Identifying Code of Individual Features in Client-side Web Applications

Josip Maras, Maja Štula, *Member, IEEE*
Jan Carlson, and Ivica Crnković, *Member, IEEE*

Abstract—Web applications are one of the fastest growing types of software systems today. Structurally, they are composed out of two parts: the server side, used for data-access and business logic, and the client-side used as a user-interface. In recent years, with developers building complex interfaces, the client side is playing an increasingly important role. Unfortunately, the techniques and tools used to support development are not as advanced as in other disciplines. From the user's perspective, the client-side offers a number of features that are relatively easy to distinguish. However, the same cannot be said for their implementation details. This makes code understanding, maintenance, and reuse difficult. The goal of the work presented in this paper is to improve reusability, maintainability and performance of client-side web applications by identifying the code that implements a particular feature. We have evaluated the approach based on three different experiments: extracting features, extracting library functionalities, and page optimization. The evaluation shows that the method is able to identify the implementation details of individual features, and that by extracting the identified code considerable savings in terms of code size and increased performance can be achieved.

Index Terms—Web applications, Feature location, Program Slicing, Code extraction

1 Introduction

The web application domain is one of the fastest growing and most wide-spread application domains. Web applications are used in almost every facet of our lives: at work, as parts of our social experiences, or for e-commerce. From a structural perspective, web applications consist of two equally important parts: the server side, realized as a procedural application implementing data-access and business logic and the client side, realized as an event-driven application that acts as a user-interface (UI). The client-side of a web application is developed with a combination of three languages based on entirely different paradigms: *i)* HTML, a markup language, for defining structure and content; *ii)* CSS, a style sheet language, for presentational aspects, and *iii)* JavaScript, a scripting language, for the behavior. Alongside code, a web application includes resources such as images, videos and fonts. The interplay of these elements produces the result displayed in the browser.

In recent years, the client-side is playing an increasingly important role. By utilizing fast, modern browsers and advanced scripting techniques, developers are building highly interactive applications that can compete with standard desktop applications in terms of user-experience and responsiveness. This has led to the increased complexity of web applications. Unfortunately, techniques and tools used to support development are not as advanced as in some other, more traditional software engineering disciplines. This is especially true in the area of development, analysis, and reuse tools and methods.

From the user's perspective, an application has a number of different features. A feature is implemented by a subset of the whole application's code and resources. However, identifying the exact subset is a challenging task: code responsible for the desired feature is often intermixed with irrelevant code, there is no trivial mapping between the source code and the application displayed in the browser. The ability to exactly identify the code and resources of individual features facilitates reuse, code understanding and maintenance.

The main contribution of this paper is a method for identifying and extracting code and resources that implement individual features in a client-side web application. In order to locate the implementation code, we have to be able to track dependencies between different parts of the application. To address this, we introduce a client-side dependency graph, show how it is constructed, and how it can be used to identify the code and the resources that implement a feature. In addition, we apply the approach in three different cases: extracting features, extracting library functionality, and page optimization. The evaluation of the results shows that the method is able to identify the implementation details of individual features, and that by extracting the identified code considerable savings in terms of code size, and increased performance can be achieved.

J. Maras and M. Štula, Faculty of Electrical Engineering, Mechanical Engineering, and Naval Architecture, University of Split, Croatia.
E-mail: josip.maras@fesb.hr, maja.stula@fesb.hr
J. Carlson and I. Crnković, Mälardalen University, Västerås, Sweden.
E-mail: jan.carlson@mdh.se, ivica.crnkovic@mdh.se
Manuscript received XX, 2012; revised YY,2012.

The paper is organized as follows: Section 2 describes the motivation and the background, and Sections 3 – 5 describe the overall approach to the identification process, define the dependency graph, and show how it is related to each step in the identification process. Section 6 describes the tool that implements the whole process, and Section 7 presents the evaluation. Finally, Sections 8 and 9 present related work and the conclusion.

2 Background

In this section we motivate the usefulness of the approach and present a view on the client-side web application. We also give an introduction to the inner workings of client-side web applications and introduce feature location and program slicing – two techniques fundamental for our approach.

2.1 Motivation

The ability to exactly identify the source code and resources of a particular feature can be used to support a number of software engineering activities, such as program understanding, debugging, feature extraction and page optimization. While program understanding and debugging are important activities regardless of the application domain, feature extraction and page optimization are activities specific in the web application domain.

Fig. 1. Example web application: 1 - the news cycler UI control, 2 - buttons, 3 - image, 4 - captions.

Feature extraction – Consider a web application with a simple news slider – a part of the web application UI (marked with a dashed square in Figure 1) that allows the user to switch between a number of news items, where each item is represented by a large image (marked with 3), a caption and a link to the full news text (marked with 4); and where each change is accompanied by an animation. This is a fairly common feature and can be found in a large number of web applications. The feature manifests when a user clicks on a button (marked with 2 in Figure 1), which initiates a change of news items. The main image, the title and the link are faded out, and when the current elements completely disappear, the source of the image, the text of the title, and the link are changed with the data from the new news item. Next, the elements with the new data are slowly faded in. Once this effect is finished, the button with the index of the previous news item is visually marked as deselected, and the clicked button is visually marked as selected.

Consider a web developer trying to extract the news slider feature from the example application. This feature manifests at runtime when the user clicks on a button with the following changes to the UI: fading out of images, text, and links; changing their data; fading them in; and visually deselecting and selecting buttons. In order to extract the code and resources that implement the feature, the developer has to identify code and resources responsible for each of those changes. This is a difficult and time-consuming task: the developer has to go through several files containing different implementation languages (HTML, JavaScript, CSS) based on different paradigms, and exactly identify the code and resources defining each of the changes.

Page optimization – In client-side web applications, all code is transfered to and executed on the client. Large code bases lead to slower and less responsive web applications, which in turn increases the likelihood of users abandoning the application [17]. By identifying code that implements each application feature, we can identify code that is included in the application, but that does not contribute to any feature (dead code). By removing dead code, considerable savings in terms of page loading time, and increased performance can be achieved. This is especially true for web applications that use wide-spread client-side libraries (around 53% of all web applications [13]), since in order to increase their potential, library developers often include a wide range of features.

2.2 Client-side Web application conceptual model

In this section we present a conceptual model of client-side web applications (Figure 2) that will be used to define a mapping between a feature and the implementing code. A client-side application can be viewed as a collection of visually and behaviorally distinct UI elements (or UI controls). A UI control, even though it does not exist as a separate, standalone, easily identifiable entity in code, is defined with a certain structure, the presentational aspects of that structure, and its behavior.

Features – An application offers a number of features. The meaning of the term *feature* depends on the context. For example, the IEEE defines the term as [26]: a distinguishing characteristic of a system item (includes both functional and nonfunctional attributes such as performance and reusability), while in the

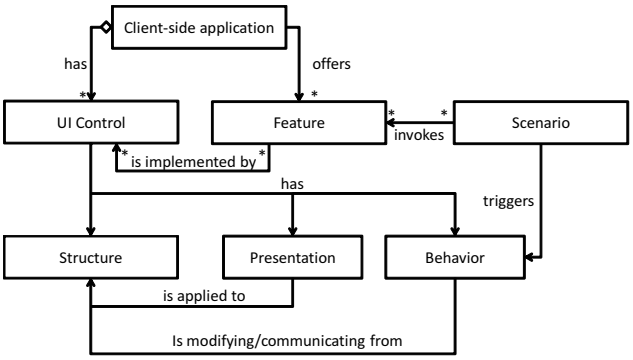

Fig. 2. Client-side web application conceptual model.

program understanding community a software feature is taken to be a specific functionality provided by the software that is accessible by and visible to the developers [21] [14] [24]. In our work we use the term *feature*, in accordance with [14], as an abstract description of a system's expected behavior that manifests itself at run-time, when the user provides the system with adequate input.

Scenarios – Since client-side web applications are UI applications, a user who wants to invoke a certain feature of the system has to provide the application with adequate input – a sequence of user actions. Such sequences of user inputs that trigger actions of a system with observable results are called scenarios [14].

Since client-side applications are UI applications to the server-side, a feature manifests through a number of structural changes on the client-side and/or communications with the server-side. Since a UI control encapsulates structure and the behavior on that structure, we define that a single feature is implemented by at least one UI control (Figure 2). A UI control can, in general, implement any number of features.

Example. In the example application (Section 2), the feature of cycling between news items is implemented by a UI control (Figure 1, mark 1). When the user performs a scenario consisting of a single input: clicking on a button, the feature manifests with a series of changes to the structure of that UI control.

In the next subsections we will go into more details about the inner workings of client-side web applications that are necessary to understand the process of identifying code and resources of individual features. We will also give a short introduction to Feature Location and Program slicing, techniques that are an integral part of our approach.

2.3 Web application primer

A client-side web application is an HTML page that includes JavaScript code, CSS code, and various resources (e.g. images and fonts). The HTML code defines the structure, JavaScript code the behavior, and CSS code the presentation. The interplay of these elements produces the result displayed in the browser.

JavaScript is a weakly typed, imperative, object-oriented scripting language with prototype based inheritance. It has no type declarations and has only run-time checking of calls and field accesses. Functions are first-class objects and can be manipulated and passed around like other objects. JavaScript is a dynamic language: everything can be modified at runtime, from fields and methods of an object to its prototype. The language also offers an eval function which can execute an arbitrary string of JavaScript code.

CSS is a declarative language used to specify the presentational aspects of HTML elements. The CSS code is composed out of CSS rules, each rule consisting of a CSS selector and a set of property-value pairs. A CSS selector is used to specify to which HTML elements the given property-value pairs will be applied.

Client-side web applications are event-driven UI applications and a majority of their code is executed as a response to user-generated events. Their life-cycle can be divided into two phases: *i)* page initialization and *ii)* event-handling. The purpose of the page initialization phase is to build the UI of the web page. The browser achieves this by parsing the HTML code and building a representation of the HTML document – the Document Object Model (DOM). When parsing the HTML code the DOM is constructed one HTML element at a time. There are two special types of HTML elements that the browser can encounter: *i)* style and link elements that include CSS code and *ii)* script elements that include JavaScript code.

When the browser reaches a style or a link element, it parses the CSS code and constructs a set of presentational rules; and when it reaches a script element, the browser suspends the DOM building and enters the JavaScript interpretation. One important purpose of the interpreted code is to register event-handlers, which define how events are handled during the second phase of the execution. Once the code is executed, the process resumes the DOM building phase.

After the last element is parsed and the UI is built, the application enters the event-handling phase, where code is executed in response to events. All UI updates are done by JavaScript modifications of the DOM, which can go as far as completely reshaping the DOM or even modifying the code of the application.

At any time, the client-side application can initiate communication with the server side without interfering with the display of the current page. This approach to client-side development is often termed AJAX (Asynchronous JavaScript and XML; but despite the name the communication does not need to be asynchronous, nor does XML have to be used).

