I.J. Information Engineering and Electronic Business, 2015, 2, 38-50
Published Online March 2015 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijieeb.2015.02.06

A Dependency Graph Generation Process for Client-side Web Applications

Tajkia R. Toma
Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh
Email: tajkiatoma@gmail.com

Mohayeminul Islam, Mohammad Shoyaib and Md. Shariful Islam
Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh
Email: mohayeminul.islam@gmail.com, shoyaib@du.ac.bd, shariful@univdhaka.edu

Abstract—The prolific growth of the Internet density has replaced native applications with web based applications. Current trend of web applications is moving towards fat client architecture, which results in a large codebase of the client side of web applications. Manual management of this huge code is tedious and time consuming for developers. We present a technique to construct a dependency graph to provide an overview of the code showing the inter-dependency of the code elements. We conduct a dynamic analysis to make the JavaScript call graph to address the dynamic nature of JavaScript. We further integrate HTML and CSS with the JavaScript call graph to make a dependency graph. Because we can accurately identify the HTML and CSS relations, the result of the dependency graph depends on the JavaScript call graph. Our evaluation of the JavaScript call graph on six web applications demonstrates that the precision is high for the large applications and relatively low for small applications. The recall is low for large applications and relatively higher for small applications.

Index Terms—Web Application, Software Maintenance, Client-side, Dynamic Analysis, Test case, Call Graph, Dependency Graph.

I. Introduction

Mobility and platform independence has revolutionized web applications in recent years. Many popular native applications have been replaced by the web applications that provide the similar services. Structurally, a web application has two processing ends: server side for the data management and business logics implementation, and client side for the presentation of data and user interaction through web browser. In client side, we form the structure of a web page in the web browser through a markup language, define an enchanting presentation by the style sheets and employ a client-side scripting language to attain dynamicity of the page.

The standard markup language used to render a web page is known as Hypertext Markup Language (HTML). HTML provides the basic structure of a web page. In addition to HTML, a Cascading Style Sheet (CSS) is used to provide more sophisticated look and feel. The web page that is built with HTML and CSS is static because we cannot implement logic with HTML and CSS. Different scripting languages are added to them to change the page dynamically and to respond to user interactions. Among the scripting languages, JavaScript is the most popular [1]. JavaScript is supported by most modern web browsers without need of any additional plugin software.

In a web application, client's needs are fulfilled by a number of interactive features provided in client side. When a good number of features implemented in client being independent of server in server-client architecture is called fat-client architecture. This makes the application more responsive and the server more capacitive.

The interactive features in fat-client applications are commonly handled by extensive use of the client side scripting language, JavaScript, along with HTML and CSS. The scripting language processes the user interface (UI) events invoked by user interactions with the features provided in a web page. Thus a fat client application needs massive JavaScript implementation, which handles massive user interactions. This makes a large codebase for a fat-client web application and codebase often become unstructured. The large unstructured codebase of development phase makes it hard to maintain and continue supporting activities in the maintenance phase.

In the software maintenance phase, user's requests for changes in application are mostly based on specific feature [2]. Therefore, developers who are man-aging change requests need the code that implements the specific features. Since the documentation of the application does not provide the implementation detail of the application, therefore, the developers manually browse source code to locate the feature. The developers have to go through several files which is tedious and time-consuming. For a fat-client web application, the developers need to browse mostly the code of client-side implementation. The dynamic nature of JavaScript and the interplay of three different languages in a web application make the manual inspection more complicated. In cases where a developer is new in the development team having no previous knowledge about the system, faces more difficulties. In such situations, an intelligent technique that provides overview of the HTML, CSS and JavaScript implementation of a full application will help the devel

opers.

The main goal of the work is to develop a dependency graph of the HTML, CSS and JavaScript implementations of the client side of a web application. In order to increase efficiency, we distribute the work load in phases of the Software Development Life Cycle (SDLC). This will help developers finding the implementing code of a feature efficiently and change the code as per requirements, which is the starting point of impact analysis [3].

The main contribution of this paper is divided into two folds:

1) A call graph to structure all the JavaScript functions and their relationships using dynamic analysis, and
2) A dependency graph to present the dependency relations among HTML, CSS and JavaScript implementations of a web application.

The JavaScript call graph is an extension of the work in [4]. We have modified the execution trace collection method to overcome their limitations. We evaluate the resultant call graph with the call graph made by manual analysis. It is not possible for the developers to identify all the function calls with manual analysis. The proposed dynamic technique can identify those statically unpredictable functions and their relations. The evaluation of the technique shows that for the small projects the precision values are low but recalls are high. For the large projects the performance has been reversed.

We integrate HTML and CSS with the JavaScript call graph and make a dependency graph to help developers in locating feature of the full web application.

The remaining sections of this paper are organized as follows: Section II provides an overview of the domain with proper motivation of the work. Section III presents the related works. Section IV and V describe the proposed method in detail with application of the proposed methods step by step on a small web application with their result. Section VI presents the evaluation of the proposed method. Section VII concludes the paper with an overview and the future plan.

II. Background

In this section we introduce the concepts and terminologies necessary to understand the feature location process for a client side web application. First we will describe the key terminologies. Then we will describe the motivation with example to make the challenging factors clear.

A. Client-side Web Application: Conceptual and Constructional Model

A web application consists of a number of web pages. A web page has structure which has presentation and behavior. Features of a web page are individually implemented by structures of the web page. A page is provided by the server and is rendered in the client browser. This concept is illustrated in Fig. 1.

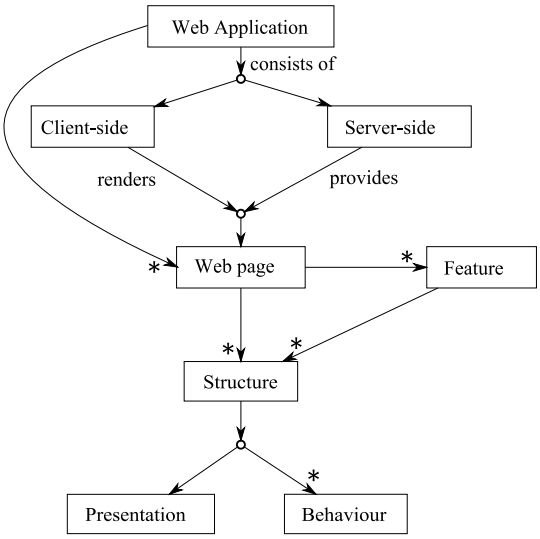

Fig. 1. The conceptual model of a client-side web application

Constructionally a web page is the point of integration of three implementing languages: markup, style sheet and scripts. The style sheet and scripts communicate with the markup through the web page. The most commonly used markup languages for web pages are HTML and Extensible HTML (XHTML). HTML uses tags to specify the structure of the page. For styling a web page CSS is the most used style sheet. It styles the web page using the HTML or XHTML as its base. A web page consisting of HTML and CSS gives control over the structure and presentation but not the behavior. This type of pages is known as static web page. To add dynamicity to a web page we add dynamic code, scripts to the web application. Among the client-side scripting languages, JavaScript is used in 87.9% web applications in respect to other client-side programming languages [1]. This constructional model is illustrated in Fig. 2.

