Delft University of Technology
Software Engineering Research Group
Technical Report Series

Invariant-Based Automatic Testing of Modern Web Applications

Ali Mesbah, Arie van Deursen and Danny Roest

Report TUD-SERG-2011-003

TUD-SERG-2011-003

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, Arie van Deursen and Danny Roest. Invariant-Based Automatic Testing of Modern Web Applictions. *IEEE Transactions on Software Engineering*, 2011.

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. X, NO. Y, 2011 1

Invariant-Based Automatic Testing of Modern Web Applications

Ali Mesbah, *Member, IEEE Computer Society*
Arie van Deursen, *Member, IEEE Computer Society*
Danny Roest

Abstract—AJAX-based *Web 2.0* applications rely on stateful asynchronous client/server communication, and client-side run-time 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 state-flow graph for all (client-side) user interface states. We identify AJAX-specific faults that can occur in such states (related to e.g., DOM validity, error messages, discoverability, back-button compatibility) as well as DOM-tree invariants that can serve as oracles to detect such faults. Our approach, called ATUSA, is implemented in 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 three case studies, consisting of six subjects, evaluating the type of invariants that can be obtained for AJAX applications as well as the fault revealing capabilities, scalability, required manual effort, and level of automation of our testing approach.

Index Terms—Automated testing, web applications, Ajax.

1 INTRODUCTION

There is a growing trend to move applications towards the Web. Well-known examples include Google's mail and office software comprising spreadsheet, word processing, and calendar applications. The reasons for this move to the web are manifold:

- No installation effort for end-users;
- Automatic use of the most recent software version by all users, thus reducing maintenance and support costs;
- Universal access from any browser on any machine with Internet access, not only to the application but also to user data;
- New collaboration and community building opportunities as supported by *Web 2.0* applications.

For today's web applications, one of the key technologies facilitating this move is AJAX, an acronym for "Asynchronous JAVASCRIPT and XML" [13]. With AJAX, web-browsers not only offer the user navigation through a sequence of HTML pages, but also dynamic rich interaction via graphical user interface components.

While the use of AJAX technology positively affects user-friendliness and interactiveness of web applications [27], it comes at a price: AJAX applications are notoriously error-prone due to, e.g., their stateful, asynchronous, and event-based nature, the use of (loosely typed) JAVASCRIPT, the client-side manipulation of the browser's Document-Object Model (DOM), and the use of delta-communication between client and web server [27].

In order to improve the dependability of AJAX applications, static analysis or testing techniques could be deployed. Unfortunately, static analysis techniques are not able to reveal many of the dynamic dependencies present in today's web applications. Furthermore, traditional web testing techniques are based on the classical page request/response model, not taking into account client side functionality. Recent tools such as Selenium,[1] offer a capture-and-replay style of testing for modern web applications. While such tools are capable of executing AJAX test cases, they still demand a substantial amount of manual effort from the tester.

The goal of this paper is to support *automated testing of* AJAX *applications*. To that end, we propose an approach in which we automatically derive a model of the user interface (UI) states of an AJAX application. We obtain this model by "crawling" an AJAX application, automatically clicking buttons and other UI-elements, thus exercising the client-side UI functionality. In order to recognize failures in these executions, we propose the use of *invariants*: properties of either the client-side DOM-tree or the derived state machine that should hold for any execution. These invariants can be generic (e.g., after any client-side change the DOM should remain W3C-compliant valid HTML) or application-specific (e.g., the home-button in any state should lead back to the starting state).

We offer an implementation of the proposed approach in an open source, plugin-based tool architecture. It consists of a crawling infrastructure called CRAWLJAX,[2] as well as a series of testing-specific extensions referred to as ATUSA. We have applied these tools to a series of AJAX applications. We

- *A. Mesbah is with the department of Electrical and Computer Engineering, University of British Columbia, 2332 Main Mall, V6T1Z4 Vancouver, BC, Canada. E-mail: amesbah@ece.ubc.ca*
- *A. van Deursen, and D. Roest are with the Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Mekelweg 4, 2628CD Delft, The Netherlands. E-mail: arie.vandeursen@tudelft.nl d.roest@student.tudelft.nl.*

1. http://selenium.openqa.org
2. http://crawljax.com

report on our experiences in this paper, evaluating the proposed approach in terms of fault-finding capabilities, scalability, automation level, and the usefulness of invariants.

This paper is a substantially expanded and revised version of our paper from early 2009 [28]. Since the first publication on CRAWLJAX [24], a range of improvements to the tool and the underlying algorithms have been realized. Furthermore, the tool and testing approach have been applied to several AJAX applications (see, e.g., [33], [6]). In this paper, we provide an integrated presentation of the full approach, incorporating the most recent developments concerning the crawling algorithm, the testing approach, the available plugins, and the application of the approach to a range of different AJAX applications, of which six are covered in substantial detail.

The paper starts with a survey of related work (Section 2), followed by an analysis of AJAX testing challenges (Section 3). We then explain the crawling algorithms (Section 4) as well as the invariant-based testing approach built on top of it (Sections 5 and 6). After covering the architecture of our plugin-based tool set (Section 7), we describe three case studies, totalling six different AJAX applications (Section 8). We conclude with a discussion of our findings, a summary of our contributions, and an outlook towards future work.

2 RELATED WORK

Modern web interfaces incorporate client-side scripting and user interface manipulation which is increasingly separated from server-side application logic [36]. 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. We survey these in Sections 2.1 and 2.2. We describe current AJAX testing approaches in Section 2.3, after which we provide a short overview of the use of invariants for web testing in Section 2.4.

2.1 Traditional Web Testing

Benedikt *et al.* [4] 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 6.

Tools such as WAVES [18] and SecuBat [19] 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.

Alfaro *et al.* apply model-checking [9] to web applications using their tool called MCWEB [10]. Their work, however, was targeted towards web 1.0 applications.

A model-based testing approach for web applications was proposed by Ricca and Tonella [31]. 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 [24].

Logging user session data on the server is also used for the purpose of automatic test generation [11], [34]. 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 [27] 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 [16], [17] present their static analysis of server-side Java code to extract web application request parameters and their potential values. They use [15] symbolic execution of server-side code to identify possible interfaces of web applications. Such techniques have limitations in revealing faults that are due to the complex (client-side) runtime behavior of modern rich web applications.

2.2 GUI Application Testing

Reverse engineering a model of the desktop (GUI) in order to generate test cases has been proposed by Memon *et al.* [23]. AJAX applications can be seen as a hybrid of desktop and web applications, since the user interface is composed of components and the interaction is event-based [27]. 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 [22], and therefore require other testing tools and techniques.

2.3 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[3] can be used to test JAVASCRIPT on a functional level.

The most commonly-used AJAX testing tools are currently *capture/replay* tools such as Selenium IDE,[4], WebKing[5], and

3. http://jsunit.net
4. http://selenium.openqa.org
5. http://www.parasoft.com/jsp/products/home.jsp?product=WebKing

Sahi[6], which allow DOM-based testing by capturing events fired by user interaction. Other web application testing tools such as WebDriver[7] or Watij[8] take a different approach: rather than being a JAVASCRIPT application running within the browser, they use a wrapping mechanism and provide API's to control the browser. Such tools demand, however, a substantial amount of manual effort on the part of the tester to create and maintain a test suite, since every event-trail and the corresponding DOM assertions have to be written by the tester.

Marchetto *et al.* [21] discuss a case study in which they demonstrate that traditional web testing techniques (e.g., code coverage testing [31], model-based testing [1], session based testing [11], [34]) have serious limitations when applied to modern *Web 2.0* applications. They propose 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.

Our approach is the first to exercise automated testing of *Web 2.0* applications by simulating real user events on the web user interface and inferring an abstract model automatically.

2.4 Invariants

The concept of using invariants to assert program behavior at run-time is as old as programming itself [8]. For the domain of web applications, any approach that performs validation of the HTML output (e.g., [4], [2]) could be considered to be using invariants on the DOM. This paper makes the use of invariants for testing web applications explicit by defining different types of (client-side) invariants, providing a mechanism for expressing those invariants and automatically checking them through dynamic analysis.

