Automated Verification of Pattern-Based Interaction Invariants in Ajax Applications

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

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

Yoshinori Tanabe
National Institute of Informatics
Tokyo, Japan
y-tanabe@nii.ac.jp

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

***Abstract*—When developing asynchronous JavaScript and XML (Ajax) applications, developers implement Ajax design patterns for increasing the usability of the applications. However, unpredictable contexts of running applications might conceal faults that will break the design patterns, which decreases usability. We propose a support tool called JSVerifier that automatically verifies interaction invariants; the applications handle their interactions in invariant occurrence and order. We also present a selective set of interaction invariants derived from Ajax design patterns, as input. If the application behavior breaks the design patterns, JSVerifier automatically outputs faulty execution paths for debugging. The results of our case studies show that JSVerifier can verify the interaction invariants in a feasible amount of time, and we conclude that it can help developers increase the usability of Ajax applications.**

***Index Terms*—Ajax; Reverse Engineering; Model Checking; Design Pattern;**

I. Introduction

Asynchronous JavaScript and XML (Ajax) applications have become essential platforms for daily life [1] and are an integral part of Google services, Facebook, and Twitter. Asynchronous technologies, such as Ajax, make Web applications responsive [2] and are credited with the 500% increase in Web users compared to a decade ago [3].

A key factor in attracting Web users is usability of Web applications [4]; the ease of using Websites corresponds to well designed navigation [5]. When developing Ajax applications, developers can implement Ajax design patterns [6] for increasing the usability of the applications. Although developers intend to correctly implement the design patterns, unpredictable contexts while running applications might conceal faults that will break the design patterns. We claim that such faults decrease usability; therefore, a technique for verifying whether the application correctly runs according to the implemented design patterns is required.

Several studies have been conducted on state-based analysis and testing of Ajax applications. Some have succeeded in leveraging dynamic analysis techniques because Ajax applications can interactively manipulate an interface by using the document object model (DOM)[1] [7], [8], [9], [10]. Although these dynamic techniques can leverage actual DOMs as the states, DOM-based testing does not help verify the correctness in execution paths that are not part of the scenarios and environments given by the developers. As a static approach, Guha et al. presented a technique for extracting the behavior relevant to asynchronous communications using a control-flow analysis [11]. This technique can be used for detecting runtime server requests that do not match the extracted behavior; however, it is presumed that developers can correctly understand and implement this behavior.

Since developers might incorrectly implement their requirements such as Ajax design patterns (`Implementing`), we previously presented a method for extracting state machines focusing on interactions with applications in order to support program understanding [12]. We assume the interactions as triggers that can change the states of the applications, as described in Section III. However, developers needed to manually determine whether the extracted state machines satisfy interaction invariants, which does not enable them to exhaustively find faulty execution paths.

For this study, we propose a support tool called *JSVerifier*, which automatically verifies interaction invariants in Ajax applications using the *Spin*[2] model checker. The verification method of JSVerifier is mainly divided into the following three steps:

Step1: JSVerifier translates the extracted state machines into an application model that Spin can interpret. To simulate the application behavior, JSVerifier also generates an interaction model (`Extractor`). **Step2:** When developing applications, developers can store information about implemented Ajax design patterns (`IADP information`) into a repository. By obtaining the information via the repository, JSVerifier determines interaction invariants based on Ajax design patterns and property patterns and generates verification formulas (`Formulator`). **Step3:** JSVerifier runs Spin for verifying the correctness of the application behavior with the invariants. If the application behavior does not satisfy the invariants, JSVerifier suggests faulty execution paths (`Verifier`). We assume that developers can debug the applications using these suggestions (`Debugging`). Figure 1 gives an overview of this verification method.

We address the following research questions.

[1]http://www.w3.org/DOM/

[2]http://spinroot.com/

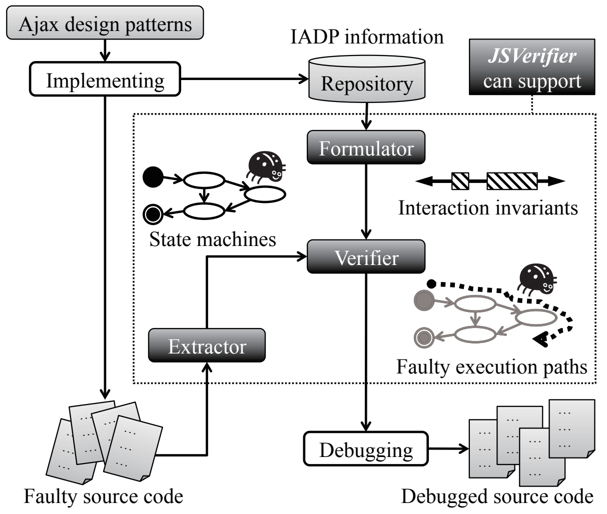


Fig. 1. Overview of verification method used in JSVerifier

RQ1 Can JSVerifier automatically verify interaction invariants with given IADP information?

RQ2 Can JSVerifier output verification results in a feasible time?

Our contributions are as follows:

- An automated verification of interaction invariants in Ajax applications.
- A selective set of interaction invariants based on Ajax design patterns and property patterns.
- Implementation of the verification method of JSVerifier.
- An evaluation experiment with case studies that shows that JSVerifier can verify the invariants in a feasible amount of time.
- Actual faults that JSVerifier can expose in real-world Ajax applications. Some of the faults are difficult to be exposed using testing techniques.

The remainder of this paper is organized as follows. First, we provide background on Ajax application development and give a brief example to explain our work in Section II. In Section III, we describe our proposed tool, JSVerifier. We then discuss our evaluation in Section IV and discuss related work in Section V. Finally, we conclude our work in Section VI.

II. Background

In this section, we note how developers develop Ajax applications focusing on interactions and increase usability of the applications based on Ajax design patterns. We then give a brief example that illustrates an issue of interaction invariants in Ajax applications.

A. Ajax Applications Developments

Asynchronous JavaScript and XML (Ajax) are breed approaches in Web applications [13]. Our research target is the interactions with Ajax applications as shown in Figure 2. By leveraging an Ajax engine on the client side, the applications can asynchronously receive necessary data from servers and partially update a Web page without page transitions, so that it can continuously process user requests on the client side. Thus, this approach makes Web applications responsive [2].

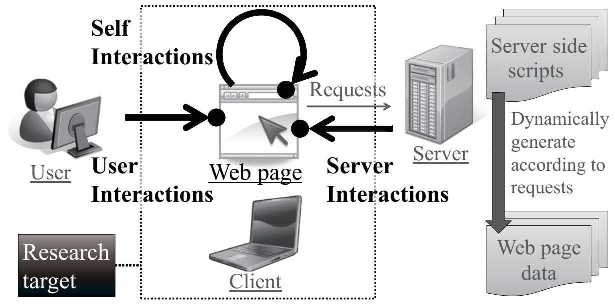


Fig. 2. Interactions with Ajax applications

When developing and maintaining rich Internet applications containing Ajax applications, interactions with the applications need to be considered in order to improve user experience [14]. Since the applications are primarily aimed at providing rich user experience [15], developers are concerned with the following.