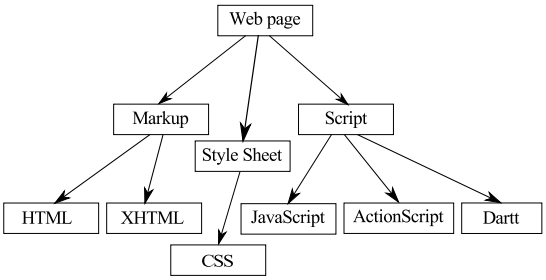

Fig. 2. The constructional model of a client-side web application

A web page consisting of HTML and CSS gives control over the structure and presentation but not the behavior. A web page that always serves the same contents to every user unless the file is manually changed in server side is called a static web page. While a dynamic web page is where the server file is the same but it displays different data depending on information such as the time of day, the user who is logged in, the date, the search term it has been given to look for [5]. To add dynamicity to a web page we add dynamic code, scripts to the web application. Scripts can be written for both server and

client side. A script that is embedded within an HTML file is a client side script. A script is interpreted at runtime.

Among the client-side scripting languages, JavaScript is used in 87.9% web applications in respect to other client-side programming languages [1]. There are also supporting libraries/frameworks for the advance use of JavaScript. Five widely used JavaScript frameworks are: jQuery [6], Modernizr [7], MooTools [8], Prototype [9] and ASP .NET Ajax [10][11]. The Asynchronous JavaScript and XML (Ajax) can send request to the web server and receive data in different formats such as JSON, XML, HTML and even text files without reloading the current page in the browser [12].

B. Key Terminologies

Throughout this paper, we have used some domain specific terminologies. We introduce the terminologies and concepts related to our work in the following subsections.

Feature: According to IEEE [13] the term feature means "A distinguishing characteristic of a system item (includes both functional and nonfunctional attributes such as performance and reusability)". While according to the program understanding community, a specific functionality that is accessible by and visible to the developers as well as users, which is specified from a user requirement, is called a feature [14][15][16]. The definition varies from context to context [14]. In this paper we used the term from both the perspectives except that we exclude the non-functional attributes from the definition of IEEE in our feature list. We only considered the observable behaviors that can be triggered by users or developers.

Scenario: A scenario is a sequence of tasks or user inputs that invokes a feature of an application [17]. A scenario describes a feature from an abstraction level [14]. Scenarios can be of two types: supported scenario and avoided scenario. A supported scenario is a state of an execution of a system that will be in the system, whereas an avoided scenario should not exist in the system [18].

Test case: A test case is the documentation which specifies a combination of test inputs, execution conditions and expected results [19][20], where the expected results are worked out before testing the application for the given test inputs and execution environment. A test case is used to exercise a particular program path or to test the correctness of the behavior of a functionality or feature of an application. The test input should satisfy the pre-condition before the test execution starts and the expected output should satisfy the post-condition after the end of the execution. Test cases should also include the output of unexpected inputs and error cases. Test cases cover the complete code and combine features in many ways, whereas, scenarios invoke all relevant features but as few other features as possible [14].

Execution trace: Execution trace is a record of the sequence of instructions executed during the execution of a computer program. It often takes the form of a list of code labels encountered as the program executes [19]. According to [21], an execution trace is a sequence of events that represent the important moments in the execution of the program. In our system a trace is a set of functions executed sequentially and collected dynamically by executing test cases.

Call graph: Call graph is a diagram that identifies the functions of a computer program and shows which functions call one another [19]. Generally a call graph is a directed graph where the start node of an edge is the caller function and the end node is the callee function. Call graphs can be dynamic or static. A dynamic call graph is an exact record of an execution of the program. Therefore, a dynamic call graph is usually exact. However, it only describes one execution of the program. A static call graph is a call graph that represents every possible execution of the program. Static call graph algorithms are usually over approximations because it is an undecidable problem. That means static call graph may present some relationships which may never occur in reality.

Dependency Graph: A dependency graph is a directed graph where we present the relationship among the elements of a web application. The elements are represented by the nodes of the graph and the edges of the graph picture the relationship among the nodes. An edge from a parent node to a child node expresses the dependency of the parent node on the child node.

C. Motivating Examples

A web application is the interplay of three different types of languages: markup, style sheet and script. Surveys influence us to select HTML as markup language, CSS for style sheet and JavaScript for scripting [1]. The combination and coordination of three different languages at a time makes it difficult to manage. The output of HTML and CSS are more or less manually interpretable, but the execution flow of JavaScript is quite untraceable for its dynamic nature. Also there are very few tool support for the traceability.

In JavaScript, there are several mechanisms whereby executable code can be generated at runtime, (e.g., eval). Static reasoning about dynamically generated code is very difficult [22]. Members of an object can be modified at runtime, even an object can be redefined at any stage of a program's execution.

Listing 3 in Appendix is the JavaScript implementation of a web application which searches for documents cashed in browser memory (DocSearch). In addition to document search there is also a feature that provides suggestion while typing in search box. The example demonstrates some of the key properties of JavaScript. Here we will explain these properties by code snippets from the application.

JavaScript is a weakly typed object-oriented language which uses prototype-based inheritance. The variables of JavaScript are dynamically typed, i.e. can hold values of different types over the course of program execution. This makes the understanding of call checking and field access in run-time. The dynamic typing property has been demonstrated throughout the Listing 3 whenever a variable has been declared using var, e.g., Lines 4, 5 and 6.


```
3       input = input.toLowerCase();
4       var matches = [];
5       for (var i in source) {
6       var src = source[i].toLowerCase();
7       if (matcher(src, input)) {
```

The functions of JavaScript are first class objects. This means, functions are passed as arguments to other functions, returning them as the values from other functions and assigning them to variables [23]. In the following code snippet, at Line 2 there is a function named findMatches and it takes three arguments. The third argument is used at Line 7 as a function which matches a given word in an array of string. While calling the function from Line 24 and 59, they pass two functions with different logics for matching in third argument. Also the function is assigned to a property of helper on Line 13. These demonstrate the first-class object property of functions in JavaScript.

```
1       (function (helper, undefined) {
2        function findMatches(source, input,
          matcher){
3         input = input.toLowerCase();
4         var matches = [];
5         for (var i in source) {
6          var src = source[i].toLowerCase();
7          if (matcher(src, input)) {
8             matches.push(src);
9           }
10        }
11        return matches;
12       }
13       helper.findMatches = findMatches;


24      var matches = helper.findMatches(
         getSuggestionsSet(), input,
25        function (src, inp) {
           return src.substring(0, inp.length)
26          === inp;
        });

59      var results = helper.findMatches(
          getParagraphsSet(), searchString,
          function (src, inp) {
60          return src.indexOf(inp) >= 0;
61      });
```

Objects in JavaScript do not have a fixed set of properties. Properties can be created simply by assigning values at anywhere in the code and can even be deleted [24] [25]. The dynamic property creation for an object is shown at Lines 13 and 51-53 in the above code snippet.

In JavaScript, a function can be defined inside another function. The scope of the inside function is only its parent function. Thus we say the function as a function of local scope. In Lines 2 of the above code snippet and 56 of the code snippet below, functions in local scope have been defined which are not reachable from outside of their parent function.

```
50      }
51      suggestion.showSuggestion = showSuggestion;
52      suggestion.setSuggestion = setSuggestion;
53      suggestion.setSuggestionsVisible =
         setSuggestionsVisible;
54      }(window.suggestion = window.suggestion ||
          {}));
55      (function(search){
56       function showSearchResult() {
57        var searchString = $('#search-box')
             .val();
```

All the dynamic properties of JavaScript make it difficult to understand the full execution path and executed functions of a full application from the source code browsing. The properties of JavaScript hamper the formulation of call graph by manual inspection of a developer. That is why JavaScript is considered separately for the generation of dependency graph.