Automatic detection of invariants is another direction that has gained momentum. The best-known work is that of Ernst *et al.* on Daikon [12], a tool capable of inferring *likely invariants* from program execution traces. A more recent tool is DoDom [30] capable of inferring DOM invariants. We have also started exploring ways of automatically detecting DOM and JAVASCRIPT invariants in web applications [14].

3 AJAX TESTING CHALLENGES

In order to test AJAX applications automatically, we need to face the following challenges:

- Find a method to simulate a user's interaction with the web application;
- Gain access to various dynamic DOM states;
- Develop a method to assess the correctness of the obtained states.

In traditional web applications, states are explicit, and correspond to pages having a unique URL. In AJAX applications, the state of the user interface is determined dynamically, through event-driven changes in the browser's DOM tree that are only visible after executing the corresponding JAVASCRIPT code. Ultimately, an AJAX application could consist of a single-page [25] with a single URL.

The event-driven nature of AJAX presents the first serious challenge for automation, 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.

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

One way to simulate a web user and gain access to dynamic states of AJAX applications automatically 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 crawl through different UI states and infer a model of the navigational paths and states. 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 as well as different (random or user-specified) input data. We have proposed such a crawler for AJAX, called CRAWLJAX [24], which is substantially extended for this work and explained in Section 4.

Automating the process of assessing the correctness of test case output is a challenging task, known as the oracle problem [38]. The problem is even more demanding when we consider AJAX in which all the state changes are manifested through modifications on the DOM tree. 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, this manual approach becomes practically impossible. An approach taken in practice is to use a version of the application to obtain a baseline, also known as the Gold Standard [7]. The shortcoming of this approach is that it presumes that the baseline represents a correct version of the system, from which initial states can be collected and reused as oracles in subsequent test executions. The web testing literature has mainly used HTML comparators [35] and validators [2] for testing web applications. Such validators are, however, not capable of capturing complex faulty DOM states that are present in modern web applications. To automate the test oracles, we propose to use generic and application-specific invariants on the DOM-tree.

The details of our solutions for the challenges mentioned in this section are presented in the following sections.

4 DERIVING AJAX STATES

The testing method we propose is based on CRAWLJAX[9] a crawler capable of automatically deriving a state machine from an AJAX web application, which we originally proposed in early 2008 [24]. One year later (early 2009), we described how

6. http://sahi.co.in/w/
7. http://seleniumhq.org/docs/09_webdriver.html
8. http://watij.com/
9. http://crawljax.com

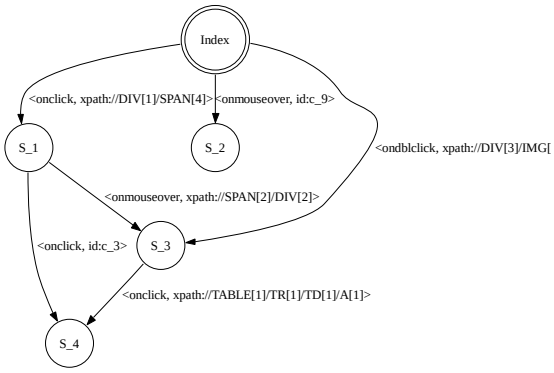

Fig. 1. An inferred state-flow graph.

this crawler can be applied for testing purposes. Since then, we have used CRAWLJAX in a range of projects (see, e.g., [33], [6]), resulting in numerous improvements to the crawler and the testing approach. In the subsequent sections we provide an integrated presentation of our most recent developments concerning the crawling algorithm.

Algorithms 1 and 2 show the overall crawling process. Central to our approach is the automatic inference of a *state-flow graph* from the web application. To infer such a graph automatically, we open the web application in a web browser, we examine the DOM-tree looking for *candidate elements* to fire events on, and detect user interface state changes. We conduct the analysis and *navigation* part recursively for all possible states. For *input fields*, we provide random data if no custom data is available. And finally, we provide various options for *controlling the crawling phase*. The following subsections discuss these steps in more detail.

Algorithm 1 Crawling process with 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: f ← analyseForms(Δupdate)
13: Set C ← getCandidateClickables(Δupdate, tags, f)
14: for c ∈ C do
15:     generateEvent(cs, c)
16: end for
17: end procedure
```

Algorithm 2 Firing events and analyzing AJAX states

```
1:  procedure GENERATEEVENT (State cs, Clickable c)
2:  robot.enterFormValues(c)
3:  robot.fireEvent(c)
4:  dom ← browser.getDom()
5:  if stateChanged(cs.getDom(), dom) then
6:      xe ← getXpathExpr(c)
7:      ns ← sm.addState(dom)
8:      sm.addEdge(cs, ns, Event(c, xe))
9:      sm.changeToState(ns)
10:     runOnNewStatePlugins(ns)
11:     testInvariants(ns)
12:     if stateAllowedToBeCrawled(ns) then
13:         crawl(cs)
14:     end if
15:     sm.changeToState(cs)
16:     if browser.history.canGoBack then
17:         browser.history.goBack()
18:     else
19:         {We have to back-track by going to the initial state.}.
20:         browser.reload()
21:         List E ← sm.getPathTo(cs)
22:         for e ∈ E do
23:             re ← resolveElement(e)
24:             robot.enterFormValues(re)
25:             robot.fireEvent(re)
26:         end for
27:     end if
28: end if
29: end procedure
```

4.1 The State-Flow Graph

The crawler we propose is a tool that can exercise client side code, and identify elements[10] 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.

Definition 1: 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 UI changes in a directed graph, by recording the paths (events) to the DOM changes, to be able to navigate between the different states. For that purpose we define a *state-flow graph* as follows:

Definition 2: A **state-flow graph** for an AJAX site $\mathbf{A}$ is a 3-tuple $\langle \boldsymbol{r}, \boldsymbol{V}, \boldsymbol{E} \rangle$ where:

1) $\boldsymbol{r}$ is the root node (called Index) representing the initial state after $\mathbf{A}$ has been fully loaded into the browser.
2) $\boldsymbol{V}$ is a set of vertices representing the UI states. Each $\boldsymbol{v} \in \boldsymbol{V}$ represents a unique run-time DOM state in $\mathbf{A}$.
3) $\boldsymbol{E}$ is a set of edges between vertices. Each $(\boldsymbol{v}_1, \boldsymbol{v}_2) \in \boldsymbol{E}$ represents a clickable $\boldsymbol{c}$ connecting two states if and only if state $\boldsymbol{v}_2$ is reached by executing $\boldsymbol{c}$ in state $\boldsymbol{v}_1$.

As an example, Figure 1 displays the state-flow graph of a simple AJAX site. From the index page three different states can be reached directly. The edges between states are labeled with an identification (e.g., XPath expression) of the element and the event type to reach the next state.

4.2 Inferring the State Machine

The state machine (line 4 Algorithm 1) 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 (lines 7-8 Algorithm 2). The following components participate in the construction of the graph:

- CRAWLJAX uses an *embedded browser* interface (with different implementations: IE, Firefox, and Chrome) supporting all technologies required by modern dynamic web applications;

10. For the sake of simplicity, we call such elements 'clickables' in this paper, however, they could have any type of DOM event-listener such as ondblclick or onmouseover.

