Characterizing the Control Logic of Web Applications' User Interfaces

Carlos Eduardo Silva and José Creissac Campos

Departamento de Informática, Universidade do Minho
& HASLab/INESC TEC
Braga, Portugal
{cems,jose.campos}@di.uminho.pt

Abstract. In order to develop an hybrid approach to the Reverse Engineer of Web applications, we need first to understand how much of the control logic of the user interface can be obtained from the analysis of event listeners. To that end, we have developed a tool that enables us to perform such analysis, and applied it to the implementation of the one thousand most widely used Websites (according to Alexa Top Sites). This paper describes our approach for analyzing the user interface layer of those Websites, and the results we got from the analysis. The conclusions drawn from the exercise will be used to guide the development of the proposed hybrid reverse engineering tool.

Keywords: Web applications, user interfaces, reverse engineering

1 Introduction

Reverse Engineering [1, 2] is the process of moving from more concrete representations of a system, typically its source or binary code, to more abstract representations of the same system. Doing this we gain knowledge about the design of the system, avoiding details that are relevant only to its implementation. Reverse engineering can be useful in a number of different situations, be it during development, testing or maintenance. Two basic types of techniques can be used. Static techniques start from the code (the source code or some form of byte-code) analyzing it statically. Dynamic techniques take a black box approach and analyze the behavior of the system while executing.

We are particularly interested in applying reverse engineering to Web applications. While development for the Web has changed how the software industry approached software development and distribution, it is recognized that applying adequate standards and methodologies is still a challenge [3]. Through reverse engineering we aim to provide support to the process of engineering and re-engineering Web applications. By helping the process of gaining an understanding of an existing system, it becomes possible to better support its testing and validation, or its maintenance and evolution. For example, by enabling the comparison of different versions of the same software.

More specifically, we are interested in the user interface layer of Web applications. This is both because they are typically more susceptible to changes (due to changes in user requirements or the technology), and are particularly hard to develop and maintain (due, not only to the event based nature of the code, but also the the wide variety of technologies available for that development).

Early attempts at applying static analysis techniques to analyze the user interface layer of Web applications [4], have shown that this type of technique, while feasible for native applications (see, for example, [5]), suffers from a number of shortcomings when applied to the Web. The two main issues are related to the existence of many alternative implementation technologies, on the one side, and to the on the fly generation of the user interfaces, on the other.

The fact that many different languages, toolkits and frameworks exist, which can be used to implement Web applications, and their user interfaces in particular, means that a static analysis approach will either have to be able to parse and interpret the different languages and/or toolkits/frameworks, or commit to a particular technology. The first solution is not viable, in practice, as it means that developing such an approach would be prohibitively expensive. The second, means that the utility of the approach becomes rather restricted. This is specially the case when one considers the fast pace of technological development in the field.

Dynamic analysis avoids the above problems by taking a black box view of the system under test, and exploring its behavior at run time. Here what is needed is some adapter layer to programmatically interact with the user interface. Frameworks such as Selenium [6] provide this layer for the case of Web applications, and have already been used in tools such as Crawljax [7].

The fact that dynamic analysis takes a black box view of the system under test, however, also poses some limitations. Most notably, apart from very simple user interfaces, it is not possible to be sure that all possible behaviors/states of the system have been explored. Additionally, when in presence of alternative system behaviors, it is not easy to determine what are the conditions that triggers each alternative behavior [8]. The danger, then, is that models obtained through a dynamic approach to reverse engineering will be incomplete, both regarding the full behavior of the system under test, and regarding the semantics of the observed behaviors.

To avoid these issues, we propose to use an hybrid approach to reverse engineering [9]. The approach combines dynamic analyzes of the application, with static analyzes of the source code of the event handlers found during interaction. Information derived from the source code is both directly added to the generated models, adding semantics to the model, and used to guide the dynamic analysis, thus enabling it to be more complete.