2.4 Feature Location

The goal of our approach is to locate the implementation details of a particular feature. This process is

known as feature (or concept) location [18][19]. The most common types of analyses, used by feature location processes, include textual analysis, static analysis, and dynamic analysis (and their combinations). Textual approaches analyze the source code text based on the idea that identifiers and comments encode domain knowledge, and that a feature may be implemented using a similar set of words throughout the system. Static analysis examines structural information such as control or data flow dependencies, for all possible program inputs, often overestimating the code related to a feature [21]. Dynamic analysis, on the other hand, relies on examining the execution of an application, and it is often used for feature location when features can be invoked and observed during runtime. Feature location using dynamic analysis generally relies on execution trace analysis, and feature-specific scenarios are developed that invoke only the desired feature. Then, the scenarios are exercised and execution traces that record information about the code that was invoked are collected. Dynamic analysis for feature location is often used [14] [20] since most features can be mapped to execution scenarios. However, there are some limitations associated with dynamic analysis – the scenarios used to collect traces may not invoke all of the code that is relevant to the feature, meaning that some of the feature's implementation may not be located [21].

In the client-side domain, features are implemented through the interplay of HTML, CSS, and JavaScript code. This means that, in addition to all difficulties inherent in the feature location process, the client-side feature location also has to take into account the dependencies that exist between different parts of web application code.

2.5 Program Slicing

Program slicing [5] is a method that starting from a subset of a program's behavior, reduces that program to a minimal form which still produces that behavior. A program slice consists of the parts of a program that affect the values computed at a point of interest – the slicing criterion. A program can be sliced statically [5], for all possible program inputs, or dynamically [6], for specific program inputs. A static slicing criterion is usually specified by a program point and a set of variables, while a dynamic slicing criterion is typically composed out of an input, the occurrence of program statement, and a set of variables. Slicing can be a very powerful technique, but it requires starting slicing criteria, and there may be no easy way to identify slicing criteria that correspond to a user-described feature.

In the client-side domain, slicing has certain specifics: *i)* it has to be performed across three different languages (HTML, CSS, and JavaScript); and *ii)* since client-side applications are UI applications, the slicing does not have to be performed only to keep parts of a program that affect the values computed at a point of interest – slicing can also be used to keep only parts of the application's UI.

3 The Identification process

The goal of our approach is to identify code and resources that implement a client-side feature invoked by a scenario. Since a feature is something abstract that manifests at runtime triggered by a scenario, and is implemented by a number of UI controls – in order to identify the implementation of a feature, we have to identify the implementation details of UI controls participating in the scenario.

When executing a scenario, a feature is manifested as a sequence of: *i)* UI modifications to the structure of the implementing UI controls, and/or *ii)* Server-side communications from the structure of the implementing UI controls. These structural changes and server-side communications represent UI control behavior in a particular scenario and we refer to them as *feature manifestations*. A feature manifestation matches an evaluation of a JavaScript expression executed when demonstrating a scenario, an evaluation that either modifies the structure of the page or communicates with the server-side. Feature manifestations, in essence, cause the manifestation of a feature.

One of the key insights that we use in this process is: in order to identify the whole code that implements a feature in a scenario, we have to identify both the feature manifestations and the code responsible for each feature manifestation (in essence, we have to perform dynamic slicing for each feature manifestation). Since a feature manifests when a user performs a certain scenario, feature manifestations can only be known dynamically – we base the approach on the analysis of the execution trace recorded while executing scenarios.

The main advantages of the approach are: *i)* it does not require any formal specification of the feature (something that is rarely done in web application development) – the user can easily specify exact feature behavior; and *ii)* it enables us to dynamically track code dependencies (something that can not be accurately done statically for a language as dynamic as JavaScript). The downsides are: *i)* the approach is primarily suited for functional features with observable behaviors (non-functional features, such as robustness, security, or maintainability do not have determinable feature manifestations) and *ii)* the accuracy and the completeness of the captured feature is dependent on the quality of the scenarios.

Scenarios are an integral part of our approach and in the current process they have to be set up manually by a user. We assume that the user understands the behavior of the target feature (i.e. is aware of different sets of input that trigger feature variation), and knows

which features are invoked by a scenario. This may require knowledge of the internal details of the system. This is in line with the assumptions presented in [14]. However, the scenarios could potentially be generated with a test case generation process (e.g [22], [23], [35]).

In order to identify the implementation of a certain feature, we have to track dependencies between different parts of the application. For this reason, we have defined a *Client-side Dependency Graph*, which is the main artifact used in the process. The overall process is shown in Figure 3 and consists of two phases: *Interpretation* and *Graph Marking*.

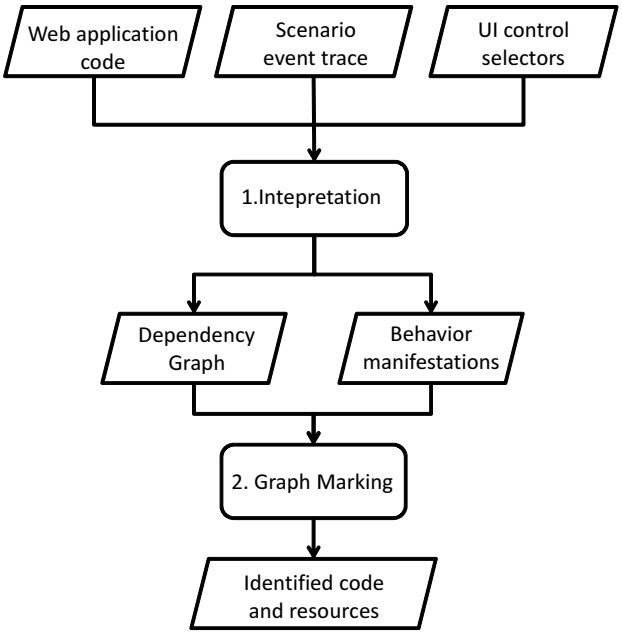

Fig. 3. Identifying code and resources of a feature based on a scenario.

Phase 1 – *Interpretation* – receives as input the whole web application code, an event trace of the scenario that causes the manifestation of the desired feature, and a set of UI control selectors. A UI selector is either a CSS selector[1] or an XPath expression[2] that specifies root HTML elements of the UI controls that implement the feature. The goal of this phase is to build the client-side dependency graph, and the process does that by interpreting the whole application with the event trace as a guideline. During the interpretation phase, as code is being executed, matching nodes with all their dependencies are created and inserted in the graph. When the interpretation process reaches a point in the application execution (i.e. evaluates a code expression) that represents a *feature manifestation*, that point is stored.

In Phase 2 – *Graph marking* – by traversing the dependency graph for every HTML node of every specified UI control and for every feature manifestation, all code and resources that directly or indirectly contribute to the demonstrated feature are marked. In essence, the graph marking phase performs dynamical program slicing with the HTML nodes of the specified UI controls and feature manifestations as slicing criteria. If we merge all computed slices – serialize the marked code and download resources from those marked nodes, we end up with a subset of the original application still able to reproduce the scenario – the implementation of a feature, for this particular scenario, is identified and extracted.

User's perspective

In order to facilitate the feature identification process we have developed a plugin for the Firefox browser that enables the user to select the UI controls in the web application and record the events demonstrated by the user. Next, the plugin determines the UI control selectors of the selected UI controls, accesses the client-side code of the web application and starts the whole process. In the end the user receives the code and resources of the feature demonstrated by the scenario and implemented by the selected UI controls.

Technical perspective

The Firefox plugin used to select UI controls and record user demonstrated events contains a modified JavaScript interpreter[3] that, in addition to evaluating JavaScript expressions, tracks the relationships between expression values and code constructs. The interpreter is executed within the web browser and has access to all browser features (e.g. obtaining DOM element properties, relating CSS rules and matching HTML nodes). The JavaScript interpreter executes code according to standard rules of JavaScript interpretation, which means that in any point of execution, it is aware of the complete application state. This enables us to track data and control dependencies during the evaluation of web application code and precisely determine when an expression that manifests the behavior is evaluated.

3.1 Example

In the following sections we will illustrate the identification process with a running example shown in Listing 1.

This very simple web application has two features: *i)* it enables the user to mark an image as a favorite and sends that decision to the server, and *ii)* displays the message returned from the server (note that they could as well be considered as a single feature, but in this example, for the sake of presentation, we will consider them as separate). Both features are triggered

1. http://www.w3.org/TR/CSS2/selector.html
2. http://www.w3.org/TR/xpath/
3. https://github.com/jomaras/Firecrow

```
/*01*/<html>
/*02*/ <head>
/*03*/   <style>
/*04*/   .fav{background-image: url("fS.png");}
/*05*/   .noFav{background-image: url("nS.png");}
/*06*/   #star { width: 32px; height: 32px;}
/*07*/  </style>
/*08*/ </head>
/*09*/ <body>
/*10*/  <div class="imageRaterContainer">
/*11*/   <img alt="Image" src="atom.png"/><br/>
/*12*/   <div id="star" class="noFav">Note</div>
/*13*/  </div>
/*14*/  <div id="notif"></div>
/*15*/<script>
/*16*/ var star=document.getElementById("star")
/*17*/ var notif=document.getElementById("notif")
/*18*/ star.onclick = function () {
/*19*/  var dec = star.className == "noFav"
      ? "fav" : "noFav";
/*20*/  star.className = dec;
/*21*/  var req = new XMLHttpRequest();
/*22*/  req.open("GET", "d.php?d="+dec, false);
/*23*/  req.send();
/*24*/  notif.textContent = req.responseText;
/*25*/};
/*26*/</script>
/*27*/</body>
/*28*/</html>
```

Listing 1. Example application

by the scenario in which the user, by clicking on the star (div element with the id *"star"*), toggles the image as a favorite. On each click, a request is sent to the server with the information about the state of the star.

The UI of the application is composed out of two containers: the first (*imageRaterContainer*) is used as a container for the image element and the star element and defines the structure related to the first feature; the second (*notif*) is used for displaying status messages returned from the server and defines the structure related to the second feature.

From the application's behavior point of view there are three crucial JavaScript expressions in Listing 1: lines 20, 23, and 24; expressions that directly modify the DOM of the page (lines 20 and 24) or communicate with the server side (line 23). From the feature point of view: expressions in lines 20 and 23 contribute to the behavior of the first feature, and the expression in line 24 to the behavior of the second feature. Our approach is, in essence, dealing with the identification of such feature manifestation expressions, determining whether or not they are important from the perspective of the selected feature, and then performing dynamic program slicing with those expressions as slicing criteria. In order to be able to perform dynamic program slicing, we have to have a way of capturing dependencies. This is usually done with a dependency graph. For this reason, in the next section, we define a client-side dependency graph that is capable of capturing dependencies in a multi-language, multi-paradigm environment that is the client side of the web application.