- Interactions that the application can handle. We argue that interactions can be classified into user, server, and self interactions corresponding to nondeterministic elements such as user events, asynchronous server responses, and timeouts, as shown in Figure 2.
- Application behavior when handling interactions.

Developers can also control whether the application will handle certain interactions. For determining the application behavior, developers need to recognize the effects of enabling and disabling these interactions [11]. Hence, we can argue that the following is also a concern for developers.

- Application behavior when enabling and disabling interactions.

Unfortunately, developers have difficulties in correctly implementing interactions while considering all possible execution paths because an interaction-based behavior depends on unpredictable contexts at runtime. Despite their greatest efforts, developers may miss certain paths to be executed, which can result in unexpected behavior. Therefore, a support tool is helpful to extract state machines from applications and to determine whether the extracted model contains any unexpected behavior.

B. Ajax Design Patterns

The success of Ajax applications lies in their asynchronous technologies in e-commerce, social networking services, and enterprise systems [16]. These applications can provide rich user experience derived from their asynchronous nature. To attract an increasing number of Web users, Web usability is a key factor in Web applications [4]. Fortunately, Ajax design patterns [6] contain seventy comprehensive findings in term of usability by surveying many real-world Ajax applications. Thus, developers can leverage the Ajax design patterns for increasing usability of applications. For this study, we leverage the findings relevant to an interaction-based behavior in Ajax applications.

```
1  <html><head>
2    <script type="text/javascript"
3       src="./js/prototype.js"></script>
4    <script type="text/javascript"><!--//
5  window.onload = setEventHandler;
6  function setEventHandler() {
7    $("reg_type").onchange = calcPrice;
8    $("reg_attendee").onchange = calcPrice;
9    $("reg_payment").onchange = calcPrice;
10   $("reg_addcart").onclick = addCart; };
11 function calcPrice() {
12   /* calculate and display total price */ };
13 function addCart() {
14 /*  disableAddCard(); // proper control */
15    if(isValidInput()) {
16      reqRunTrans();
17    } else {
18      alert("Invalid user inputs");
19 /*  enableAddCart(); // proper control */ }};
20 function reqRunTrans() {
21   new Ajax.Request("runTrans.php", {
22      method: "GET", parameters: getParams(),
23      onSuccess: succeeded }); };
24 function enableAddCart()  {
25   $("addcart").disabled = false; };
26 function disableAddCart() {
27   $("addcart").disabled = true;  };
28 function succeeded() {
29    disableAll();
30    jumpToConfirm(); };
31 //--></script>
32 </head><body >...
33 Price: $<span id="price">500</span>
34 <div>Type</div>
35 <select id="reg_type">
36 <option id="all" value="350">All days</option>
37 <option id="cnf" value="250">Conference</option>
38 <option id="wsp" value="100">Workshop</option>
39 </select>
40 <div>Attendee...</select>
41 <div>Payment...</select>
42 <span>Quantity: </span>
43   <input id="quantity" type="text" value="1" />
44 <input id="addcart"
45    type="submit"  value="Add to Cart" />
46 </body></html>
```

Fig. 3. Source code of our brief example: Shopping Website.

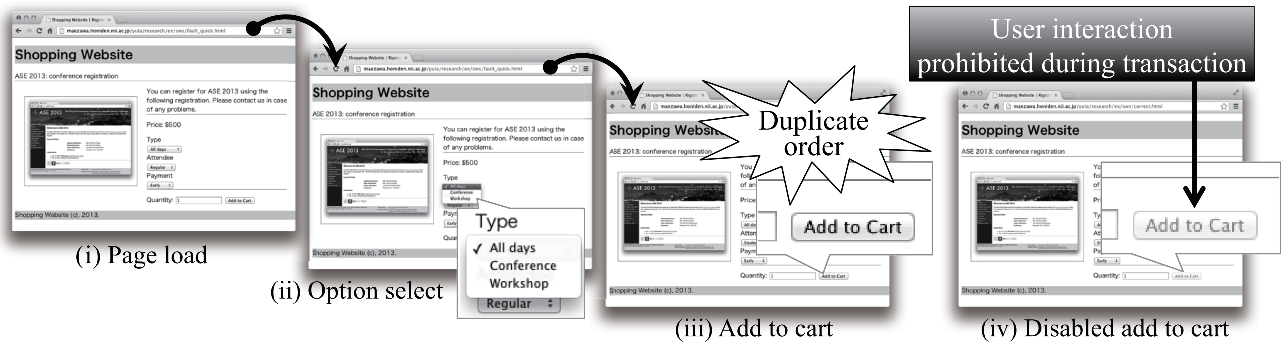


Fig. 4. Screenshots of our brief example shown in Figure 3

However, unpredictable contexts of running applications might conceal faults that will break the design patterns. Although developers test whether the application expectedly runs according to the design patterns, testing techniques do not help verify the correctness of all execution paths. For example, the `User Action` design pattern suggests that applications should register user events at page load; we call such property as a "`user event registration`" in Section III-C. This is because executing user event callback functions before displaying Web page elements might cause erroneous behavior. The current implementation immediately completes the loading of all page elements; however, iterative and incremental development might lengthen the loading time, causing unexpected behavior. Since such software evolution over time might break design patterns [17], we address a challenging issue of automatically verifying whether Ajax application behavior contains faults that are currently concealed but will be exposed.

C. Brief Example

We give the source code and screenshots of an Ajax application as a brief example[3] in Figures 3 and 4. This is a typical Ajax application for shopping on a website where users (i, ii) select item options and (iii, iv) add the item to their cart. This application has a fault that may cause duplicate orders on e-commerce websites such as Amazon[4] and eBay[5]. We illustrate how developers implement and test this application using Ajax design patterns.

First, developers implement the option selection functionality based on the user event registration property. **(i) Page load:** An `onload` event is first evaluated when users visit the Website (line 5). Then, the application calls back a function `setEventHandler` (lines 6-10). **(ii) Option select:** When users select options of an item, the browser evaluates an `onchange` event corresponding to the option widget (lines 7-9). In the interface, users can see the total price according to their selections which is calculated at a callback function `calcPrice` of the events (lines 11-12). Then, developers visit the Website and select the options for testing whether this functionality satisfies the property. Since the application displays the correct price, this test is successful.

