Extracting Client-side Web Application Code

Josip Maras
University of Split
Split, Croatia
josip.maras@fesb.hr

Jan Carlson, Ivica Crnković
Mälardalen University
Västerås, Sweden
{jan.carlson, ivica.crnkovic}@mdh.se

ABSTRACT

The web application domain is one of the fastest growing and most wide-spread application domains today. By utilizing fast, modern web browsers and advanced scripting techniques, web developers are developing highly interactive applications that can, in terms of user-experience and responsiveness, compete with standard desktop applications. A web application is composed of two equally important parts: the server-side and the client-side. The client-side acts as a user-interface to the application, and can be viewed as a collection of behaviors. Similar behaviors are often used in a large number of applications, and facilitating their reuse offers considerable benefits. However, due to client-side specifics, such as multi-language implementation and extreme dynamicity, identifying and extracting code responsible for a certain behavior is difficult. In this paper we present a semi-automatic method for extracting client-side web application code implementing a certain behavior. We show how by analyzing the execution of a usage scenario, code responsible for a certain behavior can be identified, how dependencies between different parts of the application can be tracked, and how in the end only the code responsible for a certain behavior can be extracted. Our evaluation shows that the method is capable of extracting stand-alone behaviors, while achieving considerable savings in terms of code size and application performance.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Algorithms, Theory

Keywords

Web applications, Slicing, Code extraction, Reuse

1. INTRODUCTION

Highly interactive web applications that offer user experience and responsiveness of standard desktop applications are becoming increasingly popular. They are composed out of two equally important parts: the server side, realized as a sequential application implementing data-access and business logic, and the client side, realized as an event-driven application representing the user-interface (UI). On the client side, a web page structure is defined with HTML code, presentation with CSS (Cascading Style Sheets), and behavior with JavaScript code. Alongside code, a web page usually contains resources such as images, videos, or fonts. All this code and resources are transfered to and evaluated in the user's web browser, and the interplay of these basic elements produces the end result.

From the UI perspective, a web page can be viewed as a collection of visually and behaviorally distinct elements, the so called UI controls. However, this distinctiveness does not usually exist in code, since there is no predefined way of organizing code into neatly packed components.

A client side web application can also be viewed as a collection of behaviors: from simple behaviors implementing a single functionality, through complex UI behaviors provided by UI controls, all the way to a single, complex behavior that represents the functionality of the whole page. Similar behaviors are often used in a large number of web applications, and facilitating their reuse offers significant benefits. However, this is a challenging task. Due to the underlying event-driven paradigm and the fact that a single behavior can be implemented with a combined effect of three different languages (HTML, CSS, and JavaScript) based on entirely different paradigms, it is difficult to identify code responsible for a certain behavior. This is especially true, because the most complex language – JavaScript is an dynamic scripting language. In addition to facilitating reuse the ability to set code into relation to behavior can also be used to detect and remove dead code. On top of increasing code maintainability, dead code removal also has a positive effect on web application performance, because all code is transfered and interpreted in the user's web browser.

In this paper we present a method for extracting client-side web application code. Our main contribution is a code extraction method based on the analysis of application execution traces recorded while demonstrating a web application usage scenario. In order to be able to extract the code that implements a certain behavior we have to identify it. We define a client-side web application dependency graph, describe how it is constructed, and show how it can be used to locate and extract code responsible for a certain behavior. Our work is motivated by three different usages: *i)* extracting library functionality, *ii)* extracting UI controls, and *iii)* removing dead code. We have evaluated the approach for each one, and have found out that considerable savings in terms of code size and performance can be made, while still being able to reproduce the demonstrated usage scenario.

The paper is organized as follows: in Section 2 we define the overall approach, and give a short introduction to the inner workings of client-side web applications necessary to understand the extraction algorithm. Section 3 gives a more detailed description of each step of the approach, defines the dependency graph, its construction, and shows how it can be used to locate code implementing a demonstrated usage scenario. Section 4 shows the results of the evaluation, while Section 5 describes related approaches. Section 6 gives the conclusion and presents possible continuation of the research.

2. THE OVERALL APPROACH

Our goal is to provide a method for extracting behaviors from client-side web applications: from a single functionality mapped to a function call, through complex UI behaviors implemented by UI controls, all the way to the behavior of the whole web page. Ideally the code implementing a certain behavior would be identified with static analysis, which would guaranty the completeness of the behavior, but in an environment as dynamic as the client side web application, this is not possible. Because of the extreme dynamicity of JavaScript, the underlying event-driven paradigm, and the fact that a behavior is usually defined through the interplay of different web page elements, it is difficult to make a connection between a behavior and the implementing code. For these reasons we have based our approach on the analysis of usage scenarios. The main advantages of the approach are: *i)* it does not require any specification of the behavior to be extracted and *ii)* the analysis of the application execution trace recorded while demonstrating the usage scenario enables us to dynamically track code dependencies (something that cannot be accurately done statically for a language as dynamic as JavaScript). The downside is that the accuracy and completeness of the captured behavior in terms of how much it relates to the desired behavior is completely dependent on the developer demonstrating the usage scenario.

