Combining Static and Dynamic Analysis for the Reverse Engineering of Web Applications

Carlos Eduardo Silva
Departamento de Informática/Universidade do Minho & HASLab/INESC TEC
Braga, Portugal
cems@di.uminho.pt

José Creissac Campos
Departamento de Informática/Universidade do Minho & HASLab/INESC TEC
Braga, Portugal
jose.campos@di.uminho.pt

ABSTRACT
Software has become so complex that it is increasingly hard to have a complete understanding of how a particular system will behave. Web applications, their user interfaces in particular, are built with a wide variety of technologies making them particularly hard to debug and maintain. Reverse engineering techniques, either through static analysis of the code or dynamic analysis of the running application, can be used to help gain this understanding. Each type of technique has its limitations. With static analysis it is difficult to have good coverage of highly dynamic applications, while dynamic analysis faces problems with guaranteeing that generated models fully capture the behavior of the system. This paper proposes a new hybrid approach for the reverse engineering of web applications' user interfaces. The approach combines dynamic analyzes of the application at runtime, 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, and used to guide the dynamic analysis.

Author Keywords
Static Analysis; Dynamic Analysis; Web applications.

ACM Classification Keywords
H.5.2. Information Interfaces and Presentation (e.g. HCI): User Interfaces; D.2.7. Software Engineering: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering.*

General Terms
Human Factors; Reliability.

INTRODUCTION
Reverse engineering techniques can be useful for both testing and maintaining a software system [4]. For interactive computing systems, reverse engineering can be used to extract information about the structure of the user interface as well as its behavior. This can be achieved either by looking at the code (static analysis), or by analyzing the running application (dynamic analysis).

Regarding web applications, static analysis faces problems related to the highly dynamic nature of the user interfaces. In many situations, the relation between user interface controls and the corresponding event handlers is only defined at runtime. Even the structure of the user interface might be defined dynamically at runtime, with only a basic skeleton defined statically in the code. Additionally, the diversity of technologies that are available to program such systems (both server-side and client-side), makes it difficult to develop a generic approach.

Dynamic analysis solves some of these issues by analyzing the actual running systems. However, it faces problems of its own. On the one hand, the behavior of the applications depends on both the internal state of the interactive computing system, and on the inputs provided. There is the risk that relevant parts of the user interface might be left unexplored. On the other hand, what is observed is the behavior of the application. The reasons for that behavior have to somehow be inferred. In any case, using dynamic analysis alone, the resulting model will likely be incomplete. It might miss relevant aspects of the user interface, and it will be ambiguous regarding what conditions trigger which alternative behaviors.

In this paper we report on work that aims to develop an hybrid approach to the reverse engineering of web applications. The approach takes advantage of the fact that information about the code behind a given user interface is available through the browser. Using a dynamic reverse engineering approach, a first model of the user interface is obtained. Then, by static analysis of the event handlers attached to each user interface control, relevant conditions over the input values in the user interface are determined. This provides two benefits. It supports completing the model by determining which input values should be provided so that all user interface behaviors are observed. It supports disambiguating the model by making explicit the conditions that lead to each alternative behavior of the user interface.

The paper is structured as follows: the next section provides an overview of the state of the art of Reverse Engineering applied to interactive computing systems; after that, an example application is described which will be used to illustrate the approach; the following section analyses the application emphasizing the dynamic analysis' shortcomings when attempting to reverse engineer its user interface; our proposal to solve

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EICS'13, June 24–27, 2013, London, United Kingdom.
Copyright 2013 ACM 978-1-4503-2138-9/13/06...$15.00.

these problems is then presented; the paper concludes with a discussion and conclusions on our approach.

STATE OF THE ART

As stated above, the two main approaches for Reverse Engineering are: static analysis and dynamic analysis. When both approaches are combined, we talk of hybrid analysis techniques. This section presents a brief review on the use of these techniques to reverse engineer user interfaces.

