Automated Regression Testing of Ajax Web Applications

Msc Thesis

Danny Roest

Automated Regression Testing of Ajax Web Applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Danny Roest
born in Rotterdam, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Automated Regression Testing of Ajax Web Applications

Author:	Danny Roest
Student id:	1150510
Email:	dannyroest@gmail.com

Abstract

There is a growing trend of moving desktop applications to the Web by using AJAX to create user-friendly and interactive interfaces. Well-known examples include GMail, Hotmail, Google Wave and office applications. One common way to provide assurance about the correctness of such complex and evolving systems is through regression testing. Regression testing classical web applications has already been a notoriously daunting task because of the dynamism in web interfaces. AJAX applications pose an even greater challenge since the test case fragility degree is higher due to extensive run-time manipulation of the DOM tree, asynchronous client/server interactions, and (untyped) JAVASCRIPT. In this research, we propose a technique, in which we automatically infer a model of the web application and use this as an input for the test suite. We apply pipelined oracle comparators along with generated templates, to deal with the dynamic non-deterministic behavior in AJAX user interfaces. Our approach, implemented in the open source testing tool CRAWLJAX, provides a set of generic oracle comparators, template generators, and preconditions for handling the dynamic aspect of AJAX applications. We present a plugin that shows visualizations of test failure output in terms of differences between expected and observed DOM trees.We describe four case studies evaluating the effectiveness, scalability, and required manual effort of the approach and the applicability of CRAWLJAX in a real world production environment. The various improvements made to CRAWLJAX in this research are also discussed.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. Ir. A. Mesbah, Faculty EEMCS, TU Delft
Committee Member:	Ir. B.R. Sodoyer, Faculty EEMCS, TU Delft
Committee Member:	Dr. H.G. Gross , Faculty EEMCS, TU Delft

Preface

This thesis is the result of the work I did during the past year, which I really enjoyed. I always have had an interest for AJAX web applications and had some experience with them. When I came to Arie van Deursen for a talk about graduation, he asked me what I thought of AJAX and I responded ”COOOL!”. He explained CRAWLJAX and we went to Ali Mesbah where Arie introduced me by saying: ”This is Danny, he thinks AJAX is awesome, boom, cool, kick!”. I was very enthusiastic about my project and looked forward working on it, in contrast to the fact that I normally do not really like school (understatement).

This thesis would not have succeeded without the help of some people. First, I want to thank Ali for all his help and brainstorm sessions from which I learned so much. Ali did not only teach me a lot of things, but also inspired me and gave me insights in how fun doing research can be. I think we are a good team and I am proud to have a paper published with your and Arie’s name on it. Secondly, I want to thank Arie for all this support, sharp comments, and enthusiasm. It is an honor to work with somebody like Arie who has so much experience. Also, thank you for arranging my internship at Google. This brings me to the next person, Mike Davis, who also taught me so much and helped me very good with my internship. I also want to thank the other Googlers for all the fun times at TGIF and in the pubs. Finally, I want to thank my girlfriend Annemarie and my parents for always supporting me and listening to my occasional rambling about CRAWLJAX and Google.

This project was a really good experience and I am happy that I could work with such great and smart people. I learned a lot and got new insights which changed my life. This last year was by far the most best/fun year of my educational period.

Danny Roest

Krimpen aan den IJssel, The Netherlands
3 February 2010

Contents

List of Figures

Chapter 1

Introduction

There is a growing trend of moving desktop applications to the Web. Well-known examples include GMail, Hotmail, Google Wave and office applications. One of the implications of this move to the web is that the *dependability* [24] of web applications is becoming increasingly important [6, 13].

These new generation websites use AJAX [8] (Asynchronous Javascript And XML) for more responsive and richer user interfaces. AJAX is a technique for improving the interactivity that uses JAVASCRIPT for asynchronous client server communication to alter the Document Object Model (DOM). This leads to new user interfaces. In contrast to the old static pages where a link loads an entire page, a web application that uses AJAX can be partially updated.

While the use of AJAX technology positively affects the user-friendliness and the interactiveness of web applications [21], it comes at a price: AJAX applications are notoriously error-prone due to (1) the asynchronous client-server communication, (2) extensive use of the DOM, (3) and (untyped) JAVASCRIPT on the client.

Testing software is considered a very important aspect of software development. As software evolves, one common way to provide assurance about the correctness of the modified system is through *regression testing*. Regression testing involves selective re-testing of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements [9]. This involves saving and reusing test suites created for earlier (correct) versions of the software [3], with the intent of determining whether modifications (e.g., bug fixes) have not created any new problems/bugs.

For example, many software applications are developed and maintained by multiple developers in often different geographic regions. As a consequence a programmer can introduce errors in other parts of the system which he is unaware of.

An often used way to obtain an oracle for regression testing, is to compare new outputs with saved outputs obtained from runs with an earlier version of the system. For web applications, *oracle comparators* [30] typically analyze the differences between server response pages [28].

For highly dynamic web applications, this comparison-based approach suffers from a high probability of *false positives*, i.e., differences between old and new pages that are irrelevant and do not point to an actual (regression) fault [28, 32].

For AJAX applications, the run-time dynamic manipulation of the DOM-tree as well as the asynchronous client/server interactions make it particularly hard to control the level of dynamism during regression testing. It is the objective of this research to find solutions for this *controllability* problem, thus facilitating robust comparison-based regression testing of AJAX applications.

As much as half of the code of software is currently constituted by the *user interface* [17, 23], thus testing these user interfaces is very important, but can be very time consuming. *Automatic test case generation* has become more popular because it has the potential to reduce testing costs and increase software quality [5].

The focus of this research is proposing an automated approach for conducting regression testing on AJAX web applications. It resulted in a research paper, accepted for presentation at and publication in the proceedings of the 3rd IEEE International Conference on Software Testing, Verification and Validation [27]. The title of this paper is 'Regression Testing Ajax Applications: Coping with Dynamism'. Our research also resulted in a Google Techtalk at Google London about CRAWLJAX which is published on Youtube[1].

This thesis is based and extended on the excellent work and paper by Mesbah *et al.* [22] which received the highly valued *ACM SIGSOFT Distinguished Paper Award*.

We give a brief summary (Section 3.2) in this thesis of earlier work on automated testing of AJAX applications [20, 22], since this provides the context and motivation for the present research. Our results, however, are applicable in a wider context as well: most of our methods and techniques can also be applied if the test cases are obtained manually (e.g., through a capture-and-replay tool such as Selenium IDE[2]).

This thesis starts with the related work by other researches in Chapter 2. Then we explain our approach in Chapter 3 and discuss the various challenges when conducting regression testing on AJAX applications. The following three chapters (4, 5, 6) discuss our approach in more detail. Creating a regression test suite is discussed in Chapter 4. An important aspect of the test suite is determining state equivalence (Chapter 5) which is done by: (1) using oracle comparators, (2) combining these comparators, and (3) dealing with dynamism by adopting preconditions. Modeling structures with templates for handling the dynamic states is discussed in Chapter 6. During this research, various improvements are made to CRAWLJAX which are presented in Chapter 7. Our evaluation and case studies at Google London are discussed respectively in Chapter 8 and Chapter 9. We conclude this thesis with concluding remarks and suggestions for future work in Chapter 10.

[1] `http://www.youtube.com/watch?v=rYAO94GnBlY`

[2] `http://seleniumhq.org/projects/ide/`

Chapter 2

Related Work

To the best of our knowledge, there does not exists earlier work that focuses on the challenges of regression testing AJAX applications. Common faults in AJAX web applications are classified by Marchetto *et al.* in [12]:

1. Authentication: authentication faults are incorrect user rights, logging in without a password etc.
2. Multi-lingual: a web site can have different languages. A fault can be a switch in languages, which leads to displaying the wrong language.
3. Functionality: a functional fault can lead to unexpected behavior.
4. Portability: an application can behave different than expected in different browsers.
5. Navigation: a navigation fault can occur when a link is invalid.
6. Asynchronous communication: when there is a communication error, the DOM can come in an unintended state.
7. Session: session errors can occur when user data or settings are incorrectly saved.
8. Form construction: faults in forms can lead to incorrect storing of data.

While regression testing has been successfully applied in many software domains [19, 25], web applications have gained very limited attention from the research community [33, 32].

Alshahwan and Harman [1] discuss an algorithm for regression testing based on session data [29, 7] repair. Their automated approach examines the structure of the newer version of the web application, detects the changes, and applies repair actions by creating new links between pages or splitting the session. Session-based testing techniques, however, merely focus on synchronous requests to the server and lack the complete state information required in AJAX regression testing.

The traditional way of comparing the test output in regression testing web applications is through *diff*-based comparisons, which report too many false alarms. Sprenkle *et al.* [30] propose a set of specialized oracle comparators for testing web applications's HTML output, by detecting differences in the content and structure through *diff*. Two classes of oracle comparators are defined: Content-based and Structure-based. They created different HTML oracles comparators and performed experiments on different web sites in which different oracle comparators were combined. According to their results, choosing the best oracle comparators and the best combination of oracle comparators is very application specific. In general, the union of a date comparator and a style comparator performed best.

In a recent paper, Soechting *et al.* [28] proposed a tool for measuring syntactic differences in tree-structured documents such as XML and HTML, to reduce the number of false positives. They calculate a distance metric between the expected and actual output, based on an analysis of structural differences and semantic features.

The problems related to user interface (regression) testing are the large amount of event sequences, testing the output and determining inputs. In [19, 18] the authors propose an approach which automatically generates and repairs test cases. This is achieved by modeling the GUI with *control-flow graphs* and *call graphs* and analyzing these graphs.

In [10] Koopman *et al.* propose an automated model-based testing system for on-the-fly testing of thin-client web applications. They consider the many combinations of input and output types, and try to model these relations by specifying functions from state and input to functions from output to the new states.

Stepien *et al.* [31] test web applications by extracting data from the pages and verifying this against an oracle. Templates are used to model different structures like tables to extract the important data. The navigation behavior and sequence of actions (request and response) performed by the application are also verified.

A technique called *web slicing* is used by Xu *et al.* [33] for regression testing which is based on the work of Ricca and Tonella [26]. They consider the links in a web application to detect dependencies between pages. When there is a change in the application the idea is to only test the pages depended on the changed page.

In [14] semantic interactions are introduced by Marchetto *et al.* to reduce test suites. There is a semantic interaction between two events if the effect on the DOM state of the two associated callbacks are not independent. This means that if the ordering of two events changes, the state of the application also changes.

Cost-benefit models are used by the authors in [4] for assessing and comparing different regression testing techniques. Factors like time constraints and incremental resource availability are included, which, according to their results, can affect the assessment of different regression techniques advantages.

Memon *et al.* [16] present a framework named DART for daily testing. The focus is on high automation and the test cases and oracles are automatically generated via graph traversal and enumerating the events.

The disadvantages of the mentioned approaches are that they do not consider the client-side JAVASCRIPT code and the dynamic behavior of AJAX applications.

In our approach, each comparator is specialized in stripping the differences from the expected and actual output and then use a simple string comparison. In addition, we combine the comparators by pipelining the stripped outputs, to handle the highly dynamic behavior of AJAX user interfaces. We also we constrain the state space to which a certain comparator has to be applied by using preconditions.

Chapter 3

Approach

This chapter discusses our approach for regression testing AJAX web applications and the related challenges.

3.1 Main Idea

This section discusses our general idea for regression testing a web application.

Conducting regression testing on AJAX applications needs a test suite that compares the current version with the gold standard [3] (trusted version). We infer a model of the web application which acts as the oracle in the test suite. The test suite can be executed on new versions of the web application to detect changes. These different types of changes, coming with several challenges, are discussed in this chapter. Our approach for regression testing can be described as follows:

1. Infer a model M (the oracle) of a the web application W_0 (the gold standard)
2. Save M
3. Create a the test suite T with as input M
4. Run each test case t in T on W_i where $i > 0$ and t is a path in W_0
5. Repeat steps 3 and 4 for different versions

When the application is updated, step 1 can be repeated in to create a new oracle from the gold standard. In every test case the current state is compared to the gold standard and is tested whether every test case can be fully executed to detect missing/changed links.

3.2 Background

To conduct regression testing on a web application a model is needed of the application. Since manual creating a model of the application is not very practical and scalable we use an automated approach for inferring this model as a state machine.

Mesbah *et al.* [20] proposed a new type of web crawler, called CRAWLJAX, capable of exercising client-side code, detecting and executing doorways to various dynamic states of AJAX-based applications within browser's dynamically built DOM. While crawling, CRAWLJAX infers a *state-flow graph* (the state machine) capturing the states of the user interface, and the possible event-based (e.g., onclick, onmouseover) transitions between them, by analyzing the DOM before and after firing an event.

Our approaches are implemented in Java and integrated in the discussed open source testing tool CRAWLJAX. Inferring a state machine with CRAWLJAX can be described as follows.

When CRAWLJAX is started, it loads the initial URL and the loaded page is considered as the initial state. Every found state is crawled as shown in Algorithm 1. When a state is loaded, CRAWLJAX scans the DOM tree for clickable elements (line 2), and tries to click all the found elements (line 5). Before elements are clicked, form values (Section 7.3) are entered (line 4), because state transitions can be dependent on form values (Section 9.2). After clicking an element, CRAWLJAX checks whether the current state is changed by using oracle comparators (Chapter 5). If the state is changed, the current state is added as a new state to the state machine (line 7). User-defined tests are performed in lines 9 and 10 by running plugins (Section 7.4) and testing invariants (Section 7.1.1). If the newly found state is allowed (Section 7.1.2) to be crawled (line 11), this state is recursively crawled (line 12). As can be seen, CRAWLJAX uses a depth-first algorithm and when its needs to backtrack to a certain state *X*, the initial URL is reloaded and the elements leading to state *X* are clicked.