Next, developers iteratively implement the item addition functionality. To avoid the duplicate order problem, developers

[3]Running examples are available from http://goo.gl/JE9Vd

[4]Amazon Help: http://goo.gl/Mkbfc

[5]eBay Answer Center: http://goo.gl/WVXqO

require the application of handling the add-to-cart click only once. The User Action design pattern also suggests that Ajax applications can prevent multiple calls of specific user event handlers; we call such property as a "`user event handler singleton`" in Section III-C. **(iii) Add to cart:** Users can also add the item to their cart by clicking a submit button labeled `Add to Cart`. When the button is clicked, an `onclick` event occurs (line 10) and the application processes an `addCart` function (lines 13-19). If the selections are valid (line 15), the application sends an asynchronous request to run a transaction for taking inventories on the server side (lines 16 and 20-23). Otherwise, an alert box appears for users to give valid inputs (lines 17-18). Finally, the application asynchronously receives a server response (lines 23 and 30-32) and jumps to a confirmation page (line 32). To test this property, developers click the button with valid inputs and see that the application cannot handle the click due to the immediate jump. Since a previous test case also passed, developers finally confirm that the application expectedly runs according to two properties derived from the User Action design pattern.

However, the duplicate order problem arises when users unexpectedly double-click the add-to-cart button. As we mentioned in the previous section, it is difficult to expose this duplicate order problem using a testing technique that leverages execution results. This is because the current implementation does not execute such faulty paths in a reliable network and quickly processes the lightweight transaction. Otherwise, the duplicate order problem will be revealed. **(iv) Disabled add to cart:** To avoid the duplicate order problem, developers need to implement the appropriate enabling and disabling of the click so that users cannot interact with the button while the transaction is running (lines 14 and 19).

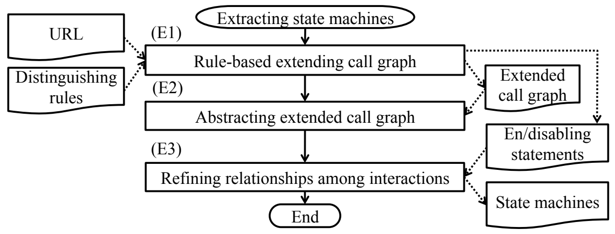
In summary, an interaction-based behavior in Ajax applications is important but difficult for developers to figure out due to unpredictable contexts. Hence, if developers intend to implement and test Ajax design patterns for increasing usability, unpredictable contexts might conceal faults, resulting in an erroneous behavior. Therefore, we investigated the automated verification of interaction invariants based on Ajax design patterns in Ajax applications for revealing faults that are currently concealed but will be exposed.

III. Approach and Implementation

JSVerifier automatically 1) extracts state machines from Ajax applications, as shown in Figure 5, and 2) verifies the correctness and suggests faulty execution paths of interaction invariants, as shown in Figure 6. We present a selective set of interaction invariants based on Ajax design patterns.

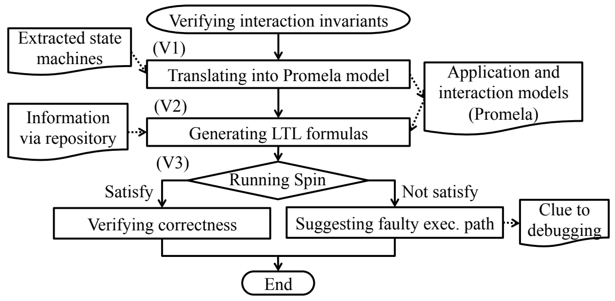
A. Extracting State Machines from Ajax Applications

JSVerifier extracts interaction-based state machines from Ajax applications using a rule-based static analysis technique. To show that JSVerifier deals with nondeterministic behavior of Ajax applications, we explain how it works, although this technique has been proposed in our previous study [12].



Extracting state machines
URL
Distinguishing rules
(E1)
Rule-based extending call graph
(E2)
Abstracting extended call graph
(E3)
Refining relationships among interactions
Extended call graph
En/disabling statements
State machines
End

Fig. 5. Flowchart for extracting state machines



Verifying interaction invariants
Extracted state machines
(V1)
Translating into Promela model
Application and interaction models (Promela)
Information via repository
(V2)
Generating LTL formulas
(V3)
Running Spin
Satisfy
Not satisfy
Verifying correctness
Suggesting faulty exec. path
Clue to debugging
End

Fig. 6. Flowchart for verifying interaction invariants

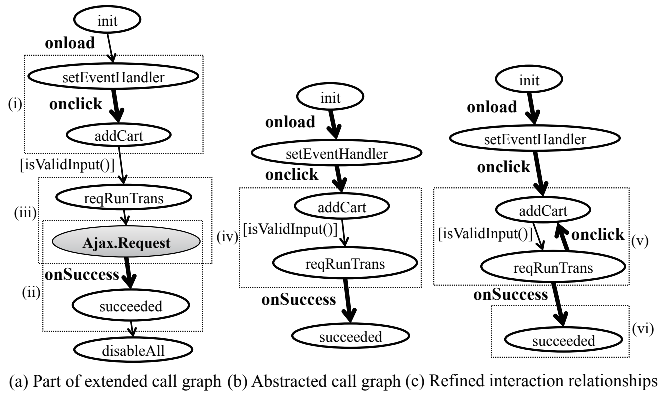


Fig. 7. Partial example of extending, abstracting, and refining call graph of our brief example in Figure 3

We assume the interactions shown in Figure 2 act as triggers that can change the states of applications. The interaction that we focus on in this study corresponds to a function call in response to an event firing. Hence, states and transitions in the extracted state machines represent function calls and the relationships between them. Our analysis technique for extracting state machines consists of three steps, as shown in Figure 5 (E1, E2, and E3).

First, developers input a URL of an Ajax application and rules for distinguishing interaction-related code fragments (distinguishing rules) into JSVerifier, and then it analyzes the source code of the application. Since HTML, CSS, and JavaScript parsers cannot distinguish event attributes, callback objects, event handling functions, and control attributes (for example, `onchange`, `onSuccess`, `Ajax.Request`, and `disabled`) from other syntax elements, developers need to define these elements relevant to the interactions (interaction elements) in the distinguishing rules. However, state machines constructed by combining the interactions might contain many impossible execution paths, resulting in a state space explosion.

