Delft University of Technology
Software Engineering Research Group
Technical Report Series

Regression Testing Ajax Applications: Coping with Dynamism

Danny Roest, Ali Mesbah and Arie van Deursen

Report TUD-SERG-2009-028

TUDelft

SERG

TUD-SERG-2009-028

Published, produced and distributed by:

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

ISSN 1872-5392

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

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

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

Danny Roest, Ali Mesbah and Arie van Deursen. **Regression Testing AJAX Applications: Coping with Dynamism**. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10), Paris, France*. IEEE Computer Society, 2010.

© copyright 2009, by the authors of this report. Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the authors.

Regression Testing Ajax Applications: Coping with Dynamism

Danny Roest
Delft University of Technology
The Netherlands
D.Roest@student.tudelft.nl

Ali Mesbah
Delft University of Technology
The Netherlands
A.Mesbah@tudelft.nl

Arie van Deursen
Delft University of Technology
The Netherlands
Arie.vanDeursen@tudelft.nl

***Abstract*—There is a growing trend to move desktop applications towards the web using advances made in web technologies such as AJAX. 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 and asynchronous client/server interactions. In this paper, we propose a technique, in which we automatically generate test cases and apply pipelined oracle comparators along with generated DOM templates, to deal with dynamic non-deterministic behavior in AJAX user interfaces. Our approach, implemented in CRAWLJAX, is open source and provides a set of generic oracle comparators, template generators, and visualizations of test failure output. We describe two case studies evaluating the effectiveness, scalability, and required manual effort of the approach.**

***Keywords*-regression testing; ajax; web applications**

I. INTRODUCTION

There is a growing trend of moving desktop applications to the Web. Well-known examples include Google's mail and office applications. One of the implications of this move to the web is that *dependability* [13] of web applications is becoming increasingly important [4], [7].

One of the key enabling technologies currently used for building modern web applications is AJAX, an acronym for "Asynchronous JAVASCRIPT And XML" [6]. While the use of AJAX technology positively affects the user-friendliness and the interactiveness of web applications [11], it comes at a price: AJAX applications are notoriously error-prone due to, e.g., the stateful and asynchronous nature as well as the extensive use of the Document Object Model (DOM) and (untyped) JAVASCRIPT on the client.

As software evolves, one common way to provide assurance about the correctness of the modified system is through regression testing. 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.

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* [17] typically analyze the differences between server response pages [15].

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 [15], [18]. 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 paper to find solutions for this *controllability* problem, thus facilitating robust comparison-based regression testing of AJAX applications.

We start our paper with a brief summary of our earlier work on automated testing of AJAX applications [10], [12], since this provides the context and motivation for the present paper. 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).

Next, in Section III we describe in detail four challenges involved in the regression testing of AJAX applications. In Section IV, we then cover our approach to tackle these challenges. In particular, we propose a number of ways to control and refine the process of comparing two user interface states occurring in AJAX applications.

In Section V we summarize our implementation of the proposed methods in our web testing infrastructure [12], which we subsequently use to conduct two case studies (a simple open source contact management system, as well as the highly dynamic Google Reader application). In Section VII we reflect on the lessons we can draw from this research, after which we conclude with a summary of related work, contributions, and future research.

II. BACKGROUND

In our previous work [10], we 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

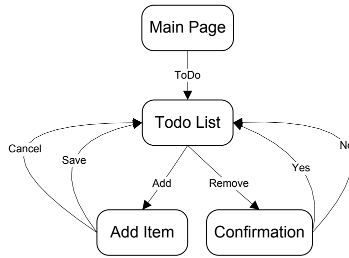

Figure 1. Navigational Model of TODO.