4 The Dependency Graph

The client-side is composed of four different parts: CSS, HTML, JavaScript, and resources that are intertwined and must be studied as a part of the same whole. Because of this, we define the client-side dependency graph consisting of four types of nodes: HTML nodes, CSS nodes, JavaScript nodes, and resource nodes; and three types of edges: structural dependency edges, data flow edges, and control flow edges. Also, since the client-side of the web application is extremely dynamic (e.g. new HTML elements are regularly created by JavaScript code and inserted into the DOM of the application, but also new JavaScript and CSS code can be dynamically created with JavaScript code), for each node type we also differentiate between static (directly present in the source code) and dynamic nodes (dynamically constructed with JavaScript code).

Dependency	Notation	Condition
Structural	$n_2 \longrightarrow n_1$	n_2 is a child of n_1
Data	$n \dashrightarrow j$	data in n is written by j
Data	$j \dashrightarrow n$	j reads data from n
Data	$h \dashrightarrow c$	the style of h is defined in c
Data	$n \dashrightarrow r$	n reads data from resource r
Control	$j_2 \curvearrowright j_1$	Execution of j_2 depends on j_1

Fig. 4. Edges in the client side dependency graph

Figure 4 shows the definition of different edge types, a straight full arrow represents structural dependencies, a straight dotted arrow data dependencies, and a curved dotted arrow control dependencies; h denotes HTML nodes, j JavaScript nodes, c CSS nodes, r resource nodes, and n denotes a node of arbitrary type. Because of the inherent hierarchical organization of HTML documents, the HTML layout translates very naturally to a graph representation. Except for the top one, each element has exactly one parent element, and can have zero or more child elements. The parent-child relation is the basis for forming dependency edges between h-nodes. A directed structural dependency edge between two h-nodes represents a parent-child relationship from a child to the parent. HTML elements can include different resources (e.g. images, videos, sounds) so there can exist structural dependency edges between an r-node and an h-node.

CSS rules are represented with c-nodes. All CSS

code is contained within an HTML element, so each c-node has a structural dependency towards the parent h-node. Also, since a CSS style can be created with JavaScript code, there can exist a data dependency between a c-node and a j-node. CSS styles often reference resources such as images (e.g. defining the background of an HTML element), so there are data dependencies between c-nodes and r-nodes. Since the main goal of a CSS style is to define styling parameters for HTML elements, there can exist a data dependency between an h-node and a c-node.

JavaScript code constructs that occur in the program are represented with j-nodes (a simplified Abstract Syntax Tree). All JavaScript code is contained in an HTML element, so each j-node has a structural dependency towards the parent h-node. Two j-nodes can also have structural dependencies between themselves denoting that one construct is contained within the other (e.g. a relationship between a function and a statement contained in its body). Data dependency edges can be formed between j-nodes and all other types of nodes: a data dependency from one j-node to another denotes that the former is using the values set in the latter; an edge from a j-node to an h-node, that JavaScript code is reading data, while an edge from the h-node to the j-node means that JavaScript code is writing data to the HTML element. An edge from a j-node to a c-node means that JavaScript code is reading data from CSS code. A j-node can also have a control dependency towards another j-node (e.g. statements in an if-statement towards the if-statement condition).

Example. Figure 5 shows the full dependency graph built while interpreting the web application code presented in Listing 1 based on the execution trace recorded while demonstrating a scenario that will be described in Section 5.1. Circles represent h-nodes, squares c-nodes, trapezoids r-nodes, and rectangles j-nodes. The numbers near each node represent the id of the node; full lines represent structural dependencies, straight dotted lines represent data dependencies, and curved dotted lines represent control dependencies.

In the following section we will describe how the graph is created and traversed to identify the implementation code of a feature in a scenario.

5 Identification process – Detailed description

The process of identifying feature code consists of two phases, and is centered around the client-side dependency graph (Algorithm 1).

As input, Algorithm 1 receives the whole code of the web application, the event trace representing the scenario, and a set of selectors for HTML elements that define the UI controls which implement the feature (CSS selectors or XPath expressions). In the first two lines, two global variables – an empty graph (*dGraph*) and an empty array *fManfs* are initialized. The purpose of the *fManfs* array is to hold the information about all feature manifestation points that were encountered while interpreting the application. The feature manifestation points and the h-nodes that define UI controls (specified with selectors) are the basis for traversing the graph in the graph marking phase (*markGraph*). We start the more detailed description in the next section, by explaining how the event execution trace is obtained.

Algorithm 1 Code Identification

```
1: function IDENTIFYCODE(code, eventTrace, selectors)
2:     global dGraph ← empty graph
3:     global fManfs ← []
4:     interpret (code, eventTrace, selectors)
5:     markGraph(selectors)
6: end function
```

5.1 Obtaining event traces

Due to the dynamicity of client-side applications the process is based on the analysis of the event trace recorded while demonstrating scenarios designed to invoke certain application features. The event-trace specifies the flow of the application, and while the user demonstrates the scenario, all raised events are logged. In general, the event trace captures all relevant information about executed events (e.g. mouse positions, key presses, the values of input elements). From a technical perspective, we obtain the event trace with a plugin to the Firefox browser which communicates with the JavaScript interpreter. Using this information, it is possible to determine the function called as an event handler, the event handling arguments, and the control-flow of the application.

Example. The developer wants to identify code and resources that are responsible for the implementation of the first feature (marking the image as favorite and sending the decision to the server). The developer selects the structure of the feature (*imageRaterContainer*) and demonstrates the following scenario: clicks on the empty star, causing it to change into a full star and send a message to the server. Listing 2 shows the event trace in JSON [12], generated by the specified scenario on the example application from Listing 1. In this case, Listing 2 shows a trace with a single triggered event – a click on an HTML element whose position in the page structure at the time of event handling is defined with an XPath expression (the div element with the id "star"). By using the information about the node the event was raised on (*thisValue*, *originalTarget*, *currentTarget*), the event parameters (e.g. *clientX*, *clientY*), and the starting code line of the event handler function (*line*) it is possible to determine the event handling function, and in turn establish the control-flow of the application.

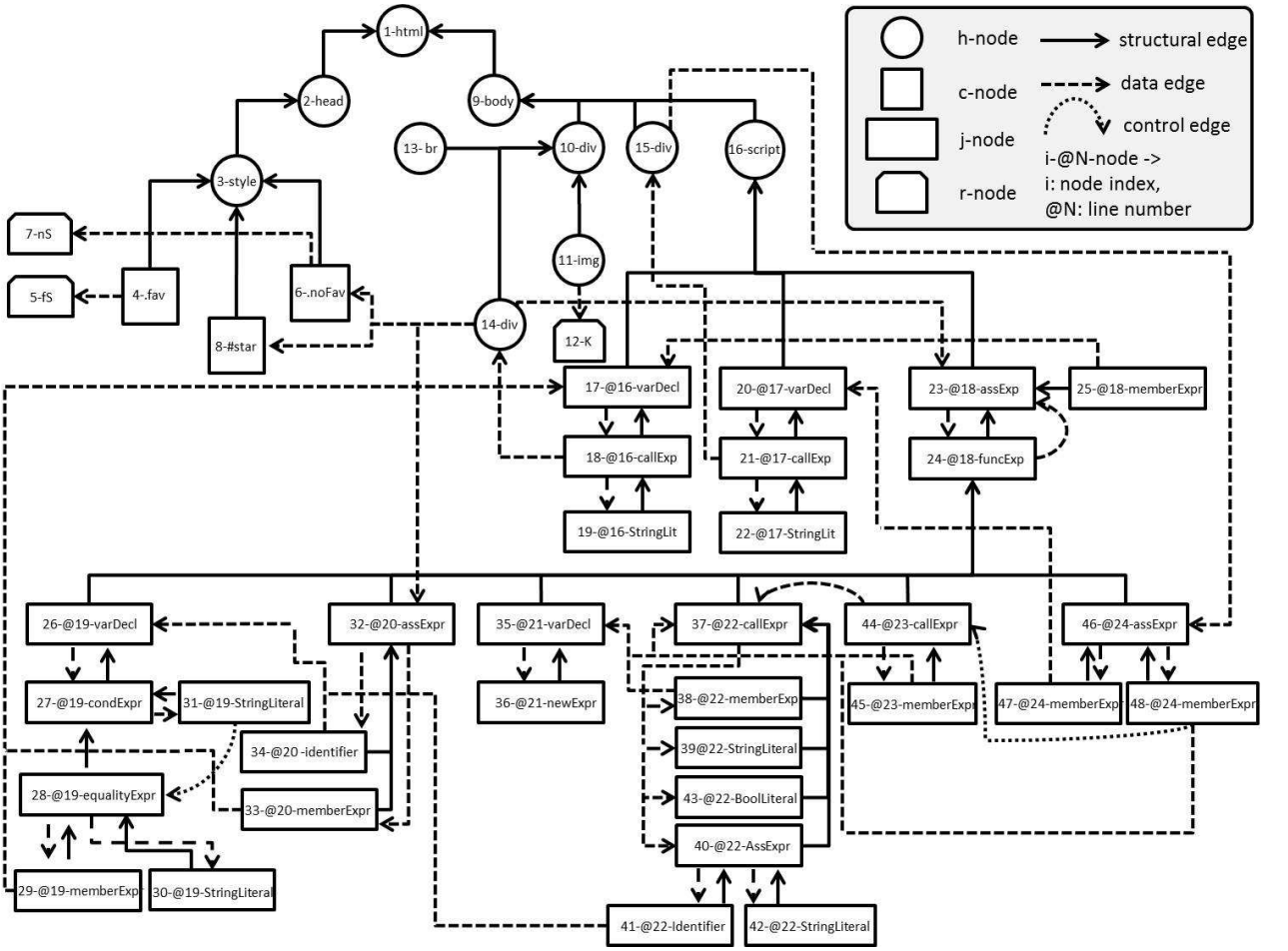

Fig. 5. Dependency graph of the web application from Listing 1.

```
{[{
   "filePath": "example.html",
   "line": 19,
   "currentTime": 1346312751631,
   "thisValue": "/html/body/div/div",
   "args": {
     "target": "/html/body/div/div",
     "originalTarget": "/html/body/div/div",
     "currentTarget": "/html/body/div/div",
     "explicitOriginalTarget": "/html/body/div/
         div",
     "rangeParent": "/html/body/div",
     "clientX": 124, "clientY": 242,
     "screenX": 1490, "screenY": 24,
     "type": "click",
     "inputStates": []
}}]}
```

Listing 2. Example event trace

5.2 Interpretation

After the event trace has been recorded, the process starts the first phase – interpretation (Algorithm 2). As an input, this phase receives the web application code, the recorded event trace, and the selectors for nodes chosen for extraction. From a technical perspective, we have developed a custom JavaScript interpreter based on the process by which the browser executes the web page. Besides evaluating web application code, the interpreter is capable of keeping track of information necessary to establish dependencies between different parts of the application. The interpretation algorithm has two phases: *page initialization* (line 2–4) and *event-handling* (lines 5–7).