```
1 <script>
2   $(".news").click(function() {
3     $("#content").load("news.html");
4 });
5 </script>
6 <div class="news">...</div>
```

Fig. 2. Attaching an onClick event listener to a DIV element with attribute class="news".

- A *robot* is used to simulate user input (e.g., `click`, `hover`, 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 tree.

As it can be seen in Algorithm 1, the browser, robot and state machine are initialized first (lines 2-4) and from there the `crawl` procedure is called. The steps taken by the crawl procedure to infer the state machine are discussed below.

4.3 Detecting Clickables

To illustrate the difficulties involved in crawling AJAX, consider Figure 2. This is a highly simplified example, showing how an `onclick` event listener can be attached to a `DIV` element at runtime through JAVASCRIPT. Traditional crawlers as used by search engines simply ignore all such clickables. Finding these clickables at runtime is a non-trivial task for any modern crawler.

To tackle this challenge, CRAWLJAX implements an algorithm in which a set of *candidate elements* (line 13 Algorithm 1) are exposed to an event type (e.g., click, mouseover) (line 3 Algorithm 2).

In automatic mode, the crawler examines all elements of the type `A`, `DIV`, `INPUT`, and `IMG`, since these elements are often used to attach event listeners to. If the user wishes to define their own criteria for selection, this list can be extended or adapted. The candidate clickables can be labeled as such based on their HTML tag element name and attribute constraints. For instance, all elements with a tag `SPAN` having an attribute `class="menuitem"` can be set to be considered as candidate clickable. For each detected candidate element on the DOM tree, the crawler fires an event on the element in the browser to analyze the effect. A candidate clickable becomes an actual clickable if the event fired on the element causes a DOM change in the browser. In that case, using the clickable element an edge is created and added to the state machine, which connects the previous state to the current state (lines 5-9 Algorithm 2).

The general approach to use CRAWLJAX is to select a large set of elements to examine (for good coverage) and to exclude elements that are not of importance or can cause problems (delete items or log the user out).

4.4 Creating and Comparing States

After firing an event on a candidate clickable, the algorithm inspects the resulting DOM tree to see if the event results in a modified state (line 5 Algorithm 2). If a similar state is part of the state flow graph already, merely an edge is created, identifying the type of click and the location clicked. If the next state is not part of the graph already, a new state is created and added first (line 7 Algorithm 2).

The level of abstraction achieved in the resulting state-flow graph is largely determined by the algorithm used to compare DOM trees (which reflect the states in the flow graph). A generic and effective way is to use a simple string edit distance algorithm such as Levenshtein [20]. This has the advantage that it does not require application-specific knowledge, and that the algorithm can be fine-tuned by means of a similarity threshold (between 0 and 1).

Alternatively, we propose the use of a series of "comparators" that each can compare specific aspects of two DOM trees. Each comparator can eliminate specific parts of the DOM tree, such as (irrelevant) attributes, time stamps, or styling issues. The resulting simplified DOM tree is subsequently pipelined to the next comparator. Figure 3 shows an example of how the pipelining works by stripping the differences and passing the result forward. At the end, after all the desired differences are removed, a simple string comparison determines the equality of the two DOM strings.

Web applications often contain structures with repeating patterns, such as tables and lists. Since the actual elements are not always relevant, we propose comparators that can abstract from the specific elements. In particular, our technique scans the DOM tree for elements that recur within a structure, and automatically generates a template capturing the pattern. The comparators can use these templates to ignore the given repeating patterns.

While state comparators can be a powerful means to ignore non-deterministic differences, they can also be too permissive grouping DOM trees that should be considered different. To address this problem, state comparators are equipped with preconditions: Boolean predicates that the test engineer can add to ensure that a given comparator is only used on states meeting certain constraints.

Finally, to enable fast comparison with existing states, the resulting stripped DOM tree is used to calculate a hashcode, which is used in all subsequent comparisons in the state machine.

4.5 Processing Document Tree Deltas

After a new state has been detected, the crawling procedure is recursively called (line 13 Algorithm 2) to find new possible candidate elements in the partial changes made to the DOM tree. CRAWLJAX computes the differences between the previous document tree and the current one (line 11 Algorithm 1), by means of an enhanced *Diff* algorithm to detect AJAX partial updates, which may be due to a server request call that injects new elements into the DOM. This differencing method is an optimization step, needed to scan and search for candidate clickables merely in the changed parts of the DOM-tree, as

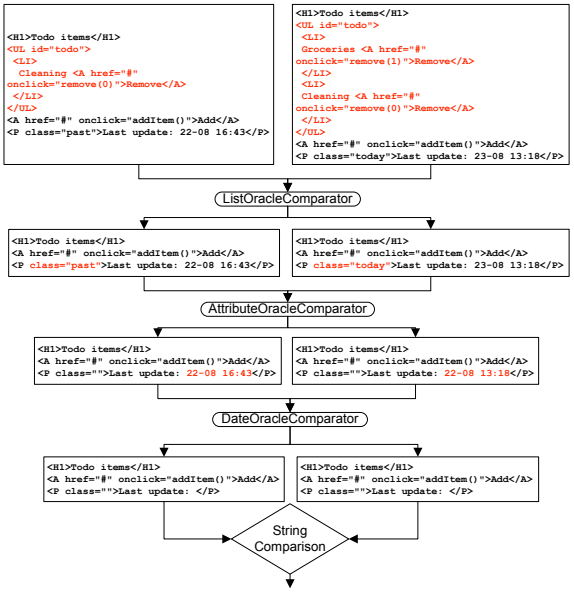

Fig. 3. Pipelining state comparators.

apposed to examining all of the elements of the new DOM-tree.

4.6 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 (line 6-7 Algorithm 2) 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 (lines 16-27 Algorithm 2).

CRAWLJAX adopts XPath to identify the clickable elements. After a reload or state change, DOM elements, can easily be deleted, changed, or replaced. As a consequence the XPath expression used for navigation can become invalid. To tackle this problem, our approach uses a mechanism called Element Resolver (line 23 Algorithm 2), which examines the clickable elements before they are used to make state transitions. This examination is needed to make sure we have access to the correct element. To detect the intended element persistently, we use various (saved) properties of the element such as their attributes and text value. Using a combination of these properties, our element resolver searches the DOM for a match, which gives us some degree of reliability in case clickables are removed or changed. Note that despite our element resolving mechanism, because of side effects of server-side state, there is no guarantee that we find the same element on the DOM-tree and can reach the exact same state.

TABLE 1
Random values for form input elements.

Type	Value
Text fields	Random string of 8 alpha characters.
Checkboxes	Checked with $p < 0.5$
Radio buttons	Checked with $p < 0.5$
Lists	Random item from the list

4.7 Data Entry Points

Besides clicks to proceed along links, buttons, etc., certain user interface states will require data entered by the user. In order to provide input values in AJAX web applications, we have adopted a reverse engineering process to extract all exposed data entry points. To this end, we have extended our crawler with the capability of detecting *input elements* on each newly detected state (line 12 Algorithm 1).

While crawling, before the robot clicks on an element, it checks the DOM for input elements and enters the corresponding values (line 2 Algorithm 2). The related input fields are saved with the clickable element causing the state transition, so that the crawler knows which values to enter in which fields the next time it clicks on the element (while backtracking). For supplying values in the input fields our approach considers three categories:

Random Input Values. Automation is an important aspect of our approach. Therefore our tool enters random values in the form elements by default. With this approach, many states that need input values can be reached without any human effort. Table 1 shows the random values used while crawling if no custom values are provided.

If there is already some value in an input field, that value is retrieved and used instead of a random value.

Custom Input Values. Specific input values are often needed for testing or to reach certain states. For example a valid e-mail address as input is needed to add a contact. With our approach it is possible to provide custom input values by specifying them for the crawler.

Multiple Custom Input Values. Entering multiple values for input fields can be useful for testing or to reach more states. For example, entering a normal string, an empty string, and a string with non alpha-numeric characters in a field that requires a text value could be required for testing.

The challenge here is to know when to enter which value. Our current approach is based on grouping the input elements on each state, by specifying the related clickable element (e.g., submit button). For each input value, n number of values are provided. These values are inserted into the input fields and the associated clickable is clicked, n times where n is the position of the provided input values. A more complete approach would be to try every combination of the input values, at the cost of increasing the running/testing time.

4.8 Controlling the Crawling Phase

In order to have more control on the crawling paths, we use *crawl conditions*, conditions that check whether a state should be visited. A state is crawled only if the crawl conditions are

