Extracting Interaction-Based Stateful Behavior in Rich Internet Applications

Yuta Maezawa
The University of Tokyo
Tokyo, Japan
maezawa@nii.ac.jp

Hironori Washizaki
Waseda University
Tokyo, Japan
washizaki@waseda.jp

Shinichi Honiden
The University of Tokyo
National Institute of Informatics
Tokyo, Japan
honiden@nii.ac.jp

***Abstract*—Although asynchronous technologies such as Ajax make Rich Internet Applications (RIAs) responsive, they can result in unexpected behavior due to nondeterministic client-side processing and asynchronous communication. One difficulty in understanding such erroneous behavior lies in the unpredictable contexts of the running system. Dynamic behavior analysis techniques do not help to verify the correctness of certain "blind spots" in the execution path. In this work, we present a static approach for extracting all possible state transitions described in source code from the RIAs. Our approach is based on the assumption that user, server and self *interactions* with the RIAs can change the states of the application. Our method consists of four steps: 1) using given specifications of Ajax relevant to interactions as rules, 2) creating a call graph, annotating interactions on it and extracting interaction controls, 3) abstracting the call graph to extract relationships among the interactions, and 4) refining the relationships with the interaction controls. By extracting the state machines of test scenarios of the correct and wrong behavior, it can help developers to pinpoint the statements in the source code that lead to the erroneous behavior. Our approach has been evaluated against a few experimental cases and we conclude that it can extract comprehensible state machines in a reasonable time.**

***Keywords*-Rich Internet Applications; Ajax; Reverse Engineering; Program Comprehension;**

I. Introduction

Rich Internet Applications (RIAs) are a new generation of web applications by introducing client-side processing and asynchronous communications, achieving high usability and rendering attractive contents [4], [13]. Although asynchronous technologies such as Asynchronous JavaScript and XML (Ajax) make RIAs responsive, they can result in unexpected behavior due to nondeterministic user events and server responses. Therefore, it is important to control how RIAs behave according to their interactive state changes.

When developers incrementally implement functions in RIAs, they initially determine what kind of *interactions*, as shown in Figure 1, RIAs can handle, and then how RIAs should behave during run-time. We assume that user, server, and self interactions can change the states of the applications. At this prototyping phase, developers intentionally make RIAs enable and disable some interactions in certain states. For example, some survey applications might initially disable a submit button unless users input in a

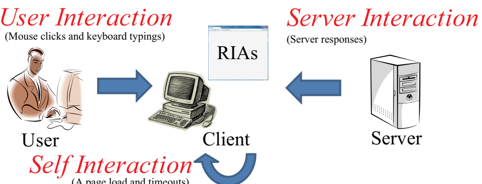

Figure 1. Interactions that have potential to change states in RIAs.

form. Developers want to ensure that RIAs correctly control the interactions according to their intentions. However, one difficulty in understanding correct or erroneous behavior lies in the unpredictable context of the running system. Therefore, to support the development of such complex applications, application models, such as state machines, are helpful for understanding the source code [14].

Successful extraction is possible by regarding the document object model (DOM) as the state of web applications [11]. Considering the interactive aspect of RIAs, some researchers have leveraged the DOMs as the states and extracted state machines from Ajax-based RIAs by using dynamic analysis [8], [1], [10]. These approaches can leverage heuristic knowledge to extract state machines by preparing execution scenarios and executing RIAs on their environments. However, extracted state machines depend on the scenarios and environments, i.e., the model might not contain erroneous behavior such as communication failures, if the analysis was done in reliable network conditions.

Therefore, we propose a static approach for extracting state machines from Ajax-based RIAs. In our approach, we assume that *interactions* have the potential to change states in RIAs as shown in Figure 1. We deal with the interactions as state transitions, because extracting DOMs as states inevitably leads to a state space explosion due to interactive DOM manipulations. Our approach is devided into four main steps: 1) using given specifications of Ajax relevant to interactions as rules to distinguish them, 2) creating a call graph, annotating them on it and extracting statements to control the interactions as *interaction controls* (*rule-based annotation of interaction: RBAI*), 3) leveraging a call graph to extract relationships among the interactions (*annotation-based abstraction of call graph: ABAC*), and 4) refining these relationships by analyzing the interaction controls (*interaction control-based refinement: ICBR*). By

```
1 new Ajax.Request ( url, {
2   onSuccess:
3     function(request) { setData(request);};});
4 useData();
```

Figure 2. Example of asynchronous commnucation fault.

extracting the state machines of test scenarios of the correct and wrong behavior, it can help developers to pinpoint the statements in the source code that lead to the erroneous behavior.