Algorithm 2 Interpretation

1: **function** INTERPRET(*code*, *eventTrace*, *selectors*)
2: astNode ← getRoot(getAST(*code*))
3: hNode ← createHNode(*astNode*, *dGraph*)
4: buildSubtree(*hNode*, *astNode*, *selectors*)
5: **for all** *event* in *eventTrace* **do**
6: interpretJs(getHandler(*event*), getNode(*event*), *selectors*)
7: **end for**
8: **end function**

5.2.1 *Initialization phase*

According to Algorithm 2, first the source code of the application is parsed and a simplified AST tree is built (encompassing code from all three languages – HTML, CSS, and JavaScript). Next, a static h-node is created for the root of the AST and added to the dependency graph (lines 2 and 3). The initialization

phase is then continued in a recursive *buildSubtree* function (Algorithm 3).

Subtree building

In the subtree building phase for each encountered HTML child element a corresponding static h-node is created (line 3). If the HTML element is a media element (image, video, etc.), an r-node is created along with the dependency from the h-node to the r-node (lines 6–8). When a CSS HTML element is reached, for each CSS rule a static c-node with a structural dependency to the containing h-node is created (lines 9–17). If a CSS rule references a resource, matching r-nodes are created along with the dependencies from the c-node to the r-nodes (lines 13–16). Every time an h-node is created and inserted into the graph, current CSS rules are checked and data dependencies from the h-nodes to matching c-nodes are created (line 5). If the HTML element is none of the before handled elements, the function is called recursively (line 21).

Algorithm 3 Subtree building

```
1:  function BUILDSUBTREE(hNode, astNode, selectors)
2:    for all astChld in getChildren(astNode) do
3:      hChldNd ← createHNode(astChld)
4:      addSDep(hChldNd, hNode)
5:      traverseCssRulesAndCreateDeps(hNode)
6:      if isMediaElement(astChld) then
7:        rNd ← createRNd(media(astChld))
8:        addDDep(hChldNd, rNd)
9:      else if isCssElement(astChld) then
10:       for all rule in astChld do
11:         cNode ← createCNode(rule)
12:         addSDep(cNode, hChldNd)
13:         if containsResource(rule) then
14:           rNds ← createRNds(media(rule))
15:           addDDep(cNode, rNds)
16:         end if
17:       end for
18:     else if isScriptElement(astChld) then
19:       interpretJs(astChld, hChldNd, selectors)
20:     else/*is any other HTML element*/
21:       buildSubtree(hChldNd, astChld, selectors)
22:     end if
23:   end for
24: end function
```

Complexity. In total, the *buildSubtree* function will be executed once per HTML node. In line 9, Algorithm 3, if the HTML element contains CSS rules, the loop in line 10 is executed once per contained CSS rule. In line 19, the function invokes the *interpretJs* function, whose execution directly depends on the number of execution steps in a particular scenario. In total, the complexity of the algorithm can be approximated as: $O(|h| + |c| + |s|)$, where h is the set of HTML nodes, c a set of CSS rules, and s a sequence of evaluated expressions in a scenario.

Example. Figure 5 shows the graph built while processing the code in Listing 3 according to the scenario from Section 5.1. The algorithm starts by

```
/*01*/<html>
/*02*/ <head>
/*03*/   <style>
/*04*/   .fav{background-image: url("fS.png");}
/*05*/   .noFav{background-image: url("nS.png");}
/*06*/   #star { width: 32px; height: 32px;}
/*07*/  </style>
/*08*/ </head>
/*09*/ <body>
/*10*/  <div class="imageRaterContainer">
/*11*/   <img alt="Image" src="atom.png"/><br/>
/*12*/   <div id="star" class="noFav">Note</div>
/*13*/  </div>
/*14*/  <div id="notif"></div>
/*15*/<script>...
```

Listing 3. An excerpt of the page HTML and CSS code from Listing 1

creating three h-nodes: *1-html*, *2-head*, and *3-style* node, with corresponding structural dependencies (e.g. *2-head→1-html*). Since the style node includes CSS code, the process starts creating c-nodes – three static c-nodes (*4-.fav*, *6-.noFav*, and *8-#star*) are created based on three CSS rules, and each c-node has a static structural dependency towards the parent style node. Nodes *4-.fav*, and *6-.noFav* reference resources, so two r-nodes (*5-fs.png* and *7-nS.png*) are created along with data-dependencies from the c-nodes to the newly created r-nodes. Next, the graph building is continued, and the process creates the h-nodes with indexes 9–11 and 13–16, as well as the necessary structural dependencies (e.g. *9-body* → *1-html*); and one r-node (*12-K.png*). Since the creation of each h-node initiates the search for a matching CSS rule, when the *14-div* node is created, data dependencies from that node to the *8-#star* and *6-.noFav* c-nodes are created (the HTML node matches those two CSS selectors).

Interpreting JavaScript code

When the process encounters an HTML element containing JavaScript code, it switches to the creation of j-nodes (Algorithm 3, line 19), and enters the JavaScript interpretation mode, as shown in Algorithm 4. In this phase j-nodes are created as each JavaScript expression is evaluated for the first time – the "getJNodeAddSDep" function (line 4, Algorithm 4) either returns an existing, or creates a new node along with a static structural dependency to the parent h-node. Next, depending on the position of the evaluated node in the source code, necessary control dependencies are created from the current node (lines 6–12, e.g. an expression in the body of an if statement depends on the condition of the if statement).

If the evaluated expression is handling an event, a control dependency is created towards the node which has performed the event registration (line 14). When exiting an execution context (function or program exit), control dependencies to all executed condition statements (e.g. if-conditions, loop-conditions)

Algorithm 4 Interpreting JavaScript

```
1: function INTERPRETJS(code, hNode, selectors)
2:   programAST ← getAST(code)
3:   while astNd ← getNextNode(programAST) do
4:     jNd ← getJNodeAddSDep(astNd, hNode)
5:     evalRes ← evalute(astNd)
6:     if isInLoopOrBranchStatement(astNd) then
7:       addCDep(jNd, getCondExprNode(evalRes))
8:     else if isInCatchStatement(astNd) then
9:       addCDep(jNd, getErrThrowNode(evalRes))
10:    else if isInFunction(astNd) then
11:      addCDep(jNd, getCallExpNode(evalRes))
12:    end if
13:    if isHandleEvent(evalRes) then
14:      addCDep(jNd, getEvntRegNode(evalRes))
15:    else if isExitingContext(evalRes) then
16:      addCDeps(jNd, getCondsInCntxt(), 'ExitCnt')
17:    end if
18:    if isAccessingIdentifiers(evalRes) then
19:      addDDep(jNd, getLastAssignNds(evalRes))
20:    end if
21:    if isReadingArrayObject(evalRes) then
22:      addDDep(jNd, getAllModifNds(evalRes))
23:    end if
24:    if isCreatingJsCode(evalRes) then
25:      parseAddedCodeCreateASTNodes(evalRes)
26:    else if isCreatingHtmlNodes(evalRes) then
27:      dHNds ← createDHNodes(evalRes)
28:      addDDep(dHNds, jNd)
29:      traverseCssRulesAndCreateDeps(dHNds)
30:    else if isCreatingCssNode(evalRes) then
31:      dCNode ← createDCNode(evalRes)
32:      addDDep(dCNode, jNd)
33:      traverseCssRulesAndCreateDeps()
34:    else if isModifyingDOM(evalRes) then
35:      modifNds ← getModifNodes(evalRes)
36:      addDDep(modifNds, jNd))
37:      traverseCssRulesAndCreateDeps(modifNds)
38:      if matchSelectors(modifNds) then
39:        push(fManfs, point(jNd, lastDIndex(jNd)))
40:      end if
41:    else if isSendingAjaxRequest(evalRes) then
42:      addCDep(jNd, getOpnConnNode(evalRes))
43:      push(fManfs, point(
44:      jNd, lastDIndex(jNd), 'isAjaxSend'))
45:    else if isAccessingAjaxResponse(evalRes) then
46:      addCDep(jNd, getSndRqNode(evalRes))
47:    end if
48:    labelCurrDependencies(getEvalPos(evalRes))
49:  end while
50: end function
```

are also created (line 16). If the evaluated expression reads identifiers (e.g. objects, properties, and functions), then a data dependency from the current node to the node matching the last assignment of the identifier is created (line 19). Currently, we are not slicing arrays, so if the accessed object is an array, data-dependencies from the current node to all nodes that match the last assignments of each array item are also created (line 22). Also, some of the evaluated code expressions (e.g. a call to the createElement method of the document object or an assignment to the innerHTML property of an HTML element) can create dynamic nodes (lines 24–33), and in that case

```
/*15*/<script>
/*16*/ var star=document.getElementById("star")
/*17*/ var notif=document.getElementById("notif")
/*18*/ star.onclick = function () { ... }
```

Listing 4. An excerpt of the JavaScript initialization code from Listing 1

a data-dependency is created from the dynamic node to the currently evaluated JavaScript node. JavaScript code often modifies the DOM of the page, so dynamic structural dependencies between the modified nodes and the currently evaluated j-node are also created (lines 35–40, Algorithm 4).

We consider that a server-side communication is a part of a feature implemented by a UI control if that communication is, in any way, dependent on HTML elements that define the UI control. Since in the interpretation phase the dependencies are not yet followed, each communication is treated as a potential feature manifestation point (line 43, Algorithm 4), but with a flag that marks it as such. Finally, all dependencies created while evaluating an expression (lines 4–47) are labeled with the current evaluation position (to differentiate between dependencies created on different function calls, loop executions, etc.).

Complexity. The number of loop iterations directly depends on the number of evaluated expressions in a scenario – one iteration for each evaluated expression. The complexity of the algorithm is $O(|s|)$, where s is a sequence of evaluated expressions in a scenario.

Example. The processing of code in Listing 4 has reached line 15 and is entering the JavaScript interpretation mode – a variable declaration node *17-varDecl* is created. On the right hand side there is a method call on the document object (which is a special object provided by the browser, acting as an interface to the DOM). The method invocation returns an object mapped to the HTML element with the id "star" (*14-div*), so a data dependency from the call expression to the h-node is created. Also, since the call expression is the initialization part of the variable declaration expression, a data dependency from the variable declaration to the call expression is created (*17-varDecl* → *18-callExp*). Nodes 20 – 22, along with their data dependencies are similarly created. Next, an assignment expression node (*23-assExp*) is created. The right hand side creates a function expression, and the left hand side a member expression. The member expression accesses the object referenced with the identifier "star", and has a data dependency towards the *17-varDecl* variable declaration. The object whose property is being set is an HTML object, and the "onclick" property is a property used to set an event-handler – a data dependency from the matching h-node to this assignment expression is created (*14-div* → *23-assExp*). Since there is no more JavaScript code for sequential execution, the process exits the inter-

pretation mode. Also, all h-nodes have been created and the *page initialization* phase is finished.