In order to avoid the pitfalls of static analysis, the goal is to keep the analysis of event handlers as simple as possible. This, however, begs the question of whether enough control logic is present in the event handlers of Web applications, and whether we can adequately identify and process the event handlers. To answer those questions, we have carried out a study of the top one thousand

most used Websites according to Alexa Top Sites[1]. In this paper we describe how the study was setup, and the results that were achieved.

The paper is organized as follows: Section 2 defines the criteria used in the analysis; Section 3 explains how information about event handlers was extracted from the Web pages, and Section 4 describes the tool which was developed; Section 5 describes the analysis of the Websites, and the paper concludes with Section 6.

2 Criteria for Analysis

Since the interest is in identifying possible alternative behaviors of the system, and the conditions under which these alternative behaviors occur, the focus of the static analysis are the conditions in if statements and loops in the event handlers.

The analysis of the Websites was thus made according to a set of criteria defined to help understand how much information could be obtained from the event handlers. The following criteria were defined:

- **Number of hrefs** – The analysis of how many hrefs are used in the page. A high number of hrefs and the failure to detect events handlers might lead us to infer the usage of Web 2.0 (i.e. static) techniques in the page.
- **Number of Events** – This criterion is the total number of events we were able to find on the Website.
- **Number of Click Events** - It is important to discern, from all the events in the page, those which are triggered by clicks, since they are easier for us to simulate. Moreover, only the visible click events are being considered in this criterion.
- **Number of Ifs** – This criterion is important since we need to measure how much used these constructs are, in comparison to others that affect the control flow of the application.
- **Number of other Control Flow constructs** – The other type of constructs related to the control flow we are analyzing, these include while/for loops and ternary conditional operators.
- **Number of Element Variables** – This criterion counts the number of variables we found, whose value is obtained from elements in the page. We are currently not distinguishing between input and non-input elements. Moreover, this analysis was performed by detecting usage of classic JavaScript *getElementBy* function calls only.
- **Number of Synthesized Variables** – Variables related to function calls that we could not ascertain were used to retrieve elements from the page.
- **Number of Object Variables** – Variables associated with objects or object properties.
- **Number of Global Variables** – Variables whose declaration is made outside the scope of the handler/function being analyzed.

[1] `http://www.alexa.com/topsites` (last accessed: May 8, 2014)

- **Number of Control Flow Variables** – Variables that might be assigned different types depending on the control flow structures.
- **Number of Hybrid Variables** – Variables that can be classified using more than one of the previously defined types, at different points in the code.
- **Event Delegation** – If our analysis infers the Website uses the event delegation approach for event handlers (see Section 3).

3 Extracting Web page behavior information

In order to combine dynamic analysis with the static analysis of event handlers, we must be able to identify those event handlers at run time. Identifying the event handlers in a page, depends on how those event handlers were added to the page in the first place.

There are two main approaches to add events to a Web page. The classic approach is to add an event handler to an element. However, even using this approach, there are several different ways of adding event handlers to the elements. A simple example is:

```
element.onclick = event_handler;
```

An event handler added in this way is retrieved simply by querying *element.onclick* in JavaScript. However, in our analysis, we soon discovered that most Websites use other methods for adding event handlers. For example:

```
element.addEventListener('click', event_handler, false)
```

The above source code is another option of adding an *onclick* event to an element. Using *element.onclick* in this case will not retrieve any results. Moreover, we also have to consider all the different JavaScript frameworks, for instance, in jQuery [10] the event handler is added as follows:

```
$(element).click(handler);
```

To address this we resorted to Visual Event, an open source framework[2] which is able to parse several JavaScript libraries and retrieve the event handlers. It currently works with a number of different libraries, and can be extended by developing new parsers for those not supported and adding them to the framework.

