Client-side Web Application Slicing

Josip Maras
Department for Modelling and Intelligent Systems
University of Split
Split, Croatia
josip.maras@fesb.hr

Jan Carlson, Ivica Crnković
Mälardalen Real-Time Research Center
Mälardalen University
Västerås, Sweden
{*jan.carlson, ivica.crnkovic*}*@mdh.se*

***Abstract*—Highly interactive web applications that offer user experience and responsiveness of standard desktop applications are becoming prevalent in the web application domain. However, with these clear benefits come certain drawbacks. For example, the event-based architectural style, and poor default support for code organization, often lead to a condition where code responsible for a certain behavior is intermixed with irrelevant code. This makes development, debugging and reuse difficult. One way of locating code implementing a certain behavior is program slicing, a method that, given a subset of a program's behavior, reduces the program to a minimal form that still produces that behavior. In many domains, program slicing is a well researched topic, but this is not the case in web application development. In this paper we present a semi-automatic client-side web application slicing method and the accompanying tool that can be used to facilitate code reuse and debugging of client-side web applications.**

***Keywords*-web application, dynamic program slicing, JavaScript, code reuse, debugging**

I. Introduction

The web application domain is one of the fastest growing and most wide-spread application domains today. Web applications usually do not require any installation, are easily deployed and updated, and apart from a standard web browser put no special constraints on the client. In the beginning they were relatively simple, but recently applications such as Facebook, Gmail, Google docs, etc. have set new standards in terms of what users expect from a web application. Web developers now routinely use sophisticated scripting languages and other client-side technologies to provide users with rich experiences that approximate the performance of desktop applications [17].

Web page structure is defined by HTML code, presentation by CSS (Cascading Style Sheets) code, and behavior by JavaScript code. Alongside code, a web page usually contains resources such as images, videos, or fonts. The interplay of these basic elements produces the end result that is displayed in the user's web browser. Visually and functionally, a web page can be viewed as a collection of distinctive user-interface (UI) elements that encapsulate a certain behavior. Unfortunately, this behavioral and visual distinctiveness is not mapped to neatly packed code units, since there is no predefined way of organizing related code and resources into some sort of components. This often leads to a situation where the web application code is hard to understand: there is no trivial mapping between the source code and the page displayed in the browser; code is usually scattered between several files and code responsible for one functionality is often intermixed with irrelevant code. This makes code maintenance, debugging, and reuse difficult.

Program slicing [16] 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. The technique has found many applications in the areas of program differencing and integration [5], software maintenance [3], and simple code reuse [9]. Even though this is a well researched topic in many domains, it has not yet been applied to client-side web development. Up to recently, these applications were considered trivial, and there was no need for advanced software engineering methods. This assumption is no longer true because modern web applications such as Google Calendar, Grooveshark, 280 Slides, etc. even surpass the complexity of standard applications. Also, most of the research has addressed strongly typed compiled languages such as C or Java, and the same techniques cannot easily be applied to a weakly typed, dynamic scripting language such as JavaScript.

In this paper we present a technique, and the accompanying tool, for client-side web application slicing. Our main contribution is a dynamic program slicing method based on the analysis of recorded execution traces. We present a client-side web application dependency graph, describe how it is constructed, and show how it can be used to slice a web application based on a slicing criterion. We have used the described process to support debugging and enable reuse of web application code.

This paper is organized as follows: Section II further describes the background of the problem, Section III describes the overall approach, and Section IV gives details about the dependence graph. Section V describes the slicing and its use in the web application development, and Section VI describes the developed tool suite and technical challenges we had to solve. Section VII shows how we have evaluated the approach, and Section VIII presents the related work. Finally Section IX gives a conclusion.

II. Background

In web application development, there is only a rudimentary level of code separation: JavaScript and CSS code can be moved to separate files, but in general, all web page code is included through a single file which contains HTML code. JavaScript and CSS code can be included either directly (by embedding the code inside an HTML node) or indirectly (by giving a link to a code file through special HTML nodes), but in a way, all code is included as HTML node content.