5.2.2 Event handling

Once the whole code file has been traversed, and all contained JavaScript code executed in a sequential fashion, the process enters the event-handling phase (Algorithm 2, lines 5–7). Information about each event is read from the execution trace, and the dependency from the j-node matching the event handling function to the j-node matching the event-registering expression is created. Similarly to the initialization phase, JavaScript code is processed using Algorithm 4.

Example. In the demonstrated scenario, the only raised event was the click on the div element representing the star. The process reads the event trace and finds that the function in line 18, Listing 1, was executed as a click event-handler on an HTML element with id "star" (*14-div*). This creates a control dependency from the function expression construct towards the node matching the event-registering construct (*24-funcExp* → *18-assExp*). Next, the function body is interpreted.

```
/*18*/ star.onclick = function () {
/*19*/  var dec = star.className == "noFav"
      ? "fav" : "noFav";
/*20*/  star.className = dec;
/*21*/  var req = new XMLHttpRequest();
/*22*/  req.open("GET", "d.php?d="+dec, false);
/*23*/  req.send();
/*24*/  notif.textContent = req.responseText;
/*25*/};
```

Listing 5. An excerpt of the event-handling JavaScript code from Listing 1

In line 19, Listing 5, a variable declaration node is created (*26-varDecl*). On the right hand side there is a conditional expression, so a conditional expression node is created (*27-condExpr*) along with a data dependency from the variable declaration to the conditional expression node. A conditional expression causes the creation of the equality expression which in turn causes the creation of a member expression node (*star.className*) with a data dependency towards the variable declaration (*29-memberExp* → *17-varDecl*). Since in this scenario the condition evaluates to true, the process only creates j-nodes related to the first condition (*31-StringLiteral*), and since the execution of *31-StringLiteral* depends on the value of *28-equalityExpression*, a control dependency *31-StringLiteral* → *28-equalityExpr* is also created.

The evaluation of the *32-assExpr* causes the creation of *33-memberExp* and *34-identifier*, with data dependencies towards variable declarations (*33-memberExp* → *17-varDecl*, *34-identifier* → *26-varDecl*). Since the *"star"* identifier (*33-memberExpr*) refers to an HTML element (*14-div*), this means that this assignment expression is, by writing to one of the properties (*"class-Name"*) of an HTML element, modifying the DOM (changing the UI). The modified element matches the selector, and the current position (j-node matching the evaluated expression and the last created dependency) is added to the array of feature manifestation points (line 39, Algorithm 4). The process continues similarly to the execution of line 23, Listing 5, where an HttpRequest is sent. According to Algorithm 4, line 43, the currently evaluated node, along with its last dependency will be stored in the array of feature manifestation points as a potential feature manifestation point.

5.3 Graph Marking

So far, in the interpretation phase, we have identified points in the execution where the feature is manifested, but to identify all responsible code, we have to track all direct and indirect dependencies of those feature manifestation points – this part of the process is handled by the graph marking phase.

Algorithm 5 Graph Marking

```
function GRAPHMARKING(selectors)
  exitContextPoints ← empty array
  for all fManf in fManfs do
    mNd ← getNode(fManf)
    d ← getDependency(fManf)
    if isAjax(bhvDep) then
      if depsOnImprtntNd(mNd, d, selectors) then
        markGraph(mNd, d)
      end if
    else
      markGraph(mNd, d)
    end if
  end for
  for all hNode in dGraph do
    if matchesSelector(hNode, selectors) then
      for all d in getDependencies(hNode) do
        markGraph(hNode, d)
      end for
    end if
  end for
  for all exitContextPoint in exitContextPoints do
    dep ← getDependency(exitContextPoint)
    if isIncluded(getTargetNode(dep)) then
      markGraph(getNode(dep), dep)
    end if
  end for
end function

function MARKGRAPH(node, dep, exitContextPoints)
  markAsIncluded(node)
  for all curDep in getPriorDepends(node, dep) do
    if isExitContextDep(curDep) then
      add(exitContextPoints, point(node, curDep))
    else
      markAsTraversed(curDep)
      markGraph(getTargetNode(curDep), curDep)
    end if
  end for
end function
```

As is described in Algorithm 5, the dependency graph is traversed for all feature manifestation points

(lines 3–13), and for all h-nodes (lines 14–20) that match the input selectors. The *markGraph* function describes the process of traversing the graph in order to mark all code nodes that influence the feature manifestation points. The key point in the algorithm is the selection of the dependencies that will be followed (*getPriorDepends* function). In the interpretation phase, all dependencies have been labeled with evaluation position ids, and the *getPriorDepends* selects all previous non-traversed dependencies according to the evaluation position. Notice that any encountered *'ExitContext'* dependencies are stored in a separate array, and are traversed at the end of the whole process only if at least one child expression of the target's parent node is included. In order to better explain the rationale behind this decision we present the code in Listing 6.

```
/*01*/var a = {
/*02*/  b:4,
/*03*/  calcSum: function() {
/*04*/    this.num = 0;
/*05*/    if(this.b) { this.num += this.b; }
/*06*/  }
/*07*/};
/*08*/a.calcSum();
/*09*/a1 = a.num;
/*10*/a.b = null;
/*11*/a.calcSum();
/*12*/a2 = a.num;
```

Listing 6. Example for merging results for multiple traversals

For code shown in Listing 6, the values of variables *a1* and *a2* are 4 and 0 respectively. If we run the identification processes separately, the results would be that the necessary code expressions for *a1* are contained in lines 1–9, and for *a2* in lines 1, 3, 4, 11–12. By executing only the necessary lines, the values of *a1* and *a2* would still be 4 and 0. If we wanted to identify the code that affects both *a1* and *a2*, the straight forward way would be to join the code expressions responsible for the value of *a1* and *a2* (the result is the whole code except for *assExpr@10*). However, this is not correct – in that case the value of *a2* would be 4 instead of 0 (the value of *b* property of object *a*, even thought it does not directly contribute to the value of a2, influences the control-flow of the *calcSum* function called from *callExp@11*). For this reason on each context exit (exiting functions or programs) we create dependencies between the calling construct and every condition that was executed in that context (line 16, Algorithm 4: there exist *'ExitContext'* control dependencies *callExpr@8* → *ifCond@5*, and *callExpr@11* → *ifCond@5*). Now, by repeating the identification process, for *a1* we again get expressions in lines 1–9, but for *a2* we get expressions in lines 1–5 and 10–12, and joining these expressions gives the correct result. However, notice that if we are only interested in *a2* we would end up with more code than is actually needed (expressions in lines 2, 5, and 10 are not necessary for the standalone value of *a2*). Because of this, as the marking algorithm is executed, all *'ExitContext'* dependencies are stored and traversed at the end of the whole process only if another graph traversal has already included the condition statement (lines 23–25, Algorithm 5).

Complexity. Let $G = \langle n, e \rangle$ be a dependency graph built in the interpretation phase; where n is a set of nodes and e a set of edges; and let s be a sequence of evaluated expressions in a scenario. The execution of the algorithm depends on the three for loops. For the first loop, the length of *fManfs* is upper bound by $|s|$ – there can not be more feature manifestations than evaluated expressions; and every execution of the *depsOnImprtntNd* function can at most go through the whole graph (every edge can be traversed at most once) – the number of executions is upper bound by $|e|$. The second loop is executed for each HTML node in every specified UI control, so the number of iterations is upper bound by $|n|$. The third loop is executed for each *exitContextPoint*, and since *exitContextPoints* are created on each context exit, their number is always bound by $|s|$. The *markGraph* function, which is called in each loop, can at most (across all invocations) visit all edges of the graph – has an upper bound of $|e|$. So the upper bound of the graph marking algorithm is: $O(|s||e|^2 + |n||e|)$. However, it is important to note that, in a typical case, the number of feature manifestation and exitContextPoints is, even though upper bound by $|s|$, much less than $|s|$.

Example. For the example in Listing 1, two feature manifestation points were identified: one for the execution of *32-assExpr* that modifies the part of the UI, and one for *44-callExpr*, labeled as an ajax behavior point. First, the *32-assExpr* is marked as important, and its dependencies traversed: the *33-memberExpr* with a dependency to *17-varDecl* is also marked as important, along with the node matching the initialization call expression (*document.getElementById('star')* – *18-callExpr*) where the current value of the identifier was set. Since the call expression is dependent on the *14-div*, it is also marked as important. This also causes the marking of h-nodes: *10-div*, *9-body*, *1-body* due to structural dependencies, c-nodes: *6-.noFav*, *8-#star* (which causes the marking of *3-style* and *2-head*), and r-nodes: *5-fS.png*, *7-ns.png*. Since *14-div* is also dependent on *23-assExpr@* all of its dependencies are also included. Similarly, for the node matching the right hand side of the assignment expression in line 20 (*34-identifier*), all dependencies are also traversed and marked.

Next, the second feature manifestation point is processed. Since it is an *'AjaxSend'* feature manifestation point, first its dependencies are followed in order to determine if it is in any way dependent on any important part of the UI. Since it is indirectly dependent on *14-div*, the graph is traversed, and all expressions

```
/*01*/<html>
/*02*/ <head>
/*03*/   <style>
/*04*/   .fav{background-image: url("fS.png");}
/*05*/   .noFav{background-image: url("nS.png");}
/*06*/   #star { width: 32px; height: 32px;}
/*07*/  </style>
/*08*/ </head>
/*09*/ <body>
/*10*/  <div class="imageRaterContainer">
/*11*/   <img alt="Image" src="atom.png"/><br/>
/*12*/   <div id="star" class="noFav">Note</div>
/*13*/  </div>
/*14*/
/*15*/<script>
/*16*/ var star=document.getElementById("star");
/*17*/
/*18*/ star.onclick = function () {
/*19*/   var dec = star.className == "noFav"
        ? "fav" : null;
/*20*/   star.className = dec;
/*21*/   var req = new XMLHttpRequest();
/*22*/   req.open("GET", "d.php?d="+dec, false);
/*23*/   req.send();
/*24*/
/*25*/};
/*26*/</script>
/*27*/</body>
/*28*/</html>
```

Listing 7. Example application extracted code

in lines 21–23 are marked as included. The algorithm then goes through all h-nodes that define the selected UI controls and traverses their dependencies. In this example, this does not include any more expressions, since everything important was already included in previous traversals.

With this, the process has identified all code expressions responsible for the implementation details of a feature demonstrated by the scenario. In essence, the process has identified that the first behavior (selecting the image as favorite) is mapped to the execution of the assignment expression in line 20, and the second behavior (sending the decision to the server) is mapped to the call expression in line 23. By traversing the dependencies of those two expressions the process identifies code responsible for the whole feature. By generating code from the nodes marked as important we get the code shown in Listing 7.