The other approach to adding event handling code to a Web page is called event delegation[3]. The idea is to take benefit of the browsers' event bubbling features. That is, when we have nested elements in HTML, triggering an event handler in an element also implies triggering the handlers for that event in the parent elements. For example, in Figure 1 on the left side we have the

[2] `http://www.sprymedia.co.uk/article/Visual+Event+2` (last accessed: May 8, 2014)

[3] `http://icant.co.uk/sandbox/eventdelegation/` (last accessed: May 8, 2014)

```
<html>
<head></head>
<body>
<div id"a">a
    <div id="b">b
        <div id="c">c
        </div>
    </div>
</div>
</body>
</html>
```

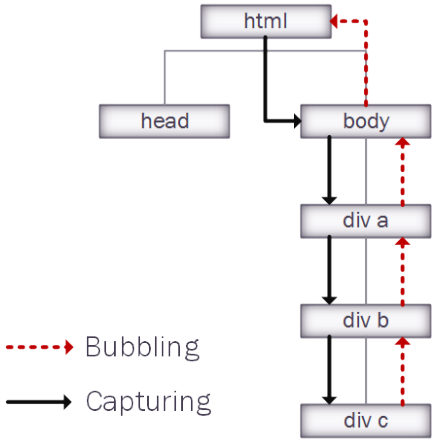

Fig. 1. Bubbling and Capturing Example

source code of a simple Web page with three *divs*. Triggering an event on the innermost element, that is, *div c* , causes the browser to traverse the Document Object Model (DOM) from the root node (*html*) to the element, this is called the capturing phase. Afterwards the browser also needs to traverse the DOM from the element to the root node, this is called the bubbling phase. Depending on the browser and how event handling is configured, relevant event handlers will be triggered in each element in either the capturing or bubbling phase.

This behavior of the Web browsers led to the usage of event delegation. That is, instead of assigning event handlers to each element, we assign a single handler to a parent container (the extreme case is to add the handler to *body*) and that container then controls the events depending on which child element has triggered the event.

Discerning which technique is being used is important since the approach to reverse engineering the events is completely different. After all, on the classic approach we can simply analyze the handlers for each element whereas in the event delegation approach we must use additional logic to identify which element triggers which behavior.

4 Approach

In order to gather the data of the Websites we have developed a framework to analyze pages according to the criteria defined in Section 2. The architecture of our tool is depicted in Figure 2. The framework is composed by three components: DOM Analyzer, Event Detection and Variables Analyzer. Selenium WebDriver[4], a tool to automate the Web browser, was used to perform the communication with the Web browser.

[4] `http://docs.seleniumhq.org/projects/webdriver/` (last accessed: May 8, 2014)

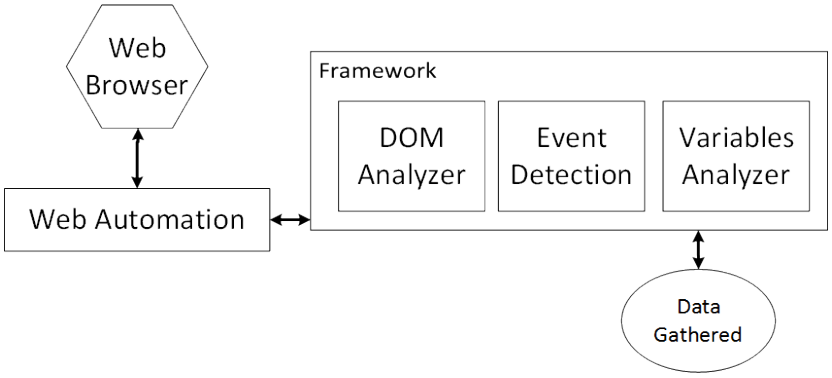

Fig. 2. Framework's architecture

4.1 DOM Analyzer

This component is responsible for parsing the HTML and gathering data from the DOM. The DOM provides an Application Programming Interface (API) for valid HTML and well-formed XML documents[5]. Whenever a Web page is loaded, the Web browser creates a DOM of that page, which in terms of HTML is the standard on how to manipulate HTML elements.

We parse the DOM with an HTML cleaner to overcome malformed HTML tags that may exist. Afterwards we inject JavaScript to discover which elements are visible and which elements are invisible. This is important since our analysis of the elements that are responsible for the behavior of the page will focus on the visible elements only.

