Delft University of Technology
Software Engineering Research Group
Technical Report Series

Invariant-Based Automatic Testing of AJAX User Interfaces

Ali Mesbah and Arie van Deursen

Report TUD-SERG-2009-005

TUD-SERG-2009-005

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: *This paper is a pre-print of:*

Ali Mesbah and Arie van Deursen. Invariant-Based Automatic Testing of AJAX User Interfaces. In Proceedings of the 31st International Conference on Software Engineering (ICSE'09), Research Papers, Vancouver, Canada, IEEE Computer Society, 2009.

Invariant-Based Automatic Testing of AJAX User Interfaces

Ali Mesbah
Software Engineering Research Group
Delft University of Technology
The Netherlands
A.Mesbah@tudelft.nl

Arie van Deursen
Software Engineering Research Group
Delft University of Technology
The Netherlands
Arie.vanDeursen@tudelft.nl

Abstract

AJAX-based Web 2.0 applications rely on stateful asynchronous client/server communication, and client-side runtime manipulation of the DOM tree. This not only makes them fundamentally different from traditional web applications, but also more error-prone and harder to test.

We propose a method for testing AJAX applications automatically, based on a crawler to infer a flow graph for all (client-side) user interface states. We identify AJAX-specific faults that can occur in such states (related to DOM validity, error messages, discoverability, back-button compatibility, etc.) as well as DOM-tree invariants that can serve as oracle to detect such faults. We implemented our approach in ATUSA, a tool offering generic invariant checking components, a plugin-mechanism to add application-specific state validators, and generation of a test suite covering the paths obtained during crawling. We describe two case studies evaluating the fault revealing capabilities, scalability, required manual effort and level of automation of our approach.

1 Introduction

Recently, many new web trends have appeared under the *Web 2.0* umbrella, changing the web significantly, from read-only static pages to dynamic user-created content and rich interaction. Many *Web 2.0* sites rely heavily on AJAX (Asynchronous JAVASCRIPT and XML) [8], a prominent enabling technology in which a clever combination of JAVASCRIPT and Document Object Model (DOM) manipulation, along with asynchronous client/server delta-communication [16] is used to achieve a high level of user interactivity on the web.

With this new change comes a whole set of new challenges, mainly due to the fact that AJAX shatters the metaphor of a web 'page' upon which many classic web technologies are based. One of these challenges is testing such applications [6, 12, 14]. With the ever-increasing demands on the quality of *Web 2.0* applications, new techniques and models need to be developed to test this new class of software. How to automate such a testing technique is the question that we address in this paper.

In order to detect a fault, a testing method should meet the following conditions [18, 20]: *reach* the fault-execution, which causes the fault to be executed, *trigger* the error-creation, which causes the fault execution to generate an incorrect intermediate state, and *propagate* the error, which enables the incorrect intermediate state to propagate to the output and cause a detectable output error.

Meeting these reach/trigger/propagate conditions is more difficult for AJAX applications compared to classical web applications. During the past years, the general approach in testing web applications has been to request a response from the server (via a hypertext link) and to analyze the resulting HTML. This testing approach based on the page-sequence paradigm has serious limitations meeting even the first (reach) condition on AJAX sites. Recent tools such as Selenium[1] use a capture/replay style for testing AJAX applications. Although such tools are capable of executing the fault, they demand a substantial amount of manual effort on the part of the tester.

Static analysis techniques have limitations in revealing faults which are due to the complex run-time behavior of modern rich web applications. It is this dynamic run-time interaction that is believed [10] to make testing such applications a challenging task. On the other hand, when applying dynamic analysis on this new domain of web, the main difficulty lies in detecting the various doorways to different dynamic states and providing proper interface mechanisms for input values.

In this paper, we discuss challenges of testing AJAX (Section 3) and propose an automated testing technique for finding faults in AJAX user interfaces. We extend our AJAX crawler, CRAWLJAX (Sections 4–5), to infer a state-flow

[1] http://selenium.openqa.org

graph for all (client-side) user interface states. We identify AJAX-specific faults that can occur in such states and generic and application-specific invariants that can serve as oracle to detect such faults (Section 6). From the inferred graph, we automatically generate test cases (Section 7) that cover the paths discovered during the crawling process. In addition, we use our open source tool called ATUSA (Section 8), implementing the testing technique, to conduct a number of case studies (Section 9) to discuss (Section 10) and evaluate the effectiveness of our approach.

2 Related Work

Modern web interfaces incorporate client-side scripting and user interface manipulation which is increasingly separated from server-side application logic [23]. Although the field of rich web interface testing is mainly unexplored, much knowledge may be derived from two closely related fields: traditional web testing and GUI application testing.

Traditional Web Testing. Benedikt *et al.* [3] present VeriWeb, a tool for automatically exploring paths of multi-page web sites through a crawler and detector for abnormalities such as navigation and page errors (which are configurable through plugins). VeriWeb uses SmartProfiles to extract candidate input values for form-based pages. Although VeriWeb's crawling algorithm has some support for client-side scripting execution, the paper provides insufficient detail to determine whether it would be able to cope with modern AJAX web applications. VeriWeb offers no support for generating test suites as we do in Section 7.

Tools such as WAVES [10] and SecuBat [11] have been proposed for automatically assessing web application security. The general approach is based on a crawler capable of detecting data entry points which can be seen as possible points of security attack. Malicious patterns, e.g., SQL and XSS vulnerabilities, are then injected into these entry points and the response from the server is analyzed to determine vulnerable parts of the web application.

A model-based testing approach for web applications was proposed by Ricca and Tonella [19]. They introduce ReWeb, a tool for creating a model of the web application in UML, which is used along with defined coverage criteria to generate test-cases. Another approach was presented by Andrews *et al.* [1], who rely on a finite state machine together with constraints defined by the tester. All such model-based testing techniques focus on classical multi-page web applications. They mostly use a crawler to infer a navigational model of the web. Unfortunately, traditional web crawlers are not able to crawl AJAX applications [14].