While running the crawler can be considered as a first smoke test pass, the derived state machine itself can be further used for testing purposes. More recently, Mesbah et. al. proposed an approach for automatic testing of AJAX user interfaces, called ATUSA [22], in which they automatically generate (JUnit) test cases from the inferred state-flow graph. Each test case method represents the sequence of events from the initial state to a target state. The state transitions are thus accomplished by firing actual DOM events on the elements[1] of the page in a real browser.

The gold standard [3] is composed of structural invariants (see [22, 2]) defined by the tester and the saved output (DOM trees) of a previous execution-run on a trusted version of the application.

3.2.1 Running Example

In this paper we illustrate the challenges of regression testing and our approach with a simple TODO AJAX application in which new items can be added and existing ones removed. The navigational model of TODO is depicted in Figure 3.1. To obtain the state-flow graph shown in Figure 3.2, CRAWLJAX:

1. clicks Todo - Add - Save - Remove - Yes. Cycle detected: Stop in *state1*. (# of Items in the list: 0)
2. backtracks to *state4*: Todo - Add - Save - Remove. (# of Items: 0)
3. clicks No. Cycle detected: Stop in *state2*. (# of Items: 1)
4. backtracks to *state2*: Todo - Add (# of Items: 1)

[1]For the sake of simplicity, we call such elements 'clickable' in this thesis, however they could have any type of event-listener.

Algorithm 1 The Algorithm of Crawljax for Crawling a State

```
1:  procedure CRAWLSTATE($state$)
2:  $elements \leftarrow$ findClickableElementsInState($state$)
3:  for all $element \in elements$ do
4:      enterFormValues()
5:      clickElement(element)
6:      $newState$ = getStateInBrowser()
7:      if stateChanged($newState$) then
8:          updateStateFlowGraph(newState, element)
9:          runPlugins()
10:         testInvariants()
11:         if stateAllowedToBeCrawled($newState$) then
12:             crawlState($newState$)
13:         end if
14:     end if
15: end for
16: end procedure
```

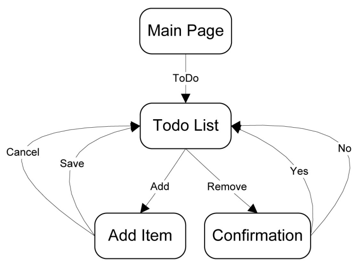


Figure 3.1: Navigational model of TODO.

5. clicks Cancel, ends up in *state3* (# of Items: 1), and stops.

Note that the generated graph has an extra state compared to the navigation model, since *state1* (empty list) and *state3* (list with one item) on the *Todo List* page are not equal. When CRAWLJAX is finished with crawling it has crawled the following paths.

1. Event sequence: Todo - Add - Save - Remove - Yes
2. Event sequence: Todo - Add - Save - Remove - No
3. Event sequence: Todo - Add - Cancel

The crawled paths by CRAWLJAX form the basis for regression test suites as discussed in Chapter 4.

3.3 Challenges

This section discusses the challenges when conduction regression testing on AJAX applications.

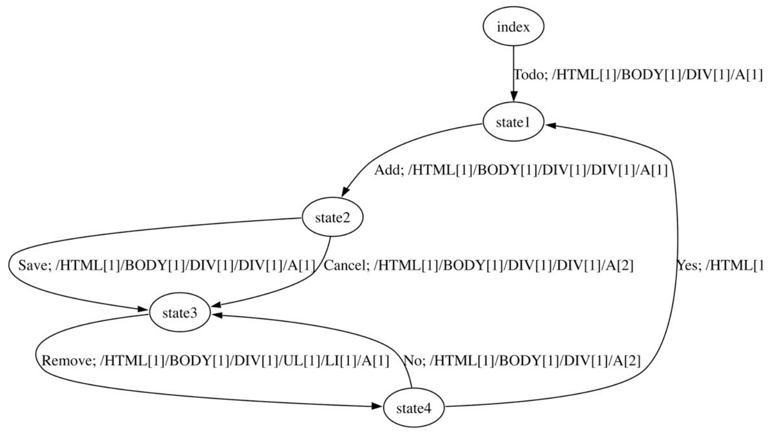

index
Todo; /HTML[1]/BODY[1]/DIV[1]/A[1]
state1
Add; /HTML[1]/BODY[1]/DIV[1]/DIV[1]/A[1]
state2
Save; /HTML[1]/BODY[1]/DIV[1]/DIV[1]/A[1]
Cancel; /HTML[1]/BODY[1]/DIV[1]/DIV[1]/A[2]
Yes; /HTML[1
state3
Remove; /HTML[1]/BODY[1]/DIV[1]/UL[1]/LI[1]/A[1]
No; /HTML[1]/BODY[1]/DIV[1]/A[2]
state4

Figure 3.2: Inferred state-flow graph for TODO.

3.3.1 Nondeterministic Behavior

Most regression testing techniques assume [25] that when a test case is executed, the application under test follows the same paths and produces the same output, each time it is run on an unmodified program. Multiple executions of an unaltered web application with the same input may, however, produce different results. Causes of such nondeterministic [15] behavior can be, for instance, (HTTP) network delays, asynchronous client/server interactions, non-sequential handling of requests by the server, randomly produced or constantly changing data on real-time web applications.

For instance, consider a web page that displays the current date-time (see Figure 3.3). Simply comparing the actual state with the gold standard results in many failures (of nonexistent bugs) that are due to the mismatches in the date-time part of the page. This characteristic makes regression testing web applications in general [30] and AJAX in particular notoriously difficult.

3.3.2 Dynamic User Interface States

A recurring problem is when a certain test case is executed multiple times and each run changes the state on the back-end. Consider a test case that adds a to-do item to the list. Figure 3.3 shows two DOM fragments of the TODO example at different times, displaying the list of to-do items. The test case expects the first fragment and the second fragment represents the current DOM in the browser. When compared naively, the test reports a failure since (in addition to the date-time part) the content of the lists does not match. An often used but cumbersome approach for this problem is to reset the back-end to an initial state before executing the test case. The challenge here is making the oracle comparator oblivious to such valid differences.

```
<H1>Todo items</H1>
<UL id="todo">
  <LI>
    Cleaning <A href="#" onclick="remove(0)">Remove</A>
  </LI>
</UL>
<A href="#" onclick="addItem()">Add</A>
<P class="past">Last update: 02-01-2010 16:43</P>
```

```
<H1>Todo items</H1>
<UL id="todo">
  <LI>
    Groceries <A href="#" onclick="remove(1)">Remove</A>
  </LI>
  <LI>
    Cleaning <A href="#" onclick="remove(0)">Remove</A>
  </LI>
</UL>
<A href="#" onclick="addItem()">Add</A>
<P class="today">Last update: 06-01-2010 12:50</P>
```

Figure 3.3: Two DOM string fragments of TODO at different timestamps.

3.3.3 State Transitions

In AJAX applications, state transitions are usually initiated by firing events on DOM elements having event-listeners. To be able to traverse the state space during testing, such clickable elements have to be identified on the run-time DOM tree first. One common way to identify such elements is through their XPath expression, which represents their position on the DOM tree in relation to other elements. DOM elements, however, can easily be moved, changed, or replaced. As a consequence the clickable element cannot be detected, and therefore the test fails. A worse scenario is that a similar element is mistakenly selected, which causes a transition to a different state than expected.

For instance, consider a test case that uses the XPath expression of the `remove` clickable of the *Cleaning* item on the first DOM fragment of Figure 3.3 (e.g, `/HTML[1]/BODY[1]/DIV[1]/UL[1]/LI[1]/A[1]`). When the same test case is run on an updated list, which contains the *Groceries* item, naively firing the *onclick* event on the element found with the XPath expression will remove the *Groceries* item instead of the intended *Cleaning* one. Spotting the correct clickable element with a high degree of certainty is thus very important for conducting robust regression testing of AJAX user interfaces.

Chapter 4

Regression Test Suite

This chapter discusses how we create a test suite that is capable of regression testing AJAX applications.

4.1 Main Idea

The main idea of our approach is that the model in Section 3.1 represents the crawled paths by CRAWLJAX with the corresponding states and clickable elements. A crawled path is a traversal from the initial state to a state in the state-flow graph. The behavior and states of the application can be dependent on the order of the executed crawl paths, thus the paths are therefore executed in the same order as they were crawled.

The test suite consists of test cases, where a test case represents a crawled path that is executed the same as in the crawling phase. A test case tests whether:

1. every state in the path can be reached
2. the visited states are equivalent with the states in the gold standard

4.2 Approach

In Chapter 3 we discussed our approach and several challenges when performing regression testing on AJAX applications. This section discusses our approach to cope with these challenges.

4.2.1 Dynamic States

To detect the changes in the application one needs a comparison mechanism which determines whether two states are equivalent. Chapter 5 discusses our approach that uses oracle comparators for a robust method for comparing states. We combine the oracle comparators to cooperate and use preconditions to dynamically determine whether an oracle comparator should be used.

4.2.2 Network Delays

Test cases should be stable and not flaky, thus (reasonable) network delays should not result in failing test cases. Naively waiting for a finite amount of time when loading a state can result in comparing states when one state is not completely loaded (the waiting time is too short) or in a very slow test suite (the waiting time is too long).

To deal with this problem we use an approach which is based on the fact that the test suite knows which state to expect. Information about this state can be used to check when the expected state loaded. Our approach uses a wait strategy that waits for important properties of the expected state to be loaded. For example, a strategy could be to wait for all the clickable elements and form elements in a state.

In this approach there is still a timeout needed because when an unexpected state is loaded, the test case could wait forever. However, this timeout can be relatively long because the test suite only waits this long when there is something wrong. As a consequence, when such a timeout occurs it can be considered as a test fault.

4.2.3 State Transitions

Our approach for robust navigation uses a mechanism (ElementResolver) that resolves the clickable elements which are used to make the state transitions. When inferring the model for the test suite, information about the clickables is saved. Each clickable element first has to be identified on the browsers DOM tree. If such a clickable is moved or changed, the XPath expression used initially will not be valid any longer. To detect the intended element persistently in each test run, we use various properties of the element such as attributes and the text value. Using a combination of these properties our element resolver searches the DOM for a match with a high level of certainty. When a clickable element is not found, it is considered as a fault.

Algorithm 2 shows the algorithm of our element resolver for resolving clickable elements. ElementResolver has as input the original element (line 1) which is used to find the equivalent element in the current state. The element, if found, with the same XPath expression as the original element is checked for equivalence (lines 2-4) and else other elements with the same tag name are checked for equivalence (lines 6-11). The procedure *isEquivalent* (lines 15-29) determines the equivalence of two elements by checking whether different element properties are equal. This is done by comparing the properties in a specific order: (1) text and attributes, (2) attributes, (3) id, and (4) text.

4.3 Implementation

We implemented the regression testing functionality in a plugin called *RegressionTester*. Conducting regression testing with this plugin consists of two phases:

Saving the crawl session. By running CRAWLJAX with the *SaveCrawlSessionPlugin* the inferred state-flow graph and crawled paths are saved to an XML file. We choose for an XML file instead of a database because the advantage of using this XML file is that (1) it is

Algorithm 2 Element Resolver

```
 1: procedure ResolveElement($originalElement$)
 2: $candidateElement \leftarrow$ getElememtByXpathInCurrentDom($originalElement$)
 3: if isEquivalent($originalElement$, $candidateElement$) then
 4:     return $candidateElement$
 5: else
 6:     for all $candidateElement \in$getElementsWithSameTagName($originalElement$) do
 7:         if isEquivalent($originalElement$, $candidateElement$) then
 8:             return $candidateElement$
 9:         end if
10:     end for
11:     return $null$
12: end if
13: end procedure
14:
15: procedure isEquivalent($originalElement$, $candidateElement$)
16: if attributesAndTextEqual($originalElement$, $candidateElement$) then
17:     return $true$
18: end if
19: if attributesEqual($originalElement$, $candidateElement$) then
20:     return $true$
21: end if
22: if elementIdEqual($originalElement$, $candidateElement$) then
23:     return $true$
24: end if
25: if textEqual($originalElement$, $candidateElement$) then
26:     return $true$
27: end if
28: return $false$
29: end procedure
```

portable, (2) it is easy to maintain different model versions of the web application, and (3) it does not need special requirements like a database server.

Running the regression tests. The XML file generated by the SaveCrawlSession-Plugin is used as input to reconstruct the crawled paths. RegressionTester replays all the crawled paths inferred by CRAWLJAX in the browser where the execution of every path represents a test case.

4.3.1 RegressionTester

Algorithm 3 shows the algorithm of the RegressionTester plugin. For each test case (lines 2-4), the browser is first put in the initial index state (start URL) of the web application (line 8). From there, values are entered (see Section 7.3) in the input elements if present (line 10) and events are fired on the clickables (line 11). Whether the clickable can be found and fired is tested in lines 12-15. After each event invocation, the resulting state in the browser is checked whether it is loaded (lines 16-20), checked against the invariants (Section 7.1.1) (lines 20-23) and compared with the expected state from the saved crawl session (lines 24-30).

A test case succeeds if and only if:

1. every event can be fired on the correct element.
2. there are no timeouts when loading states.

3. all the invariants are satisfied.
4. every visited state is equivalent with the expected state.

When one of the above conditions is not satisfied the test fails. Note that tests 2,3, and 4 are also tested for the initial state but left out to simplify Algorithm 3.

Ideally, for the unchanged application the test cases should all pass, and only for altered code, failures should occur, helping the tester to understand what has truly changed. In practice, many issues and challenges may arise, which were discussed in Section 3.3.

