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 benefits come certain drawbacks. For example, the event-based architectural style, and poor support for code organization, often lead to a situation 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 this paper we present a semi-automatic client-side web application slicing method, describe the web page dependency graph, and show how it can be used to extract only the code implementing a certain behavior.**

***Index Terms*—JavaScript, code reuse, dynamic program slicing, web application**

I. Introduction

The web application domain is one of the fastest growing and most wide-spread application domains. 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. In terms of its structure, a web page is defined by HTML code, presentation by CSS code, and behavior by JavaScript (JS) 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 [7] is a method that, starting from a subset of a program's behavior, reduces that program to a minimal form that still produces that behavior. The technique has found many applications in the areas of program differencing and integration [3], maintenance [2], and simple code reuse [4]. Even though this is a well researched topic in many domains, it has not yet been applied to client-side web development. Until recently, these applications were considered trivial, and there was no need for advanced software engineering methods. This is no longer the case because modern web applications such as Gmail, Facebook, Gdocs, etc. even surpass the complexity of standard applications. Also, most of the research on program slicing 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 JS.

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 across different languages (HTML and JS) based on a given slicing criterion.

II. Background and Related Work

JS 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. JS is extremely dynamic: everything can be modified, 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 JS code.

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 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. If the browser reaches an HTML node that contains JS code, it suspends the DOM building process and enters the JS interpretation process. In this phase this means sequentially executing the given JS 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 JS 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 some event. All updates to the UI are done by JS modifications of the DOM, which can go as far as completely reshaping the DOM, or even modifying the code of the application.

Weiser [7] 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 the web engineering domain Tonella and Ricca [6] 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 application in terms of information of interest to the user. They also 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. Our work differs from their approach in that we target the client-side application, which even though is related to server-side application, is based on a different development paradigm, and cannot be treated equally.

There also exists a tool – FireCrystal [5], 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 JS 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.

III. The Client-side Web Application Slicing Process

Most of the current slicing techniques have been developed for handling sequential programs implemented in static languages such as Java or C, where the control-flow and data dependencies can be statically determined. Because of the extreme dynamicity of JS and the accompanying event-driven application model these methods cannot also be used in client-side web application development. Therefore, we have taken a more dynamic approach, basing the analysis on application execution traces.

In order to be able to slice the web application we have to determine the control flow and data-dependencies between code constructs. The main idea is that we record the flow of application execution, while the desired application behavior is being executed. 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 JS interpreter which in the analysis phase interprets JS 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 of vital importance for identifying 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. The choice of the slicing criterion depends on the slicing usage. For example, the slicing process can be used for extracting library code, UI control reuse and debugging.

A. Extracting Library Code

In many application domains the overall code size does not have a significant impact on the performance of the application. However, this is not the case in client-side web application development, where all code is transfered and executed on the client and where larger code bases lead to slower web pages. So, when using JS libraries, we only want to transfer the part of the library used by the application. By creating a suite of through unit tests and making function return values as slicing criteria, we can record a representative execution trace which can be used to extract the desired functionality from the library.

B. Extracting UI Controls

The web application UI is composed of visually distinctive UI elements that encapsulate a certain behavior – the so called UI controls. Similar controls are often used in a large number of web applications and facilitating their reuse could lead to faster web development. Each UI control is primarily defined with an HTML node that defines its structure and JS code that defines its behavior. Its main method of communicating with the user is by modifying the DOM of the associated HTML node. This means that in order to extract the UI control we have to slice the web application with DOM modifying expressions as slicing criteria. In order to be able to that, the process starts with the user selecting the HTML node that defines the UI control and executing the behavior of the UI control that he/she wants to reuse. In the recording phase, all DOM mutation events are logged (triggered when DOM of the UI control is modified) and executions that cause them are set as slicing criteria.

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 JS language, and the close integration of the DOM and JS code. 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.

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. The 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, JS 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 JS code are important. The HTML code is used as a basis for building the DOM, while all functionality is realized by JS interactions with the web page's DOM. The JS code and the DOM are intertwined and must be studied as parts of the same whole.

A. Graph Description

In our approach, the web application dependence graph is composed of two types of nodes: *HTML nodes* and *JS 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.

JS nodes represent code construct that occur in the program. All JS 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.

B. Building the Dependence Graph

As input, the dependency graph construction algorithm receives the HTML code of the web page, the code of all included JS 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. When it reaches a HTML node that contains JS code (the script node), it switches to the creation of JS nodes – the process enters the execution-trace guided interpretation mode where code nodes are created as each code construct is evaluated. Once the whole code file has been traversed, and all contained JS 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. JS nodes are also created for each code construct executed as a part of the event-handler code.

C. Slicing the Dependence Graph

Program slicing is always performed with respect to a slicing criterion. In Weiser'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 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.

The basic 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 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. The structural dependencies of each traversed node are 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.

V. Tool

In order to further investigate the proposed approach, we have developed two applications: *i)* Firecrow, a plugin to the Firefox web browser, built on top of the Firebug[1] web debugger, which records application execution traces, and *ii)* a Web code slicer. 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

[1]http://getfirebug.com

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 slicing criteria.

Web code slicer is a standard Java application composed of several modules: JS, CSS, and HTML parsers, JS 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

VI. Evaluation

We have evaluated our approach by extracting functionalities from an open-source vector and matrix math library – Sylvester[2], which includes functions for working with vectors, matrices, 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 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 only the code essential for the stand-alone functioning of the method is extracted. In all cases the method extraction was successful, and the use-case could be repeated for the extracted code.

Table I 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 behavior.

The set of use cases and the accompanying code can be downloaded from the Firecrow homepage.

TABLE I
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%

VII. 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 gave outlines on how it can be used in the areas of code extraction, UI control reuse, and debugging. The approach has been evaluated by extracting functionality from an open-source JS 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. We consider this an interesting fact, and in the future we plan to investigate how much library code is generally used in web applications by performing an empirical study. We also plan to extend the slicing process to cover whole web applications, by expanding it 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] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17:751–761, 1991.

[3] 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.

[4] 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.

[5] 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.

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

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

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