION` publicly available.

## 7. RELATED WORK

We classify related work into two broad categories: code-completion and JavaScript code analysis.

### 7.1 Code Completion

Prior work has focussed on providing code-completion based on different forms of input provided by the developer. For example, Brandt et al. [7] embedded a task specific search engine in IDE that can assist the programmers in finding relevant code on web. Han et al. [16] used machine learning algorithms for completing code based on abbreviations provided by the developer. Little et al. [25] used keywords as inputs to provide code-completion for Java programs. Omar et al. [34] used graphical methods to provide code-completion for parameters of object initialization methods. Sahavechaphan et al. [37] developed a framework that can be used by the developers to query a sample repository for code snippets.

There has also been a significant amount of work on improving the quality of code-completion tools. For example, Hou et al. [17] used hierarchical information to sort list of code-completion suggestions, and context based information to filter out invalid suggestions. Bruch et al. [8] mined existing code repositories and used the results to filter and rank the list of suggestions. Robbes et al. [36] used program history to assess and improve the quality of code-completion tools.

The main difference between these studies and ours is that we focus on providing code-completion for interactions between JavaScript code and the DOM. This is challenging because JavaScript is a loosely typed language, and the DOM is a highly dynamic and evolving entity. Because we dynamically analyze the DOM states in addition to dynamically evaluating the JavaScript code, we can correlate the JavaScript code with the DOM. Further, we use the DOM element locator methods used by the programmer, and the hierarchical DOM information, as inputs to the code completion process. To the best of our knowledge we are the first to provide

<sup>2</sup><http://www.ece.ubc.ca/~kbajaj/dompletion/data.zip>

code-completion for DOM based interactions within the JavaScript code.

### 7.2 JavaScript Code Analysis

There have been numerous techniques for statically analyzing JavaScript code. For example, Jensen et al. [22] presented a static program analysis infrastructure that can infer type information for JavaScript programs. Madsen et al. [27] statically analyze calls to JavaScript libraries and the DOM API to generate the program's call graph. None of these approaches consider the DOM state and hence cannot reason about DOM-JavaScript interactions. Jensen et al. [21] developed a DOM model and used static analysis to reason about DOM-JavaScript within the web application. However, like all static analyses, their approach is conservative, as it needs to take into account all possible static code paths and hence suffers from false-positives. Further, they use their approach for finding errors in applications, but not for code completion, which has a different set of tradeoffs from bug finding (e.g., speed of analysis).

In recent work, Ocariza et al. [33] propose a technique to generate fixes for DOM-JavaScript interaction errors in web applications. Similar to our work, they dynamically analyze the JavaScript code and the DOM states of the web application to generate suggestions for code that has an erroneous DOM-JavaScript interaction. However, there are two main differences between their work and ours. First, they consider complete web applications, where the developer has made a mistake in DOM interactions, while our goal is to provide code completion for DOM interactions in web applications under development. Second, they base their suggestions on an empirical study of common fixes applied by programmers in fixing DOM-JavaScript interaction errors. Consequently, their suggestions are biased towards bug fixes. In contrast, our goal is to help programmers while they are developing the web application, rather than during bug fixing activities.

## 8. CONCLUSION AND FUTURE WORK

In this paper, we introduce a code completion approach for JavaScript based web applications to help programmers with DOM-JavaScript interactions. Our approach is based on dynamic analysis of the DOM and JavaScript code, and provides code completion suggestions based on DOM element locators. We have implemented our approach in a tool called `DOMPLETION`, which we evaluate using three open source web applications. We find that `DOMPLETION` can provide code completion with a recall of 87% and a precision of 90% with an average time of 2.5 seconds.

We plan to extend this paper in number of ways. First we have focussed on sequential JavaScript code. However, JavaScript also support non-sequential execution i.e., executing code based on timeouts, intervals and DOM based events. We will extend our technique to support non-sequential code. Second, the `DOMPLETION` tool supports limited DOM element locators, and libraries. Although `DOMPLETION` achieves acceptable recall with the limited DOM element locators, expanding this set would allow us to deploy `DOMPLETION` on a larger set of web applications.

## 9. ACKNOWLEDGEMENTS

This work was supported in part by a Strategic Project Grant from the Natural Science and Engineering Research Council of Canada (NSERC), a research gift from Intel Corporation, and a MITACS fellowship. We thank the reviewers of ASE'14 for their suggestions to improve the paper.