JavaScript primer. JavaScript is a weakly typed, imperative, object-oriented script language with delegation inheritance based on prototypes. 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 just like other objects. They are also variadic, meaning that they can receive any number of parameters. An object is simply a set of properties and associated values. Each object has an internal prototype property which refers to another object. That object can also have a reference to its own prototype, forming a prototype chain. The prototype chain is used to implement inheritance and shared properties – a property lookup involves searching the current object and its prototype chain, until the property is found. JavaScript is extremely dynamic: everything can be modified, from fields and methods of an object to its prototype. This makes the object model extremely flexible and it is difficult to constrain the behavior of any given object. As many other script languages, it offers the *eval* function which can take an arbitrary string of JavaScript code and execute it.

Web application execution style. Client-side web applications are mostly event-driven UI applications, and most of the code is executed as a response to user-generated events. The lifecycle 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 – Document Object Model (DOM) [14]. When parsing the HTML code the DOM is filled one HTML node at a time. If the browser reaches an HTML node that contains JavaScript code, it stops with the DOM building process and enters the JavaScript interpretation process. In this phase this means sequentially executing the given JavaScript code. Some of that code can register event-handlers. Once the JavaScript code in that node is executed, the code interpretation stops and the process again enters the DOM building phase. After the last HTML node is parsed and the whole UI is built, the application enters the event-handling phase – all code is executed as a response to some event. 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.

III. The Client-side web application slicing process

The dynamic nature of JavaScript, accompanied by the event-driven application model, makes it very difficult to statically determine the control-flow and data dependencies in a web application. Therefore, we have taken a more dynamic approach, basing the analysis on application execution traces.

The main idea is that, while a user initiates a series of interactions that correspond to the desired application behavior, in the background, we record the flow of application execution. In that way we establish the control-flow through the application. Identifying data-dependencies between code construct requires total knowledge about the state of the application at every execution point. Unfortunately, it is extremely impractical to gather and store this much detailed information during the recording phase for any but the most trivial programs. For this reason we have developed a custom JavaScript interpreter which in the analysis phase interprets JavaScript code based on recorded application execution traces. In this way the state of the simulated execution completely matches the state of the application at recording time. Unlike standard interpreters, this interpreter not only evaluates code expressions, but also keeps track of code constructs that are responsible for current values of all identifiers – information which is of vital importance for constructing data dependencies between code constructs.

Once the dependencies between code constructs have been established during the execution trace guided interpretation, the application code is sliced based on a slicing criterion specified by the user. In the end, after the slicing process is finished, only the code influencing the slicing criterion will be extracted from the whole application code.

IV. Client-side web application dependence graph

A basis for any slicing algorithm is a dependence graph which explicitly represents all dependencies between program's code constructs. The client-side web application code is composed of three parts: HTML code, JavaScript code, and CSS code. CSS code is used by the browser to specify rendering parameters of HTML nodes, and can be safely ignored from the perspective of application behavior. The remaining two code types: HTML and JavaScript code are important. The HTML code is used as a basis for building the DOM, while all functionality is realized and all UI updates are done by JavaScript interactions with the web page's DOM. The JavaScript code and the DOM are intertwined and must be studied as a part of the same whole.

In order to ease code manipulation, we do not work directly with code strings, but with an object representation of the source code. The code model is basically a simplified Abstract Syntax Tree (AST) and facilitates code construct manipulation in terms of the expression type it represents,

the position in the source code, and structural relationships with other code constructs (the JSON [8] code model of the source code given in Listing 1 is shown in Listing 2). Its structure is derived from the JavaScript grammar [7].

```
/*1*/var boxElm=document.getElementById("box");
```

Listing 1. Source code example

Although a lot more verbose then the source code from which it was generated, the code model facilitates working with and creating relationships between code constructs.

```
{ "type":"variableDeclarationStatement",
"stLn":1,  "vars":[{"type":"varDecl","stLn":1,
"name":"boxElm", "value": { "type": "init",
"stLn": 1, "value": {"type": "call", "stLn": 1,
"expr": { "type": "memberExpr", "stLn": 1,
"mainExpression": { "type": "identifier", "stLn":
1, "id": "document", "enLn": 1},"suffixes":
[{ "type": "propertyReferenceSuffix",
"stLn": 1, "identifier": "getElementById","enLn":
1}], "endLn": 1}, "args": {"type": "args","stLn":
1, "args": [{"type": "stringLiteral","stLn": 1,
"value":"ImJveCI=","quotation": "double","endLn":
 1,}], "endLn": 1}, "suffixes": [], "endLn":
1},"endLn": 1},"endLn": 1}], "endLn": 1}
```

Listing 2. Code model

A. Graph description

In our approach, the web application dependence graph is composed of two types of nodes: *HTML nodes* and *JavaScript nodes*, and three types of edges: *control flow* edges denoting the flow of application control from one node to another, *data flow* edges denoting a data dependency between two nodes, and *structural dependency* edges denoting a structural dependency between two nodes.