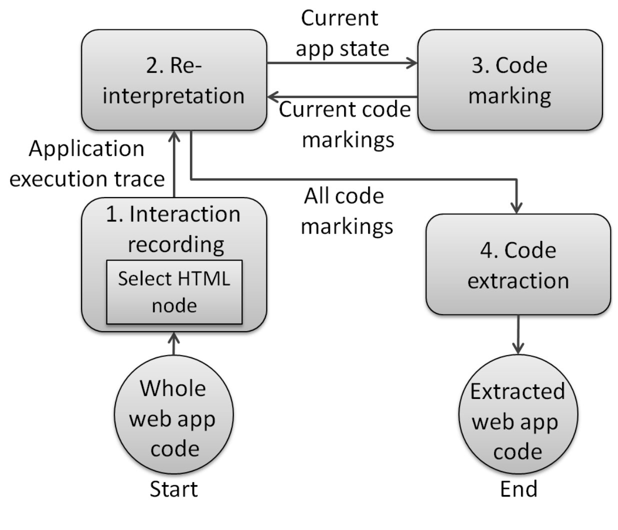

Figure 1: The Extraction process

The overall extraction process is shown in Figure 1, and goes as follows. Phase 1 – *Interaction recording* – requires input from the developer in a sense that he/she has to choose the HTML node the process should focus on (because the goal is to track UI modifications), and demonstrate the usage scenario while in the background the application execution trace is being recorded. In Phase 2 – *Reinterpretation* – the whole web application code is reinterpreted, and code dependencies are tracked with the application execution trace as a guideline. When the reinterpretation reaches a point in the execution trace that changes the structure of the HTML node we are interested in, the reinterpretation pauses and the process enters Phase 3 – *Code marking*. In this phase, starting from the code expression making the modification, all dependencies are traversed, and all code that directly or indirectly influences the modification code expression are marked as important. Once the dependency traversal is done, the process again resumes the reinterpretation in Phase 2. This cycle is repeated as long as there are node modifications in the application execution trace. When the reinterpretation is done the process enters the final phase, Phase 4 – *Code extraction* – where the code of the entire web application is traversed; code expressions identified in the code marking phase are kept, while the rest are removed – the end result being a subset of the original application still able to reproduce the demonstrated usage scenario.

The main component of the process is the tracking of dependencies between different parts of the web page. To better understand the whole process, we give a short web application primer.

2.1 A Web Application Primer

A client-side web application is, in its essence, an HTML page that includes JavaScript code, CSS code and various resources such as images or fonts. The HTML code defines the structure of a web page, JavaScript code the behavior, and CSS (Cascading Style Sheets) code the presentation. The interplay of these basic elements produces the end result displayed in the user's web browser. JavaScript is a weakly typed, imperative, object-oriented script language with prototype based delegation 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 also a dynamic language: everything can be modified at runtime, from fields and methods of an object to its prototype. As many other script languages, it offers the eval function which can execute an arbitrary string of JavaScript code. CSS is a declarative language used to specify the presentational aspects of HTML nodes. The CSS code of the application 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 nodes the given property-value pairs will be applied to.

Client-side web applications are mostly event-driven UI applications, and majority of the code is executed as a response to user-generated events. The life-cycle of the application can be divided into two phases: *i)* page initialization and *ii)* event-handling phase. The main 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 an object-oriented representation of the HTML document – the Document Object Model (DOM). When parsing the HTML code the DOM is filled one HTML node at a time. There are two special types of HTML nodes that the browser can reach: *i)* the style and link nodes, that enable the inclusion of CSS code, and *ii)* the script node that enables the inclusion of JavaScript code.

When the browser reaches either the style or the link node it parses the included CSS code and constructs a set of presentational rules. Each rule has a CSS selector specifying to which HTML nodes the rule will be applied to at any point of application execution. If the browser reaches the script node it suspends the DOM building process and enters the JavaScript interpretation process. In this phase this means sequentially executing the given JavaScript code. One important purpose of this code is to register event-handlers, which define how events are handled later during the second phase of the execution. Once the JavaScript code in that node is executed, the process again resumes the DOM building phase. After the last HTML node is parsed and the whole UI is built, the application enters the event-handling phase, where code is executed as a response to events. All updates to the UI 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.

A web page is a collection of UI controls, which communicate with the user by modifying their UI. Each UI control is defined with a set of HTML nodes (the UI control's DOM), a set of CSS rules applied to those nodes, and JavaScript code that implements the behavior. All modifications of the UI are done with JavaScript modifications of the UI control's DOM, and a single UI behavior is often implemented with multiple DOM modifications.

```
/*01*/<html>
/*02*/ <head>
/*03*/  <style>
/*04*/    .bgRd{background-color:red;}
/*05*/    .bgBl{ background-color:blue;}
/*06*/    div{height:60px; width:60px}
/*07*/    span{height:30px; width:30px}
/*08*/  </style>
/*09*/ </head>
/*10*/ <body>
/*11*/  <div class="bgRd"id="frSq"></div>
/*12*/  <br/>
/*13*/  <div class="bgBl"id="scSq"></div>
/*14*/  <script>
/*15*/   function changeColor(elem) {
/*16*/    elem.className = elem.className
    == "bgRd" ? "bgBl" : "bgRd";}
/*17*/   var frSq = document.
    getElementById("frSq");
/*18*/   var scSq = document.
    getElementById("scSq");
/*19*/   frSq.onclick = function(){
/*20*/    if(frSq.children.length == 0) {
/*21*/     var sml = document.
    createElement("span");
/*22*/     sml.className = "bgBl";
/*23*/     frSq.appendChild(sml);}
/*24*/    };
/*25*/   scSq.onClick = function(){
/*26*/    changeColor(scSq); };
/*27*/  </script>
/*28*/ </body>
/*29*/</html>
```

Listing 1: Example application

3. THE EXTRACTION PROCESS

The extraction process will be illustrated with a running example shown in Listing 1. The UI of the web application is initially composed out of two squares: a top red one, and a bottom blue one, and two behaviors: *i)* when the top square is clicked for the first time, a smaller blue square is created inside it; and *ii)* when a bottom square is clicked it changes its color (from blue to red, and vice versa). Throughout this section we will show how the code necessary only for the execution of the first behavior is extracted. A common approach is by building a dependency graph.