```
UrlCondition condition = new UrlCondition("crawljax.com");
crawler.addCrawlCondition("Only crawl the Crawljax web site
    ", condition));
```

Fig. 4. An URL crawl condition for only crawling the CRAWLJAX web page.

satisfied (line 12 in Algorithm 2). Figure 4 shows an example of a crawl condition that enforces that only states within the `crawljax.com` domain are crawled.

In addition, to give more controllability CRAWLJAX has a set of options, such as the maximum number of states, maximum crawling time, waiting time after each reload (for the page to load fully), waiting time after an event is fired (for the DOM-tree to get updated) and the crawl depth.

5 TESTING AJAX STATES THROUGH INVARIANTS

With access to different dynamic web states we can check the user interface against different constraints. We propose to express those as *invariants*, which we can use as an oracle to automatically conduct sanity checks in any state. Although the notion of invariants has predominantly been applied to programming languages for software evolution [12] and verification [3], we believe that invariants can also be adopted for testing modern web applications to specify and constrain DOM elements' properties, their relations and occurrences.

In this work, we distinguish between *generic* and *application-specific* invariants on the DOM-tree, between DOM-tree states, and on the run-time JAVASCRIPT variables. Each invariant is based on a fault model [7], representing AJAX-specific faults that are likely to occur and which can be captured through the given invariant.

5.1 Generic DOM Invariants

5.1.1 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 to 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 an oracle to determine whether errors or warnings occur.

5.1.2 No Error Messages in DOM

A client-site web state should never contain a string pattern that suggests an error message [4] in the DOM tree. 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.

TABLE 2
Expressing state invariants.

	Satisfied if and only if
XPath expression	the XPath expression returns at least one DOM element
Regular expression	the regular expression is found in the DOM string
JAVASCRIPT expression	the JAVASCRIPT expression evaluates to true
URL condition	the current browser's URL contains the specified string
Visible condition	the specified DOM element is visible

5.1.3 Accessibility and i18n Compliant DOM

Many modern AJAX web applications pose accessibility challenges to people with disabilities due to their dynamic content and advanced user interface components. Evaluating the dynamic states against W3C standards such as the Web Content Accessibility Guidelines (WCAG 1.0)[11] or the recent Accessible Rich Internet Applications suite (ARIA)[12] can help find accessibility faults automatically.

The same is true for checking each DOM state against W3C internationalization and localization (i18n)[13] guidelines.

5.1.4 Secure States

Testing modern web applications for security vulnerabilities is far from trivial. Capturing web security requirements in terms of generic invariants that can be checked automatically is very promising. Recently, we applied this technique [6] for automatically detecting security vulnerabilities in client-side self-contained web widgets, that can co-exist independently on a single web page. We focused on two security invariants, namely (1) no widget is able to change the content (DOM) of another widget, and (2) no widget can steal data from another widget and send it to the server via an HTTP request, with promising detection results.

In addition, security vulnerabilities such as Cross-Site Scripting (XSS) in AJAX applications can be captured in the same manner. It is worth mentioning that detecting DOM-based XSS requires an analysis of the run-time generated DOM (which we have access to) and not just the pages' syntax [37].

5.2 Application-specific State Invariants

We can define invariants that should always hold and could be checked on the dynamic states, specific to our AJAX application in development. In our case studies, Section 8, we describe a number of application-specific invariants.

Constraints over the DOM-tree can be easily expressed as invariants. Typically, this can be coded into one or two

11. http://www.w3.org/TR/WAI-WEBCONTENT/
12. http://www.w3.org/WAI/intro/aria
13. http://www.w3.org/International/publications

```
//the menu item on the home page should always have the class attribute 'menuElement'
Condition correctMenuItem = new XPathCondition("//DIV[@id='menu']/UL/LI[contains(@class, 'menuElement')]");
Condition whenAtHomePage = new JavaScriptCondition("document.title=='Home'");
crawler.addInvariant("Home page menu items", correctMenuItem, whenAtHomePage);
```

Fig. 5. Example of an XPATH invariant with a JAVASCRIPT precondition.

simple Java methods. The resulting invariants can be used to dynamically search for invariant violations.

Table 2 shows the different generic ways the invariants can be expressed. We currently have support for expressing invariants in XPath, regular, or JAVASCRIPT expressions. In addition, we support conditions such as the URL or visibility of DOM elements, which can be used to express invariants. The logical operators NOT, OR, AND, and NAND can also be applied, on or between the invariants, for more flexibility. In addition, each invariant type can be constrained to a specific set of states using preconditions.

While crawling through the different states of the web application, since we have access to the run-time JAVASCRIPT, we can also specify invariants on the values of any JAVASCRIPT variable.

Figure 5 shows an example of expressing an XPATH invariant with a JAVASCRIPT precondition for checking whether the menu item on the home page contains the class attribute 'menuElement'.

The generated templates capturing DOM patterns (discussed in Section 4.4) can also be augmented and used as invariants on the DOM tree. Figure 6 shows a DOM invariant template that checks the structure of the list, whether the item is between 2 and 50 alpha numeric characters, and whether an item's identifier is always an integer value.

5.3 Generic State Machine Invariants

Besides constraints on the DOM-tree in individual states, we can identify requirements on the state machine and its transitions. Some of the generic invariants that can be defined on any state machine inferred by our tool consist of the following:

5.3.1 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.

5.3.2 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

```
<UL id="todo">
  ([\s]*<LI>[\s]+
    [a-zA-Z0-9 ]{2,50}<A href="#" onclick="remove([0-9]+)">
        Remove</A>[\s]*
  </LI>[\s]*)*
</UL>
```

Fig. 6. An augmented template that can be used as an invariant.

'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.

5.4 Application-specific State Machine Invariants

Besides generic invariants on the state machine, we can also define constraints on the temporal properties of the web application using application-specific invariants. These temporal properties are usually in a `Source - Action -> Target` format, and can be checked as invariants after the state machine is fully inferred, using for instance temporal logic model checking. Examples include:

- From any state, clicking on the `logoff` button should bring us to the logged off state (or the login page);
- From state product list, clicking on the `overview` link should take us to the overview state.

6 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.

6.1 Test Suite Generation

To generate a test suite, we use the *K shortest paths* [39] 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.

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

/*Element-info: SPAN class=expandable-hitarea */
Clickable c1 = new Eventable(new Identification(
  "xpath", "//DIV[1]/SPAN[4]"), "onclick");

assertPresent(c1);
browser.enterRelatedInputValues(c1);
assertTrue(browser.fireEvent(c1));
assertEquals(oracle.getState("S_1").getDom(),
  browser.getDom());

/*Element-info: DIV class=hitarea id=menuitem2 */
Clickable c2 = new Eventable(new Identification(
  "xpath", "//SPAN[2]/DIV[2]"), "onmouseover");

assertPresent(c2);
assertTrue(browser.fireEvent(c2));
assertEquals(oracle.getState("S_3").getDom(),
              browser.getDom());
...
}
```

Fig. 7. A generated JUnit test case.

Given a rooted directed graph G with non-negative edge weights, a positive integer K, and two vertices v_1 and v_2, the problem asks for the K shortest paths from v_1 to v_2, in non-decreasing order of length. In our algorithm, first the set of *sink* vertices (with no outgoing edges), in G is calculated. Then we use each sink in $\{s_1, s_2, ..., s_n\}$ to find the K shortest paths from the root (`index`) state to s_i. Loops are included once.

Next, we transform each path found into a JUnit test case, as shown in Figure 7. 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 a clickable element is associated with input fields, input values are first inserted in the browser's DOM tree before triggering the event.

After each event invocation the resulting state in the browser is compared with the expected state. The comparison can take place at different levels of abstraction ranging from textual [35] to schema-based similarity [25]. The states are currently compared with our oracle comparator pipelining mechanism as discussed in Section 4.4.

6.2 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 [5]. 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.

In short, a test case succeeds if:

1) every transition (edge) element can be successfully found in the state;
2) the corresponding event can be fired on the transition element;
3) there are no timeouts when loading each state;
4) the invariants are satisfied;
5) every visited state in the browser is equivalent with the expected state in the state machine.

6.3 Applications

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). For further details on how this technique is used for regression testing AJAX applications, we refer to our recent paper [33].

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.

7 TOOL IMPLEMENTATION

7.1 The Testing Framework

Our approach, called ATUSA (Automatically Testing UI States of AJAX), is implemented in Java. It is based on the crawling capabilities of our open-source crawler CRAWLJAX and provides plugin *hooks* for testing AJAX applications at different levels. More implementation details of the crawler can be found on the CRAWLJAX website.[14] The *state-flow graph* is based on the JGrapht[15] library. Apache Velocity templates assist us in the code generation process of JUnit test cases.

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.