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 HTML nodes. A directed structural dependency edge between two HTML nodes represents a parent-child relationship from a child to the parent. A dependency graph subgraph composed only of HTML nodes matches the DOM of the web page.

JavaScript nodes (JS nodes) represent code construct that occur in the program and each node contains a reference to the matching code object. All JavaScript code is contained either directly or indirectly in an HTML node, so each JS node has a structural dependency towards the parent HTML node. Two JS 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). Data-flow edges can exist either between two JS nodes, or between a JS node and a HTML node. A data dependency from one JS node to another denotes that the former is using the values that were set in the latter. A data dependency edge from a JS node to an HTML node means that the JS node is reading data from the HTML node, while a data dependency from the HTML node to the JS node means that the JS node is writing data to the HTML node. Table I 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 and J denote HTML nodes and JavaScript nodes, and N denotes a node of arbitrary type.

Table I
EDGE IN THE DEPENDENCE GRAPH

Edge	*Condition*
$N_1 \xrightarrow{s} N_2$	N_1 is a child of N_2.
$H \xrightarrow{d} J$	J writes data to H.
$J \xrightarrow{d} H$	J reads data from H.
$J_1 \xrightarrow{d} J_2$	J_1 reads data that was set in J_2.
$J_1 \xrightarrow{c} J_2$	J_2 is executed after J_1.

B. Building the dependence graph

As an input the dependency graph construction algorithm receives the HTML code of the web page, the code of all included JavaScript files, and a recorded execution trace. The graph construction algorithm mimics the way a browsers builds a web page. For each encountered HTML node it creates a matching HTML graph node, and when it reaches a HTML node that contains JavaScript code (the script node), it switches to the creation of code construct nodes. First, the containing JavaScript code is parsed, and its code model built. The process then enters the execution-trace guided interpretation mode where code nodes are created as each code object representing a certain code expression is evaluated. This means that one code object will spawn one node for each time it is interpreted. Even thought this greatly increases the number of created nodes, it simplifies the code slicing algorithm. 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 mimicking phase. 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 also created for each code construct executed as a part of the event-handler code.

```
/*01*/<html>
/*02*/ <head></head>
/*03*/<body>
/*04*/ <div id="box" class="red"></div>
/*05*/ <script>
/*06*/ var boxElm=document.getElementById("box");
/*07*/ boxElm.onclick = function()
/*08*/ {
/*09*/   if(boxElm.className == "red")
/*10*/     boxElm.className = "blue";
/*11*/   else
/*12*/     boxElm.className = "red";
/*13*/ };
/*14*/ </script>
/*15*/ <p></p>
/*16*/</body>
/*17*/</html>
```

Listing 3. Example application

C. Example

We will illustrate the graph building with an example shown in Listing 3. The UI of this simple application is composed only of a single red square, built from a div HTML element with the id *box*, which when clicked changes its color from red to blue (by changing the CSS class of the element). In the JavaScript code an HTML element object defining the square is obtained by calling the *getElementById* method of the document object (which exposes the DOM of the web page). Then, in the second code line, by setting the value of the *div* element's onclick property to a function expression, we have registered an event handler function that will be executed each time the *div* element is clicked on. On each click the function checks the value of the class attribute of the *div* element, and changes it if necessary.

The example uses a recorded execution in which the user opened the web page and performed a single click on the square, changing its color from red to blue.