Our goal is to prepare a call graph using dynamic analysis to have all the execution paths of JavaScript implementation and integrate HTML and CSS with it to make a dependency graph which can be further used for locating a feature in the code. The state of the art on JavaScript analysis provides ideas and solutions for different goals. In the next section we discuss some state of the arts, their solution processes in detail with their advantages and disadvantages.

III. Related Work

Analysis on JavaScript call graph for maintenance phase is a newly evolving field. Recently, Toma et al. [4] proposed a dynamic analysis based JavaScript call graph generation technique for a client side web application. Test cases of testing phase are their source to have the full execution path of a web application. They trace the execution flow running the test cases and from the collected traces they made and update the call graph iteratively. The problem of the work is that the process cannot detect the functions of local scope.

Feldthaus et al. [26] proposed a call graph generation mechanism for JavaScript, based on a scalable field based flow analysis. The contribution of the work is for the support of sophisticated development tools for the development phase. The call graph is generated for the IDE services for developers. The goal of the work demands it to be based on static analysis. The analysis only tracks the flow of function values from a flow graph. For the identification of object flow a field-based approach is employed where the properties of objects are modeled as a global property. The analysis goes further and simply ignores all dynamic property accesses. To make the flow graph they considered two types of flows: Intra-procedural flow and Inter-procedural flow. The authors presented two contrasting approaches for handling inter-procedural flow analysis: pessimistic and optimistic. Both the approaches are scalable and achieve very high recall. The precision value is better for the pessimistic approach than the optimistic approach.

The proposed method in [26] has its own limitations. The static analysis based technique track only the function values and ignores dynamic characteristics of JavaScript. As the analysis is field-based, it cannot distinguish different properties having same name of different objects and considered as one global property.

A points-to analysis or static analysis of JavaScript with the correlation tracking, a novel approach is proposed in [27]. They identify the correlated dynamic property accesses as a common code pattern. A code extraction has been done to analyses on the relevant code. They enhance the Andersens analysis proposed in [28]. The authors of [28] did an implementation of a field-sensitive Andersen-style analysis. The work is not able to complete analysis within a reasonable amount of time and produces very imprecise results. Thus the authors of [27] proposed a correlation tracking technique on top of the Andersens analysis. In a correlation tracking, a dynamic property read r and a property write w are said to be correlated if w writes the value read at r, and both w and r must refer to the same property name. The embedded correlation tracking improves both analysis performance and precision of [28], though the work has some remaining scalability challenges.

Wei et al. [22] proposed an analysis of JavaScript of an application using static analysis. To refine the static analysis a dynamic calling structure collected at runtime. The dynamic analyzer is used for blended points-to analysis to instrument function calls, object allocations and dynamically generated/loaded source code. The dynamic analyzer is designed in a lightweight manner. The authors analyzed multiple executions of a JavaScript code with good program coverage, in order to obtain analysis results for the entire program. The dynamic analyzer optimized by the selection of a cover set from among the executions observed. The author developed a JavaScript engine characterizing the dynamic behavior of JavaScript programs. This builds a graph representation of a JavaScript program. The dynamic analyzer further refines the static analysis using additional information collected by the dynamic analysis.

Maras et al. [17] were first to analyze feature location for client-side web applications in which a dependency graph was made including JavaScript, HTML, CSS and resources used in the application. Their approach has two main phases: Interpretation and Graph marking. The interpretation phase takes web application code, an event trace of the scenario to invoke a feature and a set of UI control selectors as input. For interpretation they used their own interpreter. They interpret the JavaScript code using dynamic slicing and code traversal. For making the dependency graph they store the feature manifestation point where either a structural change occurs or a server-client communication establishes.

In [17], the JavaScript interpretation was made using scenarios where the quality of the result may hamper depending on the provided scenario. The scenario is set up by manual effort. Thus it is depended on the user's understandability about the behavior of the system. The process only works for the functional requirements that can response with a user interaction. Also there is need of large human interaction for the identification of feature manifestation point.

To address the limitations in the state of the art, we propose a mechanism based on dynamic analysis using the test cases of a web application. The dynamic analysis includes all the execution statements and the test cases cover the entire execution path in the application. Also it needs no human knowledge involvement for the making of dependency graph.

IV. Proposed Call Graph Generation Process

According to SDLC, the testing phase comes before the maintenance phase. In the testing phase the tester executes test cases to check for the fulfillment of the user requirements. Even the result of execution providing unexpected input from a user should be included in the test cases. Thus the testing is supposed to cover all the execution paths in the system at least once. While running test cases the tester also needs to trace the execution flow of functions which we call execution trace. A call graph is then made from the execution trace. The call graph can be updated continuously whenever we have an execution trace generated from a test case. A graphical view of the process is shown in Fig. 3.

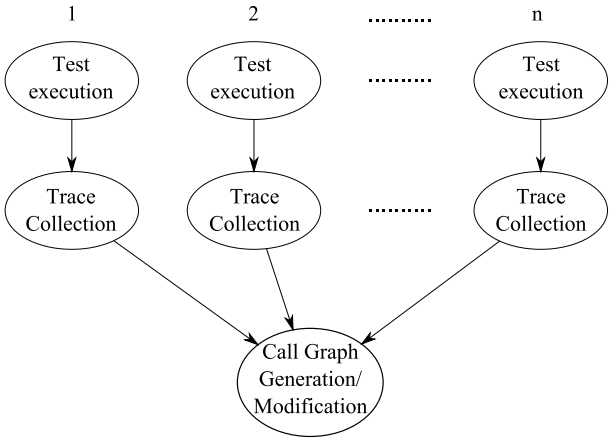

Fig. 3. JavaScript Call graph generation process

In the following sections we will describe each of the steps in detail. The output of the steps is also shown using a sample web application. We will also describe our contribution over the existing process of execution trace collection.

A. Test Execution

Execution of test cases is a part of system testing in the entire testing process and is the only testing process that we are concerned here. A test case consists of a pre-condition, a set of execution conditions and a post-condition. While executing test cases, a tester first tests if the pre-condition is satisfied before execution of the case. The tester proceeds only when the pre-condition is satisfied. Next the tester follows the steps mentioned in the execution conditions. Often there is an expected output for the intermediate steps before we get the final output of the test case. The output of the last step in the execution conditions implies the post-condition of the case. If the output satisfies the clause of the post-condition then the test status is passed for this case, otherwise it has failed. Our concern is not dependent on the status of the

test case, rather, we emphasis only on covering the execution paths.

Example: We use an example application, DocSearch (Appendix), for the examples of the steps of our technique. The implementation code of the DocSearch is added in the Appendix. The application provides two basic features: suggestions while typing in search box, and search. Two test cases for testing the major paths of the two features are shown in Table 1. The first row describes the execution steps that invoke the suggestion feature and second row describes the execution steps that invoke the search feature. The first row is the test case for testing the suggestion functionality. The tester first check whether the JavaScript file is loaded as mentioned in the pre-condition. In the next steps, the first condition is to navigate to the search page where a text box with a search button will appear. Next the tester types a letter, "a" according to the test case, in the search box and thus the set of execution conditions ends. After the execution of the last condition the post-condition is matched with the output of the last condition which states the status of the test case, either pass or fail.