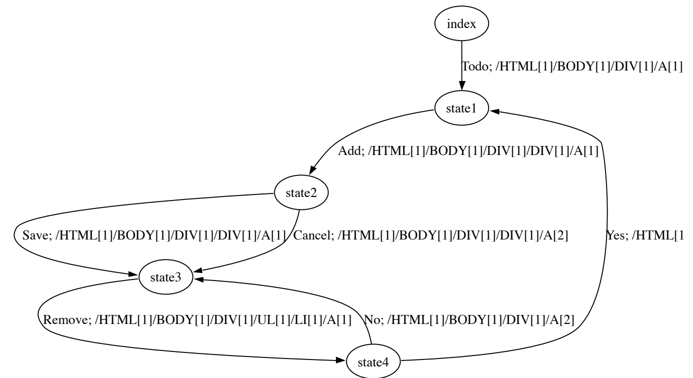

Figure 2. Inferred State-flow Graph for TODO.

dynamically built DOM. While crawling, CRAWLJAX infers a *state-flow graph* 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.

While running the crawler can be considered as a first full smoke test pass, the derived state machine itself can be further used for testing purposes. More recently, we proposed an approach for automatic testing of AJAX user interfaces, called ATUSA [12], in which we 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. Our gold standard [3] is composed of structural invariants (see [12], [2]) defined by the tester and the saved output (DOM trees) of a previous execution-run on a trusted version of the application.

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 1. To obtain the state-flow graph shown in Figure 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)
5) clicks Cancel, ends up in *state3* (# of Items: 1), and stops.

Note that the generated graph has an extra state, since *state1* (empty list) and *state3* (list with one item) on the *Todo List* page are not equivalent. After the crawling phase, based on this graph ATUSA generates four test cases that test:

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

```
public void testcase_1_2_3_4_5(){                                          1
 try {                                                                     2
  FormHandler.handleFormElements(browser);                                 3
  //Todo href=todo.php onclick=load() xpath=/DIV[1]/A[1]                   4
  assertTrue("Event fired: Todo", fireEvent(1, "Todo"));                   5
  assertTrue("Invariants passed in state: 1" ,                             6
       checkInvariants(browser));
  assertTrue("State equivalent with: 1", compareStates(1,                  7
       browser.getDom()));
  ...} }                                                                   8
```

Figure 3. A Generated Test Case for TODO.

- The initial state
- Event sequence: Todo - Add - Save - Remove - Yes
- Event sequence: Todo - Add - Save - Remove - No
- Event sequence: Todo - Add - Cancel

Figure 3 shows the generated test case partially. For each test case, the browser is first put in the initial index state of the web application. From there, input elements are filled if present and events are fired on the clickables. After each event invocation, the resulting state in the browser is checked against the invariants (line 6) and compared with the expected state in the database (line 7).

Note that a test case succeeds if: every (clickable) element in the test case can be found and its event can be fired, no invariants are violated, and every actual state is equivalent to the expected state. 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 are discussed in the next section.

III. AJAX REGRESSION TESTING CHALLENGES

Nondeterministic Behavior. Most regression testing techniques assume [14] 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 [8] behavior can be, for

```
<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: 22-08-2009 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="past">Last update: 22-08-2009 16:50</P>
```

Figure 4. Two DOM string fragments of TODO at different timestamps.

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 4). 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 [17] and AJAX in particular notoriously difficult.

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 4 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.

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

```
preCondition = new RegexPreCondition("<H1>Todo items</H1>"
     );
attributeOracle = new AttributeOracleComparator("class");
oraclePreCondition = new OraclePreCondition(
     attributeOracle, preCondition);
```

Figure 5. An attribute oracle comparator with a precondition.

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 4 (e.g, `/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.

Detecting Added and Removed States. A generated test suite is generally not able to test newly added states. At the same time, states that are removed by the developer intentionally will also result in a failure. For example, when a search option is added to TODO, previously generated test cases have to be updated to include this new part. One common solution is to update the test suite manually to include the changes. For large numbers of changes, however, this becomes a tedious and error-prone task for the tester. Another solution is to regenerate the whole test suite automatically. A side effect of this approach is that it is not possible to see which states were added and removed easily. The challenge is thus updating the test suite automatically and maintaining as much information as possible from the old test suite to detect the changes.

IV. OUR APPROACH

In this section we describe our approach for handling the challenges described in Section III.