Logging user session data on the server is also used for the purpose of automatic test generation [7, 21]. This approach requires sufficient interaction of real web users with the system to generate the necessary logging data. Session-based testing techniques are merely focused on synchronous requests to the server and lack the complete state information required in AJAX testing. Delta-server messages [16] from the server response are hard to analyze on their own. Most of such delta updates become meaningful after they have been processed by the client-side engine on the browser and injected into the DOM.

Exploiting static analysis of server-side implementation logic to abstract the application behavior is another testing approach. Artzi *et al.* [2] propose a technique and a tool called Apollo for finding faults in PHP web applications that is based on combined concrete and symbolic execution. The tool is able to detect run-time errors and malformed HTML output. Halfond and Orso [9] present their static analysis of server-side Java code to extract web application request parameters and their potential values. Such techniques have limitations in revealing faults that are due to the complex run-time behavior of modern rich web applications.

GUI Application Testing. Reverse engineering a model of the desktop (GUI), to generate test cases has been proposed by Memon *et al.* [13]. AJAX applications can be seen as a hybrid of desktop and web applications [16], since the user interface is composed of components and the interaction is event-based. However, AJAX applications have specific features, such as the asynchronous client/-server communication and dynamic DOM-based user interface, which make them different from traditional GUI applications [12], and therefore require other testing tools and techniques.

Current AJAX Testing Approaches. The server-side of AJAX applications can be tested with any conventional testing technique. On the client, testing can be performed at different levels. Unit testing tools such as JsUnit[2] can be used to test JAVASCRIPT on a functional level. The most popular AJAX testing tools are currently capture/replay tools such as Seleninum, WebKing[3], and Sahi[4], which allow DOM-based testing by capturing events fired by user (tester) interaction. Such tools have access to the DOM, and can assert expected UI behavior defined by the tester and replay the events. Capture/replay tools demand, however, a substantial amount of manual effort on the part of the tester [13].

Marchetto *et al.* [12] have recently proposed an approach for state-based testing of AJAX applications. They use traces of the application to construct a finite state machine. Sequences of semantically interacting events in the model are used to generate test cases once the model is refined by the tester. In our approach, we crawl the AJAX application, simulating real user events on the user interface and infer the abstract model automatically.

[2] http://jsunit.net
[3] http://www.parasoft.com/jsp/products/home.jsp?product=WebKing
[4] http://sahi.co.in/w/

3 AJAX Testing Challenges

In AJAX applications, the state of the user interface is determined dynamically, through event-driven changes in the browser's DOM that are only visible after executing the corresponding JAVASCRIPT code. The resulting challenges can be explained through the reach/trigger/propagate conditions as follows.

Reach. The event-driven nature of AJAX presents the first serious testing difficulty, as the event model of the browser must be manipulated instead of just constructing and sending appropriate URLs to the server. Thus, simulating user events on AJAX interfaces requires an environment equipped with all the necessary technologies, e.g., JAVASCRIPT, DOM, and the XMLHttpRequest object used for asynchronous communication.

One way to *reach* the fault-execution automatically for AJAX is by adopting a web crawler, capable of detecting and firing events on clickable elements on the web interface. Such a crawler should be able to exercise all user interface events of an AJAX site, crawl through different UI states and infer a model of the navigational paths and states. We proposed such a crawler for AJAX, discussed in our previous work [14], which will be briefly explained in Section 4.

Trigger. Once we are able to derive different dynamic states of an AJAX application, possible faults can be triggered by generating UI events. In addition input values can cause faulty states. Thus, it is important to identify input data entry points, which are primarily comprised of DOM forms. In addition, executing different sequences of events can also trigger an incorrect state. Therefore, we should be able to generate and execute different event sequences.

Propagate. In AJAX, any response to a client-side event is injected into the single-page interface and therefore, faults propagate to and are manifested at the DOM level. Hence, access to the dynamic run-time DOM is a necessity to be able to analyze and detect the propagated errors.

Automating the process of assessing the correctness of test case output is a challenging task, known as the oracle problem [24]. Ideally a tester acts as an oracle who knows the expected output, in terms of DOM tree, elements and their attributes, after each state change. When the state space is huge, it becomes practically impossible. In practice, a baseline version, also known as the Gold Standard [5], of the application is used to generate the expected behavior. Oracles used in the web testing literature are mainly in the form of HTML comparators [22] and validators [2].

4 Deriving AJAX States

Here, we briefly outline our AJAX crawling technique and tool called CRAWLJAX [14]. CRAWLJAX can exercise client side code, and identify clickable elements that change the state within the browser's dynamically built DOM. From these state changes, we infer a *state-flow graph*, which captures the states of the user interface, and the possible event-based transitions between them.

We define an AJAX UI state change as a change on the DOM tree caused either by server-side state changes propagated to the client, or client-side events handled by the AJAX engine. We model such changes by recording the paths (events) to these DOM changes to be able to navigate between the different states.

Inferring the State Machine. The state-flow graph is created incrementally. Initially, it only contains the root state and new states are created and added as the application is crawled and state changes are analyzed. The following components participate in the construction of the graph: CRAWLJAX uses an *embedded browser* interface (with different implementations: IE, Mozilla) supporting technologies required by AJAX; A *robot* is used to simulate user input (e.g., `click`, `mouseOver`, text input) on the embedded browser; The *finite state machine* is a data component maintaining the state-flow graph, as well as a pointer to the current state; The *controller* has access to the browser's DOM and analyzes and detects state changes. It also controls the robot's actions and is responsible for updating the state machine when relevant changes occur on the DOM. The algorithm used by these components to actually infer the state machine is discussed below: the full algorithm along with its testing-specific extensions is shown in Algorithms 1 and 2 (Section 8).