Table 1. Two test cases for the two features of DocSearch (Suggestion and Search Respectively)

Pre-condition: JavaScript file is loaded 1. Navigate to search page: A text box with a "Search" button should appear 2. Type "a" in the text box **Post-condition:** A list of matching words should be shown below in the suggestion box
Pre-condition: The text box is filled with text 1. Click the search button **Post-condition:** Result should be shown in result panel

As the method proposed in this work is a contribution to the maintenance phase, therefore we assume that we already have all the test cases from the testing phase and the source code from the development phase.

B. Trace Collection

The trace collection process starts with the execution of a test case. The process is done as an additional task in the testing phase. A trace collector runs in the background while test cases are in execution. Method is the granularity level for the execution trace. In the previous work we could not identify the functions of local scope. Therefore, we modified the trace collection process to overcome the problem.

We maintain a set of all functions that are already executed in the system. We monitor each of the functions of all namespace in execution while running a test case. When the control flow enters into a function, the caller function's trace is updated with the current function's information. Also the set of executed functions is updated with the current function.

We monitor whenever a function of a namespace executes and collect trace following the steps in Algorithm 1. We store necessary information of the functions, the path of the function, the name of the function and we make an id of the function to identify it uniquely. We also store the name of the HTML elements that it manipulates. This influences the making of the dependency graph in Section V.

We include the monitored function f in a set of already executed functions (Line 8). Next we search for the parent of f in the set of executed functions (Line 9). If the parent exists then we add the current function into the set of called functions of the parent (Line 11). If the parent does not exist then it is called directly from the application which is the parent of all called functions in that execution period. Thus the function needs not to be added to any other functions reference. The functions accessed from a namespace are of public scope. The local functions are not included in namespaces. Thus we cannot trace them only tracking the namespaces. To identify a function of local scope, whenever we trace a function we check the existence of the parent in the global name set. If the parent is not null, but also not exists in the global name set, then the parent is a function of local scope and we add it in the set and mark it as a function of local scope (Line 14). We continue to find the parent of a parent (Line 17) until we find that the parent function is included in the set (Line 13).

Algorithm 1 Execution Trace collection

```
1: function CollectTrace(nameSpace)
2:   for all function f ∈ functions in execution
         and f ∈ nameSpace do
3:     f.filePath ← getfilePath(f)
4:     f.lineNo ← getLineNo(f)
5:     f.name ← getName(f)
6:   f.id ← makeId(f)
7:   f.modifyingDomElements ←
         getModifyingDomElements(f)
8:     push f in functions
9:     fParent ← parentOf(f)
10:    if fParent not NULL then
11:      push f in fParent.calledFunctions
12:    end if
13:    while fParent not NULL & fParent ∉ functions
         do
14:      push fParent in functions
15:      push f in fParent.calledFunctions
16:      f ← fParent
17:      fParent ← parentOf(fParent)
18:    end while
19:  end for
20: end function
```

Complexity: The algorithm iterates a single loop with each function executing in the application. The existence checking of a function in a function list will execute in constant time as we will use hash set as collections. Also there is a while loop inside the for loop. Therefore complexity of the algorithm is $O(f^2)$ where f is the number of executed functions.

Example: We made two test cases for our DocSearch application. Executing the two test cases we get two separate execution traces of the JavaScript part. The first one is listed in Listing 1. The execution trace contains the list of executed functions with the information of a specific function: id, name, location made of filePath and lineNo, and calledFunctions. id is the qualified name of the func

tion to identify a function uniquely by its id and name is the name of the function. Location is made of filePath and lineNo where the function starts in the file. The calledFunctions property contains the name of the functions that are called by the current function. The JSON file of the execution trace is listed in Listing 1.

Listing 1: JSON of the execution trace (executing test case for suggestion feature)

```
1.  {"onfocus@CallGraph/docSearch.html:1": {
2.    "id": "onfo-
    cus@CallGraph/docSearch.html:1",
3.    "name": "onfocus",
4.    "location": "CallGraph/docSearch.html:1",
5.    "calledFunctions": [
6.      "win-
    dow.suggestion.setSuggestionsVisible"
7.    ]
8.  }, "win-
    dow.suggestion.setSuggestionsVisible": {
9.    "id": "window.suggestion.
                    setSuggestionsVisible",
10.   "name": "setSuggestionsVisible",
11.   "location": "CallGraph/docSearch.js:49"
12.   "calledFunctions": [
13.     "jQuery.fn.toggle"
14.   ]
15. }, "onclick@CallGraph/docSearch.html:1": {
16.   "id": "on-
    click@CallGraph/docSearch.html:1",
17.   "name": "onclick",
18.   "location": "CallGraph/docSearch.html:1",
19.   "calledFunctions": [
20.     "window.suggestion.setSuggestion",
21.     "window.bodyClicked"
22.   ]
23. }, "window.suggestion.setSuggestion": {
24.   "id": "window.suggestion.setSuggestion",
25.   "name": "setSuggestion",
26.   "location": "CallGraph/docSearch.js:34"
27.   "calledFunctions": [
28.     "jQuery.fn.val",
29.     "win-
    dow.suggestion.setSuggestionsVisible"
30.   ]
31. }, "window.bodyClicked": {
32.   "id": "window.bodyClicked",
33.   "name": "bodyClicked",
34.   "location": "CallGraph/docSearch.html:38"
35.   "calledFunctions": [
36.     "win-
    dow.suggestion.setSuggestionsVisible"
37.   ]
38. }, "onkeyup@CallGraph/docSearch.html:1": {
39.   "id":
    "onkeyup@CallGraph/docSearch.html:1",
40.   "name": "onkeyup",
41.   "location": "CallGraph/docSearch.html:1",
42.   "calledFunctions": [
43.     "window.suggestion.showSuggestion"
44.   ]
45. }, "window.suggestion.showSuggestion": {
46.   "id": "window.suggestion.showSuggestion",
47.   "name": "showSuggestion",
48.   "location": "CallGraph/docSearch.js:18"
49.   "calledFunctions": [
50.     "jQuery.fn.val",
51.     "window.getSuggestionsSet",
52.     "window.helper.findMatches",
53.     "jQuery.fn.html",
54.     "buildSuggestionsContent",
55.     "win-
    dow.suggestion.setSuggestionsVisible"
56. ]
57. }, "window.getSuggestionsSet": {
58.   "id": "window.getSuggestionsSet",
59.   "name": "getSuggestionsSet",
60.   "calledFunctions": []
61. }, "window.helper.findMatches": {
62.   "id": "window.helper.findMatches",
63.   "name": "findMatches",
64.   "calledFunctions": [
65.     "String.toLowerCase"
66.   ]
67. },"buildSuggestionsContent@CallGraph/
          docSearch.js:40": {
68.   "id": "buildSuggestionsContent@CallGraph/
                  docSearch.js:40",
69.   "name": "buildSuggestionsContent",
70.   "location": "CallGraph/docSearch.js:40",
71.   "calledFunctions": [
72.     "jQuery.fn.attr",
73.         "jQuery.fn.text",
74.         "jQuery.fn.append"
75.   ]
76. }}
```

C. Call Graph Generation

The caller-callee relation in an execution trace is represented with a call graph. The caller function and the callee function nodes are connected by a directed edge from the caller node to the callee node. The call graph can be updated each time an execution trace is gathered or make the graph with all the execution traces together. All the execution traces together make the call graph of the full system.