TABLE I
DEFINITIONS OF ELEMENTS IN OUR PROMELA MODEL

Element of Promela model	Definition
`active proctype App`	An application process that expresses behavior of an Ajax application
`active proctype Interaction`	An interaction process that interacts with `App`
`App_state`	A state in which an application process is
`App_event`	An event that an application process handles
`Int_event`	An event that an interaction process randomly occurs
`App_ch`	A channel for passing messages of interactions between application and interaction processes
`flg_exit`	An application process that is set to `true` when it transitions to an exit state
`goto_`*App_state*	A goto label expressing an *App_state* of an application
`d_step`	An atomic process for simultaneously setting the state of an application and making events empty

Rule-based extending call graph (E1): To obtain possible relationships among the interactions (interaction relationships), JSVerifier leverages a call graph that represents caller-callee relationships. The call graph does not contain the interaction relationships (e.g., event firing and callback), but JSVerifier extends the call graph in terms of interactions. By parsing the source code and finding event attributes and callback objects, JSVerifier creates relationships between invoked functions (`setEventHandler`) and their corresponding callback functions (`addCart`) and assigns the corresponding event types (`onclick`) to the relationships (lines 6 and 10 in Figure 3 and (i) in Figure 7). In cases of event handling functions, JSVerifier makes connections from the functions (`Ajax.Request`) to the corresponding callback functions (`succeeded`) with the corresponding event types (`onSuccess`) (lines 21 and 23 in Figure 3 and (ii) in Figure 7). Thus, by statically analyzing the source code with the distinguishing rules, it can extend the call graph to contain interaction relationships.

Abstracting extended call graph (E2): Since the extended call graph might have many relationships irrelevant to the interactions, JSVerifier then abstracts the extended call graph focusing on the interaction elements. For example, our brief example runs from `reqRunTrans` to `Ajax.Request` without any interactions ((iii) in Figure 7). In this case, our tool abstracts this caller-callee relationship into the corresponding invoked function of `reqRunTrans` ((iv) in Figure 7). Thus, it can obtain possible interaction relationships from the extended call graph.

Refining relationships among interactions (E3): Additionally, developers can implement enabling and disabling of interactions in Ajax applications. For example, in Figure 3, the add-to-cart button is enabled on line 27 and disabled on line 29. These statements in the application that set parameters on control attributes of DOM elements (en/disabling statements) can be distinguished with the distinguishing rules, similar to the interaction elements. By analyzing the en/disabling statements, JSVerifier adds possible interaction relationships and removes impossible ones. For example, in Figure 7(c), JSVerifier adds an `onclick` relationship from `reqRunTrans` to `addCart` ((v) in Figure 7). However, it does not add any relationship at `succeeded` ((vi) in Figure 7), because all interactions are disabled there (line 31 in Figure 3). As for branch nodes such as `addCart`, JSVerifier skips this analysis because the relationships from the branch nodes are irrelevant to the interactions. By constructing state machines based on the refined interaction relationships, JSVerifier automatically extracts real stateful behavior from Ajax applications, as shown in Figure 9.

In addition to reading the source code, developers can leverage the extracted state machines to understand the stateful behavior of Ajax applications. Although developers may be able to find faults relevant to the interactions using the extracted state machines, the cost may not be negligible for developers to manually and carefully check the behavior when they modify the source code. Additionally, the more interactions developers implement in the applications, the larger the scale of the state machines JSVerifier extracts. Therefore, we can argue that a model checking technique is helpful for revealing erroneous behavior because it can automatically verify the behavior of the applications with given flexible invariants.

B. *Verifying Interaction Invariants in Ajax Applications*

JSVerifier leverages a widely known model checker, *Spin*, for verifying interaction invariants in the extracted state machines. Given flexible invariants expressed as linear temporal logic (LTL) formulas, Spin verifies the correctness of nondeterministic automata described in Process Meta Language (Promela). Accordingly, the model checker is suitable for verifying the extracted state machines that model nondeterministic elements of Ajax applications. Our verification technique consists of three steps, as shown in Figure 6 (V1, V2, and V3).

Translating into Promela model (V1): JSVerifier first translates the extracted state machines into a Promela model. Figure III-B shows the code of a Promela model translated from part of the extracted state machines in Figure 7. Table I lists the definitions of the elements in the Promela model. In this model, JSVerifier outputs an `active proctype` named `App` (lines 5-45) for representing application behavior extracted in the state machines. The `active proctype` is a process that Spin initially instantiates. `App_state`,

```
 1 mtype = { /* define all labels */ };
 2 mtype App_state, App_event, Int_event;
 3 chan App_ch = [0] of { mtype };
 4 bool flg_exit = false;
 5 active proctype App() {
 6   d_step {
 7     App_state = init;
 8     App_event = empty; /* two statements are */
 9     Int_event = empty; /* "set empty" */ }
10 goto_init:
11   do // start of initial do loop
12   :: App_ch?Int_event ->
13     if :: Int_event == onload ->
14       App_event = onload;
15       d_step {
16         App_state = setEventHandler;
17         /* set empty */ }
18 goto_setEventHandler:
19       do
20       :: App_ch?Int_event ->
21         if :: Int_event == onclick ->
22           App_event = onclick;
23           d_step {
24             App_state = addCart;
25             /* set empty */ }
26 goto_addCart:
27           do
28           :: App_ch?Int_event ->
29             if :: Int_event == onSuccess ->
30               App_event = onSucess;
31               d_step {
32                 App_state = exit;
33                 /* set empty */ }
34               goto goto_exit;
35             :: Int_event == onclick ->
36               App_event = onclick;
37               d_step {
38                 App_state = addCart;
39                 /* set empty */ }
40               goto goto_addCart;
41 ...
42   od; // end of initial do loop
43 goto_exit:
44   flg_exit = true;
45 };
46 active proctype Interaction() {
47   do ::
48     if :: flg_exit -> break;
49     :: else ->
50       if
51       :: skip -> App_ch!onload;
52       :: skip -> App_ch!onclick;
53       :: skip -> App_ch!onSuccess;
54 ...   };
```

Fig. 8. Partial example of application and interaction models in Promela

`App_event`, and `Int_event` variables are defined for requirement descriptions (line 2). Additionally, the `App` process nondeterministically receives messages via `App_ch` (line 3), and `flg_exit` is set to *true* when the process exits (lines 4 and 43-44).

States in the state machines are represented in assignment statements to the `App_state` variable (lines 7, 16, 24, 32, and 38). To represent transitions in the state machines, JSVerifier leverages goto functionalities, because Promela unfortunately does not allow describing function calls. By searching all states from an initial one in the state machines, JSVerifier creates goto labels if the states initially appear (lines 10, 18, and 26). Otherwise, it assigns goto statements to corresponding labels (line 40). JSVerifier is exceptionally designed to deals with an exit state in such a way that Spin makes the application process exit (lines 34 and 43-44).

The application model alone is not sufficient to verify the application behavior because the application changes its state as it handles interactions. Therefore, JSVerifier also outputs an `active proctype` named `Interaction` as an interaction model that represents interactions of the application with the user, server, and the application itself (lines 46-54). This process randomly selects an interaction that the application can handle and sends the message of the interaction to the `App` process (lines 47, 51-53). When the `App` process reaches the exit state, Spin also makes the `Interaction` process exit (line 48). Thus, JSVerifier can apply Spin to simulate the stateful behavior of the Ajax application with the application and interaction models.