Figure 1 shows a built dependency graph for Listing 3. The graph contains HTML nodes (circles) and JS nodes (squares). Each node is indexed at creation time. Instead of showing the control-flow edges (which would clutter the figure), the control-flow path can be traced by following the indexes. Structural dependencies edges are shown with dashed arrows, and data dependencies edges with solid arrows. To reduce the overall number of connections in the figure, all nodes contained in a single code line are framed with a dashed boxes, and a dashed arrow between that box and a node shows that a structural dependency exists between each node in the dashed box and the target node. The graph building starts with parsing of HTML code and creating the *html*, *head*, *body*, *div*, and *script* nodes. The script node contains JavaScript code, so the HTML parsing stops and the process enters the JavaScript execution-trace guided interpretation phase, where JS nodes are built as expressions are evaluated. For each executed code construct a separate node is created. E.g. the execution of line 06: "var boxElm = document.getElementById('box')", creates the nodes labeled with "@6" in Figure 1. Since there is no more JavaScript code for sequential execution, the process again enters the DOM building phase, and creates the *p* HTML node (indexed with 15). After that, the final HTML node is created, page initialization is complete, and the process enters the event-handling phase. In the recorded execution one event-handler is executed (the function expression at line 7). When the event handler is executed two dependencies are created: a dependency between the event handling and the HTML node raising the event (an edge between node 16 and node 4), and the dependency between the event handler and the event-registering node (a dependency edge between node 16 and 14). The process then enters the JavaScript interpretation mode and builds graph nodes based on code constructs executed in the event-handler.

V. Client-side web application slicing

Program slicing is always performed with respect to a slicing criterion. In Weisner's original form a slicing criterion is a pair $\langle i, v \rangle$, where i is the code line number, and v is the set of variables to observe. In dynamic slicing [1], the definition is a bit different and is defined with respect to execution history as a triple $\langle i, k, v \rangle$ where i is the code line number and k is the ordinal number marking after how many executions of the i-th code line we want to observe a set of variables v.

```
/* Mark important code constructs */
foreach step in executionTrace
    interpretStepAndExtendDepGraph(step);
    if isSlicingCriteria(step)
        nodes = ∅
        foreach id in getIdentifiers(step)
            addElement(nodes, getLastCodeNode(id))
        while not empty(nodes)
            topNode = removeElement(nodes)
            mark(getCodeConstruct(topNode))
            deps = getDataDependencyNodes(topNode)
            addElements(nodes, deps)
            markAllStructuralAscendants(topNode)

/* Generate code for output */
foreach codeConstruct in applicationCodeModel
    if(isMarked(codeConstruct)
        outputSourceCode(codeConstruct)
```

Figure 2. Slicing Algorithm

The slicing algorithm is given in Figure 2 in pseudo code.The main idea is that in the recorded execution traces, some of the execution steps are marked as slicing criteria. When the interpreter reaches one of them, it halts the code interpretation and enters the code marking mode. By starting from the set of code nodes that match the observing variable