Compared to the code from Listing 1, the identification process has identified the HTML element defined in line 14, the variable declaration in line 17, and the assignment expression in line 24 as code that is not necessary for the target feature. Notice how the second expression in the conditional expression in line 19 ("noFav") is replaced by null, simply because it was not executed in the demonstrated scenario. In order to remedy this, the developer would have to change the scenario by demonstrating another click on the star element (executing the second expression).

6 Tool Support

The whole process is currently supported by the Firecrow tool[4], which is an extension for the Firebug[5] web debugger. Apart from implementing the algorithms described in this paper, the tool also supports code extraction, simple reuse, and page optimization. Currently, the tool can be used from the Firefox web browser, but it can be ported to any browser that provides communication with a JavaScript debugger. Only the event trace recording is browser dependent; dependency graph construction and feature code identification are functionalities that are provided by a custom-made JavaScript library that can be used from any browser on any operating system. Figure 6 shows the UI of the Firecrow tool.

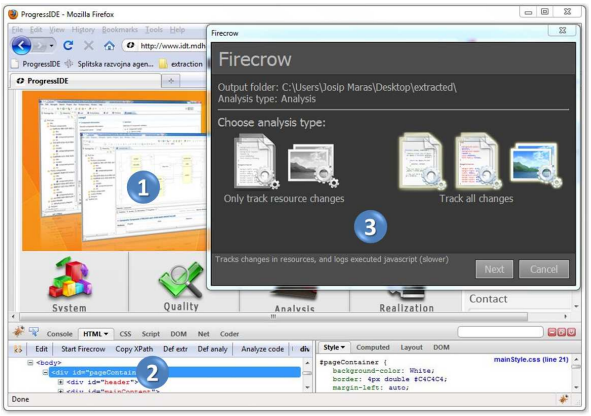

Fig. 6. Firecrow tool UI – Mark 1 shows the analyzed page, Mark 2 – the panel for selecting UI controls, and Mark 3 – Firecrow configuration window

7 Evaluation

We have performed the evaluation with two goals: *i)* to show that the process is able to identify code that implements a feature which manifests when a user demonstrates a scenario, and *ii)* to establish the gains that can be achieved by using our method when compared to a baseline obtained by profiling code. We consider the identification process successful if by extracting the identified code into a stand-alone web application we get the same functional and visual results when executing the scenario in the context of the original application, as in the context of the application composed out of the extracted code. In order to establish the gains that can be achieved with our method, we compare the lines of code (LOC) and the number of execution steps (EXE) of the extracted code (E-LOC, E-EXE) with the profiled code (P-LOC, P-EXE) which serves as a baseline. Profiling is a straightforward extraction approach – the idea is to

4. https://github.com/jomaras/Firecrow
5. http://getfirebug.com

keep all lines executed in a scenario, while maintaining syntactical correctness. The code extracted in this way is still capable of replicating the scenario. The difference in LOC represents code size gains, while the difference in EXEs represents performance gains – the same behavior is realized with less execution steps (program counter increments).

We have performed three sets of experiments: *i)* extracting client-side features, *ii)* extracting library functionalities, and *iii)* page optimization. In all cases, scenarios are defined as tests, and we consider the process successful if the tests can be successfully executed both in the original application, and in the new application composed out of the extracted code.

In addition to the metrics based on LOC and EXEs, we also present the analysis time (the time it takes to interpret the application, build the dependency graph, and generate the extracted code[6]). In the page-optimization subsection, we also include page loading-time (LT) and extracted page loading-time (E-LT), which show the gains in the page loading time when removing code that was not necessary from the application's point of view.

The data presented in this evaluation was gathered by a tool developed as a Firefox extension – all tracing data are Firefox specific, and results could be different in another browser. All applications, and the tests describing the scenarios can be downloaded from www.fesb.hr/~jomaras/download/FIdEvaluation.zip.

7.1 Extracting Features

In the first experiment – extracting features, our goal was to show that the process is capable of extracting features that are implemented with HTML, CSS, and JavaScript code, features that manifest with UI modifications of UI controls. For each test application, we have manually identified its features, the implementing UI controls of those features, and have specified the scenarios that capture the feature behavior. Each scenario is represented as a Selenium[7] test. Selenium IDE is a plugin for the Firefox browser which enables automated testing of web applications. A user records a series of actions, and defines UI properties that have to be satisfied in order for the test to be successful. We consider that a feature is successfully extracted if the same Selenium tests can be successfully executed both in the original application and in the application composed out of the extracted code.

The result of the process is a web page that contains only the implementing UI controls with all necessary code and resources required for the implementation of a feature.

The experiment was performed on ten medium sized web applications (each application contains around 11 000 lines of HTML, CSS and JavaScript code). All evaluated applications use the jQuery library, the most wide-spread library for client-side scripting. jQuery is a complex library, with about 9000 LOC, and provides functionality for simplifying work with multiple browsers, selecting DOM elements, animations, etc.

The results of the experiment are shown in Table 1 – out of 10 web applications, we were able to identify and extract 13 features. The table shows that, on average, the extracted feature requires only around 50% of executed code (E/P LOC), and that the same feature behavior can be achieved with 38% less executions (E/P EXE).

7.2 Extracting Library Functionalities

For the second experiment we wanted to validate the correctness of the extraction process against a set of externally defined behaviors. For this purpose we decided to use unit-tests specified by the developers themselves. While it is true that there does not have to be a one-to-one mapping between features and unit-tests, and that the purpose of the tests is to reveal errors and not necessarily specify features, in this experiment we consider unit-tests as externally defined behavior specification that we use to gain information about whether the extraction process is correct. We have performed the experiment on three open-source JavaScript libraries: Gauss[8] – a library for statistics, analytics, and sets; Medialize[9] – a library for working with URLs, and Sylvester[10] – a vector and matrix library.

```
/* Start Code Excerpt */
'Minimum': {
    topic: set.min(),
    '1': function(topic) {
      assert.equal(topic, 1);
    }
  }
/* End Code Excerpt */

/* Converted to: */
var a1 = set.min() == 1; //feature manifestation
```

Listing 8. Converting unit-tests in the Gauss library

Since the goal of the evaluation is to extract code responsible for each functionality represented by a unit-test, and to obtain results that show the efficiency of the extraction process, we did not want to include any non-library code. For this reason, in each unit-test, all code expressions related to unit-test libraries were replaced by functionally equivalent expressions which were then used as a basis for the feature identification process. For the example given in Listing 8, the unit-test *'Minimum'* is designed to test whether the variable

6. Tests run on Intel Core i7@1.73 GHz, 4 GB RAM
7. http://docs.seleniumhq.org/
8. https://github.com/stackd/gauss
9. https://github.com/medialize/URI.js
10. https://github.com/jcoglan/sylvester

TABLE 1
Experimental results for feature extraction. F-ID – Feature ID, T – Total, P – Profiled, E – Extracted, LOC – Lines of Code, EXE – Executions, Time – Extraction time in seconds

Page	F-ID	Scenario description
mailboxing.com	1	Wait for automatic screen-shot cycling; Click each point-button once
mailboxing.com	2	Wait for automatic change of comment title and content
mailboxing.com	3	Click left arrow three times, click right arrow three times; Click each point-button
makalumedia.com/aerospace	4	Click the more button 10 times
makalumedia.com/aerospace	5	From left to right click each circle-button once; Click first circle button
disposable.hipstamatic.com	6	From top to bottom click each list-item once; Click first list-item
sipp.cc	7	Hover and click on iPhone container; Click right arrow twice; Click left arrow twice
blip.me/broadcast	8	Wait for automatic cycling between different items
idt.mdh.se/pride	9	Click second bullet; Click first bullet
instagalleryapp.com	10	Click each thumbnail once
fourandthree.com	11	Click each tab button once
irisapp.cc	12	Wait for automatic cycling between between slides; Click each slide-link once
indubitablee.com	13	Wait for automatic page cycling; Click each button; Click right arrow, left arrow

Page	F-ID	T-LOC	P-LOC	E-LOC	P-EXE	E-EXE	P/T LOC	E/P LOC	E/P EXE	Time
mailboxing.com	1	11464	4221	1952	140113	102979	36%	46%	73%	55
mailboxing.com	2	11464	3626	1933	194325	87510	31.6%	53.3%	45%	180
mailboxing.com	3	11464	4385	2185	181796	136277	38.2%	49.8%	76%	98
makalumedia.com/aerospace	4	11171	3220	1211	192599	101255	28.2%	37.6%	52.5%	79
makalumedia.com/aerospace	5	11171	3156	1217	140567	76935	28.2%	38.5%	54.7%	48
disposable.hipstamatic.com	6	9914	2334	827	86088	38036	23.5%	35.4%	44.1%	29
sipp.cc	7	10368	2911	1615	107054	70892	28%	50.6%	66.2%	31
blip.me/broadcast	8	11141	2999	1472	137596	99555	26.9%	73.6%	72.3%	39
idt.mdh.se/pride	9	10621	3981	2537	114283	58479	37%	64%	51%	73
instagalleryapp.com	10	9609	2687	1083	64583	35072	27.9%	40.3%	54.3%	19
fourandthree.com	11	9624	2434	1070	34040	17200	25.2%	43.9%	50.5%	11
irisapp.cc	12	9903	3258	1810	176179	139802	32.8%	55.5%	79.3%	43
indubitablee.com	13	9981	3441	2011	193200	159600	34.4%	58.4%	82.6%	2701

TABLE 2
The summary of the results on extracting features from three test libraries. LOC - Lines of Code, EXE - Execution steps.

	Gauss: 678 LOC, 31/33 scenarios successful				Medialize: 1580 LOC, 30/30 scenarios successful			
	Min	Mean	Median	Max	Min	Mean	Median	Max
T-LOC	691	693.2	693	700	1595	1617.2	1611	1656
P-LOC	126	162.7	156	218	224	443.9	434.5	625
E-LOC	27	56.1	53	91	28	206.4	205.5	340
P-EXE	155	2771	1372	8950	840	5032.1	2718	30227
E-EXE	53	1498.4	1036	5260	98	2598.9	1461	13042
Time	1	2.27	1	18	1	3.02	2	13
P/T LOC	18.2%	23.5%	22.5%	31.1%	14.0%	27.4%	27.0%	38.3%
E/P LOC	21.4%	33.7%	32.9%	46.2%	10.1%	45.7%	47.0%	64.8%
E/T LOC	3.9%	7.8%	7.7%	13.0%	1.8%	12.7%	12.8%	20.9%
E/P EXE	25.7%	63.7%	71.7%	95.2%	3.1%	54.1%	54.6%	81.8%
	Sylvester: 2347 LOC, 83/83 scenarios successful				All: 147/149 scenarios successful			
	Min	Mean	Median	Max	Min	Mean	Median	Max
T-LOC	2362	2370.4	2370	2386	691	1836.5	2364	2386
P-LOC	501	585.8	563	837	126	461	527.5	837
E-LOC	35	135.4	90	467	27	132.1	87.5	467
P-EXE	3433	6819.9	4796	31114	155	5537.4	4562	31114
E-EXE	85	3470.2	1412	26537	53	2845.5	1384	26537
Time	3	7.38	5	98	1	5.31	4	98
P/T LOC	21.2%	24.7%	23.8%	35.2%	14.0%	25.0%	23.9%	38.3%
E/P LOC	7%	21.7%	17.1%	55.8%	7.0%	29.4%	28.7%	64.8%
E/T LOC	1.5%	5.7%	3.8%	19.6%	1.5%	7.7%	6.3%	20.9%
E/P EXE	2.5%	37.5%	32.1%	87.9%	2.5%	46.8%	46.1%	95.2%

topic (whose value is set by calling the method *min* on the *set* variable) is equal to 1. A functionally similar test would be to just check whether the *set.min()* equals to 1. In that way we avoid the inclusion of the unit-test framework, but still capture the essence of the test.

