Search-Based Testing of Ajax Web Applications

Alessandro Marchetto and Paolo Tonella
Fondazione Bruno Kessler - IRST
38050 Povo, Trento, Italy
marchetto|tonella@fbk.eu

Abstract

Ajax is an emerging Web engineering technology that supports advanced interaction features that go beyond Web page navigation. The Ajax technology is based on asynchronous communication with the Web server and direct manipulation of the GUI, taking advantage of reflection. Correspondingly, new classes of Web faults are associated with Ajax applications.

In previous work, we investigated a state-based testing approach, based on semantically interacting events. The main drawback of this approach is that exhaustive generation of semantically interacting event sequences limits quite severely the maximum achievable length, while longer sequences would have higher fault exposing capability. In this paper, we investigate a search-based algorithm for the exploration of the huge space of long interaction sequences, in order to select those that are most promising, based on a measure of test case diversity.

Keywords: Web Testing, Ajax Applications and Search-based Software Engineering.

1 Introduction

Web applications developed for the so-called "Future Internet" are expected to offer a richer user experience than current hypermedia navigation. The client is supposed to become a rich, interactive and highly responsive environment, offering information through advanced, adaptive means (e.g., 3D graphics). Among the technologies that are being developed to implement this vision, Ajax is one of the most promising and mature. Ajax overcomes the synchronous request-response protocol used by traditional Web applications by supporting asynchronous requests, which leave the user interface active and responsive. Combined with the possibility to update a Web page dynamically through the DOM (Document Object Model, [1]), Ajax promises to be a core component of the Future Internet technologies.

While improving the functionalities and interaction offered to the users, Ajax poses novel, additional problems with respect to those already known in the Web testing area [2, 4, 10]. Asynchronous requests and responses may interleave in an unexpected way, so Ajax programmers need to carefully program the concurrency provided by Ajax. Dynamic page update is another potential source of faults that are specific of Ajax applications. In fact, the code may be written according to wrong assumptions about the DOM state (e.g., an HTML element, such as a form or a table, is assumed to exist in the current page, while it is not there).

Recently, Marchetto *et al.* [8] investigated the use of state-based testing for AJAX Web applications and particularly focused on the specific faults introduced by this technology. The technique is based on dynamic extraction of a finite state machine for an Ajax application and its analysis with the aim of identifying sets of test cases based on *semantically interacting events* (Yuan *et al.* [13]). Empirical evidence [8, 12, 13] shows the effectiveness of this kind of technique in finding faults. Automatically generated test cases are comparable to those obtained by careful functional testing, manually performed by expert testers. Unfortunately, one of the main drawbacks of the technique based on semantically interacting events is that it generates testing suites composed of a very large number of test cases and this can limit its usefulness.

Semantically interacting event sequences are generated up to a maximum length k_{max}. Since there is an upper bound to the number of test cases that can reasonably compose a test suite, k_{max} is in practice quite small. In the case study documented by Marchetto *et al.* [8] more than 5000 test cases of length $k \leq 4$ are generated from the semantically interacting event sequences. On the other hand, the experiments performed by Marchetto *et al.* and by Yaun *et al.* show that the capability to reveal faults tends to grow at increasing event sequence length k.

In this paper, we investigate the use of a search-based approach (based on the hill-climbing algorithm) to address the problem of generating long semantically interacting event sequences while keeping the test suite size reasonably small. In order to preserve a fault revealing power comparable to that of the exhaustive test suite, we maximize the diversity of the test cases. We re-formulate the problem as an optimization problem that can be solved by applying a heuristic search algorithm guided by an objective function. Specifically, we introduce a measure of test case diversity and instead of generating exhaustively all test cases up to a given length k_{max}, we select only the most diverse test cases, without any constraint on their length k. A case study has been conducted on two real Ajax applications. The results indicate that test suites consisting of long interaction sequences generated by means of the proposed search-based approach are the most effective.

The paper is organized as follows: Sections 2 and 3 provide some background respectively on Ajax and Ajax testing (summarizing the approach by Marchetto *et al.* [8]). Our search-based approach to test case generation is presented in Section 4. Experimental results are discussed in Section 5. Related works (Section 6) and conclusions (Section 7) terminate the paper.