Detecting Clickables. CRAWLJAX implements an algorithm which makes use of a set of *candidate elements*, which are all exposed to an event type (e.g., click, mouseOver). In automatic mode, the candidate clickables are labeled as such based on their HTML tag element name and attribute constraints. For instance, all elements with a tag `div`, `a`, and `span` having attribute `class="menuitem"` are considered as candidate clickable. For each candidate element, the crawler fires a click on the element (or other event types, e.g., mouseOver), in the embedded browser.

Creating States. After firing an event on a candidate clickable, the algorithm compares the resulting DOM tree with the way as it was just before the event fired, in order to determine whether the event results in a state change. If a change is detected according to the Levenshtein edit distance, a new state is created and added to the state-flow graph of the state machine. Furthermore, a new edge is created on the graph between the state before the event and the current state.

Processing Document Tree Deltas. After a new state has been detected, the crawling procedure is recursively called to find new possible states in the partial changes made to the DOM tree. CRAWLJAX computes the differences between the previous document tree and the current one, by means of an enhanced *Diff* algorithm to detect AJAX par-

tial updates which may be due to a server request call that injects new elements into the DOM.

Navigating the States. Upon completion of the recursive call, the browser should be put back into the previous state. A dynamically changed DOM state does not register itself with the browser history engine automatically, so triggering the 'Back' function of the browser is usually insufficient. To deal with this AJAX crawling problem, we save information about the elements and the order in which their execution results in reaching a given state. We then can reload the application and follow and execute the elements from the initial state to the desired state. CRAWLJAX adopts XPath to provide a reliable, and persistent element identification mechanism. For each state changing element, it reverse engineers the XPath expression of that element which returns its exact location on the DOM. This expression is saved in the state machine and used to find the element after a reload. Note that because of side effects of the element execution and server-side state, there is no guarantee that we reach the exact same state when we traverse a path a second time. It is, however, as close as we can get.

5 Data Entry Points

In order to provide input values on AJAX web applications, we have adopted a reverse engineering process, similar to [3, 10], to extract all exposed data entry points. To this end, we have extended our crawler with the capability of detecting *DOM forms* on each newly detected state (this extension is also shown in Algorithm 1).

For each new state, we extract all form elements from the DOM tree. For each form, a `hashcode` is calculated on the attributes (if available) and the HTML structure of the input fields of the form. With this hashcode, custom values are associated and stored in a database, which are used for all forms with the same code.

If no custom data fields are available yet, all data, including input fields, their default values, and options are extracted from the DOM form. Since in AJAX forms are usually sent to the server through JAVASCRIPT functions, the `action` attribute of the form does not always correspond to the server-side entry URL. Also, any element (e.g., A, DIV) could be used to trigger the right JAVASCRIPT function to submit the form. In this case, the crawler tries to identify the element that is responsible for form submission. Note that the tester can always verify the *submit* element and change it in the database, if necessary. Once all necessary data is gathered, the form is inserted automatically into the database. Every input form provides thus a data entry point and the tester can later alter the database with additional desired input values for each form.

If the crawler does find a match in the database, the input values are used to fill the DOM form and submit it. Upon submission, the resulting state is analyzed recursively by the crawler and if a valid state change occurs the state-flow graph is updated accordingly.

6 Testing AJAX States Through Invariants

With access to different dynamic DOM states we can check the user interface against different constraints. We propose to express those as invariants on the DOM tree, which we thus can check automatically in any state. We distinguish between invariants on the DOM-tree, between DOM-tree states, and application-specific invariants. Each invariant is based on a fault model [5], representing AJAX-specific faults that are likely to occur and which can be captured through the given invariant.

6.1 Generic DOM Invariants

Validated DOM. Malformed HTML code can be the cause of many vulnerability and browser portability problems. Although browsers are designed to tolerate HTML malformedness to some extent, such errors have led to browser crashes and security vulnerabilities [2]. All current HTML validators expect all the structure and content be present in the HTML source code. However, with AJAX, changes are manifested on the single-page user interface by partially updating the dynamic DOM through JAVASCRIPT. Since these validators cannot execute client-side JAVASCRIPT, they simply cannot perform any kind of validation.

To prevent faults, we must make sure that the application has a valid DOM on every possible execution path and modification step. We use the DOM tree obtained after each state change while crawling and transform it to the corresponding HTML instance. A W3C HTML validator serves as oracle to determine whether errors or warnings occur. Since most AJAX sites rely on a single-page interface, we use a *diff* algorithm to prevent duplicate occurrences of failures that may be the result of a previous state.

No Error Messages in DOM. Our state should never contain a string pattern that suggests an error message [3] in the DOM. Error messages that are injected into the DOM as a result of client-side (e.g., 404 Not Found, 400 Bad Request) or server-side errors (e.g., Session Timeout, 500 Internal Server Error, MySQL error) can be detected automatically. The prescribed list of potential fault patterns should be configurable by the tester.

Other Invariants. In line with the above, further generic DOM-invariants can be devised, for example to deal with accessibility, link discoverability, or security constraints on the DOM at any time throughout the crawling process. We omit discussion of these invariants due to space limitations.

6.2 State Machine Invariants

Besides constraints on the DOM-tree in individual states, we can identify requirements on the state machine and its transitions.

No Dead Clickables. One common fault in classical web applications is the occurrence of *dead links* which point to a URL that is permanently unavailable. In AJAX, clickables that are supposed to change the state by retrieving data from the server, through JAVASCRIPT in the background, can also be broken. Such error messages from the server are mostly swallowed by the AJAX engine, and no sign of a dead link is propagated to the user interface. By listening to the client/server request/response traffic after each event (e.g., through a proxy), dead clickables can be detected.

Consistent Back-Button. A fault that often occurs in AJAX applications is the broken Back-button of the browser. As explained in Section 4, a dynamically changed DOM state does not register itself with the browser history engine automatically, so triggering the 'Back' function makes the browser completely leave the application's web page. It is possible to programatically register each state change with the browser history and frameworks are appearing which handle this issue. However, when the state space increases, errors can be made and some states may be ignored by the developer to be registered properly. Through crawling, upon each new state, one can compare the expected state in the graph with the state after the execution of the Back-button and find inconsistencies automatically.