Our contributions in this paper are as follows: 1) A static analysis approach for extracting state machines from Ajax-based RIAs (consisting of *RBAI*, *ABAC* and *ICBR*), based on the assumptions that user, server and self *interactions* with the RIAs can change the states of the application. 2) The *JSModeler* tool, an implementation of our approach. 3) An evaluation of our approach against some use cases. Results show that our approach can extract state machines from web pages containing Ajax code within a reasonable time.4) User experiments, which showed that the extracted state machines were comprehensible in terms of scale, label and state divisions to participants. Our approach helped them to find out erroneous behavior due to faults of interaction controls.

The remainder of this paper is organized as follows. First, we discuss issues with development and maintenance of RIAs and present our motivating example in Section II. In Section III, we introduce our static analysis approach for extracting state machines from RIAs. We then discuss our evaluation in Section IV and show related work in Section V. Finally, we conclude in Section VI.

II. Background

In this section, we explain how RIAs provide a rich user experience and show issues in developing and maintaining RIAs. We then give a motivating example that illustrate the issues of complex RIAs.

A. Rich Internet Applications

Rich Internet Applications introduce client-side processing and asynchronous communications in web applications. With these functions, RIAs can continuously process the business-logic on the client-side and asynchronously receive the necessary data to update web pages. Hence, RIAs can provide a rich user experience.

Although asynchronous technologies such as Ajax make RIAs responsive, they can result in unexpected behavior due to nondeterministic elements such as user events and server responses. We give an example of asynchronous communicationi by using a frequently used library: *prototype.js* (prototypejs.org), as shown in Figure 2. In this case, a RIA sends an asynchronous message (line 1), evaluates a successful communication event, sets the response data (lines 2-3), and uses the data (line 4). If this RIA does not immediately receive a response, it might run the use process (`useData`) before evaluating the event. This would cause unexpected behavior in certain scenarios and environments.

To avoid this issue, developers must ensure that RIAs run the use process after the set process (`setData`). Thus, it is important to control to make sure that RIAs behave in any state according to the intentions of the developers. However, developers have difficulties in understanding such erroneous behavior, because they cannot predict all possible contexts of the running system, and because debugging of such complex applications is difficult only using the source code. Although developers can make use of models, web applications rarely provide sufficient documentations due to early releases and frequently specification updates [6]. Therefore, support tools are important to automatically extract models from web applications in a reasonable time to aid developers understanding the source code [14].

B. Motivating Example

We show the source code of an Ajax-based RIA as a motivating example in Figure 3. This is a typical example of a file downloader web service. We explain two faults (lines 23 and 46). The former is caused by a communication control fault and the latter comes from a user event control fault.

(i) Countdown: When users access this web page, the web browser first evaluates an event handler: `onload` (line 6), and then calls a callback function: `countDown` set to the event handler (lines 7-13). In this function, if the `count` is greater than zero, the application updates the progress (line 9) and calls back `countDown` after 1000 msec (line 10). Otherwise, it proceeds in `getPwd` (lines 12, 15-24) then sends an asynchronous message (lines 16-26) by using an included library (prototypejs.org) (lines 2-3). After evaluating an `onSuccess` event, it sets the response data to `pwd`.

We argue that these event handling processes, which we call *interactions* in this paper, can change RIA states. For example, an `onload` event makes this application start to countdown, `setTimeout` makes it continue to countdown, and `onSuccess` makes it set a password string.

(ii) Setting an input form: Though developers intend this application to accept user inputs after it securely sets a password string, it deploys an input form regardless of the communication result (line 23). In case of a communication failure, this application might not behave as developers expected. As just described, developers must deal with error-prone complexities due to indeterministic elements such as communication results and timing of evaluating event handlers.