Furthermore, ATUSA provides a number of generic comparators (see 4.4), each of which is responsible for ignoring merely one type of difference. The list of comparators currently available includes Whitespace, Attributes, Style, Date-time, Structure, List, Table, Regex, and XPathExpression, each addressing a particular way of eliminating tree differences.

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., constraints on

14. http://crawljax.com
15. http://jgrapht.sourceforge.net

```
CrawlSpecification crawler = new CrawlSpecification(URL);
crawler.click("a");
crawler.when(aCondition).click("div").withText("close");
crawler.dontClick("a").withAttribute("class", "info").
    withText("delete");
crawler.dontClick("a").underXPath("//DIV[@id='header']");
crawler.addInvariant(new XPathCondition(xpath, preCondition
    );
crawler.addPlugin(new TestSuiteGenerator());
...
```

Fig. 8. Tool configuration example.

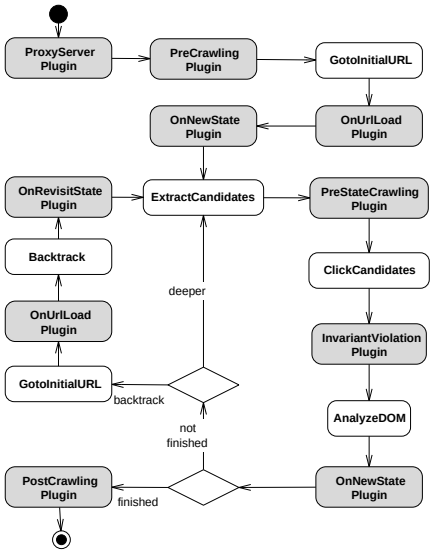

Fig. 9. Plugins Invocation Flow.

the length of the deepest paths [4], or number of clicks to a certain state. Whenever a fault is detected, the error report along the causing execution path is saved so that it can be reproduced later easily.

ATUSA explores a large number of execution paths that may result from unpredictable user behavior. This is thus complementary to that of capture/replay testing tools, which are useful for testing the correctness of a few specific paths in the web application.

ATUSA offers implemented a Java based *API* for configuring the tool with merely a few lines as depicted in Figure 8. This figure shows how (1) the user can include (click) and exclude (dontClick) certain element types from the crawling process, (2) invariants can be added for testing (3) plugins can be added for analysis and test suite generation.

7.2 Plugins

ATUSA provides the tester with APIs to implement plugins for validation and fault detection. The main interface for extending the framework is *Plugin*, which is extended by the different *types* of plugins. Each plugin type serves as an extension point that is called in a different phase of the crawling execution. Table 3 summarizes the main plugin types and their invocation

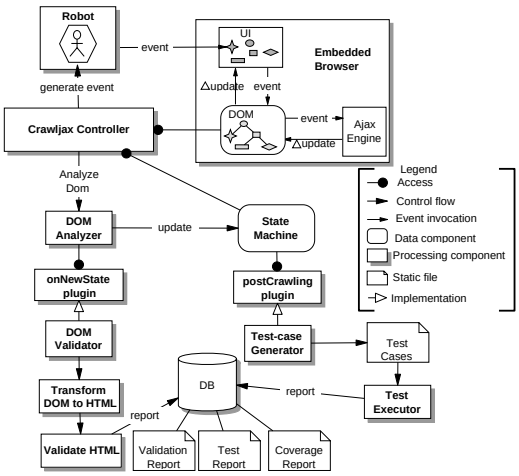

Fig. 10. Processing view of ATUSA, showing only the DOM Validator and TestSuite Generator as examples of possible plugin implementations.

phases. Figure 9 depicts the execution flow of each type of plugin and Figure 10 depicts the processing view of ATUSA, showing only the DOM Validator and TestSuite Generator as examples of possible plugin implementations.

The list of currently available plugins is shown in Table 4. Most of these plugins are open source.[16]

Understanding why a test case fails is very important to determine whether a reported failure is caused by a real fault or a legal change. To that end, our toolset generates a detailed web report that visualizes the failures. We format and pretty-print the DOM trees without changing their structure and use XMLUnit[17] to determine the DOM differences. The elements related to the differences are highlighted with different colors in the DOM trees. We also capture a snapshot of the browser at the moment the test failure occurs and include that in the report. Other important data such as the sequence of fired events, JAVASCRIPT debug variables, and the list of applied state comparators are also displayed.

8 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 [40].

8.1 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 kind of meaningful invariants can be obtained for AJAX applications and how can they be expressed in ATUSA?

16. http://crawljax.com/plugins/
17. http://xmlunit.sourceforge.net

TABLE 3
Plugin Types.

Plugin Type	Execution Phase	Examples
ProxyServer	At the initialization phase	Loading a custom proxy configuration in the embedded browser
PreCrawling	Before the crawling	Authentication plugins to log onto the system
OnNewState	When a new state is found	Validate DOM, Create Screenshots
OnRevisitState	When a state is revisited	Benchmarking
OnUrlLoadPlugin	After the initial URL is (re)loaded	Reset back-end state
OnInvariantViolation	When an invariant assertion fails	Report builder, test generation
PreStateCrawling	Before a new state is crawled	Logging candidate elements
PostCrawling	After the crawling process, the state-flow graph is fully inferred	Generating test cases from the state machine

TABLE 4
Available plugins.

Plugin Name	Functionality	Type
TestSuite Generator	generated, in JUnit format, a test suite from the inferred state machine	PostCrawling
Crawl Overview	generates, in HTML format, an overview of the dynamic states visited	OnNewState, PostCrawling
WebScarab Wrapper [6]	wraps around the WebScarab proxy. Used for modifying passing HTTP request/response	ProxyServer
SFG Exporter	exports the state-flow graph to different formats (e.g., dot, GraphML, GML, Visio)	PostCrawling
Benchmark	creates performance measurement graphs of crawl sessions	PostCrawling
Error Reporter	makes a report, in HTML format, of the detected violations	OnInvariantViolation
LogIn	utility plugin to help with logging in	PreCrawling
Mirror Generator [24]	generates a static linked HTML version of the dynamic states	PostCrawling
DOM Validator	Validates each dynamic DOM state against the W3C standard	OnNewState
InvarScope [14]	detects JavaScript and DOM invariants dynamically	ProxyServer, OnNewState, PostCrawling
CrossBrowser Tester [26]	checks each state in three different browsers looking for cross-browser incompatibilities	PostCrawling
RegressionTester [33]	conducts regression tests	PostCrawlng

RQ2 What is the fault revealing capability and effectiveness of our testing approach?

RQ3 What is the performance of the proposed approach, and how well does it scale?

RQ4 What is the automation level and how much manual effort is involved in the testing process?

8.2 Study 1: Invariants

In our fist study, our goal is to assess to what extent meaningful invariants can be obtained for AJAX applications (RQ1). To than end, we analyze four open source AJAX applications.

8.2.1 Case Study Setup

Our assessment involves the following steps:

1) We run our tool on the subject system (using the default configurations) to obtain a state-flow graph. We visualize the graph with the *Crawl Overview* plugin.
2) We analyze the graph manually, to assess its completeness, and to see if the most important user interface states are covered. If necessary, we adjust the crawler's configuration parameters and settings to increase the coverage, for example by providing specific elements that should be clicked or input values that need to be filled in;
3) We inspect the states from the graph, analyzing the DOM (with tools such as Firebug[18]), and JAVASCRIPT code, in search of candidate invariants;
4) To identify candidate invariants, we use tools such as Firefinder[19] and our own regular expression tool to extract and evaluate XPath and regular expressions over the DOM-tree;
5) We express the selected invariants in Java using ATUSA's invariant expression mechanisms (see 5.2);
6) We run our tool to check the invariants at runtime automatically.

8.2.2 THEORGANIZER

THEORGANIZER[20] is an open source web application that can be used as a task manager and organizer. It is written as a J2EE application using WebWork, Spring JDBC, and the Prototype AJAX library.

The configuration setup for THEORGANIZER was straight forward: include all images as candidate clickables and use the random input-value generator for form inputs. THEORGANIZER requires authentication, thus we wrote a plugin to log into the web application automatically.