6.3 Application-specific Invariants

We can define invariants that should always hold and could be checked on the DOM, specific to our AJAX application in development. In our case study, Section 9.2, we describe a number of application-specific invariants. Constraints over the DOM-tree can be easily expressed as invariants in Java, for example through an XPath expression. Typically, this can be coded into one or two simple Java methods. The resulting invariants can be used to dynamically search for invariant violations.

7 Testing AJAX Paths

While running the crawler to derive the state machine can be considered as a first full test pass, the state machine itself can be further used for testing purposes. For example, it can be used to execute different paths to cover the state machine in different ways. In this section, we explain how to derive a test suite (implemented in JUnit) automatically from the state machine, and how this suite can be used for testing purposes.

```
@Test
public void testcase1() {
browser.goToUrl(url);

/*Element-info: SPAN class=expandable-hitarea */
browser.fireEvent(new Eventable(new Identification(
  "xpath", "//DIV[1]/SPAN[4]"), "onclick"));

Comp.AssertEquals(oracle.getState("S_1").getDom(),
  browser.getDom());

/*Element-info: DIV class=hitarea id=menuitem2 */
browser.fireEvent(new Eventable(new Identification(
  "xpath", "//SPAN[2]/DIV[2]"), "onmouseover"));

Comp.AssertEquals(oracle.getState("S_3").getDom(),
              browser.getDom());

/*Element-info: Form, A href=#submit */
handleForm(2473584);

Comp.AssertEquals(oracle.getState("S_4").getDom(),
              browser.getDom());
}

private void handleForm(long formId) {
 Form form = oracle.getForm(formId);
 if (form != null) {
  FormHandler.fillFormInDom(browser, form);
  browser.fireEvent(form.getSubmit());
 }
}
```

Figure 1. A generated JUnit test case.

To generate the test suite, we use the *K shortest paths* [25] algorithm which is a generalization of the shortest path problem in which several paths in increasing order of length are sought. We collect all sinks in our graph, and compute the shortest path from the index page to each of them. Loops are included once. This way, we can easily achieve all transitions coverage.

Next, we transform each path found into a JUnit test case, as shown in Figure 1. Each test case captures the sequence of events from the initial state to the target state. The JUnit test case can fire events, since each edge on the state-flow graph contains information about the event-type and the element the event is fired on to arrive at the target state. We also provide all the information about the clickable element such as tag name and attributes, as code comments in the generated test method. The test class provides API's to access the DOM (`browser.getDom()`) and elements (`browser.getElementBy(how, value)`) of the resulting state after each event, as well as its contents.

If an event is a form submission (annotated on the edge), we generate all the required information for the test case to retrieve the corresponding input values from the database and insert them into the DOM, before triggering the event.

After each event invocation the resulting state in the browser is compared with the expected state in the database which serves as oracle. The comparison can take place at different levels of abstraction ranging from textual [22] to schema-based similarity [15].

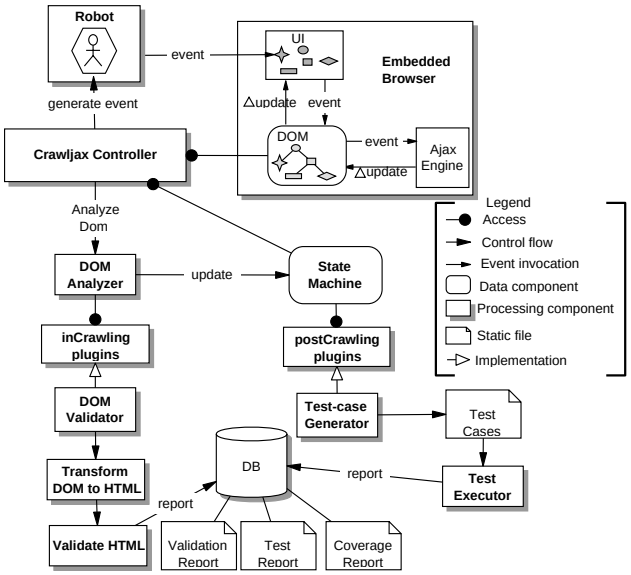

Figure 2. Processing view of ATUSA.

Algorithm 1 Pre/postCrawling hooks

```
1:  procedure START (url, Set tags)
2:  browser ← initEmbeddedBrowser(url)
3:  robot ← initRobot()
4:  sm ← initStateMachine()
5:  preCrawlingPlugins(browser)
6:  crawl(null)
7:  postCrawlingPlugins(sm)
8:  end procedure
9:  procedure CRAWL (State ps)
10: cs ← sm.getCurrentState()
11: Δupdate ← diff(ps, cs)
12: analyseForms(Δupdate)
13: Set C ← getCandidateClickables(Δupdate, tags)
14: for c ∈ C do
15:     generateEvent(cs, c)
16: end for
17: end procedure
18: procedure ANALYSEFORMS (State cs)
19: for form ∈ cs.getForms() do
20:     id ← getHashCode(form)
21:     dbForm ← database.getForm(id)
22:     if dbForm == null then
23:         extractInsertForm(form, id)
24:     else
25:         fillFormInDom(browser, dbForm)
26:         generateEvent(cs, dbForm.getSubmit())
27:     end if
28: end for
29: end procedure
```

Test-case Execution. Usually extra coding is necessary for simulating the environment where the tests will be run, which contributes to the high cost of testing [4]. We provide a framework to run all the generated tests automatically using a real web browser and generate success/failure reports. At the beginning of each test case the embedded browser is initialized with the URL of the AJAX site under test. For each test case, the browser is first put in its initial index state. From there, events are fired on the clickable elements (and forms filled if present). After each event invocation, assertions are checked to see if the expected results are seen on the web application's new UI state.