(iii) Input password and submit: In the function `setForm` (lines 29-36), the application creates and appends `input` widgets for accepting and submitting a user input then sets `inputFormText` and `doSubmit` on `onkeyup` and

```
1  <html><head>
2  <script type="text/javascript"
3          src="./js/prototype.js"></script>
4  <script type="text/javascript"><!--//
5  var count = 5;
6  window.onload = countDown;
7  function countDown() {
8    if(count > 0) {
9      updateProgress(count--);
10     setTimeout(countDown, 1000);
11   } else {
12     getPwd();
13 };};
14 var pwd;
15 function getPwd() {
16   new Ajax.Request("randomPwd.php", {
17     onSuccess: function(request) {
18       pwd = request.responseText;
19       updateProgress(pwd);},
20     onFailure: function(Request) {
21       alert("Fail to get password");}
22   });
23   setForm(); /*** control fault ***/
24 };
25 function enableDownload() {
26   /*** create a link to a download file ***/ };
27 function updateProgress(str) {
28   /*** set string in a progress field  ***/ };
29 function setForm() {
30   var ftext = document.createElement("input");
31   ftext.onkeyup = inputFormText;
32   var fsubmit = document.createElement("input");
33   fsubmit.disabled = true;
34   fsubmit.onclick = doSubmit;
35   /*** append input-form and submit-button ***/
36 };
37 function inputFormText() {
38   var len = $("ftext").value.length;
39   if(0 < len)  $("fsubmit").disabled = false;
40   else         $("fsubmit").disabled = true;
41 };
42 function doSubmit() {
43   var val = $("ftext").value;
44   if(val == pwd) {
45     /*** control fault ***/
46     /*** $("ftext").disabled = true; ***/
47     $("fsubmit").disabled = true;
48     enableDownload();
49   } else {
50     alert("Input password  is invalid");
51 };}; //--></script></head>
52 <body>
53   <div id="progress"></div>
54   <div id="form"></div>
55   <div id="download"></div>
56 </body></html>
```

Figure 3. Motivating example code of Ajax-based RIAs.

`onclick` event handlers of the widgets (lines 31 and 34). In terms of developers' intentions, this form does not initially have any user input; hence, this should disable submit (line 33). Additionally, if that users submit a valid password, they no longer need that input. However, developers can disable the submit button, but might not be able to disable the input form (line 46). In this case, this application might be confusing to users. Thus, developers can increase usability of this application by controlling widget activations, i.e., RIAs also change due to these activations as well as the indeterministic elements.

In this example, we show two types of control faults. The former causes unexpected behavior in certain scenarios and environments and the latter makes a user confuses. Although these faults lead to decreased robustness and usability of RIAs, it is hard for developers to find out such complex behavior which depends on an unpredictable run-time context.

III. Approach and Implementation

We propose *JSModeler*, which statically analyzes a web page of Ajax-based RIAs for extracting state machines. We focus on *interactions* that have the potential to change states in RIAs, as shown in Figure 1. In this section, we illustrate rules to distinguish interactions. We then note four analysis steps in *JSModeler*: Preparation, RBAI, ABAC and ICBR, as show in Figure 4. By extracting the state machines of test scenarios of the correct and wrong behavior, it can help developers to pinpoint the statements in the source code that

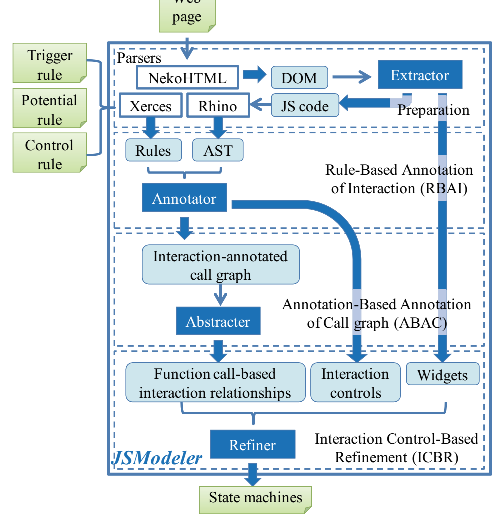

Figure 4. System overview of *JSModeler*.

lead to the erroneous behavior.

A. Input rules to distinguish interactions

Developers describe interactions by using an event handler and a callback function. For example, we show one *user interaction*, keyboard typing as follows.

```
1 <input type="text" maxlength=20
2   onkeyup="inputText();" disabled=true></input>
```

Developers set an `onkeyup` and `inputText` as an event handler and a callback function at an `input` widget (line

2). However, we cannot distinguish event handlers by other attributes (e.g., maxlength). Therefore, we input event handlers as a *Trigger rule* as follows.

```
1 <Trigger event="onkeyup" />
```

Developers also implement *interactions* by using built-in and frequently used library functions such as prototype.js, which handle an event. We call these functions *potential functions*. For example, we show a timeout description, which is a *self interaction* as follows.

```
1 function func() {
2   updateProgress("waiting...");
3   setTimeout(cb, 1000);};
```