Figure 11 depicts some parts of the inferred graph visualized by the *CrawlOverview* plugin. This graph shows the outgoing and incoming edges from each state. In addition, we can zoom into each state by clicking on the snapshot image of the state (taken during crawling), to conduct further examination.

After manually inspecting the states, we documented 5 invariants for THEORGANIZER, listed as invariants O1–O5 in Table 5. These consisted of one generic, one XPath, and one Regular expression state invariants as well as two application-specific state machine invariants (see Section 5.4).

We detected violations through invariants O2, O4, O5, as well as generic invariant A1 which was relevant for all cases. The most interesting violation was O4, in which the expected behavior was that after clicking on the logoff button, we would

18. http://getfirebug.com/
19. https://addons.mozilla.org/en-US/firefox/addon/11905/
20. http://www.apress.com/book/downloadfile/2931

TABLE 5
Invariants for Study 1.

Inv. #	Subject System	Inv. Description	Type
O1	THEORGANIZER	The Year View state `//img[contains(@src, 'head_yearView')]` contains at least one appointment item `//input[@id='edit']`	XPath expr. state invariant
O2	THEORGANIZER	'Failed to populate the list properly!'	Generic ('Fail%') DOM invariant
O3	THEORGANIZER	Appointment item (view) structure	Reg. expr. state invariant
O4	THEORGANIZER	Clicking on the logoff button `//img[contains(@id, 'logoff')]` results in a state with `//div[contains(text(), 'You have logged out')]`	application-specific SM inv.
O5	THEORGANIZER	Clicking on `//img[@id="X"]` results in a state with `//img[contains(@src, 'head_X')]`, where X is a string	application-specific SM inv.
T1	TASKFREAK	Top level body contains `div[@id='header']`, `div[@id='container']`, and at most one `div[@id='calendar']`	XPath expr., state invariant
T2	TASKFREAK	All pages displaying current tasks via `table[@id='taskSheet']` match a given template	Reg. expr. state invariant
T3	TASKFREAK	Reload button in any state found via `img[@id='frk-status']` should lead to state displaying current tasks	application-specific SM inv.
H1	HITLIST	Contact template, shown in Figure 13	Reg. expr. (template) state invariant
U1	THETUNNEL	global variable `alive` is true during the game, and false after player fails	JAVASCRIPT invariant
U2	THETUNNEL	position of ship must be 32 times higher than the wall `ship_x+32 >= right_wall`	JAVASCRIPT invariant
U3	THETUNNEL	the background value must be between 0 and 20	JAVASCRIPT invariant
A1	All systems	Back button	Generic SM inv.

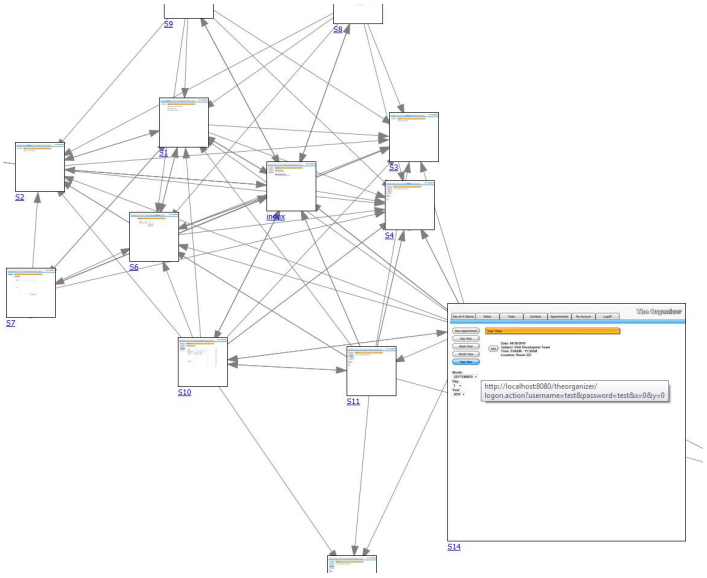

Fig. 11. The graph overview for THEORGANIZER, generated by the CrawlOverview plugin.

land on a logged off state. This invariant failed when we used Firefox as our embedded browser. Closer inspection revealed that the logoff element has an `onclick` event listener attached to it which calls a JAVASCRIPT function called `logoff()`. Firefox seems to have a conflict with this function name. One explanation could be that the word *logoff* is a hidden keyword in Firefox.

Invariant 5 is also worth mentioning. This invariant captures a pattern: clicking on an image element with `id=X` results in a state with an image as header having `src=head_X`, where X is for instance 'Tasks' or 'Contacts'. This invariant passes for all states except for 'Appointments'. Clicking on the *Appointments* element takes us to a state with an image as header having `src=head_dayView` as the header. Such inconsistencies in dynamic web applications are commonplace and usually difficult to spot manually. With automated invariant testing they can be detected and fixed in a systematic manner. The manual effort for THEORGANIZER case was less than 30 minutes.

8.2.3 TASKFREAK

TASKFREAK[21] is a simple task management and todo-list application written in PHP. Configuring CRAWLJAX required specifying the username and password to be used, as well as the HTML input fields where these needed to be entered, which was done through a simple *OnUrlLoad* plugin (Sec. 7.2). Furthermore, a quick inspection of the first page revealed that table data is clickable in TASKFREAK, which

21. http://www.taskfreak.com, TaskFreak! Original, v0.6.4

is why we specified that td- and th-elements are candidate clickables (Sec. 4.3). Since TASKFREAK includes a ticking clock in its page, we enabled the *DateOracleComparator* (Sec. 4.4), thus removing the current time before determining DOM-equality.

The random data for input fields (Sec. 4.7) works well for most of the data entry points for TASKFREAK. In order to permit reaching additional user interface states, we configured CRAWLJAX with custom values for a valid email address as well as an invalid one. Furthermore when attempting to change the password we ensured that the password entered the second time was the same as the one entered the first time. Identifying these data entry points requires a manual exploration of the application and the derived graph obtained through the *CrawlOverview* plugin, which for TASKFREAK took less than one hour.

A selection of the application-specific invariants for TASKFREAK is listed in Table 5. The first invariant T1 expresses the high-level design decision that at any time the top-level body-element of TASKFREAK consists of three div-elements: a *header* for the top navigation menu, a *container* for the actual list of todo items, and an optional *popup* area for, e.g., data entry in a calendar popping up. While simple in nature, this invariant already reveals an issue in TASKFREAK: after closing a popup, the corresponding div in the DOM-tree should be removed. In TASKFREAK, however, this is only correctly done for the calender popup when the user presses the *save* button: if *cancel* is pressed instead, the div-entry is not removed, leading to a (slowly) growing DOM-tree. The invariant that at most one calendar popup can exist at any time spots this problem.

Note that the popup-problem corresponds to a common AJAX-idiom: parts of the DOM-tree can be rendered invisible, and can be used for representing data, user-interface elements, and so on. It is the programmer's responsibility to manage these DOM-elements, and to "garbage collect" them, in order to avoid endlessly growing DOM-trees. Invariants can be used to express constraints over these parts of the DOM-tree, ensuring proper DOM-tree management.

Other invariants include the use of a template (see Sec. 5.2) to ensure that all states displaying the list of actions have the same structure (T2), as well invariants on the state machine expressing that the *reload* button always leads to the required state (T3), and that the browser's *Back* button behaves as expected (A1, which is violated in TASKFREAK).

8.2.4 HITLIST

Our third experimental subject for this study is the AJAX-based open source *HitList*,[22] which is a task manager based on PHP and jQuery.

For HITLIST, the configuration of our tool consisted of including all anchor tags as well as all input elements having a class attribute equal to add_button as candidate clickables. Furthermore, we excluded from crawling all the elements that deleted items from the application (e.g., <a title=Delete...>, <a class=delete-button...>). To ignore subtle DOM differences, we pipelined the generic *Table* and *List* oracle comparators. These comparators abstract away the differences in structures of the HITLIST tables and lists.