3.1 The Dependency Graph

The client-side web application code is composed of three different parts: CSS, HTML and JavaScript, 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 three types of nodes: HTML nodes, CSS nodes and JavaScript 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 nodes 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 and dynamic nodes. A node is static if it is directly present in the source code of the application, while it is considered dynamic if it is dynamically constructed with JavaScript code. Also, the DOM structure can be rearranged with JavaScript code, this is why there also exist two types of structural dependency edges (static and dynamic). Table 1 shows the definition of the different edge types, where edges marked s, d and c represent structural dependencies, data dependencies and control flow, respectively. H denotes HTML nodes, J JavaScript nodes, C CSS nodes, and N denotes a node of arbitrary type.

Table 1: Edges in the client side dependency graph

Edge	*Condition*
$N_1 \xrightarrow{\text{s}} N_2$	N_1 is a child of N_2.
$N \xrightarrow{\text{d}} J$	J writes data to N.
$J \xrightarrow{\text{d}} N$	J reads data from N.
$H \xrightarrow{\text{d}} C$	C style is applied to H.
$J_1 \xrightarrow{\text{c}} J_2$	J_2 is executed after J_1.

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. A dependency graph subgraph composed only of H-nodes matches the DOM of the web page.

C-nodes represent CSS rules applied to specified HTML nodes. All CSS code is contained in a HTML node, 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 structural dependency between a C-node and a J-node. The main goal of a CSS style is to define styling parameters of HTML nodes that satisfy certain conditions, for this reason, there can exist a data dependency between an H-node and the C-node.

J-nodes represent JavaScript code construct that occur in the program (a simplified Abstract Syntax Tree). All JavaScript code is contained in an HTML node, so each J-node has a structural dependency towards the parent HTML node. Two J-nodes can also have structural dependencies between themselves denoting that one code construct is contained within the other (e.g. a relationship between a function and a statement contained in its body). J-nodes can form data-dependency edges with all 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 the J-node is reading data, while an edge from the H-node to the J-node means that the J-node is writing data to the H-node. An edge from a J-node to a C-node means that the JavaScript code is reading data from the C-node.

Example. Figure 2 shows the graph built while reinterpreting the web application code based on the execution trace shown in the usage scenario. The circle nodes represent H-nodes, the square nodes C-nodes, and the rectangle nodes J-nodes. The numbers near each node represent the flow of control, the full lines represent structural dependencies, while the dotted lines represent data dependencies.

3.2 Interaction recording

The whole process is based on the analysis of application traces recorded by communicating with the JavaScript debugger while demonstrating the chosen behavior. The recording application is realized as a Firefox extension.

Example. The developer wants to extract the first behavior (creating a square within a square), so he selects the part of the UI on which the extraction process will be focused on – the HTML node with the id “frSq” and demonstrates the following usage scenario: click on the bottom square, causing it to change its color from blue to red (even though this is not required from the perspective of the first behavior), and clicking on the top square which results with the creation of a smaller blue square within it (the desired behavior).

3.3 Reinterpretation

After the application execution trace has been recorded, the process enters the second phase – Reinterpretation. As an input this phase receives the web page code, the node selected for extraction, and a recorded execution trace. The algorithm is shown in Algorithm 1, and has two phases: *page initialization* (line 1) and *event-handling* (lines 2–4). The basis of the algorithm is the process by which the browser executes the web page (described in Section 2.1.).

Building HTML and CSS nodes

In the initialization phase, for each encountered HTML code node a matching static H-node is created (Algorithm 2).

Algorithm 1 reinterpret(*webPageCode*, *extractionNode*, *appExeTrace*)

```
1: createHChildNodes(createStatHNode(),
                     createCodeTree(webPageCode),
                     appExeTrace, extractionNode)
2: for all event : appExeTrace do
3:     interpretJs(getCode(event), getNode(event),
                   getExecs(event))
4: end for
```

When an HTML with CSS code is reached, for each CSS rule a static C-node with structural dependencies to the containing H-node is created. On each DOM change, CSS rules are traversed and if an HTML node satisfies a CSS rule, a data dependency from the H-node to the C-node is created.

Example. The algorithm starts by creating the three H-nodes: html, head, and style node, where the head node has a structural dependency towards the html node, and the style node towards the head node. Since the style node is a special type of node, the process starts creating C-nodes – four static C-nodes are created based on four CSS rules, and each CSS node has a static structural dependency towards the parent style node. Next, the DOM building phase is continued, and the process creates the following H-nodes: body, div, br, div, and script and marks the necessary structural dependencies (body $\rightarrow$ html; div, br, div, script $\rightarrow$ body). Since the creation of each static node initiates the search for a matching CSS rule, when the first div node is created, data dependencies from that node to the first and the third C-node are created (since the HTML node satisfies those two CSS selectors), and when the second div node is created, data dependencies from the second div to the second and the third C-node are also created (top middle and top left part of the graph in Figure 2).