This function handles the event, *1000 msec elapsed*, and sets a callback function, cb. However, we also cannot distinguish this *potential function* from other function calls such as updateProgress. Therefore, we input *potential functions* and their arguments, which identify an event handler and a callback function, as a *Potential rule* as follows.

```
1 <Potential function="setTimeout"
2   event="after(arg_2)" callback="arg_1" />
```

We also extract *interaction control* statements which are widget managements and activations. Developers describe the controls by setting widget attributes and corresponding functions as well as *interactions*. If users input any text, this application enables users to submit (line 39 in Figure 3); otherwise, submit is disabled (line 40 in Figure 3) using the attribute disabled. Therefore, to distinguish and extract these statements as interaction controls, we input these attributes, properties and functions as a *Control rule* as follows.

```
1 <Control type="attr" keyword="disabled" />
```

Thus, we can distinguish interactions and their controls. To list event handlers, we refer HTML specification[1] provided by W3C. We also refer DOM specification[2] provided by Mozilla, and list the build-in functions relevant to user interactions and timeout. On the subject of interaction controls, we refer these specifications, and list control attributes (also W3C) and widget manipulation functions (also Mozilla). We define these application-independent specifications as rules in our tool. In addition, developers can define rules for frequently used libraries such as an onSucess event handler of prototype.js in Figure 2. They can describe such user-defined rules by refering the libraries' specifications and these rules are reusable.

B. Interaction-based analysis

Our analysis is divided into four steps shown in Figure 4.

[1]http://www.w3.org/TR/html5/webappapis.html#event-handler-attributes
[2]https://developer.mozilla.org/en/DOM/window

1) Preparation: We input a URL of a web page and XML files describing rules to the *JSModeler*. The tool parses a source code of the page and creates DOM tree, and extracts JavaScript code. Additionally, it extracts widgets which have tag name, id value, activation and display. Finally, it generates Rules and AST (abstract syntax tree) of the code.

2) Rule-based annotation of interaction (RBAI): The *JSModeler* creates a call graph by parsing the AST, which has functions and relationships between function calls as vertices and edges. We can deal with interactions as callback function calls due to event fires. Therefore, the tool annotates the interactions in the call graph with the rules and outputs it as an Interaction-annotated call graph.

3) Annotation-based abstraction of call graph (ABAC): We abstract the annotated call graph focusing on interactions at an abstraction step. In this abstraction, we remove edges which have no event and do not relate to *potential functions*. However, it might occur improper abstractions due to certain function which are called at multiple places. For example, the motivating example calls updateProgress at countDown and a callback function of onSuccess (lines 11 and 21). Though countDown and the callback function should be divided due to an onSuccess event, these function calls are abstracted to one vertex due to updateProgress. Therefore, we remove such function calls before the abstraction. Finally, we generate an function call-based interaction relationships, which express relationships among interactions in terms of function calls.

4) Interaction control-based refinement (ICBR): Since RIAs actually control such function calls by setting control attributes such as disabled, we refine interaction relationships based-on function calls by analyzing interaction controls. These controls have the following properties.

- A target widget that an application controls
- A vertex on which it sets the control statement
- The condition of it reaching the control statement

For example, in Figure 3 (line 39), we can extract an interaction control that controls a widget with an id of fsubmit (*widget*) on inputFormText (*vertex*) under $0 < len$ (*condition*).

We determine whether the interactions are controlled by widget *activations* and *displays*. For example, a user cannot input if a form is disabled by a disabled attribute (*activation*) or the form is not displayed (*display*).

In this step, we input the interaction relationships as a graph, interaction controls, and widgets. We then search from an initial vertex of the graph. By reaching a vertex that has interaction controls, we manipulate *activations* and *displays* of corresponding widgets according to the controls and refine the graph as follows:

- **Divide a state**: We create new vertices for each condition of the controls, if activations and displays of

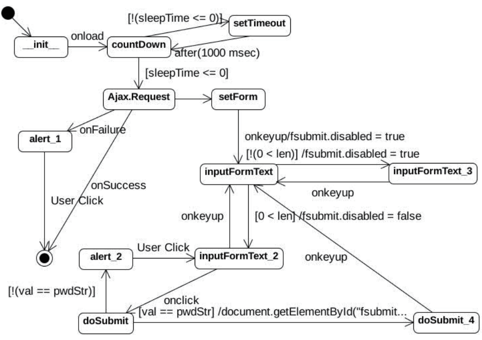

Figure 5. State machines extracted with *JSModeler* from our motivating example that contained control faults.

widgets are changed. We then create edges from the vertex to new ones and describe the conditions and control statements as guard conditions and actions on the edges.