Algorithm 3 Regression Tester

```
1: procedure REGRESSIONTEST($crawlSession$)
2: for all $crawlPath$ IN $crawlSession$.getCrawledPaths() do
3:     testPath($crawlPath$)
4: end for
5: end procedure
6:
7: procedure TESTPATH($crawlPath$)
8: gotoInitialUrl()
9: for $clickable \in crawlPath$ do
10:     enterRelatedFormValues($clickable$)
11:     $fired \leftarrow$ fireEvent($clickable$)
12:     if $fired = false$ then
13:         handleEventFailure()
14:         return
15:     end if
16:     $timeout \leftarrow$ waitForInitialState()
17:     if $timeout$ then
18:         handleTimout()
19:     end if
20:     $invariantsSatisfied \leftarrow$ testInvariants()
21:     if $invariantsSatisfied = false$ then
22:         handleViolatedInvariants()
23:     end if
24:     $currentState \leftarrow browser$.getState()
25:     $expectedState \leftarrow clickable$.getTargetState()
26:     $equivalent \leftarrow$ statesNotEquivalent($currentState$, $expectedState$())
27:     if $equivalent = false$ then
28:         handleStateNotEquivalent()
29:         return
30:     end if
31: end for
32: end procedure
```

By default, the RegressionTester creates a JUnit TestSuite object, which can be used to run the regression test as a JUnit test suite (Figure 4.1), making it easy to integrate in existing test environments.

For flexible custom handling of the faults, e.g. adding faults to a report (Section 4.4), there is a *TestFaultHandler* interface. The user can implement this interface which has the methods, `onEventFailure()`, `onStateDifference()`, `onStateLoadTimeout()`, and `onInvariantViolation()`. The related method is called when there is a test fault, which gives the tester the flexibility to handle faults as he prefers.

To determine when a state is loaded, a custom *WaitStrategy* can be also be specified by the user. By default the regression tester waits the same amount of time as CRAWLJAX for a state to be loaded, but this could be flaky as discussed in Section 3.3. We therefore supplied

```
public final class RegressionTest extends TestCase {
...
  public static TestSuite suite() {
    SavedCrawSessionReader session = new SavedCrawlSessionReader(SESSION_XML);
    RegressionTester tester = new RegressionTester(session, getCrawlConfig());
    tester.run();
    return tester.getTestSuite();
  }
..
}
```

Figure 4.1: Example of a RegressionTester JUnit 3 test suite

an optional WaitStrategy that waits for all the anchor elements and form elements which were in the original state. However, the tester can also implement its own WaitStrategy.

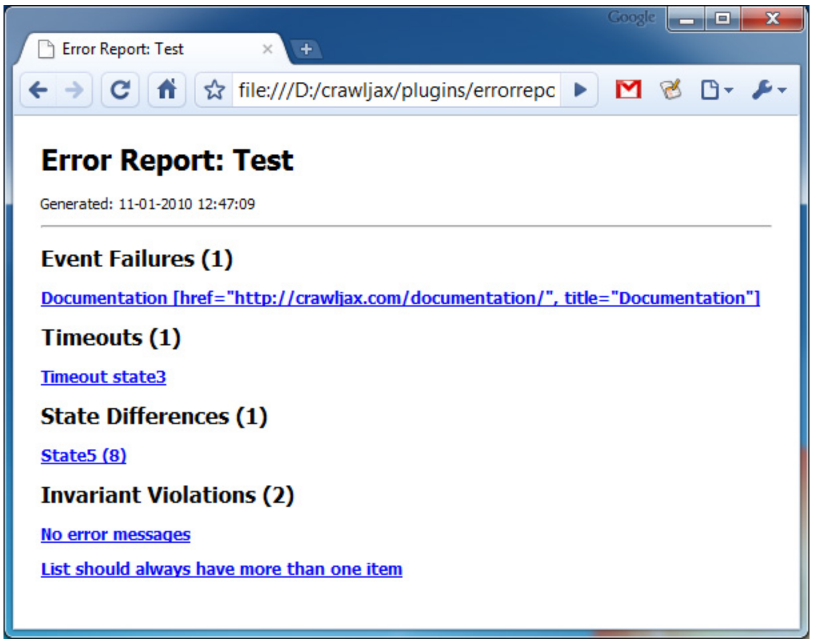
4.4 Visualizing Test Failures

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, we created a plugin called *ErrorReport* that generates a detailed web report that visualizes the failures. We format and pretty-print the DOM trees without changing their structure and use XMLUnit[1] to determine the DOM differences. The elements related to the differences/failure 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 and here the related elements are also highlighted. With this plugin, it is possible to view the stripped DOMs (done by the oracle comparators), but also the complete DOM trees to check whether differences are falsely ignored by the comparators. Other details such as the sequence of fired events (path to failure), and JAVASCRIPT debug variables are also displayed.

By default this plugin supports State Differences, Event Failures (an event cannot be fired) and Invariant Violations. However, it is also possible to add custom faults to the report.

Figure 4.2 shows the overview of an example error report, which lists the different faults by category. The user can click on a fault to see more information which is depicted in Figure 4.3. This figure shows a state difference fault with a snapshot of the browser at the moment of the failure with the differences highlighted (see also Figure 4.4). Figure 4.3 also shows the elements which were clicked before the fault occurred, thus in this case the links *Crawljax*, *Home*, and *Downloaded* where clicked (in this order) before the fault was detected. The current title by using a JAVASCRIPT expression is also displayed. One could view the DOM differences, by clicking on the *View DOMs* link, as depicted in Figure 4.5. The DOM differences are highlighted with each a color and a corresponding description.

[1] `http://xmlunit.sourceforge.net/`



Error Report: Test

Generated: 11-01-2010 12:47:09

Event Failures (1)

Documentation [href="http://crawljax.com/documentation/", title="Documentation"]

Timeouts (1)

Timeout state3

State Differences (1)

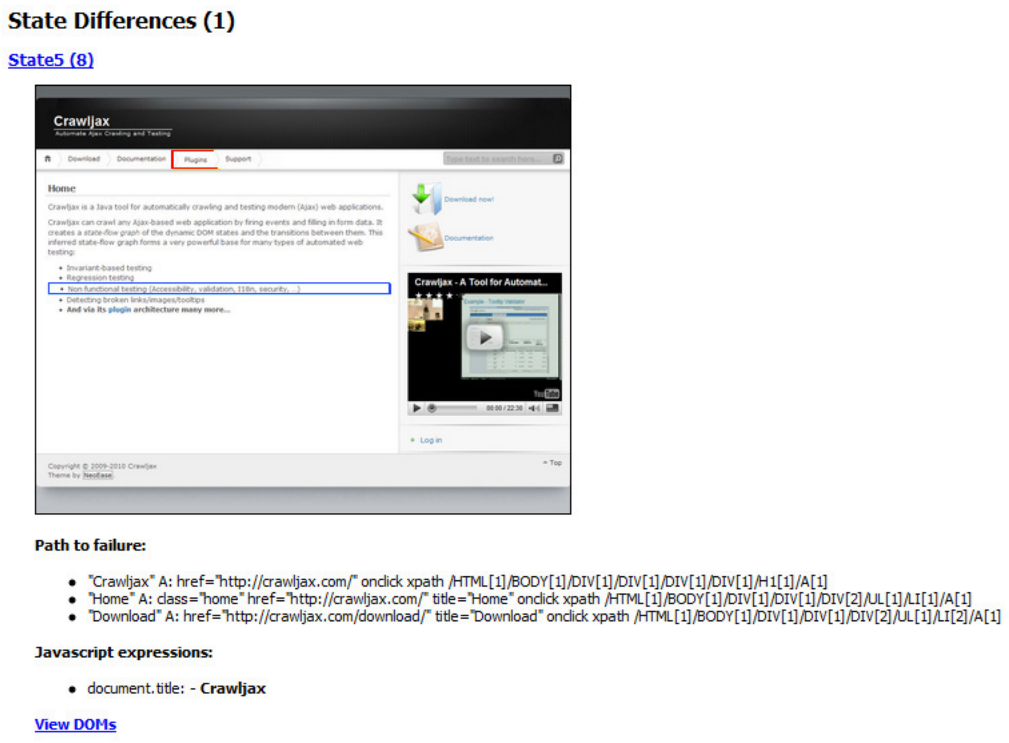
State5 (8)

Invariant Violations (2)

No error messages

List should always have more than one item

Figure 4.2: ErrorReport - Overview



State Differences (1)

State5 (8)

Path to failure:

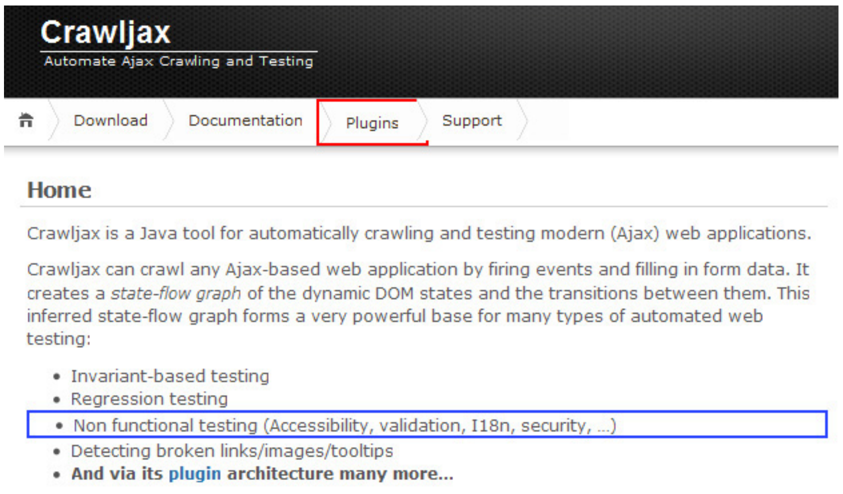
- "Crawljax" A: href="http://crawljax.com/" onclick xpath /HTML[1]/BODY[1]/DIV[1]/DIV[1]/DIV[1]/DIV[1]/H1[1]/A[1]
- "Home" A: class="home" href="http://crawljax.com/" title="Home" onclick xpath /HTML[1]/BODY[1]/DIV[1]/DIV[1]/DIV[2]/UL[1]/LI[1]/A[1]
- "Download" A: href="http://crawljax.com/download/" title="Download" onclick xpath /HTML[1]/BODY[1]/DIV[1]/DIV[1]/DIV[2]/UL[1]/LI[2]/A[1]

Javascript expressions:

- document.title: - **Crawljax**

View DOMs

Figure 4.3: ErrorReport - Details



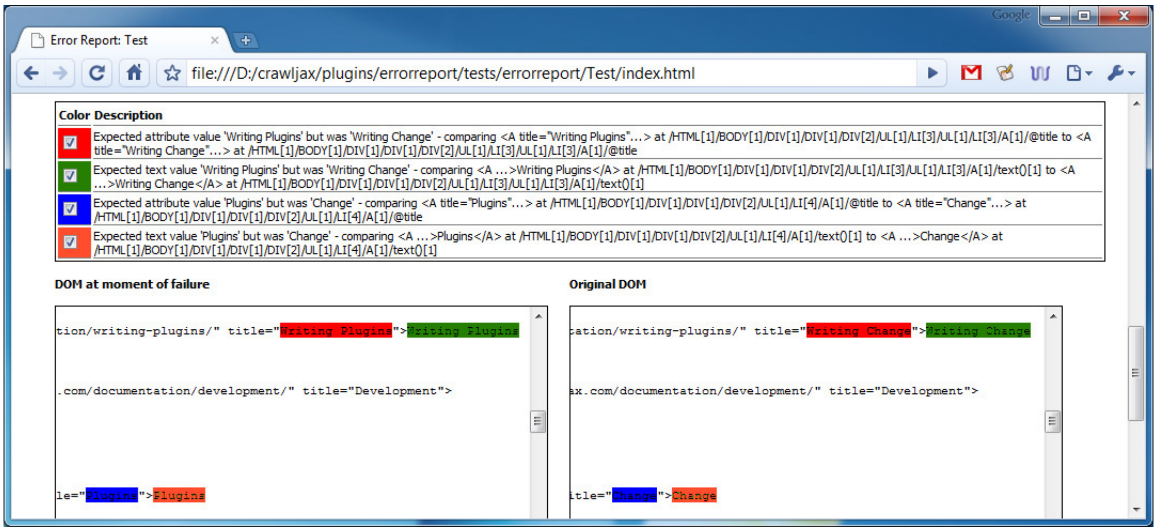
Home

Crawljax is a Java tool for automatically crawling and testing modern (Ajax) web applications.

Crawljax can crawl any Ajax-based web application by firing events and filling in form data. It creates a *state-flow graph* of the dynamic DOM states and the transitions between them. This inferred state-flow graph forms a very powerful base for many types of automated web testing:

- Invariant-based testing
- Regression testing
- Non functional testing (Accessibility, validation, I18n, security, ...)
- Detecting broken links/images/tooltips
- **And via its plugin architecture many more...**

Figure 4.4: ErrorReport - Highlighted elements


Error Report: Test
file:///D:/crawljax/plugins/errorreport/tests/errorreport/Test/index.html

Color	Description
🟥 ✓	Expected attribute value 'Writing Plugins' but was 'Writing Change' - comparing <A title="Writing Plugins"...> at /HTML[1]/BODY[1]/DIV[1]/DIV[1]/DIV[2]/UL[1]/LI[3]/UL[1]/LI[3]/A[1]/@title to <A title="Writing Change"...> at /HTML[1]/BODY[1]/DIV[1]/DIV[1]/DIV[2]/UL[1]/LI[3]/UL[1]/LI[3]/A[1]/@title
🟩 ✓	Expected text value 'Writing Plugins' but was 'Writing Change' - comparing <A ...>Writing Plugins</A> at /HTML[1]/BODY[1]/DIV[1]/DIV[1]/DIV[2]/UL[1]/LI[3]/UL[1]/LI[3]/A[1]/text()[1] to <A ...>Writing Change</A> at /HTML[1]/BODY[1]/DIV[1]/DIV[1]/DIV[2]/UL[1]/LI[3]/UL[1]/LI[3]/A[1]/text()[1]
🟦 ✓	Expected attribute value 'Plugins' but was 'Change' - comparing <A title="Plugins"...> at /HTML[1]/BODY[1]/DIV[1]/DIV[1]/DIV[2]/UL[1]/LI[4]/A[1]/@title to <A title="Change"...> at /HTML[1]/BODY[1]/DIV[1]/DIV[1]/DIV[2]/UL[1]/LI[4]/A[1]/@title
🟧 ✓	Expected text value 'Plugins' but was 'Change' - comparing <A ...>Plugins</A> at /HTML[1]/BODY[1]/DIV[1]/DIV[1]/DIV[2]/UL[1]/LI[4]/A[1]/text()[1] to <A ...>Change</A> at /HTML[1]/BODY[1]/DIV[1]/DIV[1]/DIV[2]/UL[1]/LI[4]/A[1]/text()[1]

DOM at moment of failure

```
tion/writing-plugins/" title="Writing Plugins">Writing Plugins

.com/documentation/development/" title="Development">

le="Plugins">Plugins
```

Original DOM

```
ation/writing-plugins/" title="Writing Change">Writing Change

x.com/documentation/development/" title="Development">

itle="Change">Change
```

Figure 4.5: ErrorReport - DOM differences

Chapter 5

Determining State Equivalence

An important aspect of our regression test suite (Chapter 4) is comparing states to detect changes in the web application. As discussed in Section 3.3, comparing states is a challenging task. This chapter discusses our approach for comparing states and how state equivalence determined.