The generated JUnit test suite can be used in several ways. First, it can be run as is on the current version of the AJAX application, but for instance with a different browser to detect browser incompatibilities. Furthermore, the test suite can be applied to altered versions of the AJAX application to support regression testing: For the unaltered user interface, the test cases should pass, and only for altered user interface code failures might occur (also helping the tester to understand what has truly changed). The typical use of the derived test suite will be to take apart specific generated test cases, and augment them with application-specific assertions. In this way, a small test suite arises capturing specific fault-sensitive click trails.

8 Tool Implementation: ATUSA

We have implemented our testing approach in an open source tool called ATUSA (Automatically Testing UI States of AJAX), available through our website.[5] It is based on the crawling capabilities of CRAWLJAX and provides plugin *hooks* for testing AJAX applications at different levels. Its architecture can be divided into three phases:

preCrawling occurs after the application has fully been loaded into the browser. Examples include authentication plugins to log onto the system and checks on the HTML source code.

inCrawling occurs after each detected state change, different types of invariants can be checked through plugins such as Validated DOM, Consistent Back-button, and No Error Messages in DOM.

postCrawling occurs after the crawling process is done and the state-flow graph is inferred fully. The graph can be used, for instance, in a plugin to generate test cases from.

Algorithms 1 and 2 show the hooks along the crawling process. For each phase, ATUSA provides the tester with specific APIs to implement plugins for validation and fault detection. ATUSA offers generic invariant checking components, a plugin-mechanism to add application-specific state validators, and generation of a test suite from the inferred state-flow graph. Figure 2 depicts the processing view of ATUSA, showing only the DOM Validator and Test Case Generator as examples of possible plugin implementations.

ATUSA supports looking for many different types of faults in AJAX-based applications, from errors in the DOM instance, to errors that involve the navigational path, e.g.,

[5] http://spci.st.ewi.tudelft.nl/atusa/

Algorithm 2 Incrawling hook while deriving AJAX states

```
1:  procedure GENERATEEVENT (State cs, Clickable c)
2:  robot.fireEvent(c)
3:  dom ← browser.getDom()
4:  if distance(cs.getDom(), dom) > τ then
5:      xe ← getXpathExpr(c)
6:      ns ← State(dom)
7:      sm.addState(ns)
8:      sm.addEdge(cs, ns, Event(c, xe))
9:      sm.changeState(ns)
10:     inCrawlingPlugins(ns)
11:     crawl(cs)
12:     sm.changeState(cs)
13:     if browser.history.canBack then
14:         browser.history.goBack()
15:     else
16:         browser.reload()
17:         List E ← sm.getPathTo(cs)
18:         for e ∈ E do
19:             robot.fireEvent(e)
20:         end for
21:     end if
22: end if
23: end procedure
```

constraints on the length of the deepest paths [3], or number of clicks to a certain state. Whenever a fault is detected, the error report along the causing execution path is saved in the database so that it can be reproduced later easily.

Implementation. ATUSA is implemented in Java 1.6. The *state-flow graph* is based on the JGrapht library. The implementation details of the crawler can be found in [14]. The plugin architecture is implemented through the Java Plugin Framework (JPF) and we use Hibernate to store the data in the database. Apache Velocity templates assist us in the code generation process of JUnit test cases.

9 Empirical Evaluation

In order to assess the usefulness of our approach in supporting modern web application testing, we have conducted a number of case studies, set up following Yin's guidelines [26].

Goal and Research Questions. Our goal in this experiment is to evaluate the fault revealing capabilities, scalability, required manual effort and level of automation of our approach. Our research questions can be summarized as:

RQ1 What is the fault revealing capability of ATUSA?
RQ2 How well does ATUSA perform? Is it scalable?
RQ3 What is the automation level when using ATUSA and how much manual effort is involved in the testing process?

9.1 Study 1: TUDU

Our first experimental subject is the AJAX-based open source TUDU [6] web application for managing personal todo lists, which has also been used by other researchers [12]. The server-side is based on J2EE and consists of around 12K lines of Java/JSP code, of which around 3K forms the presentation layer we are interested in. The client-side extends on a number of AJAX libraries such as DWR[7] and Scriptaculous[8], and consists of around 11k LOC of external JAVASCRIPT libraries and 580 internal LOC.

To address RQ3 we report the time spent on parts that required manual work. For RQ1-2, we configured ATUSA through its `properties` file (1 minute), setting the URL of the deployed site, the tag elements that should be included (`A, DIV`) and excluded (`A:title=Log out`) during the crawling process, the depth level (2), the similarity threshold (0.89), and a maximum crawling time of 60 minutes. Since TUDU requires authentication, we wrote (10 minutes) a `preCrawling` plugin to log into the web application automatically.

As shown in Table 1, we measure average DOM string size, number of candidate elements analyzed, detected clickables and states, detected data entry points, detected faults, number of generated test cases, and performance measurements, all of which are printed in a log file by ATUSA after each run.

In the initial run, after the login process, ATUSA crawled the TUDU application, finding the doorways to new states and detecting all possible data entry points recursively. We analyzed the data entry points in the database and provided each with custom input values (15 minutes to evaluate the input values and provide useful values). For the second run, we activated (50 seconds) the DOM Validator, Back-Button, Error Detector, and Test Case Generator plugins and started the process. ATUSA started crawling and when forms were encountered, the custom values from the database were automatically inserted into the browser and submitted. Upon each detected state change, the invariants were checked through the plugins and reports were inserted into the database if faults were found. At the end of the crawling process, a test suite was generated from the inferred state-flow graph.