Static Analysis

Static analysis performs a system's analysis without executing it. This is achieved through the analysis of the source code or the binaries of the system. Examples of static analysis tools usually involve targeting a specific language. For instance, Bouillon et al. [3] reverse engineer simple HTML pages; Staiger [15] is targeted at C/C++ applications that use user interface libraries like Qt or TK; Guha et al. [6] use static control flow analysis on JavaScript applications; Ko et al. [7] use static analysis on JavaScript applications to search for missing feedback in applications; Bellucci et al. [2] perform static analysis of HTML and CSS to support adaptation of Web Applications across platforms, the approach addresses architectural aspects keeping the same JavaScript between the adaptations.

In an attempt to be more generic, and thus reduce the effort of changing the target application's programming language/framework, Silva et al. [14] separate the parser and Abstract Syntax Tree (AST) analyzer from the rest of their tool (GUIsurfer). This enables them to reverse engineer Java/Swing and GWT applications, but also WxHaskell applications, with minimal adaptation to the tool.

When attempting to adapt the GUIsurfer approach to other web based technologies [13], however, problems arose related to the highly dynamic nature of the code. For example, the binding of event handlers to controls is, in many cases, done at execution time only. Indeed, when considering interactive applications, one of the main problems with static analysis approaches is identifying the binding of event handlers to user interface controls. The fact that there are many different ways to perform this binding, makes it hard to determine such binding from a purely static analysis of the code. In the case of web applications, this problem is exacerbated by the dynamic nature of the code. Indeed, Mesbah et al. [10] affirm that reverse engineering Ajax based on static analysis is not feasible.

Dynamic Analysis

Dynamic analysis aims to obtain a model of a system from observation of its runtime behavior. Several authors have studied its applicability to interactive systems. Memon et al. [9] describe an application called GUI Ripping which consists of a dynamic process that traverses a Graphical User Interface (GUI) by opening all its windows and extracting all the widgets and their information. Amalfitano et al. [1] use dynamic analysis to create Finite State Machines (FSMs) from Rich Internet Applications (RIAs). Crawljax is a tool that crawls Ajax based Web applications analyzing dynamically state changes and creates a FSM [10]. Morgado et al. [11] describe the ReGUI tool that automatically extracts structural and behavioral information from a GUI, producing a wide variety of views and formats of the data, thus enabling different types of analysis.

While this type of approach solves the problem faced by static analysis with dynamically generated user interfaces, it can only observe the behavior of the interface, and has problems with determining the logic behind that behavior, and with guaranteeing that all relevant behavior has been observed.

Hybrid Approaches

Hybrid approaches try to take advantage of the best feature of both static and dynamic analysis. For example, Systa [16] gathers both dynamic views (using a customized JDK debugger) and static views (parsing Java byte code), and afterwards improves them by merging aspects from both types of views. Li and Wohlstadter [8] describe an hybrid approach that enables runtime maintenance of GUIs. This tool was developed for Java/Swing applications and its focus is on supporting changes to elements of the user interface at runtime, while our focus is on creating models that describe the user interface. This is also the goal of Gimblett and Thimbleby [5] which *discover* a model of an interactive system by simulating user actions. Models created are directed graphs where nodes represent system states and edges correspond to user actions. The approach is dynamic but it also considers access to application source code.

AN ILLUSTRATIVE EXAMPLE

In order to illustrate both the limitations of static and dynamic approaches in the reverse engineering of web applications, and our proposal for an hybrid approach, a small illustrative example will be used. This is a contacts agenda application enabling users to maintain a list of contacts.

Figure 1 shows a subset of the frames of the application. We are specifically focusing on the ”Find” functionality of the application (on the left side of Figure 1) which allows a user to search for contacts in his/her contact list. As illustrated in the figure, clicking the ”Search” button can lead to three different frames. Two of them are warnings, stating no text was entered or no contact was found. The third one is the main window with the found contacts selected. The results will depend on the list of contacts of the user, the text entered in the textbox and the state of the two checkboxes for ”Match Case” and ”Whole Words”. The other two buttons (”Cancel” and ”Show”) are currently not being considered for simplification.