In order to do this, the JavaScript we are injecting in the page is using the jQuery *find(':hidden')* function. Obviously, this also means that we have to add the jQuery framework to all the sites we are analyzing. In order to avoid problems with sites using other JavaScript frameworks, since many other frameworks use the $ character as a shortcut, we are using the jQuery noconflict() method.

4.2 Event Detection

The Event Detection component analyzes the page and searches for the event handlers that are present therein. As we discussed previously in Section 3, we must be able to search for the event handlers independently of what approach is being used in the Web page. Thus, Visual Event is first used to retrieve all the event handlers assigned to the Web page using the classic approach. However, the event delegation approach is significantly harder to analyze.

Since it can be implemented in several different ways, we are currently only identifying if that approach is being used. Even to perform that identification,

[5] `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/introduction.html` (last accessed: May 8, 2014)

```
1  document.body.onclick = function(e) {
2      e = ( e ) ? e : event
3      var el = e.target || e.srcElement;
4      if (el.id == "c") {
5          //Handler code
6          if (e.preventDefault) {
7              e.preventDefault();
8          }
9          else {
10             e.returnValue = false;
11         }
12     }
13 }
```

Fig. 3. An example of using Event Delegation

we require an analysis of the entire JavaScript source code, in search of usages of JavaScript tokens similar to the ones present on Figure 3, which depicts the source code of an example of assigning an event handler to an HTML element using the event delegation approach.

We are currently using a combination of the Firebug[6] and NetExport[7] extensions of the Firefox Web browser in order to retrieve all the JavaScript files that are requested from the server when a page is loaded. Then we analyze all the JavaScript source code in search for patterns similar to the ones presented on Figure 3.

4.3 Variables Analyzer

After extracting the relevant JavaScript code from the event handlers, we create an Abstract Syntax Tree (AST). In order to do this we use Mozilla's Rhino[8] to parse the JavaScript source code and generate the AST.

The code is analyzed to identify the statements that affect control flow. The relevant constructs are: ifs/elses, ternary operators, and while/for loops. Moreover, it is also necessary to analyze the variables used in those constructs.

Initially, we statically analyzed and classified those variables, based on how they are used on the source code, into the following four categories:

- **Constants** - are variables that remain unaffected by any type of function call. Our analysis will ignore these variables since we consider them of no interest, as they cannot cause changes in the logic of the control flow.

[6] `http://getfirebug.com/` (last accessed: May 8, 2014)
[7] `http://www.softwareishard.com/blog/netexport/` (last accessed: May 8, 2014)
[8] `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino` (last accessed: May 8, 2014)

- **Element variables** - are variables that use function calls like getElementById(...) to retrieve elements from the page. Moreover, these variables are further divided into:
  - **Input variables** - elements of the page that a user can manipulate (e.g textboxes).
  - **Static variables** - elements of the page that a user cannot manipulate (e.g labels).
- **Synthesized variables** - are variables that use other types of function calls, typically these will be calls to auxiliary functions at the user interface level, or to the business logic of the application.
- **Object variables** - are variables which are associated with an object or one object property.

An analysis of more complex pieces of source code led us to the need of also identifying three more types of variables:

- **Global variables** - If the variable declaration or assignment is outside the scope of our function we consider that variable to be global.
- **Control Flow variables** - are variables that have simultaneous assignments in different parts of the source code under different control constructs. For example, lets imagine we were analyzing the source code in Figure 3. If we added a new if condition after line 11 using the *e* variable, our analysis would conclude that, at that stage, *e* would have either been affected by the previous invocation of the *preventDefault()* method, or by the assignment of *false* to its *returnValue* property. In these situations we call that variable a Control Flow variable since its origin may differ according to the flow of the application.
- **Hybrid variables** - are variables for whose classification we identified more than one type of the previously discussed types of variables (except constants since those can be ignored). For instance, in the following source code variable *b* would be obtained from both an input variable and a synthesized variable:

```
var b = document.getElementById('c');
b = b + getServerData();
if(b>0){...}
```

Another aspect we can see from the source code above is that our analysis must include all previous assignments of that variable in the source code. Otherwise, in the previous code we would define the variable as Synthesized instead of Hybrid, which would be inaccurate.