2 Ajax

Ajax (Asynchronous Javascript And XML) is a bundle of technologies used to simplify the implementation of rich and dynamic Web applications. It employs HTML and CSS for information presentation. The Document Object Model [1] is used to access and modify the displayed information. The *XMLHttpRequest* object is exploited to retrieve data from the Web server asynchronously. XML is used to wrap data and Javascript code is executed upon callback activation.

With Ajax, developers can implement asynchronous communications between client and server. To this aim, the Ajax object *XMLHttpRequest* wraps any service request or data traveling between client and server. The asynchronous response from the server is handled at the client side by Javascript code which is part of a callback triggered by the response. Upon arrival of the server message, the handler method associated with such an event is run. Depending on the received data, the message handler can change the structure or content of the current Web page through the DOM.

Since Ajax Web applications are heavily based on asynchronous messages and DOM manipulation, we expect the faults associated with these two features to be relatively more common and widespread than in other kinds of applications. So, first of all, Ajax testing should be directed toward revealing faults related to incorrect manipulation of the DOM. For example, this may be due to assumptions about the DOM structure which become invalid during the execution, because of page manipulation by Javascript code. Another example is an inconsistency between code and DOM, which makes the code reference an incorrect or nonexistent part of the DOM. Second, Ajax testing should deal with asynchronous message passing, a well-known source of trouble in software development [5]. Often Ajax programmers make the assumption that each server response comes immediately after the request, with nothing occurring in-between. While this is a reasonable assumption under good network performance, when the network performance degrades, we may occasionally observe unintended interleaving of server messages, swapped callbacks, and executions occurring under incorrect DOM state. All such faults are hard to reveal and require dedicated techniques [7].

3 Ajax Testing

An Ajax Web application is constructed around the structure of the DOM to be manipulated by the message handlers associated with user events or server messages. Correspondingly, we model an Ajax application by means of a Finite State Machine (FSM), the states of which represent DOM instances and the transitions of which represent the effects of callback executions. This model captures precisely the two distinctive features of Ajax, namely reflective DOM manipulation and asynchronous messages. Given this model of an Ajax application, test cases can be derived according to available techniques for state-based testing [11], such as path coverage criteria. However, such approaches tend to generate a high number of test cases involving unrelated events. Hence, we restricted the considered event sequences to those containing chains of semantically interacting events [8], similarly to the technique proposed for GUI-testing [12, 13]. To make the paper self contained, we now summarize how to obtain the FSM model and how to extract semantic event interactions which are turned into test cases.

3.1 Model extraction

We extract the FSM of an Ajax application through dynamic analysis, complemented by information coming from static code analysis. Dynamic analysis is by definition partial, hence a manual validation or refinement step may be required after model extraction, to ensure that the extracted model is not under-approximating the set of admissible behaviors.

The starting point for our dynamic analysis is a set of execution traces, such as the ones shown in Figure 1 for a hypothetical *Cart* application written in Ajax. *Cart* allows users to add and remove items from a cart, or to empty the cart. Execution traces may be obtained from log files generated by real user interactions, following an approach similar to the one proposed by Elbaum *et al.* [4]. We can also take

Trace	Event sequence
1	add
2	rem
3	add, add, rem
4	add, add, rem, rem, rem
5	add, empty
6	add, empty, rem

Figure 1. Traces for Cart (events only)

advantage of existing test cases, if any, produced by previous testing phases or in previous development iterations.

DOM element	Abstraction
`DIV`|`SPAN`|`P`	*null* | *empty*
`TEXTAREA`	*null* | *empty* | *notEmpty*
`OL`|`UL`	*null* | `#LI=0` | `#LI>0`
`TABLE`	*null* | `(#TD|#TR)=0` | `(#TD|#TR)>0`
`INPUT type=text`	*null* | *empty* | *notEmpty*
`A`	*null* | *notNull*
`LI`	*null* | *empty* | *notEmpty*
`SELECT`	*null* | *empty* | *sel=1* | ...
`INPUT type=text name=total`	*null* | *total=0* | *total>0*

Figure 2. Fragment of the default state abstraction function (top) and Cart-specific abstraction (bottom)