5.1 Main Idea

To determine state equivalence, we use *Oracle Comparators* which unambiguously determine whether two states are equivalent. Oracle comparators in the form of a *diff* method between the current and the expected state are usually used [28, 30] to handle nondeterministic behavior in web applications. When there are differences between states, for every difference needs to determined whether it is a fault or a legal change. A disadvantage of using a diff based comparator is that there are often many differences which are legal changes, resulting in falsely reported faults.

In this research we propose an approach where the oracle comparators filters the differences that are legal changes and then compare the filtered version. Every comparator is specialized in a type of difference and since there are different types of differences, the oracle comparators cooperate to combine their knowledge. When filtering, one has to be careful that valuable information is not filtered since it could lead to missing faults. We therefore use an technique in which for every state for every comparator is checked whether it should be applied or not.

5.2 Approach

This section discusses our approach for comparing states.

5.2.1 Oracle Comparators

In our approach, each comparator is exclusively specialized in comparing the targeted differences from the DOM trees by first stripping the differences, defined through e.g., a regular

```
<P class="past">Last update: 06-01-2010 12:50</P>
```

```
<P class="">Last update: 06-01-2010 12:50</P>
```

Figure 5.1: Example of stripping by an attribute oracle comparator.

```
<P class="past">Last update: 06-01-2010 12:50</P>
```

```
<P class="past">Last update: </P>
```

Figure 5.2: Example of stripping by a date oracle comparator.

```
preCondition = new RegexCondition("<H1>Todo items</H1>");
attributeComparator = new AttributeComparator("class");
oracleComparator = new OracleComparator(attributeOracle, preCondition);
```

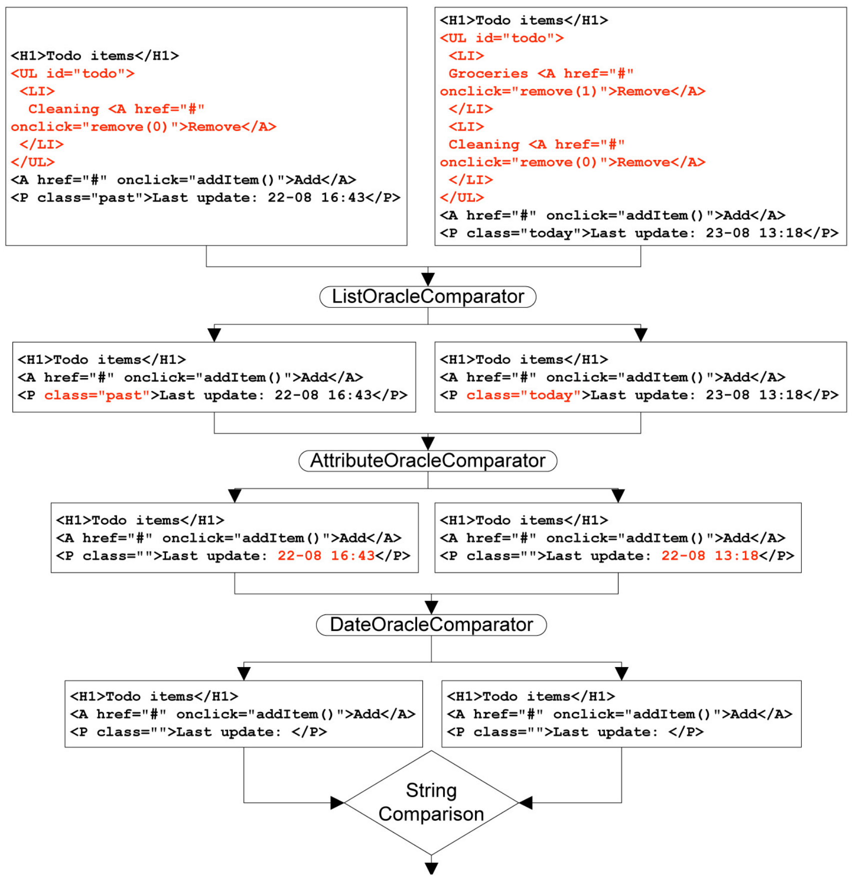
Figure 5.3: An attribute oracle comparator with a precondition.

expression or an XPath expression, and then comparing the stripped DOMs with a simple string comparison. In this research the inputs for the oracle comparators are the original DOM and the current DOM. Figure 5.1 and Figure 5.2 show examples how the comparators filter their targeted differences.

5.2.2 Oracle Comparator Preconditions

Applying these generic comparators can be very effective in ignoring nondeterministic real-time differences, but at the same time, they can have the side-effect of missing real bugs because they are too broad. To tackle this problem, we adopt *preconditions*, which are conditions in terms of boolean predicates (see Section 7.1), to check whether an oracle comparator should be applied on a given state. A comparator can have zero, one, or more preconditions, which all must be satisfied before the comparator is applied.

Going back to our TODO DOM fragments in Figure 3.3, one could use an `Attribute` comparator to ignore the `class` attribute differences of the `<p>` element when comparing the states. Since this combination is quite generic, we certainly do not wish to ignore all the occurrences of an element with a `class` attribute in this application. To that end, we combine the `Attribute` comparator with a precondition that checks the existence of a pattern, e.g., `<H1>Todo items</H1>`, before applying the comparator on each state, as shown in Figure 5.3.



```
<H1>Todo items</H1>
<UL id="todo">
 <LI>
  Cleaning <A href="#"
onclick="remove(0)">Remove</A>
 </LI>
</UL>
<A href="#" onclick="addItem()">Add</A>
<P class="past">Last update: 22-08 16:43</P>
```

```
<H1>Todo items</H1>
<UL id="todo">
 <LI>
 Groceries <A href="#"
onclick="remove(1)">Remove</A>
 </LI>
 <LI>
 Cleaning <A href="#"
onclick="remove(0)">Remove</A>
 </LI>
</UL>
<A href="#" onclick="addItem()">Add</A>
<P class="today">Last update: 23-08 13:18</P>
```

```
<H1>Todo items</H1>
<A href="#" onclick="addItem()">Add</A>
<P class="past">Last update: 22-08 16:43</P>
```

```
<H1>Todo items</H1>
<A href="#" onclick="addItem()">Add</A>
<P class="today">Last update: 23-08 13:18</P>
```

```
<H1>Todo items</H1>
<A href="#" onclick="addItem()">Add</A>
<P class="">Last update: 22-08 16:43</P>
```

```
<H1>Todo items</H1>
<A href="#" onclick="addItem()">Add</A>
<P class="">Last update: 22-08 13:18</P>
```

```
<H1>Todo items</H1>
<A href="#" onclick="addItem()">Add</A>
<P class="">Last update: </P>
```

```
<H1>Todo items</H1>
<A href="#" onclick="addItem()">Add</A>
<P class="">Last update: </P>
```

Figure 5.4: Oracle Comparator Pipelining for TODO.

5.2.3 Oracle Comparator Pipelining

When comparing DOM states, there are often multiple types of differences, each requiring a specific type of oracle comparator. As can be seen in Figure 5.1 and Figure 5.2, there would still be differences after stripping when only one of the comparators is applied to Figure 3.3, thus the states would not be determined equivalent with a string comparison.

In our approach we combine different comparators in a technique we call *Oracle Comparator Pipelining* (OCP). Oracle comparators can strip their targeted differences from the DOM and the idea is to pass this stripped output as input to the next comparator. When all differences are stripped, a simple and fast string comparison can be used to test whether two states are equivalent. Figure 5.4 shows an example of how two HTML fragments are compared through a set of pipelined comparators. The *ListOracleComparator* is a special type of oracle comparator which will be discussed in Chapter 6

```java
public class BreaklinesComparator extends AbstractComparator {

  private void StripBreaklines() {
    //strip the breaklines from the DOMs
  }

  @Override
  pulic boolean isEquivalent() {
    stripBreaklines();
    return super.compare();
  }
}
```

Figure 5.5: Example of a custom oracle comparator implementation.

5.3 Implementation

This section discusses how we implemented our approach for comparing states and determining state equivalence.

5.3.1 Oracle Comparators

The main interface for the comparators, *Comparator*, is implemented by *AbstractComparator* which contains the methods for setting and getting the (stripped) DOMs. Every comparator must implement the `isEquivalent()` method which returns true if and only if the two DOMs are equivalent. Figure 5.5 shows an example of an oracle comparator that ignores `BR` elements.

5.3.2 Comparing States

For testing, the *OracleComparator* class is used to specify the comparator and its preconditions (Figure 5.3). OracleComparators are added to the *StateComparator*, which (1) combines the different oracles comparators, (2) checks the specified preconditions, (3) and determines whether the states are equivalent.

A comparator is only applied when it has no preconditions or all its preconditions are satisfied. Algorithm 4 shows the idea of StateComparator.

5.3.3 Generic Oracle Comparators

We implemented the following generic comparators which are supplied with CRAWLJAX.

DateComparator Ignores dates and times, e.g. ”2009-01-23”, April 19, 2007”, ”16:15”, and ”10:35:04 am”.

StyleComparator Ignores style elements like `<b>`, `<i>`, `<u>`, `<strong>` and style attributes like `align`, `bgcolor`, and `height`. It also ignores properties of the `style` attribute, but not the style properties `display` and `visibility`, since these are important in general.

Algorithm 4 OracleComparator

```
1:  procedure isEquivalent($originalDom$, $testDom$)
2:  $originalDomPipelined \leftarrow originalDom$
3:  $testDomPipelined \leftarrow testDom$
4:  for all $oracleComparator \in oracleComparators$ do
5:      if allPreConditionsSatisfied($oracleComparator$, $testDom$) then
6:          $comparator \leftarrow oracleComparator$.getComparator()
7:          $equivalent \leftarrow comparator$.isEquivalent($originalDomPipelined$, $testDomPipelined$)
8:          if $equivalent$ then
9:              return $true$
10:         end if
11:         $originalDomPipelined \leftarrow comparator$.getStrippedOriginalDom()
12:         $testDomPipelined \leftarrow comparator$.getStrippedTestDom()
13:     end if
14: end for
15: return $false$
16: end procedure
```

AttributeComparator Ignores the specified attributes like `class` attributes or random attributes.

PlainStructureComparator Only compares the structure by ignoring attributes and textual content.

EditDistanceComparator Compares states by using the Levenshtein edit distance [11]. Note that this comparator does not strip anything.

ScriptComparator Ignores all the scripts in `SCRIPT` elements.

RegexComparator Ignores the specified regular expression, which could be, for example, used to ignore ip-addresses, e.g. `[0-9]{1,3}(\.[0-9]{1,3}){3}`.

XPathComparator Ignores the specified XPath expressions, which could be used to ignore certain elements/attributes, e.g. `//DIV[class='notInteresting']`.

Other powerful comparators that support structures like tables and lists are discussed in Chapter 6.

Chapter 6

Capturing Structures with Templates

This chapter discusses the use of templates for modeling structures to improve the testing of web applications.

6.1 Main Idea

Whereas textual differences caused by a changed state in the back-end can relatively easy be supported, differences in structures are more difficult to capture.

For example the structure of the to-do list page (Figure 3.3) changes each time a test case adds a new to-do item. Web applications often contain many structures with repeating patterns such as *tables* and *lists* in which these state changes are manifested. As mentioned, an example is the to-do list in Figure 3.3, but one could think of other examples like tables/lists that display a repeating pattern of items like news messages, menus, topics, navigation elements and more.

In this research we model these structures as templates, which can be used to compare states. When there are differences between states we extract the templates of these structures, check whether both the states match this template, and when they match, that part can be considered equivalent. With this approach the comparators in the test cases can ignore differences in structures which are legal changes.

6.2 Approach

This section discusses how we extract the templates and how these templates can be applied.

6.2.1 Extracting the Templates

To support the structural differences, our technique scans the DOM tree for elements with a recurring pattern and automatically generates a template that captures the pattern. The templates are defined through a combination of HTML elements and regular expressions.

Algorithm 5 shows our approach for extracting the templates. In the specified state (line 1) the related HTML (line 4) is searched for known structures (line 6) and for each

```
<UL id="todo">
  <LI>
    Groceries <A href="#" onclick="remove(1)">Remove</A>
  </LI>
  <LI>
    Cleaning <A href="#" onclick="remove(0)">Remove</A>
  </LI>
</UL>
```

```
<UL id="todo">([^<]*
  <LI>[^<]*
    <A href="[^"]*" onclick="[^"]*">[^<]*</A>[^<]*
  </LI>[^<]*)*
</UL>
```

Figure 6.1: Generated template for the TODO list.

structure is checked whether it contains embedded structures (line 8), e.g. a list in a table. The retrieved embedded structures are replaced by temporary strings (line 9) to make sure that the extracted template (line 10) does not capture these embedded structures as normal content. After the template is generated, the embedded templates are restored in the current template (line 11). A template is only added to the template list when the template does not already exists (lines 12-14). For the different structure types there are different templates (line 20-21).