## 10. REFERENCES

- [1] Alexa top 500 global sites. <http://www.alexa.com/topsites>. Accessed: 2014-04-23.
- [2] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman. Understanding JavaScript event-based interactions. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 367–377. ACM, 2014.
- [3] K. Bajaj, K. Pattabiraman, and A. Mesbah. Dompletion: DOM-Aware JavaScript code completion. Technical Report UBC-SALT-2014-002, University of British Columbia, 2014. <http://salt.ece.ubc.ca/publications/docs/UBC-SALT-2014-002.pdf>.
- [4] K. Bajaj, K. Pattabiraman, and A. Mesbah. Mining questions asked by web developers. In *Proceedings of the 11th ACM Working Conference on Mining Software Repositories*, page 10 pages. ACM, 2014.
- [5] Bing. <http://www.bing.com>. Accessed: 2014-04-23.
- [6] Brackets - the free, open source code editor for the web. <http://brackets.io/>. Accessed: 2014-07-08.
- [7] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric programming: Integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522. ACM, 2010.
- [8] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 213–222. ACM, 2009.
- [9] CSS selectors reference. [http://www.w3schools.com/cssref/css\\_selectors.asp](http://www.w3schools.com/cssref/css_selectors.asp). Accessed: 2014-04-23.
- [10] Esprima. <http://esprima.org>. Accessed: 2014-04-23.
- [11] Facebook. <https://www.facebook.com>. Accessed: 2014-04-23.
- [12] Function - JavaScript | MDN. [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function). Accessed: 2014-04-23.
- [13] Gallery 3 begins | gallery. [http://galleryproject.org/gallery\\_3\\_begins](http://galleryproject.org/gallery_3_begins). Accessed: 2014-04-23.
- [14] Google JavaScript style guide. <https://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml>. Accessed: 2014-17-09.
- [15] B. Hackett and S.-y. Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 239–250. ACM, 2012.
- [16] S. Han, D. Wallace, and R. Miller. Code completion from abbreviated input. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 332–343. ACM, 2009.
- [17] D. Hou and D. Pletcher. An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In *Proceedings of the 27th IEEE International Conference on Software Maintenance, 2011*, pages 233–242. IEEE Computer Society, 2011.
- [18] JavaScript and HTML DOM reference. <http://www.w3schools.com/jsref/>. Accessed: 2014-04-23.
- [19] Javascript coding guidelines. <http://docs.typo3.org/flow/TYP03FlowDocumentation/stable/TheDefinitiveGuide/PartV/CodingGuideLines/JavaScript.html>. Accessed: 2014-17-09.
- [20] JavaScript coding standards | drupal.org. <https://www.drupal.org/node/172169>. Accessed: 2014-17-09.
- [21] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 59–69. ACM, 2011.
- [22] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Static Analysis*, pages 238–255. Springer, 2009.
- [23] V. Kashyap, J. Sarracino, J. Wagner, B. Wiedermann, and B. Hardekopf. Type refinement for static analysis of JavaScript. In *Proceedings of the 9th ACM Symposium on Dynamic Languages*, pages 17–26. ACM, 2013.
- [24] H. Lee, M. Antkiewicz, K. Czarnecki, et al. Towards a generic infrastructure for framework-specific integrated development environment extensions. *Domain-Specific Program Development*, 2, 2008.
- [25] G. Little and R. C. Miller. Keyword programming in Java. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 84–93. ACM, 2007.
- [26] Live development with brackets (experimental); brackets blog. <http://blog.brackets.io/2013/02/08/live-development-with-brackets-experimental/>. Accessed: 2014-07-08.
- [27] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, pages 499–509. ACM, 2013.
- [28] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: Helping to navigate the API jungle. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 48–61. ACM, 2005.
- [29] M. Mirzaaghaei and A. Mesbah. DOM-based test adequacy criteria for web applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 71–81. ACM, 2014.
- [30] M. Mooty, A. Faulring, J. Stylos, and B. Myers. Calcite: Completing code completion for constructors using crowds. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 15–22. IEEE Computer Society, 2010.
- [31] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the eclipse IDE? *IEEE Softw.*, 23(4):76–83, 2006.
- [32] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. An empirical study of client-side JavaScript bugs. In *Proceedings of the ACM / IEEE International Symposium on*

- Empirical Software Engineering and Measurement*, pages 55–64. IEEE Computer Society, 2013.
- [33] F. Ocariza, K. Pattabiraman, and A. Mesbah. Vejovis: Suggesting fixes for JavaScript faults. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 837–847. ACM, 2014.
- [34] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *Proceedings of the International Conference on Software Engineering*, pages 859–869. IEEE Computer Society, 2012.
- [35] Phormer, the PHP without MySQL photogallery manager. <http://p.horm.org/er/>. Accessed: 2014-04-23.
- [36] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 317–326. IEEE Computer Society, 2008.
- [37] N. Sahavechaphan and K. Claypool. XSnippet: Mining for sample code. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 413–430. ACM, 2006.
- [38] Wikipedia. <https://www.wikipedia.org>. Accessed: 2014-04-23.
- [39] Wordpress > JavaScript coding standards « make wordpress core. <http://make.wordpress.org/core/handbook/coding-standards/javascript/>. Accessed: 2014-17-09.
- [40] Wordpress: Blog tool, publishing platform, and CMS. <https://wordpress.org>. Accessed: 2014-04-23.
- [41] WordPress/wp-content/themes/twentytwelve/js/navigation.js at master. <https://github.com/WordPress/WordPress/blob/master/wp-content/themes/twentytwelve/js/navigation.js>. Accessed: 2014-04-23.
- [42] W. Wulf and M. Shaw. Global variable considered harmful. *SIGPLAN Not.*, 8(2):28–34, 1973.