Traces contain information about the DOM states and the callbacks causing transitions from state to state (Figure 1 shows only the callbacks for space reasons). DOM states are abstracted from the concrete states by means of a state abstraction function, such as the one shown in Figure 2. Since the number of possible concrete DOM states is usually huge and unbounded, we do not represent them directly in the FSM model. We consider abstract states only instead, following an approach similar to the one implemented in the tool Adabu [3]. The state abstraction function shown in Figure 2 (top) is a default, generic function that can be used as a first approximation with any Ajax application. However, it is often necessary to refine or extend it with application-specific abstraction mechanisms. Figure 2 (bottom) contains one such example, where the text field *total* is known to actually contain a number, hence its abstraction can be defined upon the numeric value, being either 0 or greater than zero. On the contrary, the default abstraction of a text field is *null*, *empty*, or *notEmpty*. In *Cart*, *total* is always *notEmpty*, hence the default abstraction is not adequate. With regards to transitions, we represent those user events that have an effect on the DOM. All other method invocations have no effect on the DOM state, thus we can safely ignore them. We determine the set of methods reacting to events and possibly affecting the DOM by means of a static code analysis. The output of this analysis determines the set (actually, a safe superset) of the methods that need to be traced. The DOM state is logged after the invocation of each such methods.

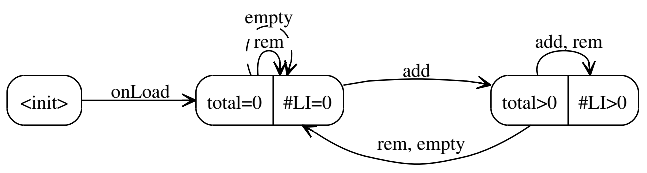

Figure 3. FSM for Cart application

Figure 3 shows the FSM obtained from the traces in Figure 1 by means of the state abstraction function in Figure 2. One admissible transition (dashed in Figure 3) is missing, the reason being that no execution trace gathered in this example covers it (dynamic analysis is partial), hence it was added manually to the model. Furthermore, the built FSM can be manually enriched by the user with conditions constraining transitions activation.

3.2 Semantic interactions -based Testing

Definition 1 (Semantically interacting events) *Events e_1 and e_2 interact semantically if there exists a state S_0 such that their execution in S_0 does not commute, i.e., the following conditions hold:*

$$\begin{gathered} S_0 \Rightarrow_{e_1;e_2} S_1 \\ S_0 \Rightarrow_{e_2;e_1} S_2 \\ S_1 \neq S_2 \end{gathered}$$

where S_0 is any state in the FSM of the Ajax application. We have a semantic interaction whenever the effects on the DOM state of the two callbacks c_1 and c_2, associated with e_1 and e_2, are not independent, i.e., swapping the order of execution brings the application to a different state. This definition can be easily extended to sequences of events, instead of single events, by just replacing e_1 and e_2 with two event sequences. In the *Cart* example, the two events *rem, add* interact semantically. In fact, the execution order $\langle$*rem, add*$\rangle$ results in a cart containing one item (state `#LI>0`). The sequence $\langle$*add, rem*$\rangle$ produces an empty cart (state `#LI=0`).

Two different criteria [8, 12, 13] can be used to generate suites of test cases based on semantically interacting events.

The first testing criterion (called *SEM*) is based on the concatenation of pairs of semantically interacting events. In the sequence $\langle e_1, \ldots, e_n \rangle$ obtained by applying this criterion, every pair of events e_i, e_{i+1} in the sequence (with i ranging between 1 and $n-1$) is a pair of semantically interacting events.

The second criterion (called *ALT*) considers the semantic interaction of the event sequence prefix with the last event in the sequence. In the sequence $\langle e_1, \ldots, e_n \rangle$ obtained by applying this criterion, the prefix $\langle e_1, \ldots, e_{n-1} \rangle$ interacts semantically (i.e., does not commute) with e_n.

Once event sequences are generated according to either of the criteria above, input values must be provided. To this aim, we take advantage of a database of input values, collected together with the traces that are used for model extraction. Values in the database are typed, so that it is possible to randomly pick-up one of the values available in the database for the particular data type required by a given callback. Each obtained event sequence is converted into a testing script that can be executed by Selenium[1]. The PASS/FAIL result of a test case execution is determined by: (1) checking the consistency of the concrete state sequence w.r.t. the related state sequence in the FSM model of the application; (2) checking any output value (e.g., DOM elements, their attributes and the textual content in the resulting page) against a manually provided oracle; (3) determining whether the application is crashed by the test case.