A. Determining State Equivalence

Stripping the Differences. Oracle comparators in the form of a *diff* method between the current and the expected state are usually used [17], [15] to handle non-deterministic behavior in web applications. 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 expression or an XPath expression, and then comparing the stripped DOMs with a simple string comparison.

Oracle Comparator Preconditions. Applying these generic comparators can be very effective in ignoring nondeterministic real-time differences, but at the same time, they

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

Figure 6. A JAVASCRIPT crawl condition, with regular expression precondition, for adding a contact once.

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, 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 4, 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 the `<p>` 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.

We have created the following types of conditions that can be used for different purposes:

1) **Regular expression condition** returns true iff the regular expression is satisfied.
2) **XPath expression condition** returns true iff the XPath expression returns at least one DOM element.
3) **JavaScript expression condition** returns true iff the JAVASCRIPT expression evaluates to true.
4) **OR condition** returns true iff one of the two specified conditions returns true.

The Regular and XPath expression conditions also have an inverse variant for more flexibility.

Crawling Conditions. The nondeterministic behavior 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 6 shows an example of a JAVASCRIPT crawl condition with a regular expression precondition.

Oracle Comparator Pipelining. When comparing DOM states, there are often multiple types of differences, each requiring a specific type of oracle comparator. In our approach, we combine different comparators in a technique

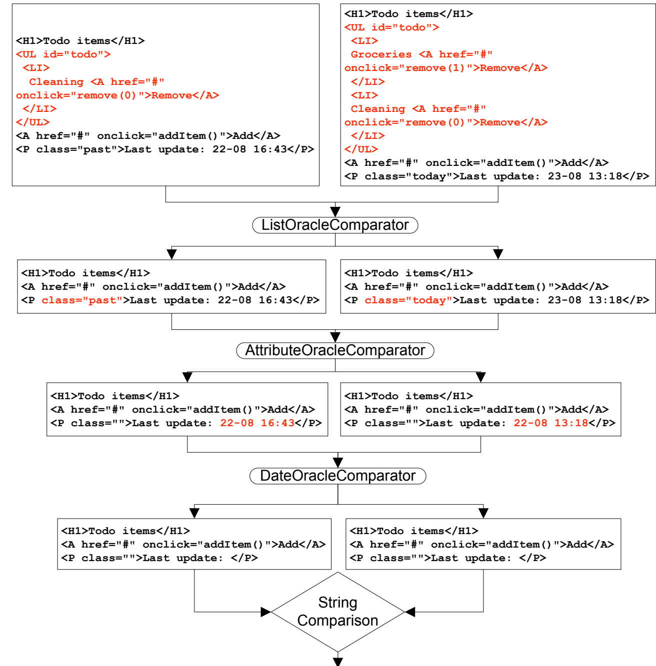

Figure 7. Oracle Comparator Pipelining for TODO.

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

Figure 8. Generated template for the TODO list.

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 7 shows an example of how two HTML fragments are compared through a set of pipelined comparators.

Template Generation. 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 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 the state changes are manifested.