To the best of our knowledge, there are currently no tools that can automatically test AJAX dynamic states. Therefore, it is not possible to form a base-line for comparison using, for instance, external crawlers. To assess the effectiveness of the generated test suite, we measure code coverage on the client as well as the presentation-tier of the server. Although the effectiveness is not directly implied by code coverage, it is an objective and commonly used indicator of the quality of a test suite [9]. To that end, we instrumented the presentation part of the server code (`tudu-dwr`) with Clover and the client-side JAVASCRIPT libraries with JSCoverage[9], and

[6] `http://tudu.sourceforge.net`

[7] `http://directwebremoting.org`

[8] `http://script.aculo.us`

[9] `http://siliconforks.com/jscoverage/`

LOC Server-side	LOC Client-side	DOM string size	Candidate Clickables	Detected Clickables	Detected States	Detected Entry Points	DOM Violations	Back-button	Generated Test Cases	Coverage Server-side	Coverage Client-side	Detected Faults	Manual Effort	Performance
3k	11k (ext) 580 (int)	24908 (byte)	332	42	34	4 forms 21 inputs	182	false	32	73%	35% (ext) 75% (int)	80%	26.5 (minutes)	5.6 (minutes)

Table 1. TUDU **case study.**

deployed the web application. For each test run, we bring the TUDU database to the original state using a SQL script. We run all the test cases against the instrumented application, through ATUSA's embedded browser, and compute the amount of coverage achieved for server- and client-side code. In addition, we manually seeded 10 faults, capable of causing inconsistent states (e.g., DOM malformdness, adding values longer than allowed by the database, adding duplicate todo items, removing all items instead of one) and measured the percentage of faults detected. The results are presented in Table 1.

Findings. Based on these observations we conclude that: The use of ATUSA can help to reveal generic faults, such as DOM violations, automatically; The generated test suite can give us useful code coverage (73% server-side and 75% client-side; Note that only partial parts of the external libraries are actually used by TUDU resulting in a low coverage percentage) and can reveal most DOM-based faults, 8 of the 10 seeded faults were detected, two faults were undetected because during the test execution, they were silently swallowed by the JAVASCRIPT engine and did not affect the DOM. It is worth mentioning that increasing the depth level to 3 significantly increased the measured crawling time passed the maximum 60 minutes, but dit not influence the fault detection results. The code coverage, however, improved by approximately 10%; The manual effort involved in setting up ATUSA (less than half an hour in this case) is minimal; The performance and scalability of the crawling and testing process is very acceptable (it takes ATUSA less than 6 minutes to crawl and test TUDU, analyzing 332 clickables and detecting 34 states).

9.2 Study 2: Finding Real-Life Bugs

Our second case study involves the development of an AJAX user interface in a small commercial project. We use this case study to evaluate the manual effort required to use ATUSA (RQ3), and to assess the capability of ATUSA to find faults that actually occurred during development (RQ1).

Subject System. The case at hand is Coachjezelf (CJZ, "Coach Yourself"),[10] a commercial application allowing high school teachers to assess and improve their teaching skills. CJZ is currently in use by 5000-6000 Dutch teachers, a number that is growing with approximately 1000 paying users every year.

The relevant part for our case is the interactive table of contents (TOC), which is to be synchronized with an actual content widget. In older versions of CJZ this was implemented through a Java applet; in the new version this is to be done through AJAX, in order to eliminate a Java virtual machine dependency.

The two developers working on the case study spent around one week (two person-weeks) building the AJAX solution, including requirements elicitation, design, understanding and evaluating the libraries to be used, manual testing, and acceptance by the customer.

The AJAX-based solution made use of the `jQuery`[11] library, as well as the `treeview`, `history-remote`, and `listen` plugins for `jQuery`. The libraries comprise around 10,000 lines of JAVASCRIPT, and the custom code is around 150 lines of JAVASCRIPT, as well as some HTML and CSS code.

Case study setup. The developers were asked (1) to try to document their design and technical requirements using invariants, and (2) to write the invariants in ATUSA plugins to detect errors made during development. After the delivery of the first release, we evaluated (1) how easy it was to express these invariants in ATUSA; and (2) whether the (generic or application-specific) plugins were capable of detecting faults.

Application-Specific Invariants. Two sets of invariants were proposed by the developers. The first essentially documented the (external) `treeview` component, capable of (un)folding tree structures (such as a table of contents).

The `treeview` component operates by setting HTML class attributes (such as `collapsible`, `hit-area`, and `lastExpandable-hitarea`) on nested list structures. The corresponding style sheet takes care of properly displaying the (un)folded (sub)trees, and the JAVASCRIPT intercepts clicks and re-arranges the class attributes as needed.

Invariants were devised to document constraints on the class attributes. As an example, the `div`-element immediately below a `li`-element that has the class `expandable` should have class `expandable-hitarea`. Another invari-

[10] See `www.coachjezelf.nl` for more information (in Dutch).

[11] `jquery.com`

Failure	Cause	Violated Invariant	Invariant type
Images not displayed	Base URL in dynamic load	Dead Clickables	Generic
Broken synchronization in IE	Invalid HTML id	DOM-validator	Generic
Inconsistent history	Issue in `listen` library	Back-Button	Generic
Broken synchronization in IE	Backslash versus slash	Consistent current page	Specific
Corrupted table	Coding error	`treeview` invariants, Consistent current page	Specific
Missing TOC Entries	Incomplete input data	Consistent current page	Specific

Table 2. Faults found in CJZ-AJAX.

```
//case one: warn about collapsible divs within expandable items
String xpathCase1 =  "//LI[contains(@class,'expandable')]/DIV[contains(@class,'collapsable')]";

//case two: warn about collapsible items within expandable items
String xpathCase2 = "//LI[contains(@class,'expandable')]/UL/LI[contains(@class,'collapsable')]";
```

Figure 3. Example invariants expressed using XPath in Java.

ant is that expandable list items (which are hidden) should have their CSS display type set to "none".

The second set of invariants specifically dealt with the code written by the developers themselves. This code took care of synchronizing the interactive display of the table of contents with the actual page shown. Clicking links within the page affects the display of the table of contents, and vice versa.

This resulted in essentially two invariants: one to ensure that within the table of contents at most one path (to the current page) would be open, and the other that at any time the current page as marked in the table of contents would actually be displayed in the content pane.

Expressing such invariants on the DOM-tree was quite easy, requiring a few lines of Java code using XPath. An example is shown in Figure 3.