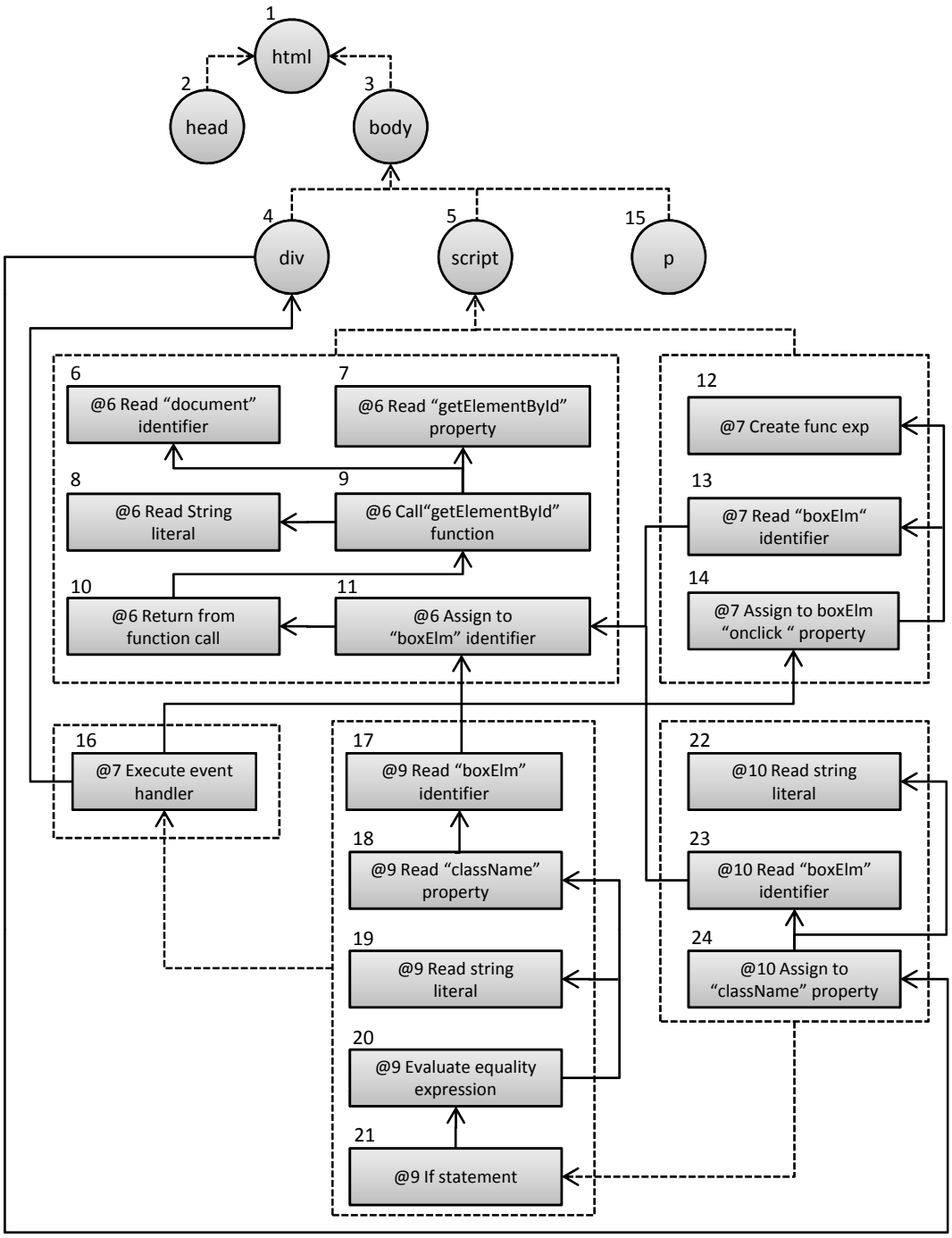

Figure 1. An example web page dependency graph: Circles represent HTML nodes, squares JavaScript code nodes. Dashed arrows represent structural dependencies, while solid arrows represent data dependencies. Dashed boxes encompass code nodes contained in the same line, and a connection between a dashed box and a node shows that a structural dependency exists between every node in the dashed box and the target node.

identifiers, it traverses the dependence graph by following data-dependency edges. The code construct belonging to each traversed node is marked as important and will be included in the final slice. Structural dependencies of each traversed node are also studied, and all structural ascendants of the traversed node are also marked as important. In the end, when all slicing criteria have been processed, and all relevant nodes have been marked, the final program slice is generated by traversing the whole code model and including only the marked code constructs.

Program slicing has found many applications, especially in program understanding and debugging. Here we show how it can be used for extracting library code, UI control reuse and debugging in client-side web application development. We only give a high-level description of library code extraction and debugging because these applications are mostly focused on choosing the appropriate slicing criteria. The UI control reuse is more complex, so we go into more details about it.

A. Extracting library code

When encountering problems that have been solved in the past, developers often perform reuse tasks [2] rather than reinventing the wheel. The most common way of code reuse is by using code libraries. In many application domains this is not a problem – the overall code size does not have a significant impact on the performance of the application. But, in client-side web application development, all code is transfered and executed on the client and larger code bases lead to slower web pages. This means that in general, we want to transfer the minimum amount of code to the client. This is often a problem when using JavaScript libraries, because often only a part of library code is used, and we transfer unnecessary kilobytes of code. One of the advantages of our web application slicing method is that it can be used to extract functionality from large libraries. By creating a suite of thorough unit tests and making function return values as slicing criteria, we can record a representative application execution trace which can be used for extracting the desired functionality from the library.

B. Extracting UI controls

Web application UI is composed of visually distinctive UI elements that encapsulate a certain behavior – the so called UI controls. Similar UI controls are often used in a large number of web applications and facilitating their reuse could offer significant benefits in terms of faster development. Each UI control is defined by a combination of HTML code, CSS code, JavaScript code and different resources (images, fonts, etc.). In order to extract an UI control, from the code and resources of the whole web application, we have to locate only the code and resources that are required for the stand-alone functioning of the UI control. This is a difficult and time consuming task. We have already published work in this area [11], where the extraction was based on application profiling, and here we expand it with a precise slicing algorithm.