The ”Search” button has an event handler which triggers the *search* function, presented in JavaScript in Figure 2. The function starts by analyzing if there is any text in the inputBox, and in case there is not, it creates an alert with the text ”No text entered”. If there is text, the findContacts function is invoked with three parameters: the text input by the user, and the states of the matchCase (mC) and wholeWords (wW) checkboxes (note that this function can be defined in the client or the server). Afterwards, the function checks if there was any result returned from that function. In case there was a result, it updates the contacts list, and closes the frame. The

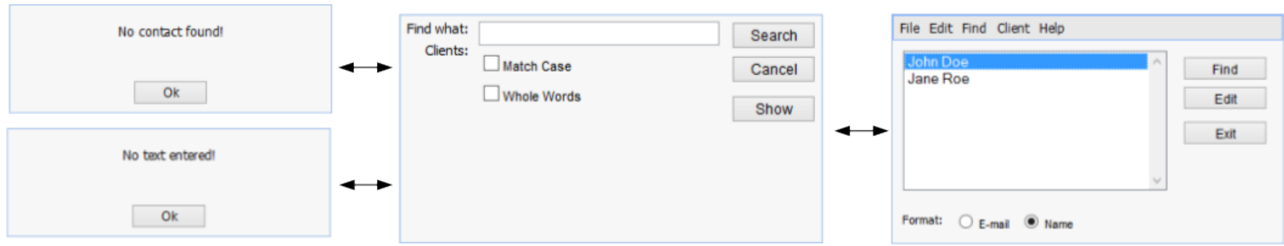

Figure 1. Subset of the frames of the Contacts Agenda applications

```
1  function search(){
2      fInput=document.getElementById("fInput").value;
3      if(fInput!=""){
4          var mC = document.getElementById("mC").checked;
5          var wW = document.getElementById("wW").checked;
6          var res = −1;
7          res = findContacts(fInput, mC, wW);
8          if(res>−1){
9              contactsListUpdate(res);
10             findExit();
11         }
12         else {
13              customAlert("No contact found!");
14         }
15     }
16     else {
17          customAlert("No text entered!")
18     }
19 }
```

Figure 2. Search function

user is thus returned to the mainForm frame (depicted in the top left of Figure 1). Otherwise, an alert is raised with the text "No contact Found".

ANALYSIS

The contacts application is an Ajax application, thus using both HTML, CSS and JavaScript to code the client side and, in this particular case, PHP to code the server side. Therefore, a purely static analysis would have to take into consideration these four languages in order to get some sound results. Not only is that a problem, but we also need to take into consideration the highly dynamic possibilities of JavaScript, as already discussed.

Analyzing the application in a purely dynamic analysis, solves the problems above. On the one hand, we do not have to deal with all the different technologies that might be used to develop web applications. On the other hand, we are able to observe the effect of the event handlers at runtime regardless of how they are registered. Using this type of approach we are able to identify the different states of the interface, but the question remains of how to infer which conditions govern the different behaviors of the application.

As an example, we built a state machine of our application using a dynamic analysis tool (Crawljax). Figure 3 presents a manually enhanced version of the resulting finite state machine. For readability purposes states have been decorated with the names of the corresponding frames, and state transitions with the names of the controls (in this case, buttons) responsible for causing them. While this information is not present on the original diagram generated by the tool, that diagram can be interactively explored and such information obtained.