To constrain the state space, we created a *CrawlCondition* that ensures a contact can only be added once during the crawling phase. This was done by checking a JAVASCRIPT condition in the *Add Contact* state, as shown in Figure 12. When the precondition (a state containing the text 'Add Contact') is satisfied, the JAVASCRIPT condition retrieves the list of contacts from the server and checks whether there are no contacts present during execution. In that case it returns true, allowing our tool to add a new contact.

```
//check on Add Contact page
preCondition = new RegexCondition("Add Contact");
//check whether 'John Doe' does not already exist
condition = new JavaScriptCondition("$.ajax({ url: 'ajax/
     home.php', async: false }).responseText.indexOf('John
     Doe')==-1");
crawlConditions.add(new CrawlCondition("AddOnce", "Only add
      John Doe once", condition, preCondition));
```

Fig. 12. A JAVASCRIPT crawl condition, with regular expression precondition, for adding a contact once in HITLIST.

From the output of the *Crawl Overview* plugin we generated a regular expression for the contact state. We manually augmented this generated template and created a custom *Contact* regular expression invariant for the contact list. This template, shown in Figure 13, serves as a DOM invariant, since it checks the structure as well as the validity of the contact's name, phone number, e-mail, and *id*.

With H1, we were able to detect a violation in a regression version of HITLIST, namely leading zeros in phone numbers were missing (e.g., 0641288822 was saved as 641288822).

8.2.5 THETUNNEL

Our last subject for this study is an open source web-based implementation of a tunnel game.[23] In this game, the player

```
<TABLE class="contact">\s*
 <TBODY>\s*
  <TR>\s*
   <TD height="[0-9]+" width="[0-9]+">\s*
    <IMG src="public/images/[a-z]+.png"/>\s*
   </TD>\s*
   <TD style="[^"]*">\s*
    [a-zA-Z ]{3,100}[^<]*<BR/>(.*?)
    [a-zA-Z0-9\._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}[^<]*<
         BR/>(.*?)
    [0-9]{10,15}[^<]*<BR/>\s*
    <A class="action view-details" href="javascript:;" id
         ="[0-9]+">(.*?)view details(.*?)</A>\s*
   </TD>\s*
  </TR>\s*
 </TBODY>\s*
</TABLE>
```

Fig. 13. Regular expression (template) invariant for HITLIST contact list.

22. HitList Version 0.01.09b, http://code.google.com/p/hit-list/

23. http://arcade.christianmontoya.com/tunnel/

controls an airplane and the objective is to avoid hitting a moving wall. It is written using jQuery.

For this web application, we were interested in documenting JAVASCRIPT invariants. Therefore, we analyzed the JAVASCRIPT source code manually and documented a number of invariants on the global variables that could be used as assertions to test the program state.

A few of the invariants we obtained are listed in Table 5. These invariants were turned into assertions and be used for regression testing the JAVASCRIPT code automatically. Assertions on global variables can be checked through ATUSA's invariant checking API's. For instance, Figure 14 shows how U2 in Table 5 can be checked through ATUSA.

```
crawler.addInvariant(new JavaScriptCondition("ship_x + 32
    >= right_wall"));
```

Fig. 14. Checking invariants on JAVASCRIPT global variables.

Checking local variables can be done by injecting the assertion code into the JAVASCRIPT source code through a proxy. The details of this technique can be found in [14].

8.2.6 Findings

Going back to our first research question (RQ1), from the four case studies described, we conclude the following:

- Writing invariants captures and requires an understanding of the design of the web application. The automatically generated crawl overview helps us in the process of program comprehension.
- Invariants over the relations of elements and their attributes on the DOM tree can be naturally expressed using XPath expressions. Invariants capturing the structure of the elements can be expressed using template-based regular expressions. Those constraining actions and their consequences can be captured in state machine invariants. Finally, code-level JAVASCRIPT design contracts can be easily expressed as JAVASCRIPT expressions.
- Invariants can be used to find various faults, including DOM memory leaks (TASKFREAK), crossbrowser inconsistencies (THEORGANIZER), and regressions (HITLIST).
- The manual effort involved in configuring CRAWLJAX and writing the described invariants in this study, is minimum, amounting to less than one hour for each of the cases covered.

8.3 Study 2: TUDU

In this study, we are particularly concerned with assessing the fault revealing capabilities, scalability, and required manual effort our approach (RQ2-RQ4).

8.3.1 Subject System

Our experimental subject in this study is the AJAX-based open source TUDU web application[24] for managing personal todo lists, which has also been used by other researchers [22]. 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[25] and Scriptaculous[26], and consists of around 11k LOC of external JAVASCRIPT libraries and 580 internal LOC.

8.3.2 Case study setup

For RQ2 and RQ3, we configured ATUSA (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. We use the *TestSuite Generator* plugin in this case study to automatically generate a test suite from the inferred state machine. To address RQ4, we report the time spent on parts that required manual work.

As shown in Table 6, 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 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 input values were automatically inserted into the browser and submitted. Upon each detected state change, the invariants were checked and reports were generated if any inconsistencies 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 [16].

To that end, we instrumented the presentation part of the server-side Java code (`tudu-dwr`) with Clover. We exclude the server-side business logic and database layers, since we are merely interested in the user interface parts. For the client-side, we instrumented JAVASCRIPT libraries and custom code with JSCoverage[27], and deployed the whole web application to an application server (Apache Tomcat). For each test run,

24. http://tudu.sourceforge.net
25. http://directwebremoting.org
26. http://script.aculo.us
27. http://siliconforks.com/jscoverage/

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.

8.3.3 Findings

The results of this study are presented in Table 6. Based on these observations we conclude that:

- The use of ATUSA can help to reveal generic faults, such as DOM violations, automatically;
- As far as RQ2 is concerned, the generated test suite can give us useful code coverage: 73% of server-side presentation code and 75% of client-side JAVASCRIPT custom code; Note that only partial parts of the external JAVASCRIPT libraries are actually used by TUDU resulting in a low coverage percentage (35%). ATUSA revealed most of the 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 past the maximum 60 minutes, but did not influence the fault detection results. The code coverage, however, improved by approximately 10%;
- 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 (RQ3).
- The manual effort involved in setting up ATUSA (less than half an hour in this case) is minimal (RQ4);

8.4 Study 3: Finding Real-Life Bugs

Our final 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 (RQ4), and to assess the capability of ATUSA to find faults that actually occurred during development (RQ2).

8.4.1 Subject System

The case at hand is Coachjezelf (CJZ, "Coach Yourself"),[28] 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.

28. See www.coachjezelf.nl for more information (in Dutch).

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`[29] 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.

8.4.2 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.

8.4.3 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 invariant 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 15.

8.4.4 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

29. jquery.com

TABLE 6
TUDU case study.

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 7
Faults found in CJZ-AJAX.

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

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

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

Fig. 15. Application-specific invariants expressed using XPath.

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 7.

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.

8.4.5 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 (RQ2);
- Application-specific invariants can help to document and test the essence of an AJAX application, such as the synchronization between two widgets (RQ1-RQ2);
- The manual effort in expressing such invariants in Java and using them in ATUSA is minimal (RQ4).

9 DISCUSSION

9.1 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.

9.2 Invariants

Our solution to the oracle problem is to include invariants (as also advocated by, e.g., Meyer [29]). 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 generic or application-specific 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 [23]) — 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.

9.3 Generated versus hand-coded JAVASCRIPT

The case studies we conducted involve two different popular JAVASCRIPT libraries (i.e., jQuery and Prototype) in combination with hand-written JAVASCRIPT code. Alternative frameworks exist, such as Google's Web Toolkit (GWT)[30] 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.

9.4 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.

9.5 Performance and Scalability

The state space of any realistic web application is huge and can cause the well-known *state explosion problem*. To constrain the state space, we provide the tester with a set of configurable options. These constraints include the maximum search depth level, similarity threshold for comparing states, 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 crawling an AJAX site depends on many factors such as the speed at which the server can handle requests, how fast the browser and client-side JAVASCRIPT can update the interface, and the size of the DOM tree.