Algorithm 2 Call Graph Generation

```
1:  function UPDATECALLGRAPH(callGraph, functions)
2:      for all f ∈functions do

3:          fNode ← makeGraphNode(f, callGraph)
4:          for all child ∈ f.calledFunctions do
5:              cNode ← makeGraphNode(child, callGraph)
6:              if edge(fNode, cNode) does not exist then
7:                  createEdge(fNode, cNode)
8:              end if
9:          end for
10:     end for
11: end function
12: function MAKEGRAPHNODE(f, callGraph)
13:     fNode ← createNode(f)
14:     if fNode ∈ callGraph then
15:         fNode ← getGraphNode(fNode)
16:     else
17:         addNode(fNode)
18:     end if
19:     return fNode
20: end function
```

The functions set processed in Algorithm 1 is passed to Algorithm 2 to make the call graph. To update the call graph with the current trace collection first we make a node of a function of the *functions* set (Line 13). If the function already exists from a previous execution trace then we get the node from the graph (Line 15), else we make a new node (Line 17). Following this process we

also make nodes for the children functions of the function and we get the children from the calledFunctions set of a function (Line 4 and 5). We update the relation of the nodes creating edges from parent node to children nodes (Line 7). We repeat the steps for all the functions in the functions set. Thus running the total process for all the functions of all the execution traces we get the full call graph for the full application. The resultant call graph will have multiple components as the fired functions are not fired from a common function.

Complexity: There is one loop (Line 4) inside of another loop (Line 2) in Algorithm 2. We use hash map for storing call graph. Therefore retrieving a node from the graph with the nodes' id can be done in constant time. Thus the complexity of the algorithm is $O(f^2)$ where f is the total number of function in the application.

Example: The call graph made from the JSON file of the execution trace of the suggestion test case (Listing 1) is shown in Fig. 4. We make the call graph evaluating the calledFunctions set of a function. The function from which the functions of calledFunctions are invoked is the parent and the called functions are the children in the graph.

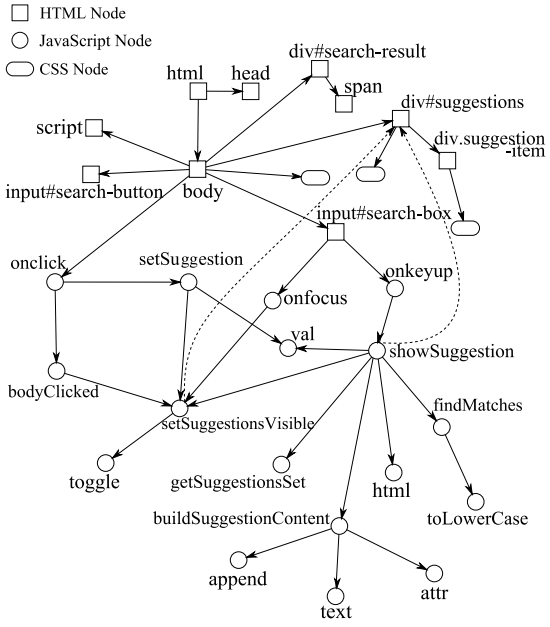

Fig. 4. Call graph after executing trace of suggestion feature

In the next section we elaborate our next contribution, dependency graph generation with detail description. The steps of the process are explained with examples.

V. Proposed Dependency Graph Generation Process

A dependency graph of a web application represents the relations among the HTML elements, CSS properties and bound JavaScript event listeners to an HTML element. Therefore, we introduce two new type of nodes: HTML node and CSS node to represent the HTML and CSS information and relations. Thus we define three different types of structures for the graph to differentiate the three types of nodes. The structure of the nodes is included in Fig. 5.

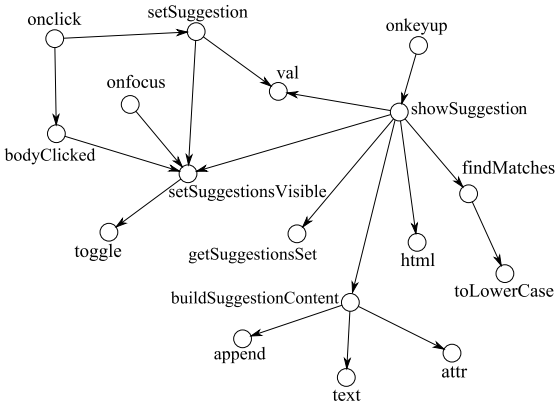

Fig. 5. Dependency graph of the example application

Algorithm 3 includes the steps for making a dependency graph. HTML is inherently a language where the element's implementation maintains a tree structure. Thus we get an HTML tree structure traversing the code and make html node for each node of the tree (Line 1 - 2). Next we traverse through the included style sheets in the document. From the style sheets we traverse through the CSS rules and make CSS node with the relevant information (Line 5). The selector's value of a CSS node gives the information about the HTML node to which the rules are to apply. We create an edge from the selector HTML node, to the CSS node (Line 8).

Algorithm 3 Dependency Graph Generation

```
1: htmlTree ← getHtmlTree()
2: htmlNodes ← createNodes(htmlTree)
3: for all styleSheet ∈ includedStyleSheets do
4:     for all c ∈ cssRules of styleSheet do
5:         n ← createCssNode(c)  ▷ Gives a structure of a
                                   node and return that
6:         selectedHtmlNodes ←
             getSelectedHtmlNodes(htmlNodes, selector of n).
                                 ▷ Get matching HTML
                                 nodes with the selector
7:         for all h ∈ selectedHtmlNodes do
8:             createEdge(h, n) . ▷ From h to n
9:         end for
10:    end for
11: end for
12: for all h ∈ htmlNodes do
13:    eventListeners ← getBoundEventListeners(h)
14:    for all e ∈ eventListeners do
15:        jsNode ← getNode(callGraph, e).
                                 ▷ Find e in callGraph and
                                 return graph node
16:        createEdge(h, e)      ▷ From h to e
17:        for all d ∈ DOM Elements of e do
18:            d ← getHtmlNode(d)
19:            createEdge(e, d)   ▷ From e to d
20:        end for
21:    end for
22: end for
```

Now we integrate the JavaScript nodes with the Dependency graph. We traverse through all the HTML

nodes and get the bound event listeners with each HTML nodes (Line 12 - 13). We use the call graph to have the relations among the JavaScript nodes. We search to the call graph for the JavaScript node (Line 15) and create an edge from the HTML node to the JavaScript node (Line 16). Also we stored the information about the HTML elements that a JavaScript function modifies. Therefore, we create an edge from the JavaScript node to the HTML node (Line 19).

Complexity: We traverse through all the defined CSS rules inside each of the included style sheets which need a loop inside another loop. Inside the inner loop we traverse through all the matching html nodes that are the selector of the CSS rule. Thus the complexity of the algorithm is $O(n^3)$.

Example: Fig. 5 is the dependency graph of the DocSearch application. The graph includes all the HTML and CSS nodes of the application. For the JavaScript nodes, we only added the call graph of the suggestion feature. The "." notation of the HTML nodes represents the class attribute of the node and the "#" notation represents the id attribute of the node.

Though the JavaScript call graph had multiple components in the graph, the dependency graph will have no component. One single root, an HTML node with html tag, will tie all the other HTML, CSS and JavaScript nodes.

VI. Evaluation

Among the three types of nodes in the dependency graph, we can accurately find the HTML and CSS nodes. However, dynamic nature of JavaScript reduces the accuracy of the JavaScript call graph. Therefore, the result of the dependency graph generation process is dependent only on the result of JavaScript call graph generation process. Thus our focus will be on the evaluation of the generated JavaScript call graph.