The UI control extraction is also based on recorded application execution traces. The process is semi-automatic and starts with the user selecting the HTML node that defines the UI control. Then, the recording phase is initiated, the page reloaded, subscriptions to the DOM mutation events [15] registered, and the initial state of the UI control logged. The initial state is composed of code executed while initializing the control, and styles and resources that initially define the control. Generally, all executed code is logged by communicating with the JavaScript debugger service, which provides hooks (or events) that activate on each execution and give information about the currently executed source code lines. In order to obtain the styles and resources that initially define the UI control, the DOM of the control is traversed and all CSS styles and resources applied and used in the control are logged. With this, a log of resources that initially define the control is obtained.

After the UI control is fully loaded, the modifications of the control are caused by user interactions and/or timing events. The executed code is again logged by communicating with the JavaScript debugger service, while any dynamic changes in styles and resources are logged by handling DOM mutation events. The advantage of handling the DOM mutation events is that by observing which code construct was executed before the event was fired we can locate the DOM modifying code constructs – code constructs that are responsible for updating the UI of the control. By setting the DOM modifying code constructs as slicing criteria, we can slice the whole web application code and gain only the code that is responsible for the behavior of the UI control. Using this approach we are able to locate all code and resources that define the control: *i)* HTML code, because the user directly selects the HTML node defining the control; *ii)* JavaScript code, because by slicing the web application code with DOM modifying constructs as slicing criteria we locate only the code responsible for the behavior of the user-control; *iii)* CSS code; and *iv)* resources, because styles and resources applied to the control during the the whole course of the execution are logged.

C. Debugging

One of the problems when debugging applications is determining which code statements have influenced the erroneous value of some variable. This is even harder in web application development because of the event-based execution paradigm, specifics of the JavaScript language, and the close integration of DOM and JavaScript source. By selecting the execution that contains the erroneous value as a slicing criterion, we can generate a program slice which will contain only the statements that lead to the bug.

D. Example

If we again take the example given in Listing 3, select a simple user-control defined by the *div* element and record the same use-case as in the example given in the graph building section, by handling the DOM mutation events we get that the slicing criterion is the assignment expression in line 10 (because it modifies the class attribute of the HTML node). By executing the slicing algorithm on the dependence graph given in Figure 1 we get the web application slice given in Listing 4. Compared to the original example, the DOM tree does not include the *p* element (because it is outside of the *div* element and it is not used by any important code nodes), and the JavaScript code does not have the else statement (lines 11,12 in Listing 3) because it was not used in the recorded application trace.

```
/*01*/<html>
/*02*/ <head></head>
/*03*/<body>
/*04*/ <div id="box" class="red"></div>
/*05*/ <script>
/*06*/ var boxElm=document.getElementById("box");
/*07*/ boxElm.onclick = function()
/*08*/ {
/*09*/   if(boxElm.className == "red")
/*10*/     boxElm.className = "blue";
/*11*/ };
/*12*/ </script>
/*13*/</body>
/*14*/</html>
```

Listing 4. Example web app slice

VI. Tool suite

In order to realize web application slicing we have built a tool suite consisting of three applications: *i)* a proxy web server, *ii)* Firecrow, a plugin to the Firefox web browser which records application execution traces, and *iii)* a Web code slicer.

A proxy web server was developed in order to modify JavaScript code for easier analysis. A big issue in most web applications is code size. Larger code size means slower page loading. Most web developers minimize their code by removing unnecessary whitespace characters. This can lead to situations where a code library with around 10 000 lines of code gets minimized to less than 50. Normally this is not a problem, but the application execution trace is recorded by communicating with the JavaScript debugger, which only gives trace information on line level. Therefore, if a single line contains many code constructs it is hard to deduce a meaningful control-flow. The proxy web server intercepts the requests for HTML and JavaScript files and formats the contained JavaScript code in a way that enables meaningful control-flow tracing based on the information that can be obtained from the debugger.

Firecrow, shown in Figure 3, is a plugin for the Firebug[1] web debugger. Currently, it can be used from the Firefox web browser, but it can be ported to any other web browser that provides communication with a JavaScript debugger, and a DOM explorer. Firecrow enables the recording of application traces and is responsible for gathering of all information that is necessary in the process of slicing. It supports two modes: extraction of UI controls mode and code extraction/debugging mode. In the extraction of UI controls, it enables the user to visually select the HTML node that represents the UI control, and in addition it gathers information responsible for the visuals of the UI control (used images, CSS styles, etc.). In the code extraction/debugging phase it shows an overview of the recorded application execution trace, and enables the user to select executions which will be used as a slicing criteria.