To verify whether the application model correctly behaves according to implemented Ajax design patterns, it is difficult for developers to define properties to be verified and to correctly express them in verification formulas. Since developers have information about the implemented Ajax design patterns, JSVerifier supports the difficult task by generating correct verification formulas using the information.

C. Mapping Interaction Invariants using Property Patterns

We show a selective set of interaction invariants in Table II. JSVerifier leverages the interaction invariants for generating correct verification formulas so that developers only input information about implemented Ajax design patterns.

Generating LTL formulas (V2): Since Ajax design patterns contain comprehensive findings for increasing usability of Ajax applications, we first define properties in terms of the interactions from the findings (`Ajax design properties`). These Ajax design properties consist of a property name and description; for example, the user event registration (name) property explains that Ajax applications should register user events at an onload callback (description).

To express these properties in correct verification formulas, we also leverage the property pattern mappings for LTL [18], which classifies raw property specifications of a GUI, concurrency logic, and communication protocol, into occurrence and order patterns. These property patterns contain template verification formulas with given states and events of running applications. By relating these property patterns to the Ajax design properties, we can describe LTL templates using `App_state`, `App_event`, and `Int_event` in Table I for JSVerifier. Note that Spin requires a negative property against the expected application behavior. For example, a negative property of the user event registration property means that a user event occurrence (Var 2) precedes that of a page load (Var 1). Therefore, we relate the `Precedence` property pattern in the order pattern to the user event registration (P1 in Table II). For the user event handler singleton property, the `Existence` property pattern in the occurrence pattern is related because the existence of multiple calls (Var 1) of the user event (Var 2) negates the design property (P2 in Table II). Thus, developers

TABLE II
A SELECTIVE SET OF INTERACTION INVARIANTS OF AJAX DESIGN PROPERTIES

P#	Property	Ajax design pattern	Prop. pattern	LTL template with **Var 1** and **Var 2** used in Table III
1	User ev. registration	User Action	Precedence	`App_event != `**PageLoadEv U**` App_event == `**UserEv**
2	User ev. handler singleton	User Action	Existence	`<>(App_state == `**PreventFunc**` && App_event == `**UserEv**`)`
3	Sever resp. before activation	On-Demand JavaScript	Precedence	`App_event != `**SvrResp U**` App_state == `**ActivateFunc**
4	User ev. before submission	Explicit Submission	Precedence	`App_event != `**UserEv U**` App_state == `**SubmitFunc**
5	Process before submission	Live Form	Precedence	`App_state != `**ProcFunc U**` App_state == `**SubmitFunc**

i) select Ajax design properties and ii) input variables in the LTL templates, and then JSVerifier can generate correct LTL formulas using the relationships listed in Table II..

Here, we explain Ajax design properties leveraged in our case studies in Section IV. An `On-Demand JavaScript` Ajax design pattern suggests that Ajax applications should activate specific functionalities (Var 2) using results of the asynchronous data retrieval after the server responses (Var 1) (a `server response before activation` property). Additionally, `Explicit Submission` and `Live Form` Ajax design patterns suggest that Ajax applications should require an explicit users operation (Var 1 in the former) and should process form data (Var1 in the latter) before data submission (Var 2 in both) (`user event before submission` and `process before submission` properties). These properties are related to the Precedence property pattern similar to the user event registration property (P3, P4, and P5 in Table II).

Running Spin (V3): When developers implement and test Ajax applications based on Ajax design patterns, they can input information about implemented Ajax design patterns (IADP information) into a repository of JSVerifier. Developers can input function and event names in the source code as variables for selected Ajax design properties. If the names do not appear in the extracted state machines, JSVerifier can find corresponding states because it stores abstraction information. This information contains to which states the functions are abstracted. Therefore, developers do not need to deeply understand how it works.

By obtaining the information via the repository, as input, JSVerifier automatically generates correct LTL formulas representing the design properties. Then, Spin traverses in the state space of the Promela model and verifies whether the model satisfies the formulas. This verification of correctness can assure developers that the application correctly behaves according to their intentions. Otherwise, Spin outputs a counterexample of the LTL formula as a fault oracle, and then JSVerifier extracts an execution path containing the fault from the oracle. Finally, JSVerifier automatically suggests a debugging clue to the developers as output.

D. Use Scenario and Results of Our Brief Example

We explain a use scenario of JSVerifier and results of our brief example in Figure 9. We assume that JSVerifier can be used in the context of test-driven development, where developers first give test cases of additional functionalities then improve the source code to pass the test cases. Developers first input interaction invariants of implemented Ajax design patterns into the repository of JSVerifier, then they can debug until the invariants are verified as correct.

We now illustrate a JSVerifier use scenario with our brief example. We assume that developers implement functionalities in Ajax applications based on Ajax design patterns. Developers first select the user event registration property and input its variables when implementing the option selection functionality (i, ii). Then, JSVerifier verifies the correctness of this implementation. Next, developers implement the item addition functionality (iii) and give the information for this additional functionality. At this time, JSVerifier determines that the current implementation does not satisfy the additional invariant and suggests a corresponding faulty execution path on the extracted state machines. We assume that developers can debug using the faulty execution path (iv). Finally, developers confirm that the application correctly runs according to the invariants.

IV. EVALUATION

To answer the following research questions, we conducted case studies and evaluated JSVerifier.