Dataset: We applied our implementation on six applications. The dataset has been collected from [17] and is located in [29]. The dataset includes only the client-side implementation of the applications. Therefore we excluded the part of the features that needs server-side response from client-side. We also made JavaScript call graph by manual inspection to compare the resultant call graph of our technique with the manual call graph.

In our evaluation we assumed that we already have test cases from the testing phase which covers all system execution paths. However, applications having all test cases are hard to find compared to having only the source code of an application. The dataset we selected includes automated test cases for some selected features they located from the applications. Therefore we prepared test cases to have all the paths to be executed except the execution paths that need server-side communication from client-side. The test cases have been automated using Mozilla Firefox's plugin Selenium IDE version 2.4.0. A Software Engineer having three years of experience of working in JavaScript helped us in making the call graph manually. A Senior Quality Assurance Engineer having three and half years of experience in testing verified the test cases we made.

We set the granularity of the manual analysis to function level. We considered only the custom functions and plugins written for the behavior of the applications to be evaluated and excluded the library function's calling hierarchy. Though our technique can identify the relations of the library functions, we did not consider the library output to compare with the manual result as the making of call graph of library functions by manual inspection is a huge work to keep track within the limited time.

Experimental Setup: We implemented the algorithms in JavaScript for a browser environment. The JavaScript implementation and the browser environment give good access to the elements of a web page. We used Mozilla Firefox version 28.0 to run the applications and the implemented algorithms.

We have implemented the execution trace collection and call graph modification algorithms and applied them to the six applications. We evaluated our implementation in respect of two terms: No of functions and No of edges. We compare the relation resulted by our mechanism in comparison with the actual relation identified by manual analysis.

We evaluated our technique of generating JavaScript call graph by calculating precision and recall. The precision (P) and the recall (R) have been calculated with the following formula:

$$P = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}} \quad (1)$$

$$R = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}} \quad (2)$$

In our case, true positive means true identification, functions those are correctly identified by our method. False positive means false identification, functions identified by our method which actually does not actually execute. False negative means Unidentification, functions those actually executed but our method failed to identify.

Result analysis: The result of the experiment of the proposed techniques, and the precision and the recall after calculating the (1) and (2) respectively are enlisted in Table 2 and 3. In Table 2 the evaluation in respect to the total number of functions in the six applications are listed and arranged in descending order. In 6th and 7th column of the table the result of our technique is shown in term of precision and recall. In Table 3 the evaluation in respect to the total number of edges in the six applications are listed and arranged in descending order. In 6th and 7th column of the table the result of our technique is shown in term of precision and recall.

The result of the JavaScript call graph generation process shows that the precision value for both the functions' identification and edges' identification is high for the large projects (Fig. 6 and 8). With the increase of the number of functions the value decreases. Again the recall is lower for large projects and higher for the small projects (Fig. 7 and 9). For small projects, we can identify all the functions and edges of the application. The result also

includes some functions and edges that do not exist in the system according to our manual inspection of the application. The unidentified functions for large projects are mostly either a function of a local scope or a JavaScript native object. We could not identify the functions of local scope as they have not called any function that is a member function of a namespace. The type of functions that we could not identify is the browser functions, JavaScript

Table 2. Result with respect to number of functions

Projects	No of Functions	True Identified Functions	False Identified Functions	Unidentified Functions	Precision	Recall	Dynamically Identified
mailboxing.com	122	85	0	37	100%	70%	17
makalumedia.com/aerospace	52	47	4	5	92%	90%	6
sipp.cc	35	32	1	3	97%	91%	0
fourandthree.com	14	12	0	2	100%	86%	0
instagalleryapp.com	12	10	3	2	77%	83%	0
idt.mdh.se/pride	10	10	3	0	77%	100%	0

Table 3. Result with respect to number of edges

Projects	No of Edges	True Identified Edges	False Identified Edges	Unidentified Edges	Precision	Recall
mailboxing.com	206	136	9	70	94%	66%
makalumedia.com/aerospace	71	62	6	9	91%	87%
sipp.cc	38	35	9	3	80%	92%
fourandthree.com	11	9	2	2	82%	82%
instagalleryapp.com	10	9	2	1	82%	90%
idt.mdh.se/pride	7	7	2	0	78%	100%

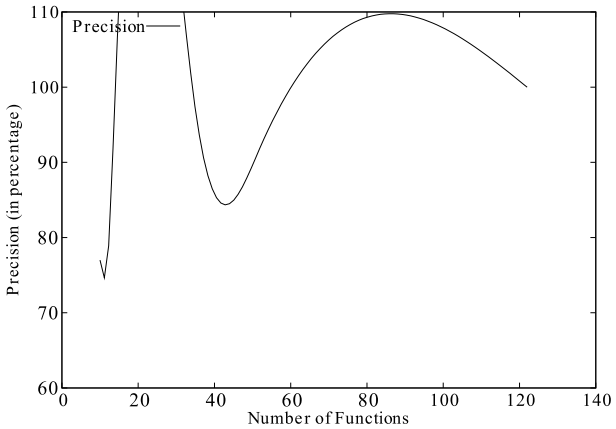

Fig. 6. Precision of the JavaScript call graph generation process in term of functions

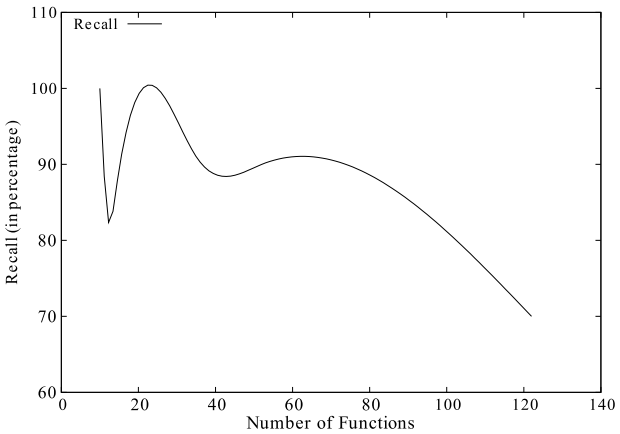

Fig. 7. Recall of the JavaScript call graph generation process in term of functions

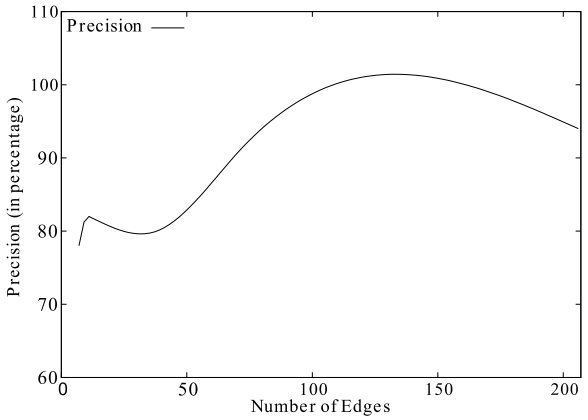

Fig. 8. Precision of the JavaScript call graph generation process in term of edges

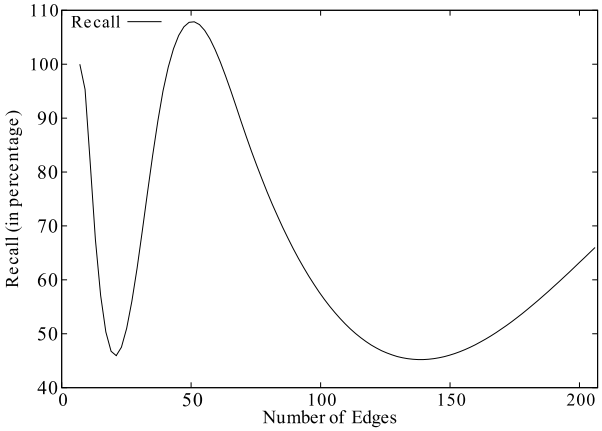

Fig. 9. Recall of the JavaScript call graph generation process in term of edges

native objects. Identifying the native objects is not in our scope.