Failures Detected. At the end of the development week, ATUSA was used to test the new AJAX interface. For each type of application-specific invariant, an `inCrawling` plugin was added to ATUSA. Six types of failures were automatically detected: three through the generic plugins, and three through the application-specific plugins just described. An overview of the type of failures found and the invariant violations that helped to detect them is provided in Table 2.

The application-specific failures were all found through two invariant types: the *Consistent current page*, which expresses that in any state the table and the actual content should be in sync, and the *treeview invariants*. Note that for certain types of faults, for instance the treeview corrupted table, a very specific click trail had to be followed to expose the failure. ATUSA gives no guarantee of covering the complete state of the application, however, since it tries a huge combination of clickables recursively, it was able to detect such faults, which were not seen by developers when the application was tested manually.

Findings. Based on these observations we conclude that: The use of ATUSA can help to reveal bugs that are likely to occur during AJAX development and are difficult to detect manually; Application-specific invariants can help to document and test the essence of an AJAX application, such as the synchronization between two widgets; The manual effort in coding such invariants in Java and using them through plugins in ATUSA is minimal.

10 Discussion

Automation Scope. User interface testing is a broad term, dealing with testing how the application and the user interact. This typically is manual in nature, as it includes inspecting the correct display of menus, dialog boxes, and the invocation of the correct functionality when clicking them. The type of user interface testing that we propose does not replace this manual testing, but augments it: Our focus is on finding programming faults, manifested through failures in the DOM tree. As we have seen, the highly dynamic nature and complexity of AJAX make it error-prone, and our approach is capable of finding such faults automatically.

Invariants. Our solution to the oracle problem is to include invariants (as also advocated by, e.g., Meyer [17]). AJAX applications offer a unique opportunity for specifying invariants, thanks to the central DOM data structure. Thus, we are able to define generic invariants that should hold for all AJAX applications, and we allow the tester to use the DOM to specify dedicated invariants. Furthermore, the state machine derived through crawling can be used to express invariants, such as correct Back-button behavior. Again, this state machine can be accessed by the tester to specify his or her own invariants. These invariants make our approach much more sophisticated than *smoke tests* for user interfaces (as proposed by e.g., Memon [13]) — which we can achieve thanks to the presence of the DOM and state machine data structures. Note that just running CRAWLJAX would correspond to conducting a smoke test: the difficulty with web applications (as opposed to, e.g., Java Swing applications) is that it is very hard to determine when a failure

occurs – which is solved in ATUSA through the use of invariants.

Generated versus hand-coded JAVASCRIPT. The case studies we conducted involve two different popular JAVASCRIPT libraries in combination with hand-written JAVASCRIPT code. Alternative frameworks exist, such as Google's Web Toolkit (GWT)[12] in which most of the client-side code is generated. ATUSA is entirely independent of the way the AJAX application is written, so it can be applied to such systems as well. This will be particularly relevant for testing the custom JAVASCRIPT code that remains to be hand-written, and which can still be tricky and error-prone. Furthermore, ATUSA can be used by the developers of such frameworks, to ensure that the generated DOM states are correct.

Manual Effort. The manual steps required to run ATUSA consist of configuration, plugin development, and providing custom input values, which for the cases conducted took less than an hour. The hardest part is deciding which application-specific invariants to adopt. This is a step that is directly connected with the *design* of the application itself. Making the structural invariants explicit not only allows for automated testing, it is also a powerful design documentation technique. Admittedly, not all web developers will be able to think in terms of invariants, which might limit the applicability of our approach in practice. Those capable of documenting invariants can take advantage of the framework ATUSA provides to actually implement the invariants.

Performance and Scalability. Since the state space of any realistic web application is huge and can cause the well-know *state explosion problem*, we provide the tester with a set of configurable options to constrain the state space such as the maximum search depth level, the similarity threshold, maximum number of states per domain, maximum crawling time, and the option of ignoring external links and links that match some pre-defined set of regular expressions.The main component that can influence the performance and scalability is the crawling part. The performance of ATUSA in crawling an AJAX site depends on many factors such as the speed at which the server can handle requests, how fast the client-side JAVASCRIPT can update the interface, and the size of the DOM tree. ATUSA can scale to sites comprised of thousands of states easily.

Application Size. The two case studies both involve around 10,000 lines of JAVASCRIPT library code, and several hundred lines of application code. One might wonder whether this is too small to be representative. However, our results are based on *dynamic* analysis rather than static code analysis, hence the amount of code is not the determining factor. Instead, the size of the derived state machine is the factor limiting the scalability of our approach, which is only moderately (if at all) related to the size of the JAVASCRIPT code.

Threats to Validity. Some of the issues concerning the *external* validity of our empirical evaluation have been covered in the above discussion on scope, generated code, application size, and scalability. Apart from the two case studies described in the paper, we conducted two more (on TaskFreak[13] and the Java PETSTORE 2.0[14]), which gave comparable results. With respect to *internal* validity, we minimized the chance of ATUSA errors by including a rigorous JUnit test suite. ATUSA, however, also makes use of many (complex) third party components, and we did encounter several problems in some of them. While these bugs do limit the current applicability of our approach, they do not affect the validity of our results. As far as the choice of faults in the first case study is concerned, we selected them form the TUDU bug tracking system, based on our fault models which we believe are representative of the types of faults that occur during AJAX development. The choice is, therefore, not biased towards the tool but the fault models we have. With respect to *reliability*, our tools and the TUDU case are open source, making the case fully reproducible.

Ajax Testing Strategies. ATUSA is a first, but essential step in testing AJAX applications, offering a solution for the reach/trigger/propagate problem. Thanks to the plugin-based architecture of ATUSA, it now becomes possible to extend, refine, and evaluate existing software testing strategies (such as evolutionary, state-based, category-partition, and selective regression testing) for the domain of AJAX applications.