To support these 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. Figure 8 depicts the generated template for the list pattern of Figure 4.

```
<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 9. An augmented template that can be used as an invariant.

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 2 can now be seen as one equal state.

These generated templates can also be augmented manually and used as invariants on the DOM tree. Figure 9 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.

To deal with huge number of state comparisons in the crawling and testing phase, for each visited state a hash code is created of the stripped DOM tree after the pipelining process. This hash code is saved for each state and can be used to quickly compare states. A new hash code is calculated only when the oracle comparator pipelining configuration changes.

B. Resolving Clickables

To replay a test case, events should be fired on elements. In order to fire events, each clickable element first has to be identified on the browser's 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 event handler(s), attributes, children nodes, and text value. Using a combination of these properties our *element resolver* searches the DOM for a match with a high level of certainty.

C. Updating the Test Suite

To detect newly added and removed states, we need to update our test suite. To that end, we crawl the new version of the application again, and reuse as much data from the old inferred state-flow graph (of the previous version) as possible, to build a new state machine.

When our crawler is in state A and it detects a new state B via event E, it tries to match B with a state in the old graph. From the old graph, candidate states that are as close as possible to our new state, in terms of equality and relations (parent and children) to other states. Candidate states are selected through four different approaches in the following order:

1) all children of A;

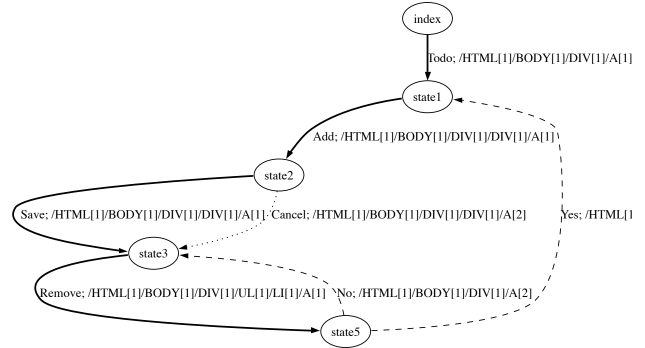

Figure 10. Building a new state-flow graph by re-using information from a previously inferred graph.

2) all states reachable via E;
3) all grandchildren of A's parent;
4) all states with the same pipelined DOM hash code.

Take as example a second crawl session on a new version of TODO, where Figure 2 is our old graph. Our crawler always starts in the *index* state, thus knowing its starting point in the old graph. When it fires an event and finds a new state, there is a high possibility that this state is one of the children (outgoing) of the *index* state. Approach 1 selects *state1* and checks whether it is equivalent to the new state. If that is the case, (a reference to) *state1* is added to the new state-flow graph, else it is added as a completely new state.

Now imagine that we are in *state3* and the confirmation page *state4* is completely replaced by a new state *state5*. We could consider this as a removing a state and adding a state scenario. None of the steps above is able to find a match in the old graph, thus *state5* is added to the new graph, as shown in Figure 10 (The dashed and dotted lines are the state transitions that are not crawled yet).

The outgoing states (e.g., *state3*) from *state5* cannot be checked by 1 because *state5* does not exist in the old graph. If the event handler (e.g., `onclick='cancel()'` of the clickable element can be determined, approach 2 is used. Events initiate state transitions and therefore the related states form plausible candidates. If 2 cannot be applied or does not return an equivalent state, approach 3 is used. Since *state5* is a new state, we cannot directly determine its children in the old graph. Approach 3 solves this problem by selecting the grandchildren of the current state's (*state5*) parent (*state3*), which are *state1* and *state3*) in the old graph. As a last resort, when no equivalent state is found, the old graph is searched entirely by comparing the hash codes (approach 4).

The test suite can easily be updated with this newly inferred state-flow graph, which contains the old matched states as well as the newly added states. In addition, the test code corresponding to the obsolete states is removed.

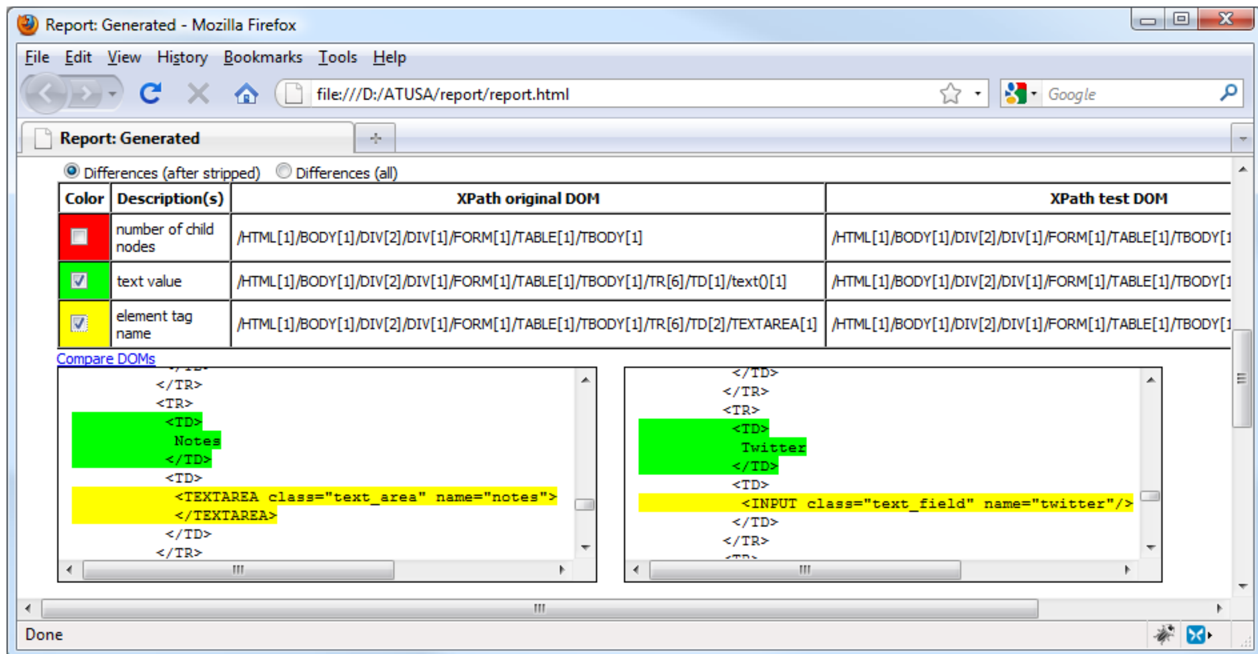

Figure 11. Visualizing the DOM tree differences that caused a regression test failure.

V. Implementation