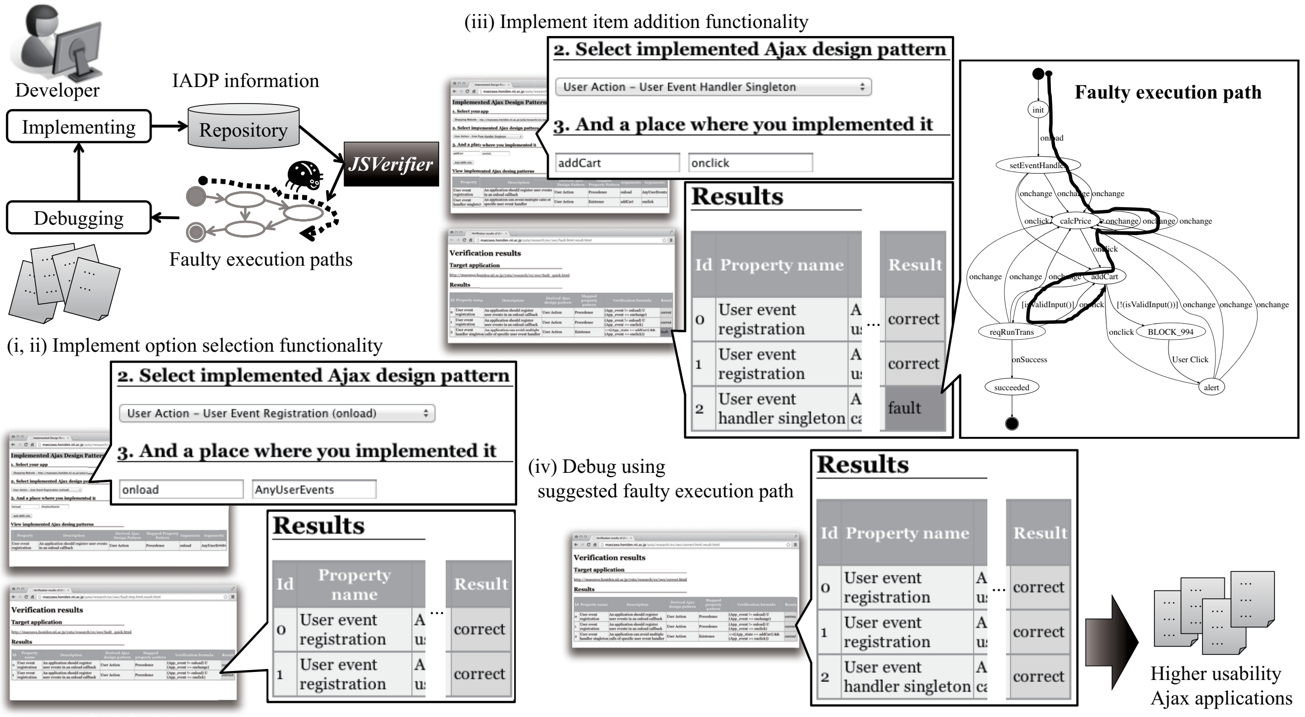
RQ1 Can JSVerifier automatically verify interaction invariants with given IADP information?

RQ2 Can JSVerifier output verification results in a feasible time?

A. Case Studies

We used two real-world Ajax applications; `sForm`[6] is an Ajax application for form validation and `Login With Ajax (LWA)`[7] is an Ajax application plugin on WordPress[8] for replacing a login widget. These applications are runnable and their source codes are available for debugging. Additionally, we prepared a sample Ajax application called `FileDLer`[9]. We had implemented this application for motivating our previous study. Table III shows HTML, CSS, and JavaScrip lines of codes in these applications (*LoC*). The 100-1K LoC range represents the small-medium size in Ajax applications.

[6]http://www.chains.ch/2008/01/26/ajax-form-validation-sform/
[7]http://wordpress.org/extend/plugins/login-with-ajax/
[8]http://wordpress.org/
[9]http://maezawa.honiden.nii.ac.jp/yuta/research/ex/fd/


(iii) Implement item addition functionality
Developer
IADP information
Implementing
Repository
JSVerifier
Debugging
Faulty execution paths
(i, ii) Implement option selection functionality
2. Select implemented Ajax design pattern
User Action – User Event Handler Singleton
3. And a place where you implemented it
addCart
onclick
2. Select implemented Ajax design pattern
User Action – User Event Registration (onload)
3. And a place where you implemented it
onload
AnyUserEvents
Faulty execution path
init
onload
setEventHandler
onchange
calcPrice
onclick
addCart
[isValidInput()]
[!(isValidInput())]
reqRunTrans
BLOCK_994
onSuccess
User Click
succeeded
alert
(iv) Debug using suggested faulty execution path
Higher usability Ajax applications

Results

Id	Property name		Result
0	User event registration	A us ...	correct
1	User event registration	A us	correct
2	User event handler singleton	A ca	fault

Results

Id	Property name		Result
0	User event registration	A us ...	correct
1	User event registration	A us	correct

Results

Id	Property name		Result
0	User event registration	A us ...	correct
1	User event registration	A us	correct
2	User event handler singleton	A ca	correct

Fig. 9. JSVerifier use scenario and results of our brief example

B. Evaluation Methodology

In our case studies, we first determined properties to be verified and corresponding variables in the three applications, as shown in Table III. Since we did not know the intent of the original developers of sForm and LWA, we conducted the determinations based on the source code fragments. Here, we explain the representative determinations.[10] In the source code of sForm, we found a `validateIt` function for validating form data and a `submit` function for submitting the data. We inferred that the validation process should be executed before the form submission, and then we determined the process before submission property and the functions as the variables in sForm. As for LWA, we determined the user event registration property because of a `jQuery.ready`[11] fragment in the source code. The `ready` is usually implemented for attaching all other event handlers as an alternative to an `onload` event. Therefore, we inferred that the application should register all user events at the ready. Note that we had known expected behavior and injected faults in FileDLer, therefore, we determined appropriate properties and variables according to our intentions. We stored the properties and the variables as IADP information into a repository of JSVerifier.

Next, we ran JSVerifier with the repository. JSVerifier measured the extraction and verification times (T_e and T_v). Additionally, JSVerifier outputted the extracted state machines, verification results and faulty execution paths. We debugged the applications using the paths, and then ran JSVerifier again. Finally, we confirmed whether JSVerifier could verify the correctness.

[10]All the determinations are available from http://goo.gl/4fQ0d
[11]http://api.jquery.com/ready/

C. Results and Discussion

Automated verification (RQ1): JSVerifier could automatically verify the correct and wrong application behavior. We tested sForm according to the faulty execution path, and then found that sForm actually handled the form submission without any user inputs in the form. Then, we debugged sForm to initially disable the submit button and confirmed that JSVerifier verified the correctness. These results represent that JSVerifier could expose executable faults in Ajax applications. Additionally, we searched the user events on the faulty execution in LWA, and then found them in the HTML source code. These implementations conformed to undesirable ones shown in the User Action Ajax design pattern. We debugged them according to a solution suggested in the design pattern so that JSVerifier could output the correct results. Note that this faulty execution path could not be executed in the current implementation. These results represent that JSVerifier could also expose inexecutable but concealed faults. As for FileDLer, we had already had correct and faulty version of the applications. We confirmed that JSVerifier could suggest the faulty execution paths as expected. Therefore, we argue that JSVerifier correctly works for our verification method.