4 Hill-Climbing Testing

We propose a search-based test case derivation technique for Ajax (called *HILL*), based on the hill climbing algorithm. It uses an objective function (also called fitness) to evolve an initial population of test cases (a test suite) with the aim of producing eventually a test suite which optimizes the objective function (i.e., maximizes the fitness). Since hill climbing is a heuristic method, the final solution will be in general a local, not the global, optimum. Even if not globally optimal, the solution found through hill climbing may be a good approximation for the problem considered.

Hill climbing is a local search algorithm that, given an initial solution (e.g., pairs of semantically iteracting events), uses the fitness function to evaluate the neighborhood solutions. A solution is a neighborhood solution if it can be reached from the current one by applying small changes to it. In our case, a neighborhood solution is produced by concatenating a semantically interacting event at the end of an existing test case. Among the neighboring solutions which improve the fitness, the one with highest fitness improvement is selected and used to form the next solution. Then, the algorithm iterates, evaluating neighborhood solutions and selecting new ones, until no improving neighborhood can be found.

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

```
(st1)Input:
- $N_{max}$: max test suite size (default: 100)
- FSM: Finite State Machine model
- k: initial event sequence length (default: 2)
- $K_{max}$: max event sequence length (default: 100)
(st2)Output:
- S: optimized test suite
(st3)Initialization of the test suite  S
 S = semInteractEventSeqs(FSM, k);
 if sizeOf(S) > $N_{max}$
  S = randomSample(S, $N_{max}$);
 end if
(st4)Evolution of the suite from length  k to k+1
 repeat
  iniFit = computeFitness(S);
  $S_{best}$ = S;
  for each tc in S such that length(tc) == k
   m = getLastEvent(tc);
   for each n in getNextSemInteractEvents(FSM, m);
    tc' = tc + n;
    if sizeOf(S) < $N_{max}$
     S' = addTestCase(S, tc');
    else
     S' = replaceTestCase(S, tc, tc');
    end if
    if computeFitness(S') > computeFitness($S_{best}$)
     $S_{best}$ = S';
    end if
   end for
  end for
  S = $S_{best}$;
 until iniFit == computeFitness(S)
(st5)Repeat (st4) with k = k + 1 until:
 - no fitness improvement occurs:
   computeFitness(S) after (st4)
   gives the same value for k and for k + 1
 - $K_{max}$ iterations have been performed
   (k == $K_{max}$)
```

Figure 4. Hill-Climbing based test suite generation

Figure 4 shows the pseudo-code of the algorithm used by HILL to generate the sequences of semantically interacting events composing the final testing suite. The test suite S being generated contains initially short sequences of events obtained by concatenating pairs of semantically interacting events. By default, the length k of such initial sequences is 2, but any arbitrary value can be used. In some cases, the number of sequences may exceed the maximum test suite size N_{max}. In such cases, sequences are sampled up to the size N_{max}.

Then, the main iteration of the hill climbing algorithm is entered (step *st4*). The purpose of this iteration is extending event sequences from length k to length $k+1$, hence improving the test suite fitness. When no improvement can be achieved by means of extensions to length $k+1$, step *st4* terminates and the algorithm continues with step *st5*. In this step, k is incremented, in order to consider longer event sequences. With this new value of k, step *st4* is re-executed if the following conditions hold: (1) the maximum sequence length K_{max} has not been reached yet; and (2) the previous execution of step *st4* has successfully performed some fitness improvement. The algorithm terminates when the maximum length is reached or the fitness does not improve any more.