Table 2 shows a summary of the results; for each library it presents mean, median, min and max values for the metrics described at the beginning of the section. In total, we have used 149 unit-tests, and in all but two cases the extracted code was able to pass the unit tests again (discussed at the end of the subsection – Failing Tests).

As can be seen in bottom right corner of Table 2, a unit-test on average executes around 25% of total library code (Profiled/Total LOC). However, some of the executed code is not necessary from the perspective of the unit-test, and by using the extraction process we get that, on average, only 29.4% (E/P LOC) of the executed code is necessary to pass the unit-test (only 7% of total library code, E/T LOC is required to pass a unit-test). In terms of overall execution steps we can see that, on average, it is possible to replicate the feature with 46% of execution steps (E/P EXE), mostly by not including initializations that do not affect the end result. It is important to note that savings are greatest for unit-tests that test simple methods, where the overall number of execution steps performed in the method is significantly smaller than the number of executions generated in the initialization phase (notice how the savings are lower in the cases where the execution spends considerably less execution time in the initialization phase – in the most complex unit-test of the Gauss library the execution savings are only around 4.8%).

We also present the time it takes to execute the whole extraction process – on average, it takes a couple of seconds to identify the implementation code (maximum of 98 seconds for the most complex test).

This experiment illustrates how libraries come with a considerable overhead, if the application uses only a small percentage of library code.

Failing Tests

Out of 149 executed tests there are two cases where the method was not able to extract code that successfully passes the unit-test. Both tests are unit-tests of the same library and fail by executing code in the same function. A simplified version of the code is given in Listing 9.

After the execution of the program the value of *b1* will be an array: [2, 3], and the value of *a1*: 2. If the identification process is centered on the value of *a1* in line 15, the result would be the inclusion of all code expressions except the initialization of the array from line 2 (*a1* depends on the array from line 5). At the core of the problem is our decision not to slice arrays (slicing arrays would cause more problems with array indexes, loops). In this case, the algorithm includes the first item ('1') of the array from line 13 (even though the value of *a1* does not depend on it) because other items from the array are required – when re-executing the unit test in the context of extracted code, an error occurs (*resultArray* is not initialized, the first value in the array is 1, and the control goes to line 9 and adds an item to a not-initialized array). Even though this is, in these two cases, an easily detectable and fixable problem, we acknowledge the possibility that more serious errors could occur. This is not a problem of the identification process (the process has correctly identified all dependencies), but the problem associated with extracting the identified code and using it in a stand-alone fashion.

```
/*01*/function fun(array) {
/*02*/  var resultArray = [];
/*03*/  for(var i = 0; i < array.length; i++) {
/*04*/    if(array[i] % 2 == 0) {
/*05*/      resultArray = [];
/*06*/      resultArray.push(array[i]);
/*07*/    }
/*08*/    else
/*09*/        resultArray.push(array[i]);
/*10*/  }
/*11*/  return resultArray;
/*12*/}
/*13*/var b1 = fun([1,2,3]);
/*14*/var a1 = b1[0];
/*15*/a1;
```

Listing 9. A simplified excerpt from a failing unit-test

7.3 Page Optimization

For the third experiment – page optimization, the goal was to show that the process is capable of identifying code of all features offered by the application. In the experiments the process identifies and removes code that does not contribute to any behavior. In order to do this, we have to know all application behaviors. For this reason, we have chosen 8 demo web applications that describe their behavior, and 2 standard web applications where it was easy to identify all application behaviors. Similar to the feature extraction experiment, based on the application behaviors, we have defined Selenium tests, and we consider that the extraction is successful if the extracted code is able to pass the predefined tests the original application has passed.

Table 3 shows the URL's of the selected pages, the scenarios that cause the manifestation of the behaviors, and all data gathered during the experiment. In all test applications the extracted code was able to pass the Selenium tests. The table shows that the optimized page generates 38%–100% executions (savings from 0%–62%), resulting in 0%–83% gains (1 - E/P LT) in page loading time. The savings are greatest in applications that use client-side libraries, while they are almost non-existent in small demo applications

TABLE 3
Experimental results for page optimization. T – Total, P – Profiled, E – Extracted, LOC – Lines of Code, EXE – Executions, Time – Extraction time in seconds, LT – Loading Time

Page	URL	Scenario description
codeBubble	jqueryfordesigners.com/demo/coda-bubble.html	Mouse over container; Mouse out container, wait for effects
fancyCheckbox	webdesign.maratz.com/lab/fancy-checkboxes-and-radio-buttons/demo.html	Click twice each checkbox; Click each radio-button
humanTypist	github.com/kennym/jQuery.humanTypist	Wait for the typing animation to finish
idtPride	idt.mdh.se/pride/	Click second, first button, second button; wait for effects
password	jamesrwhite.co.uk/testing/ password_hash/	Type a password: pa55w0d
tinySlider	sandbox.scriptiny.com/tinyslider2/	Auto-cycle all; Click left; Click right; Click each thumbnail
tabs	jqueryfordesigners.com/demo/tabs.html	Click on each tab;
fourandthree	fourandthree.com/	Click on each button
suckerFish	be.twixt.us/jquery/suckerFish.php	Mouse hover over each container
jSlideshow	line25.com/wp-content/uploads/2011/jquery-slideshow/demo/index.html	Click on each button once

Page	T-LOC	P-LOC	E-LOC	P-EXE	E-EXE	P/T LOC	E/P LOC	E/P EXE	LT	E-LT	E/P LT	Time
codaBubble	9611	2838	1383	39598	19488	30%	49%	49%	204	79	39%	27
fancyCheckbox	117	117	108	1491	1235	100%	92%	83%	36	36	100%	2.7
humanTypist	9480	2017	605	31629	12016	21%	30%	38%	204	34	17%	23.2
idtPride	10621	3981	2537	114283	58479	37%	64%	51%	285	165	58%	73.9
password	149	149	149	7047	7047	100%	100%	100%	10	10	100%	3
tinyslider	260	254	248	20704	20690	98%	98%	100%	27	27	100%	134.5
tabs	9514	2520	1234	36599	19525	26%	49%	53%	200	65	32%	24.9
fourandthree	10564	3664	2393	30523	15513	35%	65%	51%	247	200	81%	26.6
suckerFish	9663	2623	1352	75952	36152	27%	52%	48%	205	75	37%	81.8
jSlideshow	9859	3273	1944	101472	66917	33%	59%	66%	232	116	50%	52.08

(where there is no dead code).

It is important to note that the goal of the evaluation was to show that the method is capable of identifying code responsible for a behavior, and not to determine how much unnecessary code is usually included in web applications. However, the results indicate that web applications contain more code than is actually needed for their behavior, and that considerable savings could be achieved by applying this extraction method.

7.4 Threats to validity

There are several issues that might occur when attempting to generalize the experiment results. One concern is whether the selected applications are representative of real-world web applications. We tried to tackle this concern by performing experiments on a wide range of applications: from JavaScript libraries which have up to 2000 lines of code through full web pages built from around 11 000 lines of code, that make use of the most wide-spread client-side JavaScript library – jQuery. Even in this case, in order to be able to generalize the results (e.g. that web applications contain more code than is actually needed by their behavior), the experiments would have to be performed on a much larger set of web applications. We consider this as part of future work.

Another important threat to validity is whether or not our method is capable of extracting all of the code that implements a feature. Since our method is based on dynamic analysis of web application code in a particular scenario, we are aware that the quality of the scenarios is vital for the correct identification of feature code. This is why we are not claiming that our method is capable of identifying the full code of a feature, but the code of the feature manifested by the specified scenarios. However, in order to tackle this problem we have developed a method for automatic generation of feature scenarios [35] that systematically explores the event and value space of the application. The details of the method are outside the scope of this paper.

8 Related Work

Our work is closely related to program slicing, defined by Weiser [5] as a method that starting from a subset of a program's behavior, reduces that program to a minimal form which still produces that behavior. In its original form, a program is sliced statically, for all possible program inputs. Further research has led to the development of dynamic slicing [6] in which a program slice is composed of statements that influence the value of a variable occurrence for specific program inputs – only the dependencies that occur in a specific execution of a program are studied. Program slicing is usually based on some form of a Dependency Graph – a graph that shows dependencies between code constructs. Depending on the area of application, it can have different forms: a Flow Graph in original Weiser's form, a Program Dependence Graph (PDG) [7] where it shows both data and control dependencies for each expression, or a System

Dependence Graph (SDG) [8] which extends the PDG to support procedure calls rather than only monolithic programs. The SDG has also been later expanded in order to support object-oriented programs [16]. None of these graphs are fully suitable for capturing dependencies in a multi-language dynamic environment that is the client-side of the web application.

In the web domain Tonella and Ricca [9] define web application slicing as a process which results in a portion of a web application which still exhibits the same behavior as the initial web application in terms of information of interest to the user. They present a technique for web application slicing in the presence of dynamic code generation by building an SDG for server-side web applications. Even though the server-side and the client-side applications are parts of the same whole, they are based on different development paradigms, and cannot be treated equally.

Our work is also related to feature location – a technique for identifying source code locations that correspond to specific functionality. One of the earliest feature location techniques is software reconnaissance [20]. As input the technique receives two sets of scenarios: one set triggering the target feature, and the other set not triggering it. By analyzing their execution traces the technique can identify the program elements that only appear in the traces that invoke the feature. In [14], Eisenbarth and Koschke show a semiautomatic technique that reconstructs the mapping between computational units and features that are triggered by the user and exhibit an observable behavior. Their method is based on building a concept lattice, which is analyzed by the analyst. Compared to these two methods our method takes into account the specifics of technologies used in client-side web development, and supports automatic discovery of feature manifestation points, which we then use to identify code that implements the feature in a particular scenario.