Feasible verification time (RQ2): In our case studies, JSVerifier could automatically extract state machines from our case studies and verify pattern-based interaction invariants within several seconds, as shown in Table III. Most of the runtime was required for initializing HTML, CSS, and JavaScript parsers. These extraction and verification times increase linearly with the number of implemented interactions in Ajax applications. We confirmed that the applications contained the sufficient number of the interactions for using JSVerifier in our

TABLE III
RESULTS OF OUR CASE STUDIES

	LoC	P#	Property	Var 1	Var 2	T_e (msec)	T_v (msec)	Result
sForm	314	1	User event registration	onload	onblur	3170	1013	Correct
				onload	onclick		750	Correct
		5	Process before submission	validateIt	submit		992	Fault
		4	User event before submission	onclick	submit		672	Correct
LWA	2084	1	User event registration	ready	submit	6487	543	Correct
				ready	click		533	Correct
				ready	onfocus		752	Fault
				ready	onblur		686	Fault
FileDLer	251	1	User event registration	onload	onkeyup	5144	840	Correct
				onload	onclick		743	Correct
		3	Server response before activation	onSuccess	inputFormText		1035	Fault
		4	User event before submission	onclick	doSubmit		718	Correct
		2	User event handler singleton	doDownload	onkeyup		827	Fault
				doDownload	onclick		690	Correct

case studies. Additionally, JSVerifier could expose the actual faults in real-world Ajax applications in a feasible amount of time. Therefore, we argue that JSVerifier can be applicable for real-time use.

Costs for debugging faults: Although JSVerifier can suggest faulty execution paths on extracted state machines as clues to debugging, developers need to locate faults in the source code using the clues. As our future work, we plan to leverage solutions in Ajax design patterns for fault localization.

D. Threats to Validity

Internal validity threats: We considered two external factors that might affect results in our case studies. We found sForm and LWA via the Web, so these applications do not affect the internal of JSVerifier, and the results from using these applications represent the usefulness of JSVerifier for our verification method. However, we implemented FileDLer ourselves to contain the faults relevant to interaction invariants, which may be a threat to internal validity. Therefore, we intend to conduct additional case studies using real-world Ajax applications such as sForm and LWA.

Additionally, we defined Ajax design properties from Ajax design patterns and related property patterns to the properties. Although these definitions and relationships might affect the internal validity of JSVerifier, results of our case studies showed that JSVerifier could verify the correct and wrong behavior of the applications and expose the actual faults in the real-world applications. As our future work, we intend to present an exhaustive set of Ajax design properties and to evaluate the usefulness of JSVerifier for exposing actual faults in the additional case studies.

External validity threats: With regards to the generality of our approach, JSVerifier leveraged Spin, so it could only deal with requirements that were expressed in LTL formulas. However, there are requirements that are beyond the descriptive capability of LTL, for example, the reachability of certain states from any other states. To verify such behavior, we consider leveraging `SMV`[12], which can verify the correctness using computation tree logic (CTL) formulas. CTL formulas allow developers to express requirements involving the reachability. Additionally, we are currently working on outputting timed automata for `Uppaal`[13] using JSVerifier.

Moreover, in our case studies, although we could leverage `LWA` that were sufficiently practical, `sForm` and `FileDLer` were simple Ajax applications. We need to obtain more experimental results from analyzing large-scale and practical Ajax applications.

E. Limitations

Data-intensive impossible execution paths: JSVerifier analyzes only enabling and disabling statements to determine whether an Ajax application can handle the interactions. In fact, developers can implement such interaction controls also using data flows. In our brief example in Figure 3, user inputs for selecting options can never be invalid (line 15), which means that the application can never proceed to the state corresponding to invalid user inputs (lines 17-19). Such data-intensive impossible execution paths can be dealt with by DOM-based dynamic analysis [7], [8], [9], [10]. Hence, we will extend JSVerifier to leverage contributions of these related work in order to construct a hybrid approach.

However, we want to claim that the impossible execution paths would be executable fault candidates, for example, in case that other developers modify the source code of open source Ajax applications or that users install other application plug-ins. Therefore, we argue that our pessimistic analysis is valuable to verify the application behavior containing the fault candidates.

Additional Ajax design patterns: We assume that interaction invariants in Ajax applications derive from Ajax design

[12]http://www.cs.cmu.edu/~modelcheck/smv.html
[13]http://www.uppaal.com/

patterns. In fact, developers have their original Ajax design pattern and flexible requirements. When adding new design patterns, developers need to define verification properties in the design patterns and to relate appropriate property patterns to the properties. Otherwise, developers can use JSVerifier with raw LTL verification formulas.

V. Related Work

Our approach leverages a reverse engineering technique and a model checking technique. The former aims to provide alternative views of software artifacts, such as for redocumenting programs and recovering design patterns [19]. Especially, a view of state machines can improve the code understandability of developers [20]. The latter is an approach for verifying finite state machines representing concurrent systems, such as sequential circuit designs and communication protocols [21].

Ricca et al. introduced state-based analysis and testing of Web applications [22]. Although they regarded Web pages as states, an Ajax technology allows the applications to change their states in a single page. Hence, Marchetto et al. presented a state-based testing technique for Ajax applications [7], [23]. Their tool called `ReAjax` could trace execution results of actual DOMs, extract finite state machines from the trace data, and generate test cases based on the state machines. However, developers needed to manually and exhaustively execute Ajax applications for tracing sufficient execution logs.

Mesbah et al. implemented `Crawljax` that could simulate user events by finding fireable DOM elements and extract finite state machines from Ajax applications [8]. To detect DOM-based faults such as dead clickable elements, the tool analyzed invariants of the DOM structure. Additionally, they ran the tool on multiple browser environments for cross-browser compatibility testing [24]. Moreover, their extended tool called `Cilla` could find faults relevant to the presentation of the applications during crawling [25]. Although they mentioned that static analysis techniques had limitations for revealing faults of Ajax applications due to interactive DOM manipulations, we can apply a static approach for extracting and verifying state machines by focusing on the interactions with the applications.

Amalfitano et al. proposed several Ajax application-independent state change criteria and an interactive process for extracting finite state machines [9]. They constructed a tool called `CreRIA` that could suggest state changes based on the criteria and developers could accept or reject the suggestions during executing Ajax applications. Although CreRIA effectively leveraged the heuristics of developers, this interactive process was less contribution to the automation.

The above dynamic approaches leveraging execution results cannot verify the correctness of the application behavior because these tools may not execute all possible paths in the applications. Our motivation for constructing JSVerifier is that Ajax applications may have inexecutable faults to be exposed.

Additionally, Arzti et al. presented a method for prioritizing event sequences using historical execution results to improve code coverage [10]. This approach also dealt with DOM-based faults because HTML, CSS and JavaScript errors are defined in their language specifications. As for valid event sequences, there is no general definition of correct or wrong behavior. Therefore, developers leverage Ajax design patterns to define Ajax application-independent invariant occurrence and order among interactions as interaction invariants.