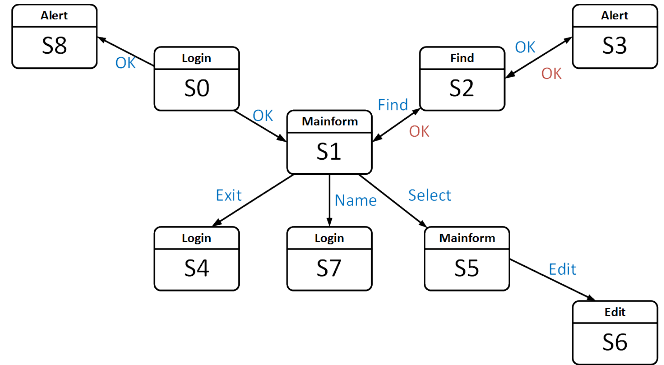

Figure 3. State Diagram based on a model extracted with Crawljax

Only a subset of the state machine is important for this analysis: the *find* frame (S2), the *mainform* frame (S1) and the "No contact found" alert (S3). Other frames (and corresponding states) of the application are not further discussed for simplification purposes. The model suffers from a number of shortcomings that we will discuss below.

Model disambiguation problems

When interacting with the application, and as can be seen in Figure 1, clicking on the Search button can lead us to a number of different frames. Through dynamic analysis we were able to (at least partially) identify this situation. As depicted in Figure 3, we were able to determine that we can go from S2 (the Find frame) to either S1 (the Main frame) or S3 (an Alert Frame).

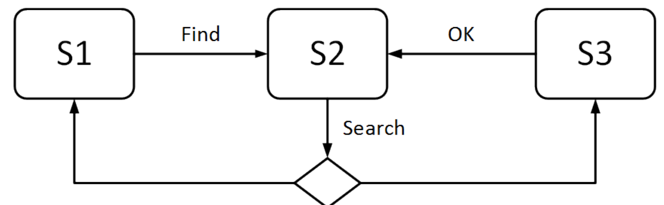

Figure 4. State diagram with buttons information

Figure 4 depicts a subset of the overall state machine with

only the states relevant to our analysis present, and the choice points more clearly identified. The problem is that, while the state diagram shows that when we click the Search button two possible next takes can be reached (S1 and S3), it says nothing about what conditions determine the behavior of the interface. In practice the model that is generated is ambiguous and needs further work.

It should be noted that while we could think of analyzing the inputs used in each case to infer the missing conditions, we could still have the same inputs going to different states, depending on what the state of the system (i.e. the current contacts list). A possible solution can be explored of using machine learning as seen in [12]. However such methods involve previous background knowledge including the encoding of the patterns for the disambiguation. Our proposal is to disambiguate the models through the static analysis of the events that trigger new states in the application.

Input space definition problems

Another aspect that becomes clear in Figure 4 is that one frame is missing from the model. In this case the "no text entered" alert was not found through dynamic analysis. This happened due to the test cases used during the dynamic exploration. A more thorough analysis, with more execution traces, would be needed to have found it.

Indeed, a common difficulty in dynamic analysis is choosing the inputs that should be used to explore the application. Normally, fully automatic dynamic approaches use random input generators or machine learning to define the inputs. Semi-automatic approaches usually rely on the tool's user to ascertain possible input values that are interesting for their intended analysis. In any case, unless knowledge about the application can be obtained and used, it is not possible to be sure that all relevant path in the behavior of the system have been covered.

Our proposal is to identify relevant input values by analyzing the conditions present in the event handlers associated with user actions.

PROPOSAL

As stated, in order to solve the two problems identified above, we propose to include an element of static analysis in the dynamic exploration of the user interface, thus creating an hybrid approach. Through dynamic analysis we are able to identify the event handlers that are associated with each user interface control. By analyzing the source code of the event handlers that trigger the state changes in the application, we are able to add more information to the dynamic exploration process, thus solving the disambiguation problem.