Algorithm 5 Extracting templates from states.

```
1: procedure FINDTEMPLATESINSTATE($state$)
2: return searchTemplates($state$.getHtml(), new List())
3: end procedure
4:
5: procedure SEARCHTEMPLATES($html$, $listTemplates$)
6: $listStructures$ ← getStructures($html$)
7: for $structure \in listStructures$ do
8:     $embeddedTemplates$ ←searchTemplates($structure$, $listTemplates$)
9:     $structure$ ←replaceTemplatesToTemporaryString($structure$, $embeddedTemplates$)
10:    $template$ ←getTemplate($structure$)
11:    $template$ ←restoreTemplates($template$, $embeddedTemplates$)
12:    if $template \ni listTemplates$ then
13:        $listTemplates$.addToHead($template$)
14:    end if
15: end for
16: return $listTemplates$
17: end procedure
18:
19: procedure GETTEMPLATE($structure$)
20: $elementType$ ← getElementType($structure$)
21: return getTemplateForElementType($elementType$, $structure$)
22: end procedure
```

Figure 6.1 depicts the generated template for the list pattern of Figure 3.3. Note that in the template newlines and white spaces are inserted for clarity.

```
<UL id="todo">
  ([\s]*<LI>[\s]+
    [a-zA-Z0-9 ]{2,100}<A href="#" onclick="remove([0-9]+)">Remove</A>[\s]*
  </LI>[\s]*)*
</UL>
```

Figure 6.2: An augmented template that can be used as an invariant.

```
<A href="#" onclick="addItem()">Add</A>
<P class="recentUpdate">Last update: 06-01-2010 12:50</P>
```

```
<A href="[^"]*" onclick="[^"]*">[^<]*</A>[^<]*
<P class="[^"]*">[^<]*</P>
```

Figure 6.3: Example of a generic extracted template

6.2.2 Applications of the Templates

There are different applications for the templates.

Oracle Comparators. The oracle comparators can use these templates by stripping the matching templates from the DOM, thereby supporting structural differences and improving the detection of legal changes.

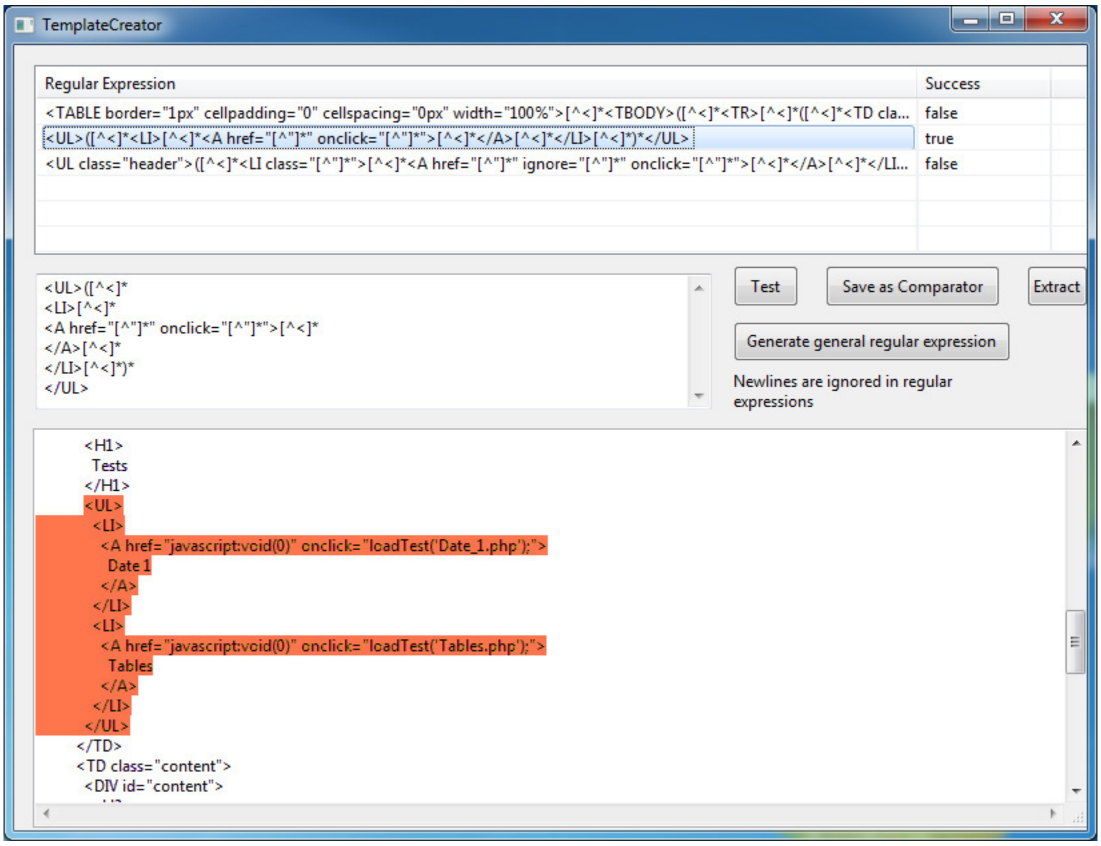
State Reduction. By supporting repeating patterns, more similar states can be determined equivalent, and as a consequence the state space for regression testing can be reduced. For example *state1* and *state3* in Figure 3.2 can now be seen as one state. State reduction can significantly improve the performance of CRAWLJAX.

Invariants. The generated templates can also be augmented manually and used as invariants on the DOM tree. Figure 6.2 shows an invariant template that checks the structure of the list, whether the to-do item is between 2 and 100 alpha numeric characters, and if an item's identifier is always an integer value.

6.3 Implementation

Extracting the templates as shown in Algorithm 5 is done with a plugin called *TemplateExtractor* which supports tables (`TABLE`), lists (`UL`, `OL`, `DL`), and drop-down boxes (`SELECT`). For every type of structure we created a comparator (like the ListOracleComparator in Figure 5.4), and we created one comparator, *StructureOracleComparator*, that supports all the structures of TemplateExtractor. With TemplateExtractor it is also possible to generate a generic template of a HTML fragment as shown in Figure 6.3.

To assist the tester, an SWT-based GUI (Figure 6.4) is implemented for creating templates. Through this user interface, the templates can automatically be extracted and generated by loading a state or by selecting the desired parts of a DOM tree. These templates



Regular Expression	Success
<TABLE border="1px" cellpadding="0" cellspacing="0px" width="100%">[^<]*<TBODY>([^<]*<TR>[^<]*([^<]*<TD cla...	false
<UL>([^<]*<LI>[^<]*<A href="[^"]*" onclick="[^"]*">[^<]*</A>[^<]*</LI>[^<]*)*</UL>	true
<UL class="header">([^<]*<LI class="[^"]*">[^<]*<A href="[^"]*" ignore="[^"]*" onclick="[^"]*">[^<]*</A>[^<]*</LI...	false

```
<UL>([^<]*
<LI>[^<]*
<A href="[^"]*" onclick="[^"]*">[^<]*
</A>[^<]*
</LI>[^<]*)*
</UL>
```

Newlines are ignored in regular expressions

```
      <H1>
       Tests
      </H1>
      <UL>
       <LI>
        <A href="javascript:void(0)" onclick="loadTest('Date_1.php');">
         Date 1
        </A>
       </LI>
       <LI>
        <A href="javascript:void(0)" onclick="loadTest('Tables.php');">
         Tables
        </A>
       </LI>
      </UL>
     </TD>
     <TD class="content">
      <DIV id="content">
```

Figure 6.4: Screenshot of the GUI for creating custom templates.

can easily be customized to use as invariants or to capture more details. Via highlighting can be tested whether the templates model the structures in the DOM correctly. From the templates, oracle comparators (in Java) can also be generated automatically and added to the pipelining mechanism.

Chapter 7

Crawljax Improvements

This chapter discusses the various improvement made to CRAWLJAX during this research.

7.1 Conditions

In this research we propose several approaches that make decisions based on the current state. We implemented *Conditions* which are boolean predicates that are tested against a state. These conditions are used by the oracle comparators (Section 5.2.2), invariants (Section 7.1.1), and crawling conditions (Section 7.1.2). We have created several generic types of conditions (Table 7.1) that can be used for different purposes. For all conditions the logical operators NOT, OR, AND, and NAND can be applied for more flexibility. The conditions make it very flexible to test state properties.

7.1.1 Invariants

In [22] the authors propose a method for invariant-based testing, where the invariants are boolean predicates that should always hold in every state. In their approach invariants could only be expressed with XPath expressions and added as plugins.

We propose a approach where the invariants are defined with the conditions discussed in Section 7.1, which makes defining invariants flexible and more user-friendly. Another new possibility is defining preconditions for the invariants, since it could be possible that an invariant should always hold, except for certain states. Figure 7.1 and Figure 7.2 show

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

Table 7.1: Generic conditions types.

```
//there should be no error messages
crawler.addInvariant("No error messages", new NotRegexCondition("Error [0-9]+");

//a menu on the home page should always have menu elements.
String xpath = "//DIV[@id='menu']/UL/LI[contains(@class, 'menuElement')]";
Condition preCondition = new JavaScriptCondition("document.title=='Home'");
crawler.addInvariant("Menu should always have menu elements", new XPathCondition(
    xpath, preCondition);
```

Figure 7.1: Example of basic invariants.

```
crawler.addInvariant("MyList should always have more than one item", new
    AbstractCondition(){

  @Override
    public boolean check(EmbeddedBrowser browser) {
      WebDriver driver = ((AbstractWebDriver)browser).getDriver();
      try{
        WebElement myList = driver.findElement(By.id("MyList"));
        return new Select(myList).getOptions().size() > 0;
      }catch(NoSuchElementException e){
        //not found, no violation
        return true;
      }
    }
});
```

Figure 7.2: Example of a custom invariant.

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

Figure 7.3: An url crawl condition for only crawling the CRAWLJAX web page.

examples of creating invariants. For regression testing invariants are applicable since invariants that hold in the gold standard should also hold in newer versions of the application.

7.1.2 Crawling Conditions

Another example of an application of the conditions are crawling conditions.

The nondeterministic behavior of AJAX applications can affect the crawling phase as well, resulting in crawling unwanted states and generating obsolete test cases. 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 satisfied or no crawl condition is specified for that state. Figure 7.3 shows an example of a crawl condition that enforces that only the CRAWLJAX web page is crawled.

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

Figure 7.4: Example of configuring clickable elements with the builder pattern.

```
CrawlSpecification crawler = new CrawlSpecification(URL);

//click all the anchor elements and buttons
crawler.clickDefaultElements();

//also click DIV elements which are clickable
crawler.click("div").withAttribute("class", "clickable");

//exclude these elements
crawler.dontClick("button").withText("Logout");
crawler.dontClick("a").underXPath("//DIV[@id='header']");
crawler.dontClick("a").underXPath("//DIV[@id='footer']");

CrawljaxConfiguration config = new CrawljaxConfiguration();
config.addPlugin(new CrawlOverview());
config.setCrawlSpecification(crawler);

CrawljaxController crawljax = new CrawljaxController(config);
try {
  crawljax.run();
} catch (CrawljaxException e) {
  e.printStackTrace();
}
```

Figure 7.5: Example of configuring CRAWLJAX with the new API.

7.2 Configuration API

In previous versions of CRAWLJAX the configuration was done with properties files, but this approach is not sufficient anymore to handle all the new features. We therefore implemented an additional new *Java* based API for configuring CRAWLJAX in which all the options can be configured. The API is designed to make configuring CRAWLJAX easy and without unnecessary code/effort.

In the main class, *CrawljaxConfiguration*, general settings like the browser, the output folder, and the plugins can be configured. Configuring the crawling properties, like which elements (not) to click, the maximum crawl depth, and input values (Section 7.3), is done with the *CrawlSpecification* class. For the most important aspect of guiding the crawling, configuring which elements should be clicked, we used the *Builder Pattern* as depicted in Figure 7.4. Figure 7.5 shows an example for configuring CRAWLJAX.

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

Table 7.2: Random values for form input elements.

7.3 Form Handling

AJAX web applications often contain many form input fields like search fields and contact forms. Certain states can only be reached when values are entered in the related form elements, thus supplying these fields with values is needed to reach high state coverage (see Section 9.2).

We implemented a whole new form handling mechanism which supports all the different types of form elements. While crawling, before CRAWLJAX clicks an element it checks the DOM for input elements and enters the corresponding values for the found elements. The related input fields are saved with the clickable elements, so that CRAWLJAX knows which values to enter in which fields the next time it clicks on the elements (while backtracking). For supplying values in the input fields our approach considers three categories.

Random Input Values. Automation is an important aspect of CRAWLJAX and therefore our approach enters random values in the form elements by default. With this approach, many states that need input values can be reached without human effort. Table 7.3 shows the random values CRAWLJAX enters. When there is already text in a text field, this is replaced by a random value since this results in higher coverage according to our case study in Section 9.2.

Manual Input Values. Specific values are often needed for testing or to reach certain states. For example, in our coverage case study (Section 9.2) an valid url was needed as input to reach good code coverage. With our approach it is possible to specify the values for input elements by their *id* or *name* attribute as shown in Figure 7.6. Every time CRAWLJAX encounters these known elements, the specified values are entered.

Multiple Manual 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 interesting.