Our approach is implemented in Java and integrated in the open source testing tool CRAWLJAX [12].[2] The implementation details of the crawler, CRAWLJAX, can be found in [10].

We have implemented a number of generic comparators, each of which is responsible for ignoring merely one type of difference. To name a few: Whitespace, Attributes, Style, Datetime, Structure, List, Table, Regex, and XPathExpression.

To assist the tester, an SWT-based GUI is implemented for analyzing various crawl sessions. Through this user interface, DOM templates can automatically be generated by selecting a state or by selecting the desired parts of the DOM tree. From these templates, oracle comparators (in Java) can also be created automatically and added to the pipelining mechanism. There is currently support for generating templates for following structural elements: `TABLE`, `UL`, `OL`, `SELECT`, and `DL`. The comparators can use these templates by stripping the matching string from the DOM tree. The Structure comparator supports all the mentioned elements.

Understanding why a test case fails is very important to determine whether a reported failure is caused by a real fault or a legal change. To that end, our toolset generates a detailed web report that visualizes the failures. We format and pretty-print the DOM trees without changing their structure and use XMLUnit[3] to determine the DOM differences. The elements related to the differences are highlighted with different colors in the DOM trees. We also capture a snapshot of the browser at the moment the test failure occurs and include that in the report. In the report, it is possible to view the stripped DOMs, 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, JAVASCRIPT debug variables, and the list of applied comparators are also displayed. Figure 11 shows a screenshot of a generated report.

VI. Empirical Evaluation

To assess the usefulness and effectiveness of our approach, we conducted two case studies [20]. 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?

To measure the effectiveness, 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. 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.

[2] http://crawljax.com

[3] http://xmlunit.sourceforge.net/

Click tags	a:{}, input:{class=add_button}
Exclude tags	a:{title=Delete%}, a:{class=delete-button}, a:{act=star; class=fav-button}, a:{act=unstar; class=fav-button%}
Generic Comparators	WhiteSpace, Style, Table, List

Table I
CONFIGURATION FOR HITLIST. % IS A WILDCARD.

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 II
NUMBER OF REPORTED STATE DIFFERENCES (SD) WITH AND WITHOUT ELEMENT RESOLVER (ER) AND ORACLE COMPARATOR PIPELINING (OCP), FOR HITLIST.

A. *Study 1:* HITLIST

Our first experimental subject is the AJAX-based open source *HitList*,[4] which is a simple contact manager based on PHP and jQuery.[5]

To asses if new features 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., `0641288822` will be saved as `641288822`. We will refer to this as the *leading-zero bug*.

Table I shows parts of the configuration of our tool for HITLIST (manual labor: 1 minute), 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 a test suite. The pipelined generic oracle comparators that are used in the test suite can also be seen in the table.

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 *CrawlCondition* that ensures a contact can only be added once during the crawling phase. This is done by checking a JAVASCRIPT condition in the *Add Contact* state, as shown in Figure 6.

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 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 II). 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. To update the test suite, we crawled V2, and there were two new states detected automatically: the search page and the search results page. Note that no other states were added, because of the List and Contact comparators. TEST SUITE B was generated from this crawl session, which passed successfully when executed on V2.

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, partially shown in Figure 11, we could easily see and analyze these differences. The crawling of V3 resulted in a total of 31 states, where 22 states were removed and 26 states were added (related to the previous versions), i.e., four states were unchanged. TEST SUITE C was generated from the inferred state-flow graph. When we executed TEST 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.