Hence, the process consists of a dynamic crawler that, for each identified frame, performs static analysis using a process which can be defined as follows:

1. Identify the user interface controls of the frame.

2. Discover which event handlers are associated with the controls.

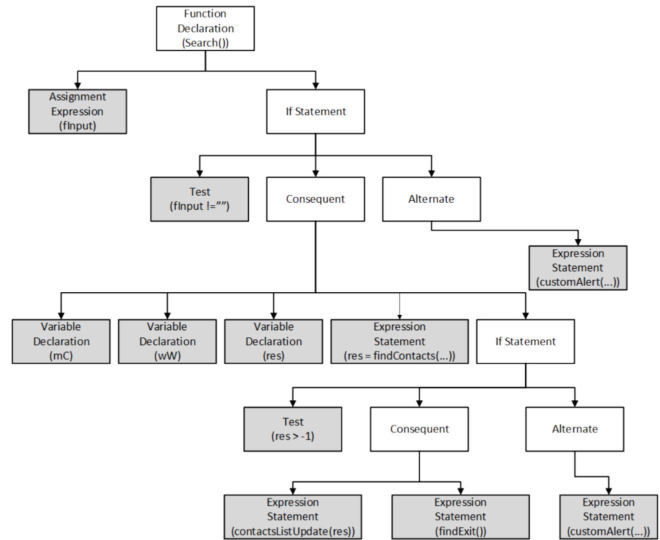

Figure 5. Abstract Syntax Tree for the Search function

3. Analyze the conditions of the discovered event handlers to determine if all behaviors have been analyzed.

4. Set up additional test cases as needed.

Identifying the user interface controls is something already done by a typical dynamic analysis tool. The event handlers can then be easily identified. Since this is done at runtime, we avoid the problems associated with the dynamic binding of event handlers. Note that although this example only uses synchronous calls to the application logic, asynchronous calls (cf. Ajax) can also be dealt with since we have access to the asynchronous request handler. Then, for each handler function, we create an Abstract Syntax Tree (AST) that represents it. Using the AST we analyze the conditions of the event handlers. Two type of variables are relevant. Variables whose value is obtained from input controls (input variables), and variables whose value is obtained from functions of the applications' logic (synthesized variables).

Enough test cases must be generated that all possible behaviors of the handler (branches of the AST) are executed. Regarding input values, this is achieved by identifying which input values must be used during the dynamic exploration. Regarding synthesized variables, determining what input values will cause the application layer to generate appropriate synthesized values is in most cases not easily achieved. This will be discussed below.

A simplified version of the AST for the search function is depicted on Figure 5. Analyzing it, we can see that it starts by assigning the variable findInput from the DOM. Afterwards we get a condition that tests if the variable is an empty string or not. Since this variable is directly associated with an input control we can manipulate, we add test cases for both an empty and an non empty string to the execution traces. And since in the automatic test we had not tested the empty string, we are able to to find a state that was not previously discovered. We called the new state SY, which is an alert of "No

text entered".

The next condition is related to a variable named *res*. Unlike the previous analysis, here the variable is not associated with a control we might manipulate but with the *findContacts(...)* function. Hence, it is a synthesized variable. We know that two different states can potentially be reached depending on the value of *res*, as the variable is used in a condition that changes the event flow. In order to fully test the behavior of the system we must, thus, find a way to establish which branch leads to which state (in this case S1 or S3, but note that the dynamic analysis could have not yet found all the possible states, as illustrated above), and that depends on the value of the synthesized variable.

Three alternative solutions can be considered:

1. We can use a debugger to analyze the values of the variable. For instance, in JavaScript we are able see the variables' values at each point using tools like Firebug. Despite being the simplest solution, we have a problem in that our execution traces might not cover all possible behaviors (e.g., if a contact is never found, we will never have the *res* variable positive and will never observe that, in that case the flow goes to S3). Moreover, using a third party debugger to inspect the variables would turn the approach to a semi-automatic process.

2. We can use code injection to change the event handler so that it generates predefined values. In the case of the example, for example, this will be a new function that is exactly the same as the Search function depicted in Figure 2 but has another line after line 7 setting the *res* value so that each of the conditional flows is executed. Hence, in one execution trace we would have assigned *res* to a random negative value and on the other to a random positive value.

   This approach has the problem that changing the event handler's result might have unexpected effects in the application.