Algorithm 2 createHChildNodes(*hNode*, *htmlCodeElem*, *appExeTrace*, *extrNode*)

```
1: for all codeChild : getChildren(htmlCodeElem) do
2:     hChldNode ← createStatHNode(codeChild)
3:     appendChild(hNode, hChldNode)
4:     addStatStrucDep(hChldNode, hNode)
5:     traverseCNodesAndCreateDependencies()
6:     if codeChild is HTMLScriptNode then
7:         interpretJs(codeChild.text, hChldNode,
                       getScriptExe(appExeTrace), extrNode)
8:     else if codeChild is HTMLCssNode then
9:         for all cssRule : parseCss(htmlChldNode.text) do
10:            cNode ← createStatCNode(cssRule)
11:            addStatStrucDep(cNode, hChldNode)
12:        end for
13:    else
14:        createHChildNodes(hChldNd, codeChild, appExeTrace)
15:    end if
16: end for
```

Building JavaScript nodes

When the process encounters an HTML node containing JavaScript code, it switches to the creation of code construct nodes, and the process enters the execution-trace guided in-

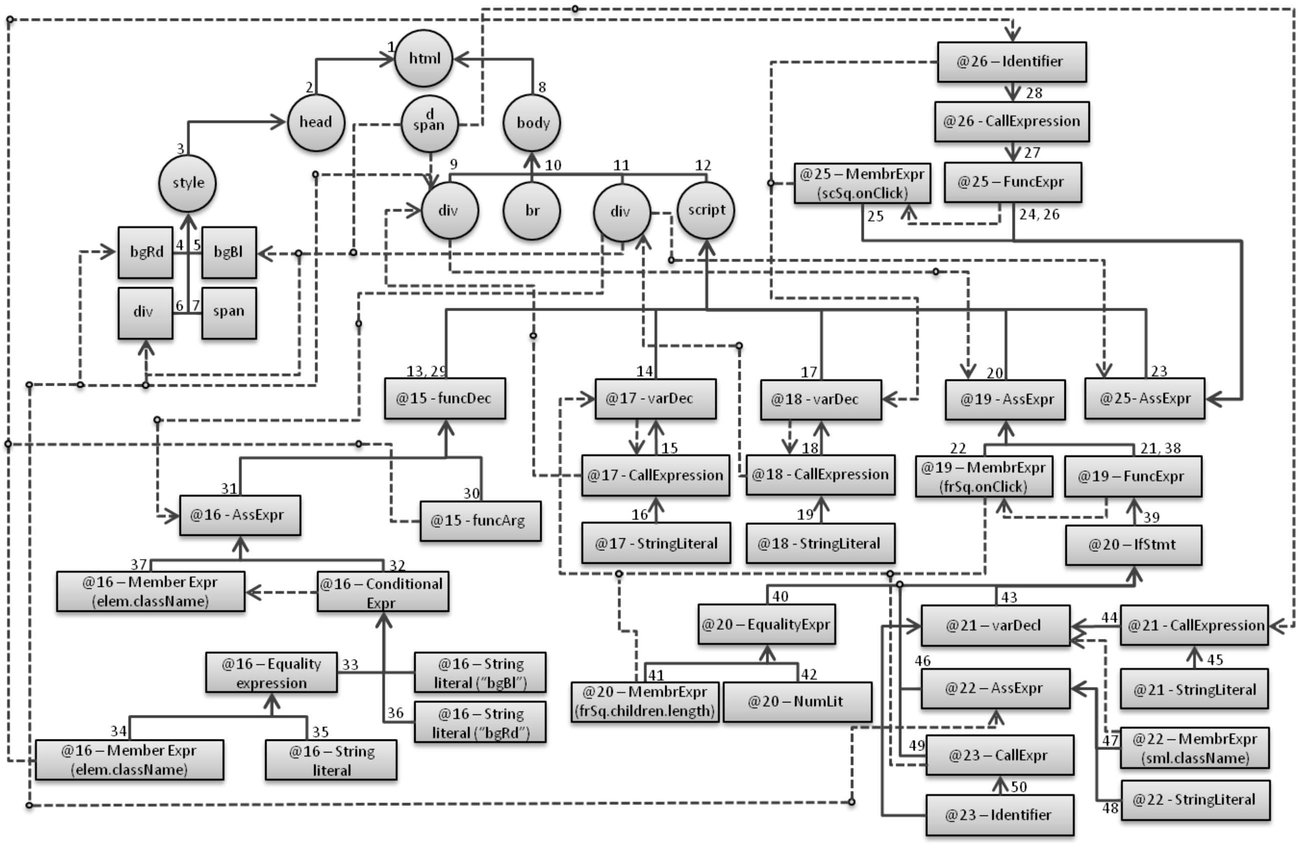

Figure 2: Dependency graph of the web application from Listing 1 (@ - denotes the line number of the code construct)

terpretation mode – code nodes are created as each code expression is evaluated (Algorithm 3). If the evaluated expression reads identifiers (a term encompassing objects, properties, and functions), then a data dependency from the current node to the node matching the last assignment of the identifier is created. Currently, we do not slice arrays, so if the accessed object is an array, data-dependencies from the current node to all nodes that match the modification expressions of that array object are also created. Also, some of the evaluated code expressions can create dynamic nodes, and in that case a data-dependency is created from the dynamic node to the currently evaluated JavaScript node.

Example. The process has reached the script node in line 14 and is entering the JavaScript interpretation mode (middle and the top right part of the graph in Figure 2), and a function declaration node from line 15 is created. Next, the control flow reaches the variable declaration in line 17, and the matching node is created. On the right hand side there is a method call on the document object (which is special object provided by the browser, acting as an interface to the DOM). The method invocation returns an HTMLObject mapped to the HTML node with the id "frSq" (the first div), so a data dependency from the call expression to the H-node is created. Also since this is a variable declaration expression, a data dependency from the identifier "frSq" to the call expression is created. Similarly the nodes and matching data dependencies for line 18 are created. For line 19 an assignment expression node 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 id "frSq" so it has a data dependency towards the variable declaration and the call expression in line 17 (because that is where the value has come from). Also, since the object whose property is being set is an HTML object, and since the "onclick" property is a property used to set an event-handler, a dependency from the matching H node to this assignment expression is created. A similar procedure is repeated for line 25. Since there is no more JavaScript code for sequential execution, the process exits the interpretation mode. Also, all H-nodes have been created and the *page initialization* phase is finished.