There are also approaches that facilitate the understanding of dynamic web page behavior: Script InSight [10], FireCrystal [11], FireDetective [29], and ReAjax [30]. Script InSight helps to relate the elements in the browser with the lower-level JavaScript syntax. It uses the information gathered during the script's execution to build a dynamic, context-sensitive, control-flow model that summaries tracing information. FireCrystal facilitates the understanding of interactive behaviors in dynamic web pages by recording interactions and logging information about DOM changes, user input events, and JavaScript executions. After the recording phase, the user can use an execution time-line to see the code that is of interest for the particular behavior. FireDetective [29] facilitates the understanding of client-server interactions in AJAX application by visualizing the execution traces, and combining client- and server-side information to link different execution traces. ReAjax [30] is a reverse-engineering tool that uses dynamical analysis to infer a finite state machine of the application's GUI, where each state represents an instance of the single-page DOM, and a transition the event that causes the DOM change. While all of these approaches facilitate the understanding of dynamic web application behaviors, compared to our approach, they work on a higher level of granularity (executed functions, DOM changes) – they make no attempts to track dependencies between different code expressions. Dependency tracking, by using the client-side dependency graph, enables us to determine the code expressions that contribute to a behavior on a finer and more exact level than compared to using only profiling.

In the domain of optimizing web application code there is also a Doloto tool [31]. Doloto takes as input the existing client-side code, traces application workloads, and outputs a rewritten version that performs dynamic loading of function code. The processed application initially transfers only the portion of the code necessary for its initialization, and the rest of the code is replaced by short stubs, while the actual implementation is transfered lazily in the background, at latest on-demand. Since the goal of Doloto is to optimize the download of web application code, they make no attempts to identify feature code. Doloto and our approach can be viewed as complementary – our method can be used to remove dead code, and Doloto can be used to dynamically load only the live code.

There are also a number of works on static [33] [34], and combined static and dynamic analysis [32]. However, our goal is to identify feature code that can later be extracted in a stand-alone fashion. This requires high precision analysis that cannot be achieved with the presented analysis techniques.

This work is a continuation and extension of our previous work [15]. Compared to that paper, we have made the following extensions: *i)* we have defined a conceptual model of the client-side that we use to reason about the relationships between features, the underlying page structure, and scenarios; *ii)* throughout the process we have added support for server-side communication, by including server-side communications as potential feature manifestation points; *iii)* we have provided a more detailed description of the algorithms for graph construction and graph marking; *iv)* we have strengthened the evaluation by performing additional experiments on three open-source libraries, and have performed the feature extraction and page optimization experiments on a larger set of web applications; *v)* for the whole approach, on almost all levels, we have provided significantly more details that facilitate the understanding of the whole process.

9 Conclusion

In this work we have shown how to identify code responsible for the implementation of a certain client-

side feature in web applications. We have demonstrated how, even in this highly dynamic, multi-paradigm, multi-language environment, dependencies can be tracked by constructing a client-side dependency graph, and how by using that graph the code responsible for a certain feature can be identified. We have evaluated the approach by performing three sets of experiments on a range of web applications, and have reached two conclusions: *i)* the method can correctly identify stand-alone behaviors by analyzing web application event traces, and *ii)* considerable savings in terms of number of executions, page loading time, and code size can be achieved while still being able to reproduce the demonstrated behavior.

The client-side dependency graph represents all dependencies in the application and it can be utilized in a number of different applications, e.g. for code understanding, dependency analysis, or clone detection. For future work, we plan to investigate some of these options.

The client-side and the server-side application are parts of the same whole and we plan to extend the process to support the analysis of server side code and resources.

References

[1] C. Kapser and M. W. Godfrey, *"Cloning Considered Harmful" Considered Harmful*, Working Conference on Reverse Engineering, pages 19–28, http://dx.doi.org/10.1109/WCRE.2006.1, IEEE Computer Society, 2006.

[2] R. Holmes, *Pragmatic Software Reuse*, University of Calgary, Canada, 2008

[3] J. Brandt, P. J. Guo, J. Lewenstein, and S. R. Klemmer, *Opportunistic programming: How rapid ideation and prototyping occur in practice*, Workshop on End-user software engineering, pages 1–5, ACM, 2008

[4] World Wide Web Consortium (W3C), XMLHttpRequest, http://www.w3.org/TR/XMLHttpRequest/ June 2012,

[5] M. Weiser, *Program slicing*, International Conference on Software engineering, pages 439–449, IEEE, 1981

[6] H. Agrawal, and J. R. Horgan, *Dynamic program slicing*, Conference on Programming language design and implementation, PLDI '90, pages 246–256, ACM, 1990

[7] S. Horwitz, J. Prins, and T. Reps, *Integrating noninterfering versions of programs*, ACM Trans. Program. Lang. Syst., volume 11, issue 3, pages 345–387, July, 1989

[8] S. Horwitz, T. Reps and D. Binkley, *Interprocedural slicing using dependence graphs*, SIGPLAN Not., volume 23, issue 7, June, 1988, pages 35–46

[9] P. Tonella, and F. Ricca, *Web Application Slicing in Presence of Dynamic Code Generation*, Automated Software Engg., volume 12, number 2, 2005, pages 259–288

[10] P. Li and E. Wohlstadter, *Script InSight: Using Models to Explore JavaScript Code from the Browser View*, International Conference on Web Engineering, 2009, pages 260–274

[11] S. Oney, B. Myers, *FireCrystal: Understanding interactive behaviors in dynamic web pages*, Symposium on Visual Languages and Human-Centric Computing, pages 105–108, 2009

[12] JSON, http://www.json.org/, 01.09.2012.

[13] W3Tech, http://w3techs.com/technologies/overview/javascript_library/all, 04.09.2012.

[14] T. Eisenbarth, and R. Koschke, *Locating Features in Source Code*, IEEE Transactions on Software Engineering, VOL. 29, NO. 3, march 2003

[15] J. Maras, J. Carlson, I. Crnkovic, *Extracting Client-side Web Application Code*, World Wide Web Conference 2012, Lyon

[16] L. Larsen, M. J. Harrold, *Slicing object-oriented software*, International conference on Software engineering, ICSE '96

[17] S. Galbraith, *Quantifying the Relationship between Website Download Time and Abandonment by Users*, http://www.simple-talk.com/dotnet/.net-tools/the-cost-of-poor-website-performance/, 06.09.2012.

[18] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, *The concept assignment problem in program understanding*, Proceedings of the 15th international conference on Software Engineering, 1993, pages 482–498

[19] V. Rajlich, and N. Wilde, *The Role of Concepts in Program Comprehension*, Proceedings of IEEE International Workshop on Program Comprehension (IWPC'02), pages 271-278, 2002

[20] N. Wilde and M. Scully, *Software Reconnaissance: Mapping Program Features to Code*, Journal of Software Maintenance: Research and Practice, vol. 7, 1995, page. 49-62.

[21] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, *Feature location in source code: a taxonomy and survey*, Journal of Software Maintenance and Evolution: Research and Practice, 2011

[22] S. Artzi, J. Dolby, S.H. Jensen, A. Møller, and F. Tip, *A framework for automated testing of javascript web applications*, Proceedings of the 33rd International Conference on Software Engineering, pages 571–580, 2011

[23] , P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, *A symbolic execution framework for javascript*, Security and Privacy (SP), 2010 IEEE Symposium on, pages 513–528, 2010

[24] 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, No. 6, pages 420-432

[25] J. Coffey, L. White, N. Wilde, and S. Simmons, *Locating Software Features in a SOA Composite Application*, In Web Services (ECOWS), 2010 IEEE 8th European Conference on, pages 99–106

[26] IEEE Standard for Software and System Test Documentation, IEEE Std 829-2008 , vol., no., pp.1,118, July 18 2008, doi: 10.1109/IEEESTD.2008.4578383

[27] A. Mesbah, A. van Deursen, D. Roest, *Invariant-Based Automatic Testing of Modern Web Applications*, Software Engineering, IEEE Transactions on 38.1 (2012), pages 35-53

[28] A. Marchetto, P. Tonella, *Using search-based algorithms for Ajax event sequence generation during testing*, Empirical Software Engineering 16.1, 2011, pages 103-140

[29] N. Matthijssen, A. Zaidman, M.A. Storey, I. Bull, A. van Deursen, *Connecting Traces: Understanding Client-Server Interactions in Ajax Applications*, Program Comprehension (ICPC), 2010 IEEE 18th International Conference on. IEEE, 2010

[30] A. Marchetto, P. Tonella, and F.Ricca, *ReAjax: a reverse engineering tool for Ajax web applications*, Software, IET 6.1, 2012, pages 33-49.

[31] B. Livshits, E. Kiciman, *Doloto: Code Splitting for Network-Bound Web 2.0 Applications*, Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. ACM, 2008, pages 350-360

[32] S. Wei and B.G. Ryder, *A practical blended analysis for dynamic features in javascript*, Technical Report TR-12-18, Department of Computer Science, Virginia Tech, 2012

[33] M. Madsen, B. Livshits and M. Fanning *Practical static analysis of Javascript applications in the presence of frameworks and libraries*, Tech. Rep. MSR-TR-2012-66, Microsoft Research, July 2012)

[34] S.H Jensen, M. Madsen, and A. Møller *Modeling the HTML DOM and browser API in static analysis of JavaScript web applications*, ESEC/FSE, 2011

[35] J. Maras, M. Štula, and J. Carlson *Generating Feature Usage Scenarios in Client-side Web Applications*, International Conference on Web Engineering, ICWE 2013

Josip Maras is a Ph.D student at the Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, University of Split. He has received his masters degree in computer science in 2009. His research interests include program analysis, web applications, and software engineering in general.

Maja Štula is associated professor of computer science at the Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, University of Split. She received a BS in electrical engineering in 1996, a MSc in 2001 and a PhD in 2005 all from the Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, University of Split. Her research interests include multi-agent systems, fuzzy cognitive maps, semantic web, and the application of intelligent technologies.

Jan Carlson is a senior lecturer at the School of Innovation, Design and Engineering, Mälardalen University, Sweden. He received his M.Sc. degree in Computer Science from Linköping University in 2000, and his doctoral degree from Mälardalen University in 2007. His research interests include component models for embedded systems, event pattern detection, formal methods and logic programming.

Ivica Crnković received the Ph.D. Degree (91) in computer science, and before that the M.Sc. (81) in computer science and M.Sc. in theoretical physics (84) all from the University of Zagreb, Croatia. After 15 years of work in industry, he moved to academia 1999. He is a professor of software engineering and chair of Software Engineering Division at Mälardalen University, Sweden, and a professor at Faculty of Electrical Engineering, University of Osijek, Croatia. He is a co-author of two books, and the co-author of more than 100 refereed publications on software engineering topics. His research interests include component-based software engineering, software architecture, software conguration management, software development environments and tools, and software engineering in general.