Let us now consider step *st4* in more detail. Each test case of length k is extended to length $k+1$ by concatenating at the end of it an event that interacts semantically (according to the *SEM* definition) with the last event in the test case. In this way a new test case of length $k+1$ is formed. It can be either added to the existing suite or it can replace the test case selected for extension, depending on the size of S, compared to N_{max}. The fitness of the resulting test suite (S') is then evaluated. If it is greater than the best extension considered so far, it is recorded as the current best improvement, into S_{best}. After examining all test cases and all possible extensions for each test case, the best possible improvement is made by replacing S with S_{best}. It should be noticed that S and S_{best} differ by exactly one test case: the one which, once extended, produces the highest fitness increase. The surrounding repeat-until loop (instructions 14-32) takes care of ensuring that all fitness improvement actions that apply to test cases of length k are actually made. After at most N_{max} iterations the loop terminates and step *st4* is over: all fitness improving extensions from length k to $k+1$ have been made. As described above, step *st5* is in charge of repeating this process at increasing event sequence length.

4.1 Fitness function

As the objective function, we propose three fitnesses based on the notion of test diversity [9]: *EDiv*, test suite diversity based on the execution frequency of each event that labels a transition in the FSM exercised by the test cases of the suite; *PDiv*, test suite diversity based on the execution frequency of each pair of semantically interacting events labeling FSM contiguous transitions exercised by each test case of the suite; and *TCov*, test suite diversity based on the FSM coverage reached by the test cases in the suite.

Test diversity is a notion related to the test distribution, which can be obtained by analyzing the execution frequency of specific software elements or artifacts (e.g., GUI events and program branches) for sets of test cases. In other words, test diversity is the degree to which a set of test cases executes a given software in diverse ways with respect to each other. HILL computes the diversity degree of a test suite according to the exercised events (*EDiv* and *TCov*) or pairs of semantically interacting events (*PDiv*).

In detail, given a test suite S, composed of a set of test cases based on semantically interacting sequences of events, its events-based diversity (*EDiv*) is computed as follows:

$$EDiv_{min}(S) = \sum_{tc \in \mathit{TCS}} \min_{tc' \neq tc} \sqrt{\frac{\sum_{e \in Events} (F_e^{tc} - F_e^{tc'})^2}{|Events|}}$$

$$EDiv_{avg}(S) = \sum_{tc \in \mathit{TCS}} \sqrt{\frac{\sum_{e \in Events} (F_e^{tc} - \overline{F_e})^2}{|Events|}}$$

where: TCS are the test cases of S; $Events$ are the FSM events that are exercised by the test cases; F_e^{tc} and $F_e^{tc'}$ are the execution frequencies of the event e in test cases tc and tc', while $\overline{F_e}$ is the average frequency of event e computed over the entire test suite S.

$EDiv_{min}$ measures how a testing suite is diverse by considering the minimun distance between the event frequencies of each test case and the event frequencies of the other test cases in the same suite. $EDiv_{avg}$ measures how a testing suite is diverse by considering the distance from the average frequencies.

For instance, in the *Cart* example, we can consider the three events add, rem, and $empty$, exercised by two test suites, $S1$ composed of two test cases $tc1 = \langle rem, add, rem \rangle$ and $tc2 = \langle rem, add, empty \rangle$, and $S2$ composed of two other test cases $tc3 = \langle rem, add, rem \rangle$ and $tc4 = \langle rem, add, rem \rangle$. The execution frequency of the events exercised by these test cases are: add=1, rem=2, and $empty$=0 for $tc1, tc3$ and $tc4$; while for $tc2$ add=1, rem=1, and $empty$=1. By considering these execution frequencies, the events-based diversities are $EDiv_{min}(S1) = 1.63$ and $EDiv_{min}(S2) = 0$; $EDiv_{avg}(S1) = 0.81$ and $EDiv_{avg}(S2) = 0$. Therefore, in this example the test cases composing the suite $S1$ are more diverse (i.e., more spread across the events of the FSM) than the test cases composing $S2$.

Similarly to the $EDiv$ measure, the diversity based on pairs of semantically interacting events ($PDiv$) can be measured as follows:

$$PDiv_{min}(S) = \sum_{tc \in \mathit{TCS}} \min_{tc' \neq tc} \sqrt{\frac{\sum_{(n,m) \in P} (F_{(n,m)}^{tc} - F_{(n,m)}^{tc'})^2}{|P|}}$$

$$PDiv_{avg}(S) = \sum_{tc \in \mathit{TCS}} \sqrt{\frac{\sum_{(n,m) \in P} (F_{(n,m)}^{tc} - \overline{F_{(n,m)}})^2}{|P|}}$$

where: TCS is the set of test cases of the suite, P is the set of pairs of events that interact semantically, $F^{tc}_{(n,m)}$ and $F^{tc'}_{(n,m)}$ are the execution frequencies of the pair (n, m) respectively in tc and tc'; $\overline{F_{(n,m)}}$ is the average execution frequency for the pair (n, m).