Web code slicer is standard Java application composed of several modules: JavaScript, CSS, and HTML parsers, JavaScript interpreter, and the slicer itself. All are custom made, except for the HTML parser, which is an open source program.

The tool suite can be downloaded from the web page: http://www.fesb.hr/~jomaras/?id=app-Firecrow

VII. Evaluation

We have evaluated our approach by extracting functionalities from an open-source vector and matrix math library – Sylvester[2]. It includes many useful functions for working with vectors, matrixes, lines and planes. As with any other library, if we only want to 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 libraries homepage 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 2000 lines of code we have extracted 20 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 for the extracted code.

Table II 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 (second column), the number of lines that were included in the final slice (third column) and the ratio between the number of executed and extracted code lines (fourth column). As can be seen, each use case executes around 10% of the total library code, and out of that executed code the slicing process extracts on average around 23%, which constitutes the parts of the code required to implement the wanted behaviour.

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

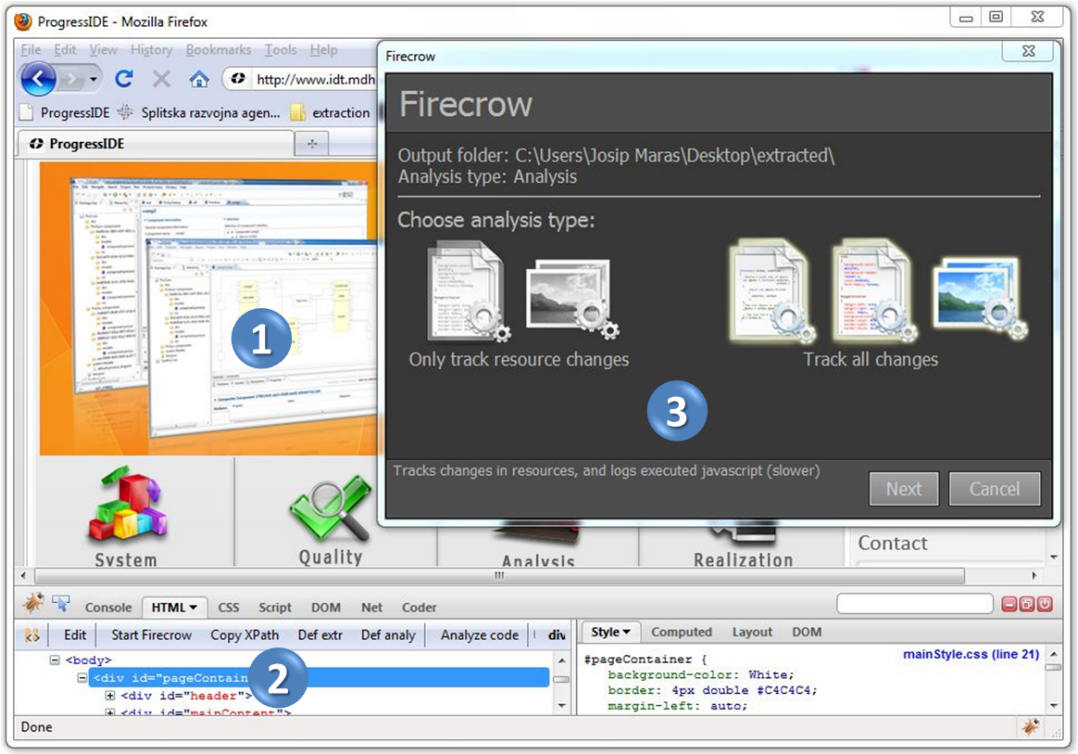

Figure 3. Firecrow plugin: Mark 1 shows the web page chosen for extraction, mark 2 Firebug's HTML panel used for selecting controls, and mark 3 Firecrow's UI