We implemented an approach in which multiple values can be specified for an input field. The problem is to know when to enter which value, which is tackled by grouping the input elements in a *Form* and by specifying the related clickable element (submit button) for this form. The submit button is clicked n times by CRAWLJAX where n is the most assigned values to an input element in the form. When $n = 0$ the first values are entered, when $n = 1$ the second values are entered, when $n = 3 \ldots$. If $n >$ the number of assigned values to an input element, its first specified value is used.

```
InputSpecification input = new InputSpecification();
//always enter the value Crawljax in the subject field
input.field("subject").setValue("Crawljax");
//always check the newsletter checkbox
input.field("newsletter").setValue(true);

Form contactForm = new Form();
//set two values in the name and e-mail field
contactForm.field("name").setValues("Danny", "Ali");
contactForm.field("email").setValues("dannyroest@gmail.com",
                                     "a.mesbah@tudelft.nl");
//always set the male field to true
contactForm.field("male").setValues(true);
//before click the save button, enter the specified values.
input.setValuesInForm(contactForm).beforeClickElement("button").withText("Save");
```

Figure 7.6: Example of setting the input values for the form elements with the new API.

Figure 7.6 shows an example of specifying manual values for input elements. The *Save* button is clicked twice in this example.

7.4 Plugins

One of the things that make CRAWLJAX powerful is its support for plugins, which is discussed in this section.

7.4.1 Plugin Framework

In the previous version of CRAWLJAX the *Java Plugin Framework* (JPF) was used for supporting plugins. However, this was not very user-friendly and portable. We therefore created a new plugin framework that is clean and easy to use.

The main interface is *Plugin*, which is implemented by the different types of plugins, e.g. before the crawling, when a new state is found, and after the crawling. Each plugin type serves as an extension point that is called in a different phase of the CRAWLJAX execution. An example is the *RegressionTester* plugin discussed in Section 4.3 and Figure 7.7 shows a code example of a *OnNewStatePlugin* that generates a mirror of the AJAX application. Other examples of plugins are discussed in the next sections.

7.4.2 CrawlOverview

It is important to know and see what CRAWLJAX has crawled to make sure your application is correctly crawled. Let the user watch the whole crawl session is not practical. We therefore created a plugin called *CrawlOverview* which visualizes what is crawled by CRAWLJAX. It does this by generating a report which contains a screenshot of every state. The elements selected by CRAWLJAX to click are highlighted (1) green when an element caused a state transition, (2) blue when it linked to a previously visited state, and (3) orange

```
public class MirrorGenerator implements OnNewStatePlugin {

  @Override
  public void onNewState(CrawlSession session) {
    try {
      String dom = session.getBrowser().getDom();
      String fileName = session.getCurrentState().getName() + ".html";

      FileWriter fw = new FileWriter(fileName, false);
      fw.write(dom);
      fw.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Figure 7.7: Example of a plugin that generates a mirror of a web application.

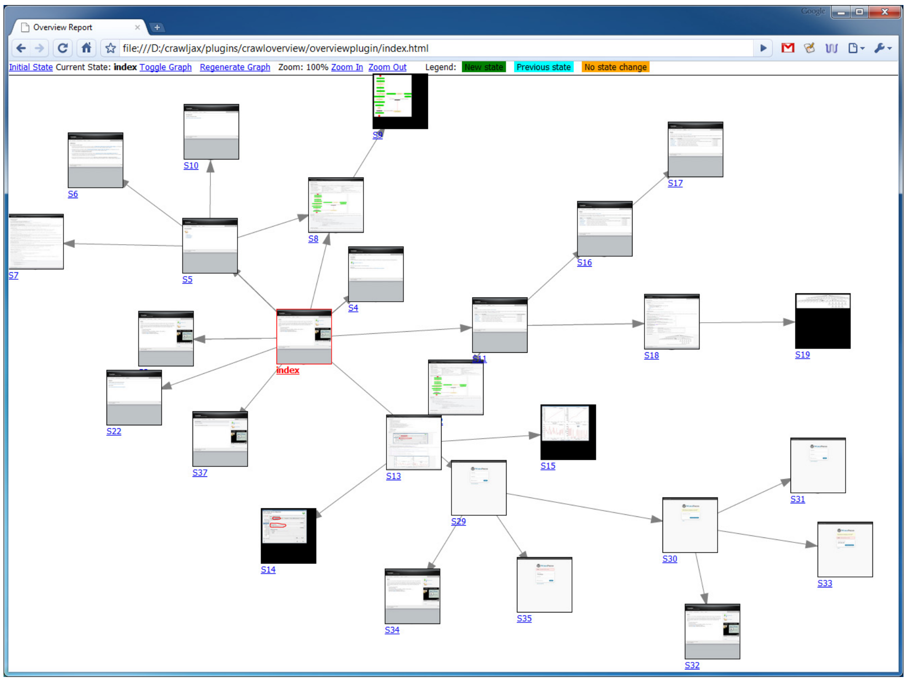
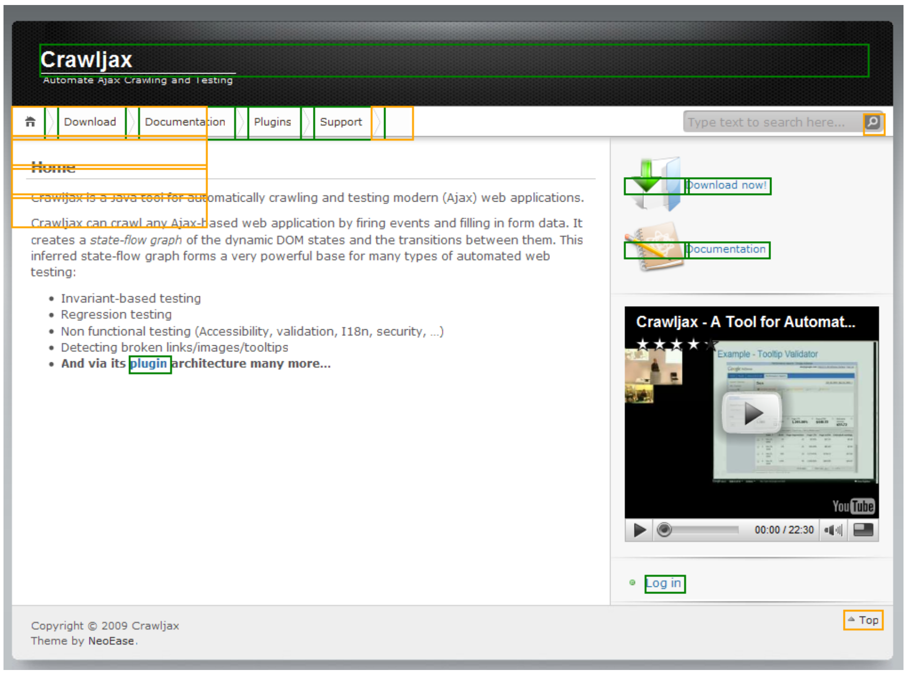

Overview Report
file:///D:/crawljax/plugins/crawloverview/overviewplugin/index.html
Initial State Current State: index Toggle Graph Regenerate Graph Zoom: 100% Zoom In Zoom Out Legend: New state Previous state No state change
S9
S10
S6
S17
S8
S16
S7
S5
S4
index
S11
S18
S19
S22
S37
S13
S15
S29
S31
S14
S30
S33
S34
S35
S32

Figure 7.8: CrawlOverview - State-flow graph

when the element caused no state transition. The report also contains the whole inferred state-flow graph by CRAWLJAX in which the user can see the navigated paths. The user can click trough the different states with the highlighted elements or via the state-flow graph. Figure 7.9 and Figure 7.8 show screenshots of the report generated by the CrawlOverview plugin.


Crawljax
Automate Ajax Crawling and Testing
Download
Documentation
Plugins
Support
Type text to search here...
Download now!
Documentation
Crawljax - A Tool for Automat...
Example - Tooltip Validator
00:00 / 22:30
Log in
Top

Home

Crawljax is a Java tool for automatically crawling and testing modern (Ajax) web applications.

Crawljax can crawl any Ajax-based web application by firing events and filling in form data. It creates a *state-flow graph* of the dynamic DOM states and the transitions between them. This inferred state-flow graph forms a very powerful base for many types of automated web testing:

- Invariant-based testing
- Regression testing
- Non functional testing (Accessibility, validation, I18n, security, ...)
- Detecting broken links/images/tooltips
- **And via its plugin architecture many more...**

Copyright © 2009 Crawljax
Theme by NeoEase.

Figure 7.9: CrawlOverview - Candidate elements

7.4.3 CrossBrowserTester

An occurring problem with AJAX applications is that some functions work in one browser but not in another browser. We therefore created a cross-browser plugin, *CrossBrowserTester* which crawls in one browser and replays the session in other browsers. For example, with this plugin broken links and errors can be detected. CrossBrowserTester uses the same approach as RegressionTester in Algorithm 3, but is immediately executed after the crawling by CRAWLJAX and requires one or more browsers as input.

The visited states cannot be compared with the generic oracle comparators discussed in Section 5.3.3 since the DOMs have large structure differences because of the different rendering of the browsers. We therefore propose an approach that compares states based on important elements. An example of such an oracle comparator would determine state equivalence by comparing important elements such as anchor elements, form elements, and elements with identifiers. An oracle comparator like this one has been successfully applied while performing case studies at Google London (Chapter 9).

7.5 Various

This section discusses some other smaller improvements for CRAWLJAX made in this research.

```
crawler.waitFor("info.php", 2000,
                new ExpectedVisibleCondition(By.id("slow_widget")));
```

Figure 7.10: Example of a waiting condition.

7.5.1 Waiting Conditions

While performing the case studies, a couple of widgets loaded slow which caused CRAWLJAX to miss these widgets, because CRAWLJAX waits a specified time after loading the page or firing an event. As a consequence CRAWLJAX did not explore important states and there were problems with backtracking.

To solve this problem we implemented *WaitConditions* which can be used to instruct CRAWLJAX that it should wait for certain elements/widgets on specified URLs. The main two types are: (1) checking whether an element is visible, and (2) checking whether an element exists in the DOM tree. This approach solved the problems for the slow loading widgets. Figure 7.10 shows an example for a wait condition that waits for a widget.

7.5.2 Performance

Crawling the large AJAX applications GOOGLE READER and ASFE 3.0 exposed some performance issues. The first issue was the ineffective backtracking of CRAWLJAX. We improved this, which decreased the crawling time up to 40%.

Currently, CRAWLJAX also uses the oracle comparators for detecting new and visited states while crawling, which can reduce the state space significantly. There was an issue with the large GWT generated DOMs that caused the state comparisons states be too slow (up to three seconds) and therefore we implemented a caching mechanism (via hashing) for state comparisons. The effect was that CRAWLJAX crawled more smoothly and the crawling time was significantly reduced.

7.5.3 User Friendliness

We improved CRAWLJAX by making it more user friendly and easier to use. One step was removing the requirement of using a database. Several interfaces are developed for easily creating oracle comparators, invariants, and templates. We also created some examples/guides that explain how to use CRAWLJAX.

Chapter 8

Empirical Evaluation

To assess the usefulness and effectiveness of our approach, we conducted two case studies following Yin's guidelines [34]. Our evaluation addresses the following research questions:

RQ1 How effective is our regression testing approach?
RQ2 How much manual effort is required in the testing process?
RQ3 How scalable is our approach?

As far as we know, there are currently no comparable tools for testing AJAX applications, thus we have no base line. To measure the effectiveness of our approach, we analyze the observed false positives and false negatives in our case studies. A false positive is a mistakenly reported fault that is caused by dynamic nondeterministic behavior and a false negative is an undetected real fault that is missed by the oracle comparator. To address RQ2, we report the time spent on parts that required manual work. RQ3 is addressed with a large dynamic AJAX web application in Section 8.2.

8.1 Case Study: HITLIST

Our first experimental subject is the AJAX-based open source *HitList*,[1] which is a simple contact manager based on PHP and jQuery.[2]

To asses whether new features/changes of an application can be detected correctly, we created three versions of HITLIST, with each version containing the features of its previous version plus:

V1 Adding, removing, editing, viewing, and starring contacts,
V2 Search functionality,
V3 Tweet information about the contact (Twitter details).

Additionally, in V3 we seeded a fault that causes the leading zeros in phone numbers to be missed, e.g., `0641287823` will be saved as `641287823`. We will refer to this as the *leading-zero bug*. This is a realistic fault because in V3 saving a contact is changed.

[1]HITLIST Version 0.01.09b, `http://code.google.com/p/hit-list/`
[2] `http://jquery.com`

```
CrawlSpecification crawler = new CrawlSpecification(URL);

//click anchors and buttons
crawler.clickDefaultElements();
crawler.click("input").withAttribute("class", "add_button");

//exclude these elements (a % is a wildcard)
crawler.dontClick("a").withAttribute("title", "Delete%");
crawler.dontClick("a").withAttribute("act", "%star");

//use oracle comparators
crawler.addOracleComparator(new WhiteSpaceComparator());
crawler.addOracleComparator(new StyleComparator);
crawler.addOracleComparator(new TableComparator());
crawler.addOracleComparator(new ListComparator());
```

Figure 8.1: Configuration for HITLIST experiment.

```
//check on home page
Condition onHomePage = new RegexCondition("Recently added contacts");
//check whether 'John Doe' is not already added
Condition johnDoeNotAdded = new NotRegexCondition("John Doe");
crawler.when(Logic.AND(onHomePage, johnDoeNotAdded)).click("a").withText("ADD")
```

Figure 8.2: Condition for adding only one contact.

Figure 8.1 shows a part of the configuration of CRAWLJAX for HITLIST (manual labor: 5 minutes), indicating the type of DOM elements that were included and excluded in the crawling phase. Based on this input, CRAWLJAX crawls the application and generates the oracle for a test suite. The pipelined generic oracle comparators that are used in the test suite can also be seen in the figure.

We measure the effects of our oracle comparator pipelining mechanism and the clickable resolving process, by enabling and disabling them during the regression testing phase. To constrain the state space, we created a *Condition* (Section 7.1) that ensures a contact can only be added once during the crawling phase which is done by checking a condition in the *Home page* state, as shown in Figure 8.2.

We made the following plan where the steps were repeated for every version.

1. Crawl HITLIST to generate a test suite,
2. Test HITLIST with this test suite,
3. Test the test suite with the next version

With this approach we can test the number of false positives and test our regression testing support.