- **Remove a transition**: We find edges coming from the vertex and remove edges having disabled events.
- **Add a transition**: We add edges that have enabled events, if the vertex does not have them already.

Finally, the *JSModeler* outputs state machines from Ajax-based RIAs.

C. JSModeler

Our *JSModeler*[7] tool implements our approach. This tool consists of four components: 1) `Extractor` extracts JavaScript code and widget data with DOM tree. 2) `Annotator` distinguishes interactions with rules and generates a call graph annotated at interaction elements. In addition, it extracts interaction controls. 3) `Abstractor` abstracts the call graph by focusing on the annotations, and 4) `Refiner` refines the abstracted graph with the controls and outputs state machines.

The extracted state machines from our motivating examples with *JSModeler* are shown in Figure 5. Developers can find that the application displays an input form (`setForm`) in which users input a password without waiting for a server response (`onSuccess`) that contains a password string. They can also guess that it continues to accept user inputs (`inputFormText`) despite the fact that users input a valid password (`doSubmit_4`). Thus, the *JSModeler* tool can help developers to find out interaction control faults with extracted state machines.

IV. Evaluation

We have derived the following research hypotheses. Can we automatically extract 1) how comprehensible state machines 2) in a reasonable time? To address these questions, we conducted case studies and evaluated our approach.

A. Case studies

We use an `sForm` (chains.ch) which is an Ajax application for form validation, for our first study. We also extracted state machines from Waseda University's home page (www.waseda.jp/top/index-e.html), because it has a fading menu widget for selecting languages.

B. Evaluation methodology

In our evaluation experiments, we measured the analysis time (T_e) to extract state machines from the case studies. Additionally, we also counted the number of interactions (N_i) in the source code and states (N_s) and transitions (N_t) in the state machines. Moreover, if we construct state machines by combining distinguished interactions, we will obtain $N'_s(= N_i + 1)$ states and $N'_t(= N_i^2 + N_i)$ transitions.

To address how comprehensible the state machines are, we conducted user experiments with two software engineering students (`p1` and `p2`). In the experiment, we asked the participants to score comprehensibility in the terms of *scale*, *labels* and *state divisions* [2].

C. Results and Discussion

We show results of our case studies in Table I.

Table I
RESULTS OF EXTRACTION OF STATE MACHINES IN CASE STUDIES.

	HTML LOC	JS LOC	N_i	N_s	N_t	T_e	N'_s	N'_t
sForm	52	236	9	10	19	77 ms	10	90
Waseda	299	959	14	10	44	309 ms	15	210

Comprehensible state machines: Compared with combining state machines, the number of model elements ($N_s + N_t$) of extracted state machines is 71 % and 76 % smaller. Additionally, extracted state machines were compact enough for the participants to comprehend the behavior of the case studies. Labels in the models were also comprehensible. In addition, we focused on interactions as transitions; still the participants noted that our approach was sufficient for correctly dividing states in the case studies. Thus, we can state that our approach is useful for comprehending the behavior of Ajax-based RIAs.

Reasonable analysis time: Table I shows that our approach could extract state machines from the case studies within 1 second. Therefore, we assume that our approach is also applicable for real-time use.

Comments from participants: We collected the following comments from the questionnaire.

- Extracted state machines helped them to understand page behavior and to find control faults. For example, in `s`Form, users can initially submit without any inputs.
- They could determine pre-conditions for firing interactions by extracting guard conditions.
- They could determine behavior that does not appear the interface, and so on.

Ajax-based RIAs have interactions that are not obvious from the source code and oblivious behavior in their interfaces. Hence, we can help developers understand behavior of the RIAs and find control faults by using extracted state machines.

D. Limitations

Widgets manipulations with strings: Our approach identifies target widgets of interaction controls. Developers can manipulate widgets with strings by using an `innerHTML` attribute of widgets. However, we limit the applications which manage widgets only through built-in functions such as `getElementById` and `createElement`. To solve this limitation, we consider to apply a symbolic execution technique to the *JSModeler*.

Data flow analysis: In addition, developers can manipulate DOM elements through variables. To identify the target widgets of the manipulations, we must analyze the data flow of variables. However, we currently ignore the data flow and analyze only variable declaration statements.

V. Related Work

Our approach is a reverse engineering technique[5] which can provide alternative views from software artifacts for re-documentation and design recover [3]. Such techniques are mainly divided into static and dynamic approaches.