Event handling

Once the whole code file has been traversed, and all contained JavaScript code executed in a sequential fashion the graph construction enters the event-handling phase (Algorithm 1). Information about each event is read from the execution trace, and the dependency from the event handling function code node to the HTML node causing the event created. JavaScript nodes are created for each expression executed as a part of the event-handler code (Algorithm 3).

Example. In the demonstrated usage scenario, the first raised event was the click on the bottom square. The process reads the application execution trace and finds that the function in line 25 was executed as a click event-handler on an HTML node with id "scSq" (bottom left and bottom

Algorithm 3 interpretJs(*code*, *hNode*, *currExecs*, *programAst*, *extrNd*)

```
for all step in curExecs do
    currAstNd ← getAstNode(step)
    jNode ← createJNode(currAstNd)
    addStaticStructDep(jNode, hNode)
    evldAstNd ← evalute(currAstNd)
    if isAccessingIdentifiers(evaldAstNd) then
        create data dependencies from the jNode to the identifier's last assignment expression
    end if
    if isReadingArrayObject(evaldAstNd) then
        add data dependencies from the jNode to all expressions that have modified the array
    end if
    if isEnteringFunction(evaldAstNd) then
        add dependency from the jNode to the call expression
    else if isReturnFromCallExpr(evaldAstNd) then
        add data dependency from jNode to retrnExpr
    end if
    if isInLoopOrInIfStatement(evaldAstNd) then
        add data dependency from jNode to conditionExpr node
    end if
    if isCreatingJsCode(evldAstNd) then
        parse the newly added code and add AST node
    else if isCreatingHtmlNode(evldAstNd) then
        create a dynamic H-node with a dataDepend to jNode
        traverseCNodesAndCreateDependencies()
    else if isCreatingCssNode(evldAstNd) then
        create a dynamic C-node with dataDepend to jNode
    else if isRearrangingDOM(evldAstNd) then
        remove dynamic struct dependency of the target H-node; add a new dynamic struct dependency from the target H-node to the new parent node
        traverseCNodesAndCreateDependencies()
        if affected nodes are descendants of extrNd then
            createCodeMarkings(jNode)
        end if
    end if
end for
```

right of Figure 2). This creates a data dependency from the function expression construct towards the H-node. The function body is interpreted and all data dependencies for the executed code are created. The procedure is repeated for the second event that was raised as a response to clicking on the first square, and goes similarly to the body of the if statement. In line 21, a new HTML element is created. This causes the creation of a new dynamic H-node with a data dependency towards the call expression in line 21. Also, since this a DOM modifying expression, the search for the matching CSS rules, is also initiated, which in this situation, creates dependencies from the dynamic H-node to the fourth C-node. In line 22, the DOM of the newly created node is modified, so a data dependency from that H-node to the assignment expression, and to the second C-node is created. In line 23 the HTML node is inserted into the DOM of the first div (a node chosen for extraction). This raises the DOM modified event, and the process enters the code marking phase (Algorithm 4).

3.4 Code Marking

As the application code is being interpreted and code dependencies built, it is important to identify executed code constructs that will be used as a basis for code extraction. Since we are dealing with UI applications, these important code constructs are the ones that are modifying the UI. All UI modifications on the client-side are done by modifying the DOM of the web application, so we track dependencies from DOM modifying statements. The main idea is that when the code interpretation algorithm raises the DOM modifying event, the dependency graph is traversed starting from the HTML node being modified by following all dependencies (Algorithm 4). When a static node is reached its matching code construct is marked as important.

Example. The call expression in line 23 is modifying the DOM of the node chosen for extraction, so the process enters the marking phase which causes the traversal of the dependency graph. The call expression in line 23 is marked because it is causing the DOM modification. Next, the identifier "sml" is marked, which leads to the marking of the variable declaration in line 21. The value of the identifier matches the dynamically created HTML span node, so its dependencies are also traversed (but the node itself is not included). This causes the inclusion of the assignment expression in line 22, and the second and the fourth C-node (and the style node, and the head node because of structural dependencies). Since the node whose DOM is being modified is the one chosen for extraction, it, and its dependencies, are also included (the first div node, the parent body node, the parent html node, the first and the third C-node, and the assignment expression in line 17). The if statement in line 20, and the script node are also included since there are statements included within them.

Algorithm 4 createCodeMarkings(*topNode*)

```
codeNode ← getCodeNode(topNode)
if isStaticNode(codeNode) then
    markAsImportant(codeNode)
end if
if topNode is JNode then
    for all node : getCntrlDepInCurrCntxt(topNode) do
        if node is Break OR Continue
        OR (ConditionExpr AND
        areOnSameLvl(topNode, node)) then
            createCodeMarkings(node);
        end if
    end for
end if
for all node in getStrucAndDataDep(topNode) do
    nodes.push(node)
end for
```

3.5 Code Extraction

After the whole application execution trace has been reinterpreted, the process goes into the last phase – *Code extraction*, where the web applications code tree is traversed and where code is generated from all static nodes marked as important in the code marking phase.