Guha et al. proposed a static approach for testing vulnerability of Ajax applications [11]. Their framework could analyze control flows in the JavaScript code and extract the `request graph` containing sequences of asynchronous communications. Developers can use this framework for detecting runtime server requests that do not match the sequences. However, their approach was presumed that developers can correctly understand and implement the application behavior. Additionally, they addressed existing faults to be detected. Considering that developers will modify the source code of open source Ajax applications and that users will install Ajax application plugins, we claim that developers should debug fault candidates that will be exposed. JSVerifier can verify the application behavior containing such the fault candidates. Furthermore, as a limitation of their approach, they pointed out that it is necessary to analyze disabling event handlers to precisely monitor Ajax application behavior. Our analysis scope covers the application behavior containing such enabling and disabling interactions.

Blewitt et al. conducted detection of GoF design patterns in Java using semantic constraints [17]. Their concern was that software evolution over time would cause breaking the design patterns in their original forms on the implementations. Additionally, Ghabi et al. addressed an issue of maintaining requirements-to-code traces [26] because the software evolution also causes invalidating a requirements traceability matrix. In this study, we assume that information about implemented Ajax design patterns are correct, and it would be interesting to how JSVerifier works with the invalid information.

VI. Conclusions and Future Work

We presented a support tool, JSVerifier, and a selective set of interaction invariants based on Ajax design patterns and property patterns. Our aim was to automatically verify the correctness of an interaction-based stateful behavior in Ajax applications according to implemented Ajax design patterns. The results of our case studies showed that JSVerifier could verify the application behavior and could expose the actual faults in the real-world Ajax applications in a feasible amount of time. We concluded that JSVerifier could help developers increase the usability of Ajax applications.

As our future work, we plan to provide an exhaustive set of the interaction invariants. We are going to exhaustively define interaction-related properties from Ajax design patterns and relate property patterns to the properties. Additionally, we intend to support developers to debug Ajax applications using the suggested faulty execution paths. We are considering using solutions in the design patterns to suggest debugging methods. Moreover, we will conduct additional case studies using real-world, large-scale, and practical Ajax applications.

References

[1] B. Stearn, “Xulrunner: A new approach for developing rich internet applications,” *IEEE Internet Computing*, vol. 11, no. 3, pp. 67–73, 2007.

[2] L. D. Paulson, “Building rich web applications with ajax,” *Computer*, vol. 38, no. 10, pp. 14–17, 2005.

[3] Internet World Stats. (2011, Dec.) World internet usage statistics news and world population stats. [Online]. Available: www.internetworldstats.com/stats.htm

[4] J. Nielsen and H. Loranger, *Prioritizing Web Usability*. Berkeley, CA: New Riders Press, 2006.

[5] C. E. Downing and C. Liu, “Assessing web site usability in retail electronic commerce,” in *Proc. Computer Software and Applications Conf. (COMPSAC’11)*, Jul. 2011, pp. 144–151.

[6] M. Mahemoff, *Ajax Design Patterns*. O’Reilly Media, Inc., 2006.

[7] A. Marchetto, P. Tonella, and F. Ricca, “State-based testing of ajax web applications,” in *Proc. Int’l Conf. on Software Testing, Verification and Validation (ICST’08)*, Apr. 2008, pp. 121–130.

[8] A. Mesbah and A. van Deursen, “Invariant-based automatic testing of ajax user interfaces,” in *Proc. Int’l Conf. on Software Engineering (ICSE’09)*, May 2009, pp. 210–220.

[9] D. Amalfitano, A. R. Fasolino, and P. Tramontana, “An iterative approach for the reverse engineering of rich internet application user interfaces,” in *Proc. Int’l Conf. on Internet and Web Applications and Services (ICIW’10)*, May 2010, pp. 401–410.

[10] S. Artzi, J. Dolby, S. H. Jensen, A. Moller, and F. Tip, “A framework for automated testing of javascript web applications,” in *Proc. Int’l Conf. on Software Engineering (ICSE’11)*, May 2011, pp. 571–580.

[11] A. Guha, S. Krishnamurthi, and T. Jim, “Using static analysis for ajax intrusion detection,” in *Proc. Int’l World Wide Web Conf. (WWW’09)*, Apr. 2009, pp. 561–570.

[12] Y. Maezawa, H. Washizaki, and S. Honiden, “Extracting interaction-based stateful behavior in rich internet applications,” in *Proc. European Conf. on Software Maintenance and Reengineering (CSMR’12)*, Mar. 2012, pp. 423–428.

[13] Garrett, Jesse James. (2005, Feb.) Ajax: A new approach to web applications. [Online]. Available: www.adaptivepath.com/ideas/ajax-new-approach-web-applications

[14] J. Duhl, “White paper: Rich internet applicationb,” IDC, Tech. Rep., Nov. 2003.

[15] M. Driver, R. Valdes, and G. Phifer, “Rich internet application are the next evolution of the web,” Gartner, Inc., Tech. Rep., May 2005.

[16] J. Farrell and G. S. Nezlek, “Rich internet applications the next stage of application development,” in *Proc. Int’l Conf. on Information Technology Interfaces (ITI’07)*, Jun. 2007, pp. 413–418.

[17] A. Blewitt, A. Bundy, and I. Stark, “Automatic verification of design pattern in java,” in *Proc. Int’l Conf. on Automated Software Engineering (ASE’05)*, Nov. 2005, pp. 224–232.

[18] Alavi, Hamid and Avrunin, George and Corbett, James and Dillon, Laura and Dwyer, Matt and Pasareanu, Corina. (2013, May) Property pattern mappings for ltl. [Online]. Available: patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml

[19] G. Canfora and M. D. Penta, “New frontiers of reverse engineering,” in *Proc. Future of Software Engineering (FOSE’07)*, May 2007, pp. 326–341.

[20] S. S. Somé and T. C. Lethbridge, “Enhancing program comprehension with recovered state models,” in *Proc. Int’l Workshop on Program Comprehension (IWPC’02)*, Jun. 2002, pp. 85–93.

[21] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[22] F. Ricca and P. Tonella, “Analysis and testing of web applications,” in *Proc. Int’l Conf. on Software Engineering (ICSE’01)*, May 2001, pp. 25–34.

[23] A. Marchetto, P. Tonella, and F. Ricca, “Reajax: a reverse engineering tool for ajax web applications,” *Software, IET*, vol. 6, no. 1, pp. 33–49, 2012.

[24] A. Mesbah and M. R. Prasad, “Automated cross-browser compatibility testing,” in *Proc. Int’l Conf. on Software Engineering (ICSE’11)*, May 2011, pp. 561–570.

[25] A. Mesbah and S. Mirshokraie, “Automated analysis of css rules to support style maintenance,” in *Proc. Int’l Conf. on Software Engineering (ICSE’12)*, May 2012, pp. 408–418.

[26] A. Ghabi and A. Egyed, “Code patterns for automatically validating requirements-to-code traces,” in *Proc. Int’l Conf. on Automated Software Engineering (ASE’12)*, Sep. 2012, pp. 200–209.