We claimed in the Section I that some functions are untraceable by manual analysis while making a JavaScript call graph manually and claimed that we can trace them by dynamic analysis. We justified our claim for some applications. These functions are not considered as a false-positive value for the result of our technique.

Discussion: we have performed the evaluation in the web page's environment. We have directly injected the code of our feature location implementation into a web page. Web page environment has some limitations like locating exact line number of a function, CSS rule etc. These could be solved by working from browser's native environment, i.e., developing a browser plugin.

We have conducted dynamic analysis which is not supposed to produce false positive result. However, we failed to get trace of some function calls. Although those functions were called, as we cannot claim it from the experiment, we have added them to false positive result. This has affected overall evaluation of the system.

VII. Conclusion

We have presented a dependency graph generation process for web applications which involves HTML, CSS and JavaScript implementation of the web application. We have also presented a dynamic analysis based JavaScript call graph generation technique and used it further for dependency graph generation. We demonstrated that a call graph can be generated for a highly dynamic language like JavaScript using information gathered in early phases of SDLC.

While evaluating the result of the dependency graph we could accurately find the HTML and CSS nodes. However, dynamic nature of JavaScript reduces the accuracy of the JavaScript call graph. Therefore, we focus on the evaluation of the generated JavaScript call graph. The result of the JavaScript call graph generation process shows that for the small projects we can identify all the functions and edges of the applications including some functions and edges which do not exist in the system according to our manual inspection of the application. In the increase of the number of functions and edges the number of unidentified functions and edges increases and the false identification decreases. We left some functions as untraceable by manual analysis while making a JavaScript call graph manually and claimed that we can trace them by dynamic analysis. We justified our claim for some applications.

The unique contribution of the work is the preprocessed dependency graph in the testing phase for future maintenance, which avoids the need of reverse engineering.

Our current implementation is specific to Firefox browser. We will implement the technique in such a way to be browser independent. The current implementation works directly in webpage environment. A browser plugin would be more suitable and would solve some inherent limitation of webpage environment. We also plan to provide a probable location of a feature in the web application. We will add probability for each node of a dependency graph to measure the degree of relevance of a node to a feature. The result will be a ranked list of nodes. We look forward to developing the feature location plugin for major browsers.

Appendix

Listing 2. html file of example web application (DocSearch)

```
1.  <html>
2.  <head>
3.      <title>Search</title>
4.      <script type="text/javascript"
                        src="jquery-
        2.1.0.js"></script>
5.      <script type="text/javascript"

        src="..\repository.js"></script>
6.      <script type="text/javascript"

        src="docSearch.js"></script>
7.
8.      <style type="text/css">
9.          #suggestions
10.         {
11.             background-color: White;
12.             position: absolute;
13.             border: 1px solid black;
14.         }
15.         body
16.         {
17.             height: 100%;
18.         }
19.        .suggestion-item
20.        {
21.             border-top: 1px solid black;
22.        }
23.     </style>
24. </head>
25. <body onclick="bodyClicked(event)">
26.     <input type="text" id="search-box"
                placeholder="enter text"

        onkeyup="suggestion.showSuggestion()"
                onfocus="suggestion.
                        setSuggestionsVisi-
 ble(true)" />
27.     <input type="button" id="search-button"
                value="Search"
                onclick="search.showSearchResult()"
 />
28.     <div id="suggestions"> </div>
29.     <div id="search-result"> </div>
30.     <script type="text/javascript">
31.       function bodyClicked(event) {
32.         if ($("#search-box")[0]!=event.target){
33.           suggestion.setSuggestionsVisible(false);
34.         }
35.       }
36.     </script>
37. </body>
38. </html>
```

Listing 3. JavaScript implementation of DocSearch

```
1.  (function (helper, undefined) {
2.    function findMatches(source, input, matcher)
    {
3.      input = input.toLowerCase();
4.        var matches = [];
5.        for (var i in source) {
```

```
6.          var src = source[i].toLowerCase();
7.            if (matcher(src, input)) {
8.              matches.push(src);
9.            }
10.         }
11.         return matches;
12.     }
13.     helper.findMatches = findMatches;
14. } (window.helper = window.helper || {}));
15.
16. (function (suggestion, undefined){
17.    var lastInput = undefined;
18.   function showSuggestion() {
19.     var input = $("#search-box").val();
20.     if (input == lastInput) { return; }
21.       var suggestionContent;
22.       if (!input || input.length == 0) {
23.          suggestionContent = "";
24.        } else {
25.         var matches = helper.findMatches(
                  getSuggestionsSet(),input,
                  function (src, inp) {
                return src.substring(0, inp.length)
                   === inp;
26.         });
27.         suggestionContent =
                buildSuggestionsContent(matches);
28.        }
29.        setSuggestionsVisible(true);
30.        $('#suggestions')
                .html(suggestionContent);
31.        lastInput = input;
32.     }
33.  function setSuggestion(sug) {
34.    $('#search-box').val(sug);
35.    setSuggestionsVisible(false);
36.  }
37.  function buildSuggestionsContent(matches) {
38.    var $content = $("<div>");
39.    for (var i in matches) {
40.      var $item = $("<span>").attr("onclick",
                'suggestion.setSuggestion("' +
            matches[i] + '")').text(matches[i]);
41.      $content.append($item).append("</br>");
42.    }
43.    if (!matches.length) {
44.       $content.append("no matches");
45.    }
46.    return $content;
47.  }
48.  function setSuggestionsVisible(visible) {
49.    $('#suggestions').toggle(visible);
50.  }
51.  suggestion.showSuggestion = showSuggestion;
52.  suggestion.setSuggestion = setSuggestion;
53.  suggestion.setSuggestionsVisible =
           setSuggestionsVisible;
54. }(window.suggestion = window.suggestion || {}));
55.
56. (function(search){
57.   function showSearchResult() {
58.     var searchString = $('#search-box').val();
59.     if (!searchString || !searchString.trim())
    {
60.       return;
61.     }
62.     var results = helper.findMatches(
                getParagraphsSet(), searchString,
                function (src, inp) {
63.             return src.indexOf(inp) >= 0;
64.     });
65.     var content =
                      buildSearchResultCon-
 tent(results);
66.     $('#search-result').html(content);
67.   }
68.   function buildSearchResultContent(results) {
69.     var content = "<p>" + results.length +
                       " matches found" + "</p>";
70.     for (var i in results) {
71.       var item = "<div class='suggestion-
    item'>"
                               + results[i] +
 "</div>";
72.         content += item;
73.     }
74.         return content;
75.   }
76.   search.showSearchResult = showSearchResult;
77. }(window.search = window.search || {}));
```


ACKNOWLEDGMENT

This research is funded by fellowship of ICT Division, Ministry of Posts, Telecommunications and Information Technology. This research is conducted in collaboration with IIT DU Optimization Group (https://sites.google.com/site/iitduoptimization/people), our sincere thanks to the group coordinator Mr. Shah Mostafa Khaled, Assistant Professor, Institute of Information Technology, University of Dhaka for his guidance and resources. We also want to thank the software engineers and quality assurance engineer who helped in the evaluation process.