3. If we have access to the source code on the server side, we are able to perform the instrumentation as in the previous approach but in this case in the actual source code of the applications, thus changing the actual application code and not having to worry about problems with having changed the event handler. Besides requiring access to the server side code, a further problem with this approach is that after changing the source code, we would have to rerun the application for the instrumentation to be applied.

Since we want to keep our analysis as dynamic as possible we have opted to implement the second choice of dynamically changing the event handlers to functions with the instrumented code.

Returning to the example, after analysis of both conditions it was then feasible to build a more detailed state diagram, like the one presented in Figure 6, where we can see which conditions were responsible to trigger which state transitions.

We can summarize the whole process as follows:

1. Use a dynamic approach to start creating the state diagram

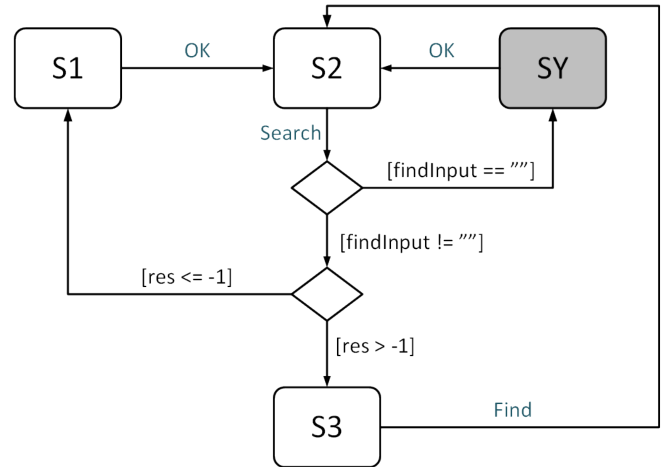

Figure 6. Complete state diagram for the Search function

2. For each state analyze the interaction widgets for the corresponding listeners

3. For each listener create an AST of the corresponding function

4. Analyze the method conditions:

   (a) Analysis which variables are used in the conditions

   (b) From those variables discover which ones are input variables (that we can control during dynamic exploration) and which are synthesized variables.

   (c) For input variables test conditions by testing the different input possibilities in the application. If test cases were missing create new test cases.

   (d) For synthesized variables alter the listener in runtime to a new function with instrumented code testing the conditions.

5. The final states are compared according to the DOM

6. Perform conditions pruning to eliminate conditions where all alternatives lead to the same state (a case that did not appear in our example, but might nevertheless happen in practice).

With such a process we are able to build state diagrams that represent the behavior of Web applications GUIs with more detail such as the one depicted in Figure 6. The added detail shows why interactions with the same widgets can lead to different states.

DISCUSSION

When comparing our analysis to other approaches to reverse engineer Web Applications we believe that we can cover more applications than previous static analysis work on Ajax Web applications such as in Guha et al. [6] and Ko et al. [7]. Moreover, we can add more detail to dynamic analysis approaches of Web applications as we see in Amalfitano et al. [1] and Mesbah et al. [10].

In terms of hybrid approaches the most similar to our approach is Systa's [16]. The main differences are that while

Systa's approach enables a static analysis of the application both before, during and after the dynamic analysis, our approach is done only during the dynamic analysis. Moreover, Systa's approach implies a full static analysis of the application. While that may be feasible on Java, it is not in Web applications since not only are the client side part based on a wide variety of different frameworks, we may also not have access to the server side of those applications. Li and Wohlstadter's approach [8] has a different focus which aims to use static analysis to propagate the changes made on the dynamic view, thus mapping widgets with code. Gimblett and Thimbleby's approach [5] is based on a semi-automatic process and hence not directly comparable. Finally, comparing our work with that of Morgado et al. [12] which use machine learning to perform disambiguation of the models, an hybrid approach has the advantage of not requiring previous background knowledge of the application and its domain.