11 Concluding Remarks

In this paper we have proposed a method for testing AJAX applications automatically. Our starting point for supporting AJAX-testing is CRAWLJAX, a crawler for AJAX applications that we proposed in our earlier work [14], which can dynamically make a full pass over an AJAX application. Our current work resolves the subsequent problems of extending the crawler with data entry point handling to *reach* faulty AJAX states, *triggering* faults in those states, and *propagating* them so that failure can be determined. To that end, this paper makes the following contributions:

1. A series of fault models that can be automatically checked on any user interface state, capturing different categories of errors that are likely to occur in AJAX applications (e.g., DOM violations, error message occurrences), through (DOM-based) generic and application-specific invariants which server as oracle.

[12] http://code.google.com/webtoolkit/

[13] http://www.taskfreak.com

[14] https://blueprints.dev.java.net/petstore/

2. An algorithm for deriving a test suite achieving all transitions coverage of the state-flow graph obtained during crawling. The resulting test suite can be refined manually to add test cases for specific paths or states, and can be used to conduct regression testing of AJAX applications.
3. An open source tool called ATUSA implementing the approach, offering generic invariant checking components as well as a plugin-mechanism to add application-specific state validators and test suite generation.
4. An empirical validation, by means of two case studies, of the fault revealing capabilities and the scalability of the approach, as well as the level of automation that can be achieved and manual effort required to use the approach.

Given the growing popularity of AJAX applications, we see many opportunities for using ATUSA in practice. Furthermore, the open source and plugin-based nature of ATUSA makes it a suitable vehicle for other researchers interested in experimenting with other new techniques for testing AJAX applications.

Our future work will include conducting further case studies, as well as the development of ATUSA plugins, capable of spotting security vulnerabilities in AJAX applications.

References

[1] A. Andrews, J. Offutt, and R. Alexander. Testing web applications by modeling with FSMs. *Software and Systems Modeling*, 4(3):326–345, July 2005.

[2] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *Proc. Int. Symp. on Software Testing and Analysis (ISSTA'08)*, pages 261–272. ACM, 2008.

[3] M. Benedikt, J. Freire, , and P. Godefroid. VeriWeb: Automatically testing dynamic web sites. In *Proc. 11th Int. Conf. on World Wide Web (WWW'02)*, 2002.

[4] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *ICSE Future of Software Engineering (FOSE'07)*, pages 85–103. IEEE Computer Society, 2007.

[5] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, 1999.

[6] E. Bozdag, A. Mesbah, and A. van Deursen. Performance testing of data delivery techniques for Ajax applications. *Journal of Web Engineering*, 0(0), 2009. To appear.

[7] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *Proc. 25th Int Conf. on Software Engineering (ICSE'03)*, pages 49–59. IEEE Computer Society, 2003.

[8] J. Garrett. Ajax: A new approach to web applications. Adaptive path, February 2005. `http://www.adaptivepath.com/publications/essays/archives/000385.php`.

[9] W. Halfond and A. Orso. Improving test case generation for web applications using automated interface discovery. In *Proceedings of the ESEC/FSE conference*, pages 145–154. ACM, 2007.

[10] Y.-W. Huang, C.-H. Tsai, T.-P. Lin, S.-K. Huang, D. T. Lee, and S.-Y. Kuo. A testing framework for web application security assessment. *Journal of Computer Networks*, 48(5):739–761, 2005.

[11] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. Secubat: a web vulnerability scanner. In *Proc. 15th int. conf. on World Wide Web (WWW'06)*, pages 247–256. ACM, 2006.

[12] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of Ajax web applications. In *Proc. 1st IEEE Int. Conference on Sw. Testing Verification and Validation (ICST'08)*, pages 121–130. IEEE Computer Society, 2008.

[13] A. Memon. An event-flow model of GUI-based applications for testing: Research articles. *Softw. Test. Verif. Reliab.*, 17(3):137–157, 2007.

[14] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling Ajax by inferring user interface state changes. In *Proc. 8th Int. Conference on Web Engineering (ICWE'08)*, pages 122–134. IEEE Computer Society, 2008.

[15] A. Mesbah and A. van Deursen. Migrating multi-page web applications to single-page Ajax interfaces. In *Proc. 11th Eur. Conf. on Sw. Maintenance and Reengineering (CSMR'07)*, pages 181–190. IEEE Computer Society, 2007.

[16] A. Mesbah and A. van Deursen. A component- and push-based architectural style for Ajax applications. *Journal of Systems and Software*, 81(12):2194–2209, 2008.

[17] B. Meyer. Seven principles of software testing. *IEEE Computer*, 41(8):99–101, August 2008.

[18] L. Morell. Theoretical insights into fault-based testing. In *Proc. 2nd Workshop on Software Testing, Verification, and Analysis*, pages 45–62, 1988.

[19] F. Ricca and P. Tonella. Analysis and testing of web applications. In *ICSE'01: 23rd Int. Conf. on Sw. Eng.*, pages 25–34. IEEE Computer Society, 2001.

[20] D. Richardson and M. Thompson. The RELAY model of error detection and its application. In *Proc. 2nd Workshop on Software Testing, Verification, and Analysis*, pages 223–230, 1988.

[21] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *ASE'05: Proc. 20th IEEE/ACM Int. Conf. on Automated Sw. Eng.*, pages 253–262. ACM, 2005.

[22] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott. Automated oracle comparators for testing web applications. In *Proc. 18th IEEE Int. Symp. on Sw. Reliability (ISSRE'07)*, pages 117–126. IEEE Computer Society, 2007.

[23] B. Stepien, L. Peyton, and P. Xiong. Framework testing of web applications using TTCN-3. *Int. Journal on Software Tools for Technology Transfer*, 10(4):371–381, 2008.

[24] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[25] J. Y. Yen. Finding the k shortest loopless paths in a network. *Manag. Sci.*, 17(11):712–716, 1971.

[26] R. K. Yin. *Case Study Research: Design and Methods*. SAGE Publications Inc, 3d edition, 2003.

TUD-SERG-2009-005
ISSN 1872-5392