REFERENCES

[1] W[3]Techs, "Usage of Client-side Programming Languages for Websites." http://w3techs.com/technologies/overview/client side language/all, 2014. [Online; accessed 28-February-2014].

[2] V. Rajlich and N. Wilde, "The Role of Concepts in Program Comprehension," in Program Comprehension, 2002. Proceedings. 10th International Workshop on, pp. 271–278, 2002.

[3] V. Rajlich and P. Gosavi, "Incremental Change in Object-Oriented Programming," Software, IEEE, vol. 21, pp. 62–69, July 2004.

[4] T. R. Toma and M. S. Islam, "An Efficient Mechanism of Generating Call Graph for JavaScript Using Dynamic Analysis in Web Application," in 3rd International Conference on Electronics, Informatics & Vision, 2014.

[5] W3C, "How Does the Internet Work." http://www.w3.org/wiki/ How does the Internet work#Static vs. Dynamic Web Sites [Online; accessed 21-April-2014].

[6] "jQuery." http://jquery.com. [Online; accessed 21-April-2014].

[7] "Modernizr." http://modernizr.com/ [Online; accessed 21-April-2014].

[8] "MooTools." http://mootools.net. [Online; accessed 21-April-2014].

[9] "prototype.js." http://prototypejs.org. [Online; accessed 21-April-2014].

[10] "ASP.NET Ajax." http://www.asp.net/ajax [Online; accessed 21-April-2014].

[11] W3Techs, "Usage of JavaScript Libraries for Websites."http:// w3techs.com/technologies/overview/javascript library/all [Online; accessed 21-April-2014].

[12] Mozilla Developer Network, “AJAX; Getting Started.” https:// developer.mozilla.org/en-US/docs/AJAX/Getting Started. [Online; accessed 21-April-2014].
[13] “IEEE Standard for Software and System Test Documentation,” IEEE Std 829-2008, July 18, 2008.
[14] T. Eisenbarth, R. Koschke, and D. Simon, “Locating Features in Source Code,” Software Engineering, IEEE Transactions on, vol. 29, pp. 210–224, March 2003.
[15] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, “Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval,” Software Engineering, IEEE Transactions on, vol. 33, pp. 420–432, June 2007.
[16] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, “Feature Location in Source Code: a Taxonomy and Survey,” Journal of Software: Evolution and Process, vol. 25, no. 1, pp. 53–95, 2013.
[17] J. Maras, M. Stula, J. Carlson, and I. Crnkovic, “Identifying Code of Individual Features in Client-side Web Applications,” Software Engineering, IEEE Transactions on, vol. 39, pp. 1680–1697, Dec 2013.
[18] S. Some, “Use Cases Based Requirements Validation with Scenarios,” in Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on, pp. 465–466, Aug 2005.
[19] “IEEE Standard Glossary of Software Engineering Terminology,” vol. 121990, no. 1, p. 1, 1990.
[20] “IEEE Standard for Software Test Documentation,” IEEE Std 829-1983.
[21] I. Andjelkovic and C. Artho, “Trace Server: A Tool for Storing, Querying and Analyzing Execution Traces,” in JPF Workshop, Lawrence, USA, 2011.
[22] S. Wei and B. G. Ryder, “A Practical Blended Analysis for Dynamic Features in JavaScript,” Technical Report TR-12-11, Computer Science, Virginia Tech, 2012.
[23] H. Abelson and G. J. Sussman. “Structure and Interpretation of Computer Programs,” 1996.
[24] G. Richards, S. Lebresne, B. Burg, and J. Vitek, “An Analysis of the Dynamic Behavior of JavaScript Programs,” in Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’10, (New York, NY, USA), pp. 1–12, ACM, 2010.
[25] S. Jensen, A. Mller, and P. Thiemann, “Type Analysis for JavaScript,” in Static Analysis (J. Palsberg and Z. Su, eds.), vol. 5673 of Lecture Notes in Computer Science, pp. 238–255, Springer Berlin Heidelberg, 2009.
[26] A. Feldthaus, M. Schafer, M. Sridharan, J. Dolby, and F. Tip, “Efficient Construction of Approximate Call Graphs for JavaScript IDE Services,” in Software Engineering (ICSE), 2013 35th International Conference on, pp. 752–761, May 2013.
[27] M. Sridharan, J. Dolby, S. Chandra, M. Schfer, and F. Tip, “Correlation Tracking for Points-to Analysis of JavaScript,” in ECOOP 2012 Object-Oriented Programming (J. Noble, ed.), vol. 7313 of Lecture Notes in Computer Science, pp. 435–458, Springer Berlin Heidelberg, 2012.
[28] “T. J. Watson Libraries for Analysis (WALA).” http://wala.sf.net. [Online; accessed 21-April-2014].
[29] “Index of Josip Maras.” http://marjan.fesb.hr/ jomaras/download/ FIdEvaluation2.zip, 2010. [Online; accessed 20-April-2014].

Authors’ Profiles

Tajkia R. Toma completed her M.Sc. in Software Engineering from Institute of Information Technology, University of Dhaka, Bangladesh in 2014. She received her Bachelor’s degree in Information Technology (Major in Software Engineering) from the same institute in 2012. Currently she is working as a Software Engineer at Jantrik Technologies Ltd. Bangladesh. Her research interest includes code analysis for web applications and software design optimization. She received fellowship from the Ministry of Post, Telecommunications and Information Technology, Bangladesh for her M.Sc. thesis.

Mohayeminul Islam completed M.Sc. in Software Engineering from Institute of Information Technology with thesis "Design Migration from Procedural to Object Oriented Program by Clustering Data Call Graph". He received his Bachelor's degree in Information Technology (Major in Software Engineering) from the same institute. Currently he is working as software engineer at Jantrik Technologies Ltd. Bangladesh. His current research interest is in Software Design Optimization. He received fellowship from the Ministry of Post, Telecommunications and Information Technology, Bangladesh for her M.Sc. thesis.

Mohammad Shoyaib received his M.Sc. degree in computer science from the University of Dhaka, Bangladesh, in 2000 and in 2012 he has completed his PhD degree from the department of the computer Engineering, Kyung Hee University, South Korea. Currently he is a faculty member of Institute of Information Technology, University of Dhaka, Bangladesh. His research interests include pattern recognition and machine learning in different areas of computer vision and image processing. He has also interest in software engineering and bioinformatics.

Md. Shariful Islam received his B.Sc. and M.Sc. degree in Computer Science from the University of Dhaka, Bangladesh, in the year 2000 and 2002, respectively. He completed his M.S degree in Information Technology from the Royal Institute of Technology (KTH), Sweden, in 2005. He obtained his Ph.D degree in Computer Engineering from Kyung Hee University, South Korea in February, 2011. He is now working as an Associate Professor in the Institute of Information Technology (IIT), University of Dhaka, Bangladesh. His current research interests include the design of routing protocols, metrics and MAC protocols for wireless mesh networks. He also worked on security issues related to Wireless AdHoc and Mesh Networks. He has published a good number of research papers in international conferences and journals. He is a member of IEEE and KICS.