A tool that automates the proposed approach is currently being developed. The tool uses Selenium to interact with the web application under analysis, and is able to differentiate, at each point, between visible and non-visible widgets, since only visible widgets are relevant for the analysis. Furthermore, the tool is able to extract a JavaScript AST for each of the event handlers registered in the widgets. This information will then be used to guide the reverse engineering process as proposed above.

CONCLUSION

Developing an understanding of an implemented web application is a complex task which can be aided by reverse engineering techniques. In this paper we have discussed the shortcomings of traditional reverse engineering techniques when applied to web applications. An approach to integrate static and analysis techniques has been presented, and illustrated with an example. By mixing the two techniques we are able to explore the best of dynamic analysis, while incorporating knowledge about the code into the models. This has enabled us to construct a more complete model of the user interface.

ACKNOWLEDGMENTS

This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-015095. Carlos Eduardo Silva is further funded by the Portuguese Government through FCT, grant SFRH/BD/71136/2010.

REFERENCES

1. Amalfitano, D., Fasolino, A. R., and Tramontana, P. Reverse Engineering Finite State Machines from Rich Internet Applications. In *Proc. 15th WCRE*, IEEE Computer Society (2008), 69–73.
2. Bellucci, F., Ghiani, G., Paternò, F., and Porta, C. Automatic reverse engineering of interactive dynamic web applications to support adaptation across platforms. In *Proc. IUI '12*, ACM Press (2012), 217–226.
3. Bouillon, L., Limbourg, Q., Vanderdonckt, J., and Mirchotte, B. Reverse engineering of web pages based on derivations and transformations. In *Proc. LA-Web '05*, IEEE Computer Society (2005), 3–.
4. Eilam, E. *Reversing: Secrets of Reverse Engineering*. Wiley, 2005.
5. Gimblett, A., and Thimbleby, H. User Interface Model Discovery : Towards a Generic Approach. In *Proc. EICS '10*, ACM Press (2010), 145–154.
6. Guha, A., Krishnamurthi, S., and Jim, T. Using static analysis for Ajax intrusion detection. In *Proc. 18th WWW '09*, ACM, Ed., ACM Press (2009), 561–570.
7. Ko, A. J., and Zhang, X. Feedlack detects missing feedback in web applications. In *Proc. CHI '11*, ACM Press (2011), 2177–2186.
8. Li, P., and Wohlstadter, E. View-based maintenance of graphical user interfaces. In *Proc. 7th AOSD '08*, ACM Press (2008), 156–167.
9. Memon, A., Banerjee, I., and Nagarajan, A. GUI ripping: reverse engineering of graphical user interfaces for testing. In *Proc. 10th WCRE '03*, IEEE Computer Society (2003), 260–269.
10. Mesbah, A., Bozdag, E., and van Deursen, A. Crawling AJAX by Inferring User Interface State Changes. In *Proc. ICWE '08*, IEEE Computer Society (2008), 122–134.
11. Morgado, I. C., Paiva, A. C. R., and Faria, J. a. P. Dynamic Reverse Engineering of Graphical User Interfaces. *International Journal On Advances in Software 5*, 3 (2012), 224–236.
12. Morgado, I. C., Paiva, A. C. R., Faria, J. P., and Camacho, R. GUI reverse engineering with machine learning. In *Proc. RAISE '12*, IEEE Computer Society (2012), 27–31.
13. Silva, C. E. Reverse engineering of rich internet applications. Master's thesis, Escola de Engenharia, Universidade do Minho, 2009.
14. Silva, J. C., Silva, C. E., Gonçalo, R., Saraiva, J., and Campos, J. C. The GUISurfer tool: towards a language independent approach to reverse engineering GUI code. In *Proc. EICS '10*, ACM Press (2010), 181–186.
15. Staiger, S. Static Analysis of Programs with Graphical User Interface. In *Proc. CSMR '07*, IEEE Computer Society (2007), 252–264.
16. Systa, T. On the relationships between static and dynamic models in reverse engineering Java software. In *Proc. 6th WCRE 1999*, IEEE Computer Society (1999), 304–313.