Example. The extraction result for the example is shown in Listing 2. Even though the function *changeColor* was executed, and has modified the UI of the second square, it is not present in the final, extracted code, simply because it did not modify the part of the UI we were interested in (node – "frSq").

```
<html>
 <head>
  <style>
    .bgRd{background-color:red}
    .bgBl{ background-color:blue}
    div{height:60px; width:60px}
    span{height:30px; width:30px}
  </style>
 </head>
 <body>
  <div class="bgRd"id="frSq"></div>
  <script>
   var frSq = document.getElementById("frSq");
   frSq.onclick = function(){
    if(frSq.children.length == 0) {
     var sml = document.createElement("span");
     sml.className = "bgBl";
     frSq.appendChild(sml);}
    };
  </script>
 </body>
</html>
```

Listing 2: Extracted example application

4. EVALUATION

We have evaluated the approach based on three applications of the method: *i)* extracting library code, *ii)* extracting UI controls, and *iii)* removing dead code of web applications. The evaluation is based on two metrics: lines of code (LOC) and the number of execution steps (EXE). For the LOC we use two baselines: total LOC of the entire application, and the executed LOC (ExLOC). The effectiveness of the approach is then measured by comparing the baselines with the extracted LOC (ExtLOC). For the total LOC, in order to better compare the results we reformat the application source code in the same format the extracted code will be in (e.g. by removing comments, braking lines according to same rules, etc.). Executed code lines represent a very simple, but straightforward extraction approach. They are derived from the application execution trace, without any analysis, simply by keeping all uniquely executed lines while maintaining syntactical correctness. This code version does not include lines that were not executed, and is capable of replicating the starting usage scenario. It is important to note that not all executed code is important from the perspective of the target behavior (e.g. some of it might do initialization, or implement functionality that is irrelevant from the perspective of the target behavior). This is why we measure the effectiveness of our approach by comparing the extracted LOC with the executed LOC. We also report the difference in the number of execution steps when executing the usage scenario in the context of the full web application code (ExEXE), and when executed in the context of extracted code (ExtEXE). This difference represents the performance gains that can be achieved by using our method, because the same behavior is realized with less execution steps. All used test applications can be downloaded from www.fesb.hr/~jomaras/www2012.

The data presented in this evaluation was gathered by a tool developed as a Firefox extension – all tracing data is Firefox specific, and results could be different in another browser. However, we do not believe that considerably different results would be attained with other modern web browsers.

4.1 Extracting Library Code

For *extracting library code* we have evaluated the approach by extracting functionalities from an open-source vector and matrix math library – Sylvester[1]. It includes functions for working with vectors, matrices, lines and planes. As with other libraries, if we only use a small subset of its functionality, then a lot of library code will be irrelevant from our application's point of view. Based on the public API given on the homepage of the library, we have developed use-cases for a subset of the public methods. We have recorded the execution of those use-cases, with the following results: from the total of 130 methods spread over 1400 lines of code we have extracted 27 methods in a way that alongside each method only the code that is essential for the stand-alone functioning of the method is extracted. In all cases the method extraction was successful, meaning that the use-case could be repeated with the same result for the extracted code. Table 2 presents the experimental data. For each tested method it provides information about the total number of uniquely executed code lines during the execution of a use case (ExLOC), the number of lines that were included in the extracted code (ExtLOC), number of executions generated while executing the usage scenario in the context of the whole web application code (ExEXE), and the number of execution steps when repeating the usage scenario in the context of the extracted code (ExtEXE). As can be seen, each method executes from around 17%–27% of the total library code, and out of that executed code the extraction process extracts anywhere from 5%–65%, which constitutes the part of the code required to implement the target behavior. In terms of the overall number of execution steps, the savings range from 42% to 97%. It is important to note that savings are the greatest in simple methods, where the overall number of execution steps performed in the method is significantly smaller than executions generated in the initialization phase. As a base line we present the number of executed lines and executions recorded while just including the library, without explicitly executing any methods. Even in that case the library has 241 executed LOC and 2279 execution steps, the reason being that all included code generates executions (e.g registering function or variable declarations, performing initialization, etc.).

4.2 Extracting UI Controls and Dead Code Removal

The goal of the UI control extraction functionality is to extract only the HTML, CSS, and JavaScript code that build the UI control, while in the case of dead code elimination we want to remove JavaScript code that is not necessary from the behavior's perspective. These two usages of the method have been evaluated on the same test data (Table 3): eight

[1]http://sylvester.jcoglan.com/

Table 2: Experimental results on extracting API functions from the Sylvester math library

Method	*ExLOC*	*ExtLOC*	*ExEXE*	*ExtEXE*
No method	241	0	2279	0
V.create	241	14	2230	68
V.cross	247	20	2464	189
V.dot	241	26	2441	174
V.random	250	24	2421	160
V.zero	241	24	2427	166
V.add	269	43	2456	275
V.dimensions	244	18	2335	75
V.distanceFrom	266	38	2515	243
V.isParallelTo	280	51	2804	409
V.max	258	31	2477	192
V.modulus	241	30	2449	185
V.multiply	267	42	2553	289
V.rotate	299	77	2668	406
V.angleFrom	276	43	2549	261
M.add	312	76	3679	1340
M.determinant	330	90	3385	1032
M.isSingular	338	98	4023	1590
M.multiply	321	83	4071	1729
M.minor	298	62	5571	3198
M.subtract	312	76	3679	1340
M.transpose	298	62	3322	1066
L.contains	269	84	2867	582
L.distanceFrom	265	76	2833	544
L.intersection	350	157	3698	1584
L.intersects	326	133	3768	1385
L.pointClosest	394	210	5501	2973
L.rotate	359	236	4621	2151
				Whole Library LOC: 1417

medium-sized web applications, the first two being standard web applications, and six being openly available smaller demo applications (examples of UI controls). None of the applications communicates asynchronously with the server side – all behavior is realized on the client-side. All evaluated applications, except the "checkbox-radioBox" application, use the jQuery library, the most wide-spread library for simplifying client-side scripting [12]. jQuery is a complex library, with about 9000 LOC, and provides functionality for simplifying work with multiple browsers, selecting DOM elements, animations, etc. For each test application, Table 3 shortly describes the interaction we record.