Table II
EXPERIMENTAL RESULTS ON EXTRACTING API FUNCTIONS FROM THE SYLVESTER MATH LIBRARY

Method name	*Executed LOC*	*Extracted LOC*	*Ratio*
Vector.cross	219	35	15%
Vector.dot	210	35	16%
Vector.random	216	35	16%
Vector.zero	211	35	16%
Vector.add	231	58	25%
Vector.dimensions	212	29	13%
Vector.distanceFrom	230	51	22%
Vector.isAntiparallel	245	66	26%
Vector.isParallelTo	247	69	27%
Vector.max	221	42	19%
Vector.modulus	211	42	19%
Vector.multiply	228	55	24%
Vector.rotate	253	93	36%
Matrix.diagonal	263	86	32%
Matrix.identitiy	254	72	28%
Matrix.rotation	239	55	23%
Matrix.zero	252	70	27%
Matrix.add	264	83	31%
Matrix.augment	258	80	31%
Matrix.isSquare	238	55	23%

The set of use cases and the accompanying code can be downloaded from the tool homepage:http://www.fesb.hr/~jomaras/?id=app-Firecrow.

VIII. RELATED WORK

Weisner [16] defines program slicing 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, Weisner's form, a program is sliced statically, that is 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) [6] where it shows both data and control dependencies for each evaluated expression, or a System Dependence Graph (SDG) [4] 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 [10].

In the web engineering domain Tonella and Ricca [13] 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. In the same work they present a technique for web application slicing in the presence of dynamic code generation where they show how to build a system dependency graph 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 also exists a tool – FireCrystal [12], an extension to the Firefox web browser that facilitates the understanding of dynamic web page behavior. It performs this functionality by recording interactions and logging information about DOM changes, user input events, and JavaScript executions. After the recording is over the user can use an execution time-line to see the code that is of interest for the particular behavior. In its current version, it does not provide a way to study which statement influence, either directly or indirectly, the statement that has caused the UI modification, and instead shows all statements that were executed up to a current UI modification.

IX. Conclusion and future work

In this paper we have presented a novel approach and the accompanying tool suite for slicing of client-side web applications. The process starts with the developer recording the execution of a set of use-cases that represent a behavior that is in accordance with some slicing goal. Based on the recorded execution-trace guided code interpretation, we have shown how a dynamic code dependency graph can be built, and how that same graph can be used to extract only the code relevant for a particular slicing criterion. Program slicing has many applications, and in this paper we have shown how it can be used in the areas of code reuse, UI control extraction, and debugging. The process has been evaluated by extracting functionality from an open-source JavaScript library. The evaluation has shown how instead of using full libraries, this process could be used only to extract the parts of the library that are actually used in the application. For future work, we plan to extend the slicing process to cover whole web applications, by expanding the process to include server-side slicing.

Acknowledgment

This work was supported by the Swedish Foundation for Strategic Research via the strategic research center PROGRESS.

References

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 246–256, New York, NY, USA, 1990. ACM.

[2] Joel Brandt, Philip J. Guo, Joel Lewenstein, and Scott R. Klemmer. Opportunistic programming: How rapid ideation and prototyping occur in practice. In *WEUSE '08: Workshop on End-user software engineering*, pages 1–5. ACM, 2008.

[3] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17:751–761, 1991.

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

[5] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 234–245, New York, NY, USA, 1990. ACM.

[6] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11:345–387, July 1989.

[7] ECMA international. ECMAScript language specification. "http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf".

[8] JSON. Json web page, May 2011. "http://www.json.org".

[9] F. Lanubile and G. Visaggio. Extracting reusable functions by flow graph based program slicing. *Software Engineering, IEEE Transactions on*, 23(4):246 –259, apr 1997.

[10] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *Proceedings of the 18th international conference on Software engineering*, ICSE '96, pages 495–505, Washington, DC, USA, 1996. IEEE Computer Society.

[11] Josip Maras, Maja Stula, and Jan Carlson. Reusing web application user-interface controls. In *International Conference on Web Engineering 2011*, ICWE '11. Springer, 2011.

[12] Stephen Oney and Brad Myers. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *VLHCC '09: Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 105–108. IEEE Computer Society, 2009.

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

[14] World Wide Web Consortium (W3C). Document Object Model (DOM), May 2011. "http://www.w3.org/DOM/".

[15] World Wide Web Consortium (W3C). Document Object Model Events, May 2011. "http://www.w3.org/TR/DOM-Level-2-Events/events.html".

[16] Mark Weiser. Program slicing. In *ICSE '81: 5th International Conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[17] Alex Wright. Ready for a Web OS? *Commun. ACM*, 52(12):16–17, 2009.