30. http://code.google.com/webtoolkit/

9.6 Application Size

The six experimental subjects involve around 20,000 lines of JAVASCRIPT library code, several hundred lines of custom application code, and several thousand dynamic DOM states. One might wonder whether the size of the subjects counts against the external validity of our study. Our results, however, are based on *dynamic* analysis rather than static code analysis; hence the amount of JAVASCRIPT code is not the determining factor in our view. The number of dynamic states is, in this case, a more realistic measure. The limiting factor for the number of states to be examined is the amount of memory available and the size of the DOM-tree. Based on our experiments, the maximum number of states can be calculated by $\frac{sizeOf(memory)}{3 \times sizeOf(DOM)}$. The average DOM size of enterprise applications is around 0.25 MB [24]. On a workstation with 4 GB of memory, this would result in around 5460 states, which is sufficient for most enterprise web applications.

9.7 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. The main goal of Study 1 (Section 8.2) was to demonstrate what type of invariants can be found in modern web applications and how different types of invariants can be expressed in ATUSA for automated testing. We have merely provided a few examples of each type and the list is not exhaustive. The small number of examples could, however, be a threat to external validity. More studies need to be done to extend the instances of each type.

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 second study (Section 8.3) is concerned, we selected them from 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 possibly towards the fault models we have. With respect to *reliability*, our tools and all the subject systems of studies 1 and 2 are open source, making these cases fully reproducible.

9.8 Ajax Testing Strategies

ATUSA is a first, but essential step in automating the testing process of AJAX applications. 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.

In our recent work [33], we have presented how our approach can be used for conducting *regression testing* of highly dynamic web applications. The initial results are very

promising: through a number of case studies, we show how generated test suites can detect regressions in different versions of a web application, through oracle comparator pipelining.

Another direction involves the application to *security testing* of *Web 2.0* widget interactions [6], which we have conducted in close collaboration with the industry.[31]

Application in the area of *accessibility testing* involves compliance to W3C accessibility standards. Initial results in this area involve an application to Google's AdSense Front End, 3.0[32] through an internship at Google (London) [32].

Further, the technique is currently being applied by Fujitsu Laboratories of America to a number of industrial web applications. The approach is also being adopted for web model-checking using the inferred state machine. Recently, we have used the approach to automate *cross-browser compatibility* testing of modern web applications [26].

10 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 [24], which can dynamically make a full pass over an AJAX application. Our current work consists of extending the crawler substantially for supporting automated testing of modern web applications. We developed a series of plugins, collectively called ATUSA, for invariant-based testing and test suite generation.

To summarize, 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 that serve as oracles.
2) A series of generic invariant types (e.g., XPath, template-based Regular Expression, JAVASCRIPT expression) for expressing web application invariants for testing.
3) 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.
4) An extension of our open source AJAX crawler, CRAWLJAX and the implementation of the testing approach ATUSA, offering generic invariant checking components as well as a plugin-mechanism to add application-specific state validators and test suite generation.
5) An empirical evaluation, by means of three 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 makes our tool 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 more testing plugins for spotting development errors and security vulnerabilities in *Web 2.0* applications. Automatically detecting dynamic structural and JAVASCRIPT invariants in modern web applications [14] is another route we will be pursuing in the future work.

31. Exact http://www.exact.com
32. http://www.google.com/adsense

Acknowledgments

The authors are grateful to Cor-Paul Bezemer, Stefan Lenselink, and Frank Groeneveld for their contributions in improving the CRAWLJAX core implementation. We would also like to thank the anonymous reviewers for their valuable and constructive comments.

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. Barnett, R. Deline, M. Fahndrich, K. Rustan, M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *J. of Object Technology*, 3(6):1–30, 2004.

[4] M. Benedikt, J. Freire, and P. Godefroid. VeriWeb: Automatically testing dynamic web sites. In *Proc. 11th Int. Conf. on World Wide Web (WWW'02)*, pages 654–668, 2002.

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

[6] C.-P. Bezemer, A. Mesbah, and A. van Deursen. Automated security testing of web widget interactions. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering (ESEC-FSE'09)*, pages 81–91. ACM, 2009.

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

[8] L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37, 2006.

[9] L. de Alfaro. Model checking the world wide web. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, pages 337–349. Springer-Verlag, 2001.

[10] L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. MCWEB: A model-checking tool for web site debugging. In *WWW'01 Posters*, 2001.

[11] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II. Leveraging user-session data to support web application testing. *IEEE Trans. Softw. Eng.*, 31(3):187–202, 2005.

[12] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.

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

[14] F. Groeneveld, A. Mesbah, and A. van Deursen. Automatic invariant detection in dynamic web applications. Technical Report TUD-SERG-2010-037, Delft University of Technology, 2010.

[15] W. Halfond, S. Anand, and A. Orso. Precise interface identification to improve testing and analysis of web applications. In *Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA'09)*, pages 285–296. ACM, 2009.

[16] 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.

[17] W. Halfond and A. Orso. Automated identification of parameter mismatches in web applications. In *Proceedings of the 16th International Symposium on Foundations of software engineering (FSE'08)*, pages 181–191. ACM, 2008.
[18] 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. *J. of Computer Networks*, 48(5):739–761, 2005.
[19] 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.
[20] V. L. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, 10:707–710, 1996.
[21] A. Marchetto, F. Ricca, and P. Tonella. A case study-based comparison of web testing techniques applied to Ajax web applications. *Int. Journal on Software Tools for Technology Transfer*, 10(6):477–492, 2008.
[22] 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.
[23] A. Memon. An event-flow model of GUI-based applications for testing: Research articles. *Softw. Test. Verif. Reliab.*, 17(3):137–157, 2007.
[24] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling Ajax by inferring user interface state changes. In *Proceedings of the 8th International Conference on Web Engineering (ICWE'08)*, pages 122–134. IEEE Computer Society, 2008.
[25] 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.
[26] A. Mesbah and M. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, 2011.
[27] 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.
[28] A. Mesbah and A. van Deursen. Invariant-based automatic testing of Ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 210–220. IEEE Computer Society, 2009.
[29] B. Meyer. Seven principles of software testing. *IEEE Computer*, 41(8):99–101, August 2008.
[30] K. Pattabiraman and B. Zorn. DoDOM: Leveraging DOM invariants for Web 2.0 application robustness testing. In *Proc. Int. Conf. Sw. Reliability Engineering(ISSRE'10)*. IEEE Computer Society, 2010.
[31] 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.
[32] D. Roest. Automated regression testing of Ajax web applications. Master's thesis, Delft University of Technology, February 2010.
[33] D. Roest, A. Mesbah, and A. van Deursen. Regression testing Ajax applications: Coping with dynamism. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10)*, pages 128–136. IEEE Computer Society, 2010.
[34] 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.
[35] 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.
[36] 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.
[37] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 171–180. ACM, 2008.
[38] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.
[39] J. Y. Yen. Finding the k shortest loopless paths in a network. *Manag. Sci.*, 17(11):712–716, 1971.
[40] R. K. Yin. *Case Study Research: Design and Methods*. SAGE Publications Inc, 3d edition, 2003.

Ali Mesbah is an assistant professor of software engineering at the University of British Columbia (UBC). He obtained his MSc degree (2003) and his PhD degree cum laude (2009) in computer science from Delft University of Technology. From 2001 until 2005 he was a software engineer at WEST consulting in the Netherlands, where he was responsible for development and maintenance of software systems. His area of research is software engineering with an emphasis on software analysis and testing of web-based systems. He is the recipient of an ACM SIGSOFT Distinguished Paper Award at the ACM/IEEE International Conference on Software Engineering (ICSE 2009).

Arie van Deursen is a full professor at Delft University of Technology, where he is heading the Software Engineering Research Group. He obtained his MSc degree in computer science in 1990 from the Vrije Universiteit, Amsterdam, and his PhD degree from the University of Amsterdam in 1994. From 1996 until 2006 he was a research leader at CWI, the Dutch National Institute for Research in Mathematics in Computer Science. His research interests include reverse engineering, program comprehension, and software architecture. He was program chair of the Working Conference on Reverse Engineering (WCRE) in 2002 and 2003, and served on numerous program committees in the areas of software evolution, maintenance, and software engineering in general. He is a member of the editorial board of the Empirical Software Engineering journal.

Danny Roest received the MSc degree in computer science from Delft University of Technology in 2010. He is currently a software engineer in the Netherlands. He has a great interest in developing user friendly and interactive web applications as well as applying software testing techniques to create dependable web applications.

TUD-SERG-2011-003
ISSN 1872-5392