*Example.*The process will be illustrated with an example application that we have developed for another project – http://www.idt.mdh.se/pride/ (Figure 3). Apart from the presentational aspects this web page also has one implemented behavior: in the central part there exist a UI control with two buttons. When the user clicks on a button, the current image and caption change with a fade effect. We will use this example with a slight modification for both usages.

Extracting UI controls

The result of the UI control extraction is a web page which contains only the selected UI control with all necessary code and resources required for its stand-alone functioning. In this evaluation, from the eight web applications we were able to successfully extract eleven stand-alone UI controls.

Table 3: Web applications

web page	*Target behavior*
idt.mdh.se/pride	Go through all items by clicking
druckbar.info	1. Wait for all image replacement 2. Mouse hover over a single item
accordion	Cycle through all items by clicking on each header
slider1	Go through items by clicking
slider2	Go through items by clicking
humanTypist	Wait page load, effect finish
checkbox radioBox	Click two times on each checkbox click once on each radio box
iPhone buttons	Click on enable, disable; on, off

Example. In the example application (Figure 3), the developer demonstrates the following behavior: he/she selects the HTML node encompassing the whole UI control, waits for the page to load, clicks on the second button and waits for the image-caption change effect to finish. When this is done, the developer clicks on the first button, and also waits for the end of the effect. When analyzing this execution trace, code dependencies are traversed and important code constructs marked when the structure of the selected HTML node (or the structure of its descendants) is being modified. The end result is the UI control from Figure 4.

Dead Code Removal

In the case of removing dead code, Table 4 shows the experimental results gained when removing dead code based on the behaviors shown in Table 3. In addition to the information described in the introduction of this section, for this set of data, we also present gains achieved in terms of page loading time (GPLT), which is measured by loading an uncached page from the local machine. This was done in order to reduce the influence of the connection speed on the end results. Loading time is an important piece of information because the longer the page loading time, the higher the probability that the user will abandon the web page [2]. As can be seen, considerable savings can be achieved in terms of page loading time (25,5% on average), and LOC (33,5% compared to executed LOC, or even 73,8% if compared with the original

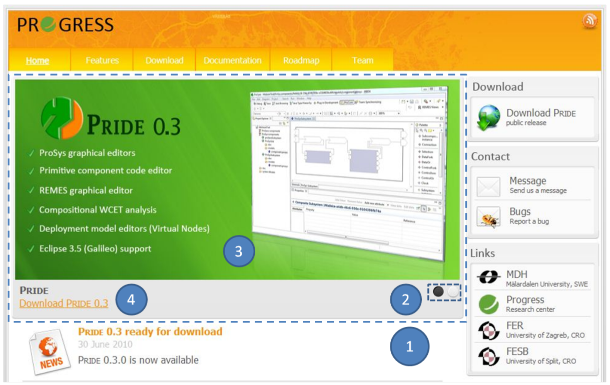

Figure 3: Pride home page; 1 – the dashed blue square marks the UI control, 2 – clickable buttons, 3 – image, 4 – captions

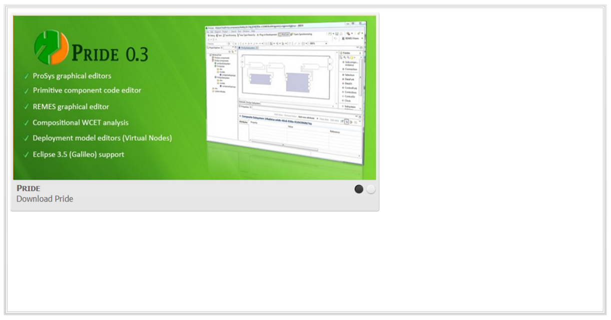
PRIDE 0.3
ProSys graphical editors
Primitive component code editor
REMES graphical editor
Compositional WCET analysis
Deployment model editors (Virtual Nodes)
Eclipse 3.5 (Galileo) support
PRIDE
Download Pride

Figure 4: The UI control extracted from the Pride home page

LOC), while still being able to reproduce the behavior of the web page. These results indicate that, in general, web applications contain more code than is actually needed by their behavior, and that considerable savings could be achieved by applying this extraction method. However, in order to strongly claim this fact, a more extensive web application test suite will have to be created.

Example. The functionality described in the previous section (*Extracting UI controls*) actually captures all complex behaviors in the example application (Figure 3), and can be used for dead code removal. The functionality is realized with the jQuery library. However, since this is not a large application, it only uses a subset of the library. This means that every time, when a user accesses the page, unnecessary code is transfered and interpreted in the browser. This leads to slower pages and increases web server traffic.

When re-executing the behavior described in the *Extracting UI controls* section, we get the following data (Table 4, first row): application behavior is implemented with 5730 JavaScript LOC (5610 jQuery, and 120 in the home page), and the average loading time is 268,9 ms[2]. Out of these 5730 LOC, in this scenario, the web application control goes through 2059 LOC. After the extraction process is finished, we end up with only 1458 LOC. This decrease in the total LOC, also decreases the loading time to 220,3 ms (18,7% faster loading time).

5. RELATED WORK

The work that we have presented in this paper is closely related to program slicing, which is defined by Weisner [13] 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. Static slicing can be difficult, and can lead to slices that are larger than necessary, especially in the case of pointer usage (e.g. in C). Further research has lead to development of dynamic slicing [1] 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 Weisner's form, a Program Dependence Graph (PDG) [5] where it shows both data and control dependencies for each evaluated expression, or a System Dependence Graph (SDG) [6] 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 [7].

In the web domain Tonella and Ricca [11] 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.