Somé et al. proposed a static analysis of state machines from C programs [12]. To extract state machines, they regarded certain variables as state variables and statically analyze data flow of them. Though they found the state variables by regular expression matching, this approach depends on how developers describe the application states.

In page-based web applications, Ricca et al. introduced model-based analysis and testing [11]. For them, HTML web pages are the central entities. They extract page transition models by analyzing hyperlinks and frames. Thus, dynamic analysis of RIAs can also deal with DOMs as state variables. However, it might cause a state space explosion due to interactive DOM manipulations.

Marchetto et al. presented state-based testing of Ajax applications [8]. They extracted finite state machines by using a dynamic approach. To avoid a state space explosion, they marked functions in which the applications process user events, asynchronous communications, and DOM manipulations, and then trace the DOM snapshots only when the applications call the functions.

In addition, Amalfitano et al. proposed state change criteria to abstract traced DOMs [1]. They prepared execution scenarios and traced DOMs by executing them. Then, they abstracted the traced data with the criteria. Finally, they extracted finite state machines.

Mesbah et al. implemented *CRAWLJAX*, which simulates user inputs by analyzing fireable DOM elements and generates finite state machines by comparing before and after DOMs [9]. They applied this technique into crawling Ajax applications in multiple environments [10].

The dynamic approaches mentioned above depend on execution scenarios and environments, prepared by developers. For example, they might not extract communication failures if working with a reliable network. Our static approach can extract all possible state machines in the source code.

VI. Conclusion and Future Work

We proposed a static approach for extracting state machines from Ajax-based RIAs. Our aim is to reveal control faults by focusing on *interactions*. We implemented the *JSModeler*[7] tool and conducted case studies and user experiments. We conclude that our approach can help developers to comprehend behavior of Ajax-based RIAs.

As our future work, we intend to conduct additional case studies and user experiments with large-scale Ajax applications. Additionally, we plan to establish a method to generate test cases based on the extracted state machines. End-users could input correct and wrong execution paths to automatically test control faults of RIAs. Then, we can localize erroneous statements in the source code by adding traceability between extracted model elements and source code fragments. Moreover, we want to determine what behavior of RIAs is correct or wrong, and devise how developers can recover from specified problems with our approach.

References

[1] D. Amalfitano et al., "An Iterative Approach for the Reverse Engineering of Rich Internet Application User Interfaces", In *Proc. of Int'l Conf. on Internet and Web Applications and Services* , pp. 401-410, 2010.

[2] S. W. Ambler, "The Elements of UML 2.0 Style", In *Cambridge University Press*, 2002.

[3] G. Canfora et al., "New Frontiers of Reverse Engineering", In *Proc. of Future of Software Engineering* , pp. 326-341, 2007.

[4] M. Driver et al., "Rich Internet Application Are the Next Evolution of the Web", In *Tech. report* , Gartner, 2005.

[5] IEEE Std 1219-1998, "IEEE Standard for Software Maintenance", 1998.

[6] M. Jazayeri, "Some Trends in Web Application Development", In *Proc. of Future of Software Engineering* , pp.199-213, 2007.

[7] *JSModeler*, www.honiden.nii.ac.jp/˜maezawa/JSModeler/.

[8] A. Marchetto et al., "State-Based Testing of Ajax Web Applications", In *Proc. of Int'l Conf. on Software Testing, Verification and Validation* , pp.121-130, 2008.

[9] A. Mesbah et al., "Crawling AJAX by Inferring User Interface State Changes", In *Proc. of Int'l Conf. on Web Engineering* , pp. 122-134, 2008.

[10] A. Mesbah et al., "Automated Cross-Browser Compatibility Testing", In *Proc. of Int'l Conf. on Software Engineering* , pp.561-570, 2011.

[11] F. Ricca et al., "Analysis and Testing of Web Applications", In *Proc. of Int'l Conf. on Software Engineering* , pp. 25-34, 2001.

[12] S. S. Somé et al., "Enhancing Program Comprehension with Recevered State Models", In *Proc. of Int'l Workshop on Program Comprehension* , pp.85-93, 2002.

[13] B. Stearn, "XULRunner: A New Approach for Developing Rich Internet Applications", In *IEEE Internet Computing* , vol. 11, pp.67-73, 2007.

[14] P. Tonella, "Reverse Engineering of Object Oriented Code", In *Proc. of Int'l Conf. on Software Engineering* , pp.724-725, 2005.