The third fitness function, *TCov*, is based on event coverage. It selects as the most adequate set of test cases those with the highest coverage of the FSM transitions. Note that this coverage measure is a special case of the events-based diversity, in which the frequency counter for each event is just a boolean value indicating if an event has been covered (exercised) by a given test case or not.

$$TCov(S) = |\bigcup_{tc \in TCS} CoveredTransitions^{tc}|$$

where: TCS is the set of test cases of the suite and $CoveredTransitions^{tc}$ the set of transitions of the FSM covered by the current test case tc.

5 Case Study

We conducted a case study experiment using two AJAX Web applications: Tudu[2] and Oryx[3]. Tudu is an application supporting management of personal todo lists. By using Tudu, the user can create, remove or change her/his todo lists and events, and can share them with other users. The application consists of more than 10k lines of codes written in Java/JSP and it uses a huge set of frameworks (e.g., Struts, Spring, Oro, Aspectj, Log4j, Velocity, Xalan, DWR). Oryx is an editor for modeling business processes in BPMN (Business Process Modeling Notation). It is distributed in three versions: demo, client and developers and, in the experiment, we just considered the client one. The application consists of around 200k lines of codes, written in Javascript/Java/JSP and it uses some libraries (Batik, HtmlParser, Xerces, Xalan).

We developed three tools to support the proposed testing technique: (1) *FSMInstrumentor*, a Javascript module able to trace the execution of the Web application under test; (2) *FSMExtractor*, a Java module to analyze the execution traces and build the application FSM; and (3) *FSMTestCaseGenerator*, a Java module to analyze the built FSM and generate test suites according to SEM, ALT, and $HILL$ (using five fitness functions: $EDiv_{min}$, $EDiv_{avg}$, $PDiv_{min}$, $PDiv_{avg}$ and $TCov$).

[2]http://tudu.sourceforge.net

[3]http://bpt.hpi.uni-potsdam.de/Oryx

5.1 Research questions and Used Metrics

The aim of the experiment is to investigate adequacy and effectiveness of $HILL$ in generating test suites, compared to those obtained by applying the SEM/ALT criteria. In detail, the experiment goal is to answer to the following research questions:

RQ1: *What is the test case distribution over sequence length k produced by HILL, with respect to SEM/ALT?*

RQ2: *What is the effectiveness in finding faults of HILL compared to SEM/ALT?*

RQ1 deals with the ability of each testing criterion to define test suites of a "reasonable" size and composed of test cases of different lengths *k*, including big values of *k*. A carefully generated test suite composed of a limited number of test cases of different lengths is potentially more adequate than a huge suite composed of test cases of small length. RQ1 can be answered by computing the composition of the testing suites in terms of length of their test cases (i.e., the number of test cases per length *k*). This helps in understanding how the different testing criteria explore the space of all test cases that can be generated for a given set of lengths *k*. We will evaluate the distribution of the generated test cases over their length k by plotting and visually inspecting the distribution plots for the test suites produced by $HILL$ and by SEM/ALT.

RQ2 deals with the effectiveness in finding faults. It can be answered by computing the number of faults revealed by each testing suite ($HILL$ vs. SEM/ALT) and comparing it with the number of faults that are known to be in the applications (e.g., injected faults). The effectiveness can be also evaluated by computing the fault detection capability ratio $rFDC(S)$:

$$rFDC(S) = \frac{\sum_{f \in F} \frac{|FR^f(S)|}{|TCS|}}{|F|}$$

where: $FR^f(S)$ is the set of test cases of the suite S that reveal fault f; F is the set of all (known) faults; and TCS is the set of test cases in the current suite of S.

5.2 Procedure

The following steps have been performed to execute the experiment for both AJAX applications:

1. we instrumented the source code of the target application by using *FSMInstrumentor*;

Id	Fault Description