There are also two tools that facilitate the understanding of dynamic web page behavior: Script InSight [8] and FireCrystal [10]. 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. Compared to our approach they make no attempts to track data dependencies between different code expressions, nor to extract the analyzed code.

In the more general domain of Java applications, G&P [4] is a reuse environment composed of two tools: Gilligan and Procrustes, that facilitates pragmatic reuse tasks. Gilligan allows the developer to investigate dependencies from a desired functionality and to construct a plan about their reuse, while Procrustes automatically extracts the relevant code from the originating system, transforms it to minimize the compilation errors and inserts it into the developer's system. This work was further expanded [3] with the possibility to automatically recommend elements to be reused based on their structural relevance and cost-of-reuse.

This work is the continuation of our previous work [9] where we have shown how web application UI controls can be reused. However in that work, for relating code and behavior we have used only the executrion trace data – all visited lines have been extracted, and there were no attempts to identify data dependencies between code constructs. As our experiments indicate, by also taking data dependencies into account, we are able able to achieve additional savings - from 5.3% to 64.6% in the number of code lines, and from 22,5% to 57,5% in the number of executions.

6. CONCLUSIONS AND FUTURE WORK

In this work we have shown how web application code implementing a certain behavior can be extracted from the whole web application code by dynamically analyzing application execution traces. We have demonstrated how, even in this highly dynamic, multi-paradigm, multi-language environment of the web application client side, dependencies can be tracked by constructing a client-side dependency graph, and how by using that graph only the code responsible for

[2]averaged on 10 uncached, local page loadings, on an Intel Core i7, 1.73 GHz

Table 4: Experimental results on dead code removal from the test applications: LOC (Original LOC), ExLOC (Executed), ExtLOC (Extracted), GPLT (page loading time gains), GLOC (extracted vs executed code gains)

Page	*LOC*	*ExLOC*	*ExtLOC*	*GLOC(vsOrig)*	*GLOC(vsExe)*	*GPLT*	*ExEXE*	*ExtEXE*	*GEXE*
pride	5730	2059	1458	74,6%	29,2%	18,7%	52374	39137	25,2%
druckbar	5694	1798	1305	77%	27,4%	12,8%	38564	29232	24,2%
accordion	6214	2315	1833	70,5%	20,8%	23,5%	68669	49100	28,5%
slider1	5656	1897	1092	80,7%	42,4%	30%	44793	29779	33,5%
slider2	5855	2233	1451	75,2%	35%	23,4%	28734	17226	40%
humanTypist	5686	1338	473	91,7%	64,6%	54,6%	21446	9122	57,5%
check, radioBox	85	56	53	37,7%	5,3%	7%	1214	941	22,5%
iPhone buttons	5627	1687	957	83%	43,2%	34,3%	27531	14929	45,8%

a certain behavior can be extracted. We have shown how this extraction process can be used for different purposes: extracting library code, extracting UI controls, and removing dead code. Based on these three applications we have evaluated the approach and have reached two conclusions: *i)* the performed method can correctly extract stand-alone behaviors by analyzing web application traces, and *ii)* considerable savings in terms of the number of executions, page loading time, and code size, can be achieved while still being able to reproduce the demonstrated behavior.

For future work we plan to extend the process to also support the analysis of server side code and resources. The client-side and the server-side of the web application, even though based on completely different paradigms, are parts of the same whole, and should be studied together. Web applications are based on a request-response paradigm: a request is sent to the server, the server processes it, accesses some data (stored in files, or in a database) and creates a response that is sent to the client. The response is usually a combination of HTML, CSS, and JavaScript that will be executed in the web browser (the client-side of the application). This is why, in order to enable complete behavior extraction, we have to extend the process to track dependencies between the generated, client-side code, and the code and resources responsible for its generation on the server-side. Also, in this work we have considered only behaviors that result with a UI modification, but there are also behaviors that result with a message exchange with the server-side (downloading data, or sending information to the server-side), so the process has to be expanded to support them, too.

We are in the process of setting up a more extensive test suite of web applications to test that the extracted code always correctly implements the desired behavior. Also, the currently performed evaluation can not be used to strongly claim that web applications contain much more code than is actually needed by their behavior. This is why the evaluation is being set up to generally determine how much code, on average, is unnecessarily included in web applications.

7. REFERENCES

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

[2] 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/".

[3] R. Holmes, T. Ratchford, M. Robillard, and R. J. Walker. Automatically Recommending Triage Decisions for Pragmatic Reuse Tasks. In *International Conference on Automated Software Engineering*. IEEE Computer Society, 2009.

[4] R. Holmes and R. J. Walker. Semi-Automating Pragmatic Reuse Tasks. In *International Conference on Automated Software Engineering*, pages 481–482. IEEE Computer Society, 2008.

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

[6] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 23:35–46, June 1988.

[7] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *International conference on Software engineering*, ICSE '96, pages 495–505, Washington, DC, USA, 1996. IEEE Computer Society.

[8] P. Li and E. Wohlstadter. Script insight: Using models to explore javascript code from the browser view. In *International Conference on Web Engineering*, ICWE '9, pages 260–274, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] J. Maras, M. Štula, and J. Carlson. Reusing web application user-interface controls. In *International conference on Web engineering*, ICWE'11, pages 228–242, Berlin, Heidelberg, 2011. Springer-Verlag.

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

[11] P. Tonella and F. Ricca. Web Application Slicing in Presence of Dynamic Code Generation. *Automated Software Engg.*, 12(2):259–288, 2005.

[12] W3Techs. Usage statistics and market share of javascript libraries for websites, October 2011. w3techs.com/technologies/overview/javascript_library/all/.

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