```
<TABLE class="contact">[^<]*
 <TBODY>[^<]*
  <TR>[^<]*
   <TD height="[0-9]+" width="[0-9]+">[^<]*
    <IMG src="public/images/[a-z]+.png"/>[^<]*
   </TD>[^<]*
   <TD style="[^"]*">[^<]*
    [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/>[^<]*
    <A class="action view-details" href="javascript:;" id="[0-9]+">(.*?)view
        details(.*?)</A>[^<]*
   </TD>[^<]*
  </TR>[^<]*
 </TBODY>[^<]*
</TABLE>
```

Figure 8.3: Augmented template for the HITLIST contact list.

8.1.1 Experiments

Version 1. From V1, TEST SUITE A was generated from 25 states and the transitions between them, consisting of 11 test cases. With the comparators and resolver enabled, TEST SUITE A reported only one failure on V1, which occurred after a new contact was added. Closer inspection revealed that this state contains a link, (`<A class="action view details" href="javascript:;" id="125" >View details</A>`) with the `id` of the new contact. Since every contact has a unique `id`, this link is also different for every contact and therefore results in a state failure. This was easily resolved by creating (1 minute) a comparator with a precondition, which stripped the value of the `id` attribute from the *Contact Added* state. The *Table* comparator turned out to be very powerful, since it modeled the contact list as a template, and thus prevented state comparison failures after adding a new contact to the list and re-executing the test suite. We manually augmented (5 minutes) this generated template (Figure 8.3) and created a custom *Contact* comparator for the contact list. This template also serves as an invariant, because it checks the structure as well as the validity of the contact's name, phone number, e-mail, and *id*.

Version 2. We executed TEST SUITE A on V2 (with the new search functionality) and all the tests passed. Without oracle pipelining, there would be many failures, because of the added search button (see Table 8.1). The *List* comparator modeled the navigation bar, which is always displayed, as a template, thus ignoring the added search button and the *Contact* comparator stripped the differences caused by adding new contacts, in the contact list.

Version 3. All the test cases of TEST SUITE B except one (test case checking the initial state), failed on V3, which contained the new Twitter information and the *leading-zero bug*. The reason was that in all of the test cases the *Contact Details* state or the *Add Contact* state were visited, which contained the extra Twitter data. In the generated report (generated by the ErrorReport plugin Section 4.4) we could easily see and analyze these differences. The crawling of V3 resulted in a total of 31 states. When we executed TEST

Test suite	Tested on	#SD	#SD (ER)	#SD (OCP)	#SD (ER, OCP)	Reduction (ER, OCP)
A	V1	60	11	60	0	100%
A	V2	194	110	64	0	100%
B	V2	60	27	60	0	100%
B	V3	415	329	232	210	36%
C	V3	95	52	80	1	98%

Table 8.1: Number of reported state differences (SD) with and without element resolver (ER) and oracle comparator pipelining (OCP), for HITLIST.

SUITE C on V3, we found a failure in the initial state, while nothing was changed in the *Contact List* page and in previous test suites there were no failures reported on this state. The Contact comparator could not match the template because of the wrong format of the phone number. The leading zero bug could therefore easily be detected and seen in the error report.

8.1.2 Results

Table 8.1 presents the number of reported state differences (false positives) with and without enabling the element resolver and oracle comparator pipelining processes. Note that the one state difference reported by TEST SUITE C on V3 using both mechanisms is an actual fault, which is correctly detected.

8.1.3 Findings

Regarding RQ1, the combination of oracle comparators and the element resolver dealt very well with dynamic DOM changes, and reduced the false positives, up to 100%. Without the oracle comparators almost every test case would have failed and the detection of changed/removed states would be much more error-prone. Using only the generic comparators would result in a false negative, namely the leading-zero bug, which is ignored by the Table comparator. However, with the custom Contact comparator, we were able to detect the faulty phone number in V3. Added or removed contacts did not result in any problems because of the generated templates. Replaced clickables were correctly identified by our element resolver mechanism. For instance, the search functionality added in V2 caused many elements to be replaced. Without the resolver, many wrong transitions would be chosen and different states would be entered than expected, resulting in many state differences, as shown in Table 8.1.

Considering RQ2, most of the false positives were dealt with by the generic oracle comparators. We did have to manually set the correct CRAWLJAX configuration for crawling and testing HITLIST and create two custom comparators, which in total cost us about 7 minutes. On average, each crawling and test execution process took less than 2 minutes.

```
CrawljaxConfiguration config = new CrawljaxConfiguration();
config.addIgnoreAttribute("closure\_hashcode\_[a-zA-Z0-9]+");

CrawlSpecification crawler = new CrawlSpecification(URL);

//click these elements (a % is a wildcard)
crawler.click("a").withAttribute("class", "link");
crawler.click("a").withAttribute("id", "sub-tree-item-\%");
crawler.click("div").withAttribute("class", "goog-button-body");
crawler.click("span").withAttribute("class", "link item-title");

//exclude these elements
crawler.dontClick("a").withAttribute("href", "href=\%directory-page");
crawler.dontClick("a").withAttribute("id", "sub-tree-subscriptions");

//use oracle comparators
crawler.addOracleComparator(new WhiteSpaceComparator());
crawler.addOracleComparator(new StyleComparator);
crawler.addOracleComparator(new TableComparator());
crawler.addOracleComparator(new ListComparator());

//first experiment
crawler.setMaximumNumberState(12);

//second experiment
crawler.setMaximumNumberState(30);
```

Figure 8.4: Configuration for GOOGLE READER experiment.

8.2 Case Study: GOOGLE READER

To examine the scalability of our approach (RQ3), we chose GOOGLE READER[3] as our second subject system. GOOGLE READER has an extremely dynamic AJAX-based user interface consisting of very large DOM trees (+500 KB).

Figure 8.4 shows the CRAWLJAX configuration (15 minutes) used for GOOGLE READER. Before the DOM tree is used, CRAWLJAX strips the predefined HTML attributes (`closure_hashcode_[a-zA-Z0-9]+`), which are different after every load, and could easily result in 200+ state differences alone per state. A *preCrawling* plugin was created (less than 5 minutes) to automatically log in the application. To control the study, we constrained the state space by choosing a maximum number of crawled states of 12 and 30, on which 10 and 25 test cases were generated by CRAWLJAX, respectively.

In this case study, we evaluated the effectiveness of our approach by counting the number of reported false positives with and without the pipelining. The element resolver is always enabled for this study. Table 8.2 shows the average number of state differences, taken in 3 test runs, and the achieved reduction percentage by pipelining the generic oracle comparators.

[3] `http://www.google.com/reader`

Test Cases	#SD (ER)	#SD (ER, OCP)	Reduction
10	51	33	35%
25	366	162	55%

Table 8.2: Number of reported state differences (SD) with and without custom oracle comparator pipelining (OCP) for GOOGLE READER.

8.2.1 Results

Note that due to the high level of dynamism in GOOGLE READER, each consecutive test run reported many (new) state differences. The time between crawling and testing also had a large influence on the number of the found state differences. To further reduce the number of false positives, we created custom comparators (10 minutes) capturing a number of recurring differences in multiple states. For instance, these comparators were able to ignore *news snippets*, *post times* e.g., '20 Minutes ago', *unread counts*, and *one-line summaries*.

Applying our custom oracle comparator along with the generic oracle comparators resulted in a total reduction of from *50%* up to *95%*. In some cases, only a few false positives remained over.

We also created a comparator based on a generated template for the *Recently read* widget (less than 10 minutes) via the GUI (Figure 6.4). This oracle comparator completely solved the differences in this widget, which reduced the number of differences drastically. On average, each crawling and test execution process took between 5 to 10 minutes.

8.2.2 Findings

The generic comparators help in reducing the number of false positives (up to 55%), but they certainly cannot resolve all differences when dealing with extremely dynamic applications such as GOOGLE READER. Custom comparators can be of great help in further reduction (up to 95%), the creation of which took us about 20 minutes. Having knowledge of the underlying web application helps in creating robust comparators faster. Although the crawling and test execution time varied, on average one test (firing an event and comparing the states) cost about 0-2 seconds, with occasional peaks of 15 seconds with very large DOM trees. It is also worth mentioning that it is very important to have a large enough delay or a good wait strategy for the DOM tree to be updated completely, after an event is fired in the test cases. Otherwise, an incomplete DOM state could be compared with a complete state in the gold standard and result in a failure.

8.3 Discussion

In this section issues regarding our approach are discussed.

Applicability. Correctly making a distinction between irrelevant and relevant state differences in regression testing modern dynamic web applications is a challenging task, which requires more attention from the research community. Our solution to control, to some degree, the nondeterministic dynamic updates is through pipelining specialized oracle comparators, if needed with preconditions to constrain the focus, and templates that can be used as structural invariants on the DOM tree. Although implemented in CRAWLJAX, our solution is generic enough to be applicable to any web testing tool (e.g., Selenium IDE) that compares the web output with a gold standard to determine correctness.

Testing is always a human activity, tools can only help. With our report generator it is possible to inspect the failures easily. One can check for false negatives and false positives. False negatives can be reduced with preconditions. However, for best bug detection one can use more detailed (template) invariants or oracle comparators.

Side effects. A consequence of executing a test case can be a changed back-end state. We capture such side effects partly with our templates. However, not all back-end state changes can be resolved with templates. For instance, consider a test case that deletes a certain item. That item can only be deleted in the first test execution and a re-run of the test case will fail. Our approach currently is to setup a ‘clean-up’ post-crawling plugin that resets the back-end after a crawl session and a `oneTimeTearDown` method that brings the database to its initial state, after each test suite execution. Clickables that are removed or changed in a way that cannot be resolved by the element resolver can also cause problems, since it is not possible to click on non-existing elements and clicking on changed elements could cause transitions to unexpected states. Test suite repairing techniques [19] may help in resolving this problem.

Threats to Validity. Concerning *external validity*, since our study is performed with two applications, to generalize the results more case studies may be necessary; However, we believe that the two selected cases are representative of the type of web applications targeted in this paper.

With respect to *reliability*, our tools and the HITLIST case are open source, making the case fully reproducible; GOOGLE READER is also available online, however, because it is composed of highly dynamic content, reduction percentages might fluctuate. Our main concern in conducting the case studies was determining to what degree the number of false positives reported during regression testing could be reduced with our technique. Although we seeded one fault and it was correctly detected, we need more empirical evaluations to draw concrete conclusions on the percentage of false negatives.

A threat to *internal validity* is that each state difference reported in the case studies, corresponds to the first encountered state failure in a test case. In other words, when a test case has e.g., five state transitions and the first state comparison fails, the results of the remaining four state comparisons are unknown. This fact can affect (positively or negatively) the actual number of false positives.

Chapter 9

Crawljax Case Studies at Google

This chapter discusses two case studies done at Google London. The focus of these case studies was not regression testing but testing CRAWLJAX in general and its applicability in a real world production environment.

From 5 October 2009 till 18 December I did an internship at Google London to perform case studies with CRAWLJAX. My manager was Mike Davis and I joined the Software Engineer in Test (SET) team which is working on the AdSense Frontend version 3.0 (ASFE 3.0). ASFE 3.0 is the new front end for managing the Google advertisements [1] which are the advertisements that a company or user can place on its web site to earn money. ASFE 3.0 is developed with Google's GWT[2] and is used as a subject for our case studies.

9.1 Case Study: Invariant-based Testing

This section discusses the case study on invariant-based testing [22] (Section 7.1.1). The invariants discussed in this section are the invariants that are tested on the DOM tree and do not include plugins. These invariants are usually only one or a couple of lines of code.

9.1.1 Approach

We chose the management feature for the *AdUnits* (the displayed advertisements on the web pages) as subject for this case study which include:

1. Viewing AdUnits (browsing, searching, sorting, and filtering)
2. Adding an normal AdUnit
3. Adding a Search Box AdUnit

The default approach for writing invariants is to inspect the requirements and specifications of the product. For ASFE 3.0 there were some requirements and specifications, but these were incomplete and out-dated, thus one could extract a couple of invariants from these documents, but this was limited. The developers used the previous version, ASFE 2.0,

[1] `https://www.google.com/adsense`

[2] `http://code.google.com/webtoolkit/`

of ASFE 3.0 for the requirements and therefore ASFE 2.0 was also inspected to extract invariants. With the requirement documents and by inspecting ASFE 2.0 we wrote several invariants which could be assigned to three categories:

Error checking. The most general and simple invariants are the invariants that check the page for error messages. An invariant in this category checked the DOM for the text "We apologize for the inconvenience..." which should never occur.

Validation. The validation invariants test whether the current state is valid. An example is the validation of error messages, like when there is "not allowed to be empty" message, checking that the field is really empty. Another example was checking the generated JAVASCRIPT code for an AdUnit.

Interaction. The simplest version of this invariant type checks whether the interaction between widgets is correct. For example when the value of element A is X, element B should always be visible. Most of the invariants were of this type. A more advanced example was an invariant that checked that when filters were applied on a AdUnit list that at least one AdUnit was displayed.

Since ASFE 3.0 is fully developed with GWT it is not useful to test the widgets or the generated JAVASCRIPT, since these parts are already throughly tested. Invariants for generated JAVASCRIPT are also not useful because when such an invariant would fail it is not understandable for the user due to the generated (obfuscated) names.

9.1.2 Results

It is difficult to derive a complete set of invariants when there is poor documentation, but from the existing documentation it was easy to derive invariants.

The *Error checking* invariant detected an error message twice when CRAWLJAX was crawling for more than one hour. However, it was not clear what was the cause of this invariant violation, and therefore this fault could not be reproduced with a simple test. ASFE 3.0 contains a debug panel in development mode, which contains the latest RPC calls and stack traces. We created a plugin from the Error Checking invariant which retrieved information from the debug panel. However, the use of this information was still not enough to reproduce the error.

The *Validation* invariants did not detect any bugs at this time. There are bugs which could be detected with this invariant type. However, we inspected the bug list and these bugs are often only discovered in a certain work-flow. It is the question whether CRAWLJAX executes the correct paths to detect these types of bugs. A bug in a previous build where invalid JAVASCRIPT code was generated would have been detected by CRAWLJAX with a validation invariant.

The simple *Interaction* invariants found two violations of the requirements. However, the violations did not appear to be real bugs but updated requirements which were not documented. Another important aspect is that the violations were only detected if the user *manually* specified the correct form values and combinations. CRAWLJAX would not always detect these bugs with random form values.

We also inspected the bug list and did not found bugs that would be found with this type of invariants. This is mainly because these faults were already detected by *Unit Tests* and after some more inspection the conclusion was that these invariants are often almost a duplication of the source code. Figure 9.1 shows an example that could be expressed as an invariant that checks the values of one widget based on the value of another widget.

```
Consider two lists, List A and List B.
If the value of List A is X, List B should contain 1,2,3
If the value of List A is Y, List B should contain 4,5,6

Pseudo code:
if(listA.getValue() == X){
  listB.items = listCaseX
}else if(listA.getValue() == Y){
  listB.items = listCaseY
}

Invariants:
!(ListA.getValue() == X && listB.items != listCaseX)
!(ListA.getValue() == Y && listB.items != listCaseY)
```

Figure 9.1: Example of an interaction invariant

A more advanced *Interaction* invariant which checked multiple interactions tested filters on a list with AdUnits. This invariant revealed the following bug:
When the user applies a filter which causes no AdUnits to be displayed, the UI shows that this filter is applied. When the user navigates to another page and then navigates back to the AdUnits page, the UI says that no filters are applied. However, the list is still empty, thus the filter is still applied.

Since not many bugs were detected with the invariants we inspected the bug list (current bugs and fixed bugs) for bugs which could be detected with invariants. Only a small percentage (2% - %5) could be expressed as invariants. Often these types of bugs required a specific execution path and these paths are not always executed by CRAWLJAX.

9.1.3 Conclusion

Error checking invariants can detect errors, but it can be very difficult (1) to know why the fault occurred and (2) to reproduce the fault. However, it is still important to know that these errors occur!

Validation invariants can be useful since a they can detect functional bugs, but it is not sure whether CRAWLJAX always detects these bugs since they could be work-flow dependent.

The most invariants were the simple *Interaction* invariants which did not detect any real bugs. However, the more advanced *Interaction* invariant for the filters showed the real power of invariant-based testing with CRAWLJAX. Testing these multiple/complicated interactions is difficult with Unit Tests, because it is almost impossible to test all the paths.

We conclude that invariant-based testing is not applicable in environments like ASFE 3.0 when:

1. there is no highly interactive user interface.

2. there are already many Unit Tests (because these reduce the bugs detected by invariants significantly).
3. the GWT widgets and the generated JAVASCRIPT are already throughly tested.
4. there is poor documentation (requirements and specifications).

However, we think that for highly interactive user interfaces invariants can be very useful because invariants can detects bugs which are almost impossible to test with Unit Tests. Also, when there are clear requirements and specifications it is more easier to derive invariants, what could lead to better invariants.

When invariants are violated it is also important to know why the invariant is violated and how the related bug can be reproduced. This is still limited possible.

9.2 Case Study: Coverage

CRAWLJAX can explore web applications automatically, but it is important to know how much of the application it actually visits. We therefore conducted a case study on code coverage achieved by CRAWLJAX, which is discussed in this section.

9.2.1 Approach

In this case study we investigated the code coverage of the ASFE 3.0 API, which is used by the ASFE 3.0 UI for the business logic. The code coverage of CRAWLJAX is compared with manual written tests. Since ASFE 3.0 is built with GWT, we measure the *Java* code coverage of the API.

Measuring the absolute code coverage is not very useful, since it is very hard to interpreted the results. The difference of code coverage between a CRAWLJAX crawl session and the execution of manual tests is interesting, but also how the different settings and manual effort for CRAWLJAX affect the code coverage.

We compared the code coverage of CRAWLJAX with the current *WebDriver*[3] tests which test all the basic functionality of ASFE 3.0. These tests open the browser and execute the paths a normal user would take. The coverage is also measured with different settings for CRAWLJAX.

The code coverage was measured by running a local version of the ASFE 3.0 server, running CRAWLJAX or the WebDriver tests, closing the server, and collecting the coverage data.

9.2.2 Results

This section does not contain actual numbers because these are considered confidential by Google. The general outcomes of this case study are therefore discussed in this section.

Figure 9.2 shows the coverage of the WebDriver tests, CRAWLJAX with random input, and CRAWLJAX with manually specified form values. As can be seen from Figure 9.2

[3] `http://code.google.com/p/selenium/`

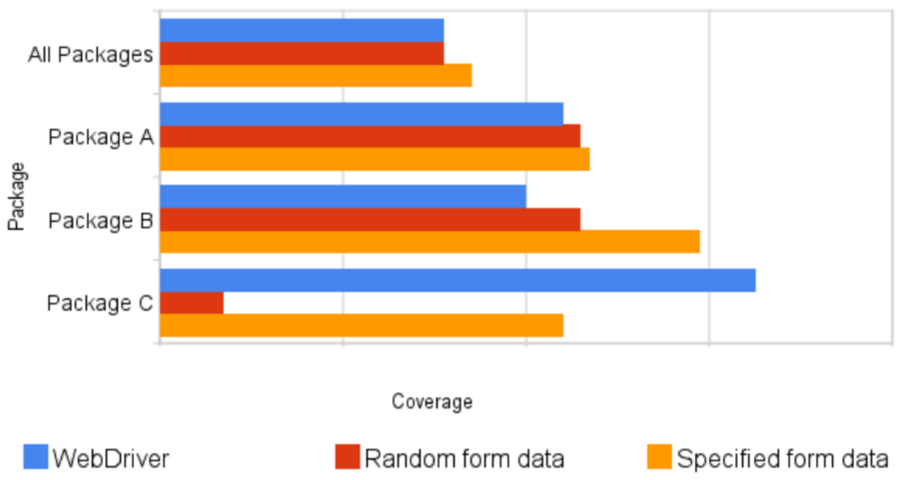


Figure 9.2: Code coverage ASFE 3.0 API - CRAWLJAX compared with WebDriver tests

CRAWLJAX has in general the same or more code coverage than the WebDriver tests. Note that this does not say that the same code is covered.

In *Package C* there is a large difference in code coverage, which was caused by a form that needed an URL and could not proceed with a random value. Without this URL a large part of the code was not executed. When a valid URL was manually specified the code coverage increased significantly.

Some other results were:

1. Without any form input data the code coverage was significantly lower than with random form input data.
2. The code coverage of CRAWLJAX is most of the time slightly higher. However, when the code coverage is less then the manual tests it is often significantly lower.
3. Overwriting existing texts in text fields gave a slightly higher code coverage than only entering text in empty fields.

9.2.3 Conclusion

From the coverage results we can conclude that CRAWLJAX can reach higher code coverage than a complete manual written test suite. It is important to inspect whether the web application is crawled completely and measuring coverage is a good method to verify this.

This case study only represents the code coverage on one application and could be different with other applications or manual written test suites. More case studies with other applications could be very useful and more research on which parts of the code are covered can be very interesting.

Chapter 10

Concluding Remarks

This chapter gives a brief overview of this research, guidelines for future work, and conclusions regarding this research.

10.1 Summary

This paper presents an approach for controlling the highly dynamic nature of modern web user interfaces, in order to conduct robust web regression testing. The contributions of this research include:

- A method, called *Oracle Comparator Pipelining*, for combining the power of multiple comparators, in which each comparator processes the DOM trees, strips the targeted differences, and passes the result to the next comparator in a specified order.
- The combination of oracle comparators with *preconditions*, to constrain the state space on which a comparator should be applied.
- Automatic generation of structural templates for common DOM structures such as tables and lists, and support for manual augmentation of application-specific templates, which can be used as invariants.
- Implementation of these methods in the open source tool CRAWLJAX, which, besides the crawling and test generation part, comes with a set of generic oracle comparators, support for visualization of test failures through a generated report to give insight in the DOM differences, and a GUI to create custom templates for modeling structures.
- An empirical evaluation, by means of two case studies, of the effectiveness and scalability of the approach, as well as the manual effort required.
- External case studies in which the applicability of CRAWLJAX in a real world production environment is tested.
- Improvements to CRAWLJAX that make it suitable for production use.

10.2 Future Work

There are several directions for future work. Conducting more case studies and adopting test repair techniques to deal with deleted clickables are directions that will be pursued.

The generating of templates can be improved by supporting more structures and not only considering the default structures like tables and lists, but also other patterns with, for example, links or `DIV` elements. We will explore automatic extracting of more detailed templates which can be used as invariants. This could lead to a new research topic about automatic detection/generation of invariants based on (DOM) structures.

We will also investigate possibilities of generating more abstract templates that can capture a model of the application at hand. In addition, finding ways of directing the regression testing in terms of what parts of the application to include or exclude forms part of our future research.

Creating new (generic) oracle comparators will be a topic of our future research. Another interesting approach is conducting research with oracle comparators that are not based on stripping, but do their comparison based on important properties. This can be done by analyzing the states for important elements like widgets, links, form input elements, and images, and comparing whether the states have the same important elements.

Another subject for future work is detecting regions in the DOM trees, and identify which parts change and which parts do not change. With this approach the oracle comparators can only consider a part of the DOM which could improve the performance. Another advantage with this approach is that there are less duplicated state differences, because some DOM parts can only be considered once.

Handling form input elements is important for the coverage and for testing purposes. More research should therefore be done on improving form handling by entering useful input without human effort and combining different input values.

10.3 Conclusion

The introduction of oracle comparator pipelining, preconditions, and templates, showed to be useful for regression testing. These techniques can therefore be used for future research with CRAWLJAX, and since our approaches are generic, they can also be with other tools like capture-and-replay tools.

Our case studies show that with CRAWLJAX and our plugins (RegressionTester, ErrorReport) it is possible to perform a largely automatic approach for the regression testing of AJAX applications. With our approach several types of regression faults can be detected, like state differences, missing/changes links, flaky pages, and application specific faults. These faults can easily be inspected by the tester by using the generated report. With the use of invariants, application-specific tests can be performed, which can be very powerful. However, there are still many things to explore in the field of regression testing AJAX applications, but our approach creates a good starting point.

Given the growing popularity of AJAX application and therefore the need for frequent testing we see many opportunities for CRAWLJAX. As shown in Section 9.2 the code coverage of CRAWLJAX is good, what is an important aspect for (regression) testing AJAX applications. We showed that CRAWLJAX can be used in a real production environment and can handle large applications.

Bibliography

[1] N. Alshahwan and M. Harman. Automated session data repair for web application regression testing. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST'08)*, pages 298–307. IEEE Computer Society, 2008.

[2] 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 Eoftware Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering (ESEC-FSE'09)*, pages 81–91. ACM, 2009.

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

[4] H. Do and G. Rothermel. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 141–151, New York, NY, USA, 2006. ACM.

[5] E. Dustin, J. Rashka, and J. Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[6] S. Elbaum, K.-R. Chilakamarri, B. Gopal, and G. Rothermel. Helping end-users 'engineer' dependable web applications. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pages 31–40. IEEE Computer Society, 2005.

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

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

[9] IEEE. *IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology*. IEEE, 1990.

[10] P. Koopman, R. Plasmeijer, and P. Achten. Model-based testing of thin-client web applications. In *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*, pages 115–132. Springer Berlin / Heidelberg, 2006.

[11] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707–+, Feb. 1966.

[12] A. Marchetto, F. Ricca, and P. Tonella. A case study-based comparison of web testing techniques applied to ajax web applications. *Int. J. Softw. Tools Technol. Transf.*, 10(6):477–492, 2008.

[13] 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.

[14] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of ajax web applications. *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 121–130, April 2008.

[15] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, 1989.

[16] A. Memon, I. Banerjee, N. Hashmi, and A. Nagarajan. DART: a framework for regression testing "nightly/daily builds" of gui applications. *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 410–419, Sept. 2003.

[17] A. M. Memon. *A comprehensive framework for testing graphical user interfaces*. Ph.D., 2001. Advisors: Mary Lou Soffa and Martha Pollack; Committee members: Prof. Rajiv Gupta (University of Arizona), Prof. Adele E. Howe (Colorado State University), Prof. Lori Pollock (University of Delaware).

[18] A. M. Memon. Automatically repairing event sequence-based gui test suites for regression testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2):1–36, 2008.

[19] A. M. Memon and M. L. Soffa. Regression testing of GUIs. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 118–127, New York, NY, USA, 2003. ACM.

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

[21] 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.

[22] 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.

[23] B. A. Myers. User interface software tools. *ACM Trans. Computer Human Interaction*, 2(1):64–103, 1995.

[24] J. Offutt. Quality attributes of web software applications. *IEEE Softw.*, 19(2):25–32, 2002.

[25] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 241–252, 2004.

[26] F. Ricca and P. Tonella. Web application slicing. *Software Maintenance, IEEE International Conference on*, 0:148, 2001.

[27] 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)*. IEEE Computer Society, 2010.

[28] E. Soechting, K. Dobolyi, and W. Weimer. Syntactic regression testing for tree-structured output. In *Proceedings of the 11th IEEE International Symposium on Web Systems Evolution (WSE'09)*. IEEE Computer Society, 2009.

[29] 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.

[30] 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.

[31] 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.

[32] A. Tarhini, Z. Ismail, and N. Mansour. Regression testing web applications. In *International Conference on Advanced Computer Theory and Engineering*, pages 902–906. IEEE Computer Society, 2008.

[33] L. Xu, B. Xu, Z. Chen, J. Jiang, and H. Chen. Regression testing for web applications based on slicing. In *Proceedings of the 27th Annual International Conference on Computer Software and Applications (COMPSAC'03)*, pages 652–656. IEEE Computer Society, 2003.

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