[4] HITLIST Version 0.01.09b, http://code.google.com/p/hit-list/
[5] http://jquery.com

Click tags	a:class=link, div:class=goog-button-body ,a:id=sub-tree-item-%, span:class=link item-title overview-item-link
Exclude tags	a:href=%directory-page, a:id=sub-tree-subscriptions
Comparators	WhiteSpace, Style, Date, Table, List
Max # states	12, 30
Stripped Attributes	`closure_hashcode_[a-zA-Z0-9]+`

Table III
CONFIGURATION FOR GOOGLE READER EXPERIMENT.

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

Table IV
NUMBER OF REPORTED STATE DIFFERENCES (SD) WITH AND WITHOUT CUSTOM ORACLE COMPARATOR PIPELINING (OCP) FOR GOOGLE READER.

Results. Table II 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.

Findings. Regarding RQ1, the combination of oracle comparators and 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 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 II. As far as updating the test suite is concerned, our approach performed very well. The added states in V2 and V3 were easily detected and added to the test suite, while keeping the rest of the test suite unchanged. The state differences could easily be analyzed in the report and through the GUI, the added and removed states were easily distinguishable.

Considering RQ2, most of the false positives were dealt with by the generic oracle comparators. We did have to manually set the correct CRAWLJAX properties 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.

B. *Study 2:* GOOGLE READER

To examine the scalability of our approach (RQ3), we chose GOOGLE READER[6] as our second subject system. GOOGLE READER has an extremely dynamic AJAX-based user interface consisting of very large DOM trees (+500 KB).

Table III shows the CRAWLJAX configuration properties set (3 minutes) for GOOGLE READER. Before the DOM tree is used, ATUSA 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, 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 IV shows the average number of state differences, taken in 3 test runs, and the achieved reduction percentage by pipelining the generic oracle comparators.

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. 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.

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.

[6] http://www.google.com/reader

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 2-5 seconds, with occasional peaks of 40 seconds with very large DOM trees. It is also worth mentioning that it is very important to have a large enough delay 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.

VII. Discussion

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) that compares the web output with a gold standard to determine correctness.

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 [9] 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.

VIII. Related Work

While regression testing has been successfully applied in many software domains [9], [14], web applications have gained very limited attention from the research community [19], [18].

Alshahwan and Harman [1] discuss an algorithm for regression testing based on session data [16], [5] repair. 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.* [17] propose a set of specialized oracle comparators for testing web applications's HTML output, by detecting differences in the content and structure through *diff*. In a recent paper, Soechting *et al.* [15] 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.

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 constrain the state space to which a certain comparator has to be applied by means of preconditions.

IX. Concluding Remarks

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 paper 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 compara-

tor should be applied as well as a method for updating the test suite.

- 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 plugins for generic oracle comparators, support for visualization of test failure thorough a generated report to give insight in the DOM differences, and a GUI to view the added and removed states in each crawl session.
- An empirical evaluation, by means of two case studies, of the effectiveness and scalability of the approach, as well as the manual effort required.

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

REFERENCES

[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] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

TUD-SERG-2009-028
ISSN 1872-5392