It is also important to notice that our analysis is focusing only on events associated with visible elements. While the depth of analysis is customizable (that is, as long as they are available in the browser, it is possible to configure the tool to analyze the source code of the functions called from the event handlers, and the ones called from those, etc.), in what follows we will only be analyzing the event handlers up to a depth of one. If an event handler has several function calls, we are also going to analyze those functions (if those functions are available),

but not the functions called by them. This happens because the goal of this analysis was to evaluate how much information about control flow we could access, analyzing as few JavaScript as possible.

Our approach to gather the data regarding variables is the following: we start by gathering the control flow constructs present in the portion of source code we are analyzing. This source code is either an event handler function or a single function in the code we retrieved by analyzing a function call. For each construct we gather which variables are used.

The following source code shows an example of an if construct:

```
if(b==document.getElementById('c'){...};
```

We differentiate our analysis according to the variable. If we have a single name token followed by any type of operator we have to analyze the previous code in search for that variable assignment, such as, *b* in the source code above. In that case we extract all the previous assignments of that variable on the source code and analyze each one for the types of variables being used. Afterwards, according to the number of different types found we identify the variable under analysis conforming to the previously explained catalog.

However, if we have more complex constructs before or after the operators, e.g., document.getElementById('c') in the above source code, we process them as a variable. Therefore, we need to identify which type of variable that part of the source code is according to our catalog, in this example we would identify it as an Element variable.

5 Top Sites analysis

In this section we describe how the tool just described was used to investigate how much control logic information might be possible to extract from event handlers. As already explained, the goal is to assess the viability of developing an hybrid approach to the reverse engineering of Web applications.

5.1 Scope of the analysis

Since the approach we have developed is fully automated, we could define any number of Websites for analysis. We decided to focus our analysis on the most popular Websites globally, thus we used the Alex top Websites list. Our analysis covered the first one thousand sites on that list. The analysis was performed on the 26th of February, 2014.

It is important to note that in order for the analysis to be automated we had to bypass several errors that could occur analyzing these applications. For example, one important problem we had was that some sites could never finish the page loading. This problem was unrelated to our tool, since opening those sites on different Web browsers, no matter how much time we waited the page would never finish loading. When this happens we cannot extract any information from the page. In order to overcome the application being set on a loop waiting for

Table 1. Events and control-flow constructs

	hrefs	Events	Click Events	Ifs	Other Control Flow Constructs	Function Calls
Mean	214.09	26.57	18.53	40.26	40.34	116.96
Standard Deviation	301.28	74.78	68.19	153.71	153.71	412.49
Median	104	5	1	3	3	9
Maximum value	3349	902	887	2109	2109	5121
Percentage of zeros	6.4	21.8	46.5	34.8	34.6	31
Mean excluding zeros	228.73	33.98	34.63	61.74	61.68	169.51

the loading we set a timeout of one minute for each analysis. The final result showed that forty five of the one thousand sites could not finish loading in the minute we set. This means that almost five percent of our analysis scope has no data gathered because of this problem.

Another problem we had in many sites was that while analyzing the event handlers source code, we got JavaScript parsing errors. In these cases we skipped only those handlers analysis. At this point we could not identify if the malformed JavaScript was coded on purpose, to prevent third party analysis such as this one, or were simply coding errors.

5.2 Data Analysis

We decided to group the criteria into two groups, one with the data on events and constructs, and the other with the data on variables. The event delegation criterion is going to be treated independently of the other criteria since is the only one with a boolean as a result and not a number.

Table 1 shows a summary of the results we retrieved from the analysis of the one thousand sites. Something we can immediately gather, from the table by analyzing the maximum value in comparison with the mean and the median, is that there clearly exist outliers in the data. Moreover, we can see that the data is quite disperse, by comparing the standard deviation values with the mean values.

In terms of hrefs, and discounting the 4.5% of sites that were not computed, numbers show that only around 2% of the analyzed pages that did not use any type of hrefs. This means that the wide majority of Websites still uses hrefs for navigation between pages. This was, of course, to be expected, but shows that the reverse engineering tool should not be restricted to single page Web applications, and consider also navigation between pages.

The analysis of the event handlers shows that only approximately 22% of sites did not have any event we could find. Moreover, when rstricting the events to only clicks we get a 46.5% in total. That means that there were approximately 25% of sites that had events we could detect but none of those events were clicks.

Table 2. Variables comparison

	Element	Synthesized	Object	Global	Control Flow	Hybrid
Mean	0.98	15.09	16.96	29.66	4.98	0.37
Standard Deviation	4.83	79.40	84.28	142.37	45.48	2.84
Median	0	0	1	1	0	0
Maximum value	49	1008	1830	2520	1155	64
Percentage of zeros	89.2	65	44.4	48.6	86.7	93.1
Mean excluding zeros	9.1	43.11	30.49	57.7	37.41	5.27

Also, having a mean of around 26.57 events and 18.53 clickable events shows that an analysis based on this type of events would have an important impact on the Web overall.

In terms of control flow structures the data we gathered showed an interesting result. The usage of if constructs is almost identical to use of the other control flows constructs. We were expecting a lot more usage of ifs than the other constructs but that was not the case in this analysis. One hypothesis is that since these are the most popular sites globally, most source code is done with performance and space constraints and a significant part of those other constructs are ternary conditional operators.

Regarding these criteria global values, finding an average of 80 control flow constructs per site with only an analysis of a depth of one function call is quite significant for our approach. Obviously, we must also take into consideration that around 35% of sites had no construct at all. We can only assume that either no logic was present on the client side, or that the event delegation approach was being used.

Concerning function calls, we found an average of 111.96 function calls per site, only on the subset of source code we were analyzing. This shows that there is a significant amount of other source code that is not analyzed in our approach. Moreover, considering those function calls might have other function calls and so on, that is even more significant. Also important is that most JavaScript code obfuscation techniques increase the number of function calls significantly to hide the logic behind the code [11]. Thus, even an increase in our analysis to a depth of 5 or more function calls might not retrieve interesting results, despite the added computational load.

Table 2 depicts a summary of the data we gathered about our analysis of variables on the source code. Element variables were not found in 89.2% of the sites. This was something we were expecting since not only there are a lot of different frameworks in JavaScript, but also there are several ways to shorten the usage of *document.getElementBy*. For instance, by wrapping those calls inside auxiliary functions:

```
function getId(id){ return document.getElementById(id); }
```

Nevertheless, about 10% the most popular sites use this construct unaltered to get elements on their event handlers. Also interesting is that excluding the sites with no variables of this type found, we got an average of 9 element variables found.

In terms of synthesized variables, they were found in 35% of sites and excluding zeros we got an average of 43.11 variables found. It is interesting, however, that the number of sites where these variables were found is quite lower than what we were expecting, particularly when comparing with object and global variables whose results were higher. Thus, most sites are not currently using these variables, which means that they might be using functional references on variables, which we are currently interpreting as object variables.

This conclusion is important because it means that we have to analyze each object variable of our analysis to see if the type of that variable is or not a function. Although this might lead to a lot more computation, this statistical analysis showed us that it is important for us to do add this feature.

Both object and global variables were found in most sites. In fact, if we exclude the 35% of sites where no control flow constructs were found, thus no analysis of variables was performed, only around 10% of sites were analyzed and got none of these variables. It is also interesting that the type of variable that clearly got more matches in our analysis was the global.

Both control flow and hybrid variables were found in a significantly smaller number than other types of variables excluding element variables. Since handling these variables is quite more complex than the others, these results were promising to our approach. Moreover, the hybrid variables were clearly the ones that were identified less in our analysis.

In terms of event delegation we identified 30.6% of sites that were using this approach for adding event handlers. It was also interesting and something unexpected, that most of these sites also had click events that we were able to identify. Thus, there are a significant number of sites that use both approaches for adding event handlers.

6 Conclusions

This paper has presented two main contributions. First a tool for analyzing Web applications, based only on the event handlers' source code, was proposed. The tool works by extracting information about the control flow of the application, based on the types of variables identified. Second that approach was used on the top thousand most popular Web sites globally. The analysis of those sites, in terms of events and variables used in control flow constructs, has been presented and discussed.

That analysis enabled us to retrieve useful information towards our goal of developing an hybrid tool for the reverse engineering of Web applications. For instance, an analysis of the two approaches of adding event handlers to a page as discussed in Section 3 shows that the classic approach is widely used, since we got results in approximately 78% of sites while the event delegation just

appeared on approximately 30% of sites. Therefore, a tool that reverse engineers sites based on the classic approach would work on the majority of Websites.

Another important result is that the amount of if constructs used in the source code we analyzed is similar to the amount of other control flow constructs. This means that if our analysis focus only on ifs we would be analyzing only half the constructs that affect the control flow of the application.

In terms of our variable's analysis we infer that both control flow and hybrid variables are used significantly less than other types of variables, thus the added computational logic we would need to handle these variables might not compensate. Furthermore, we were expecting more synthesized variables than what we found, this mean we must further inspect object variables to identify if they are functions.

Future work will comprise the usage of this data into developing a Reverse Engineering tool that will be able to extract information on most sites. Moreover, there were a few shortcomings in our analysis that could be improved such as extending the element variables identification to other frameworks or techniques, this could mean an analysis of the entire JavaScript files similar to the one we are doing to identify event delegation approaches. Furthermore, trying to find correlations between the several criteria could also yield interesting results.

Acknowledgments

This work was partly funded by project LATiCES (Ref. NORTE-07-0124-FEDER-000062) financed by the North Portugal Regional Operational Programme (ON.2 – O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), and by national funds, through the Portuguese funding agency, Fundação para a Ciência e a Tecnologia (FCT). Carlos Eduardo Silva is further funded by the Portuguese Government through FCT, grant SFRH/BD/71136/2010.

References

1. Eldad Eilam. *Reversing: Secrets of Reverse Engineering.* Wiley, 2005.
2. Alexandru C. Telea, editor. *Reverse Engineering – Recent Advances and Applications.* InTech, 2012.
3. Tommi Mikkonen and Antero Taivalsaari. Web applications – spaghetti code for the 21st century. Technical Report SMLI TR-2007-166, Sun Microsystems, 2007.
4. Carlos Eduardo Silva. Reverse engineering of rich internet applications. Master's thesis, Universidade do Minho, 2009.
5. J.C. Campos, J. Saraiva, C. Silva, and J.C. Silva. GUIsurfer: A reverse engineering framework for user interface software. In Telea [2], chapter 2, pages 31–54.
6. Roy de Kleijn. *Learning Selenium: Hands-on tutorials to create a robust and maintainable test automation framework.* Leanpub, 2014.
7. Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.

8. Ines Coimbra Morgado, Ana C. R. Paiva, Joao Pascoal Faria, and Rui Camacho. GUI reverse engineering with machine learning. In *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*, pages 27–31. IEEE, June 2012.
9. Carlos Eduardo Silva and José Creissac Campos. Combining static and dynamic analysis for the reverse engineering of web applications. In P. Forbrig, P. Dewan, M. Harrison, K. Luyten, C. Santoro, and S.D.J. Barbosa, editors, *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2013)*, pages 107–112. ACM, 2013.
10. Jenkov Jakob. *jQuery Compressed*. Jenkov Aps, 2011.
11. Sebastian Schrittwieser and Stefan Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In *Proceedings of the 13th International Conference on Information Hiding*, IH'11, pages 270–284, Berlin, Heidelberg, 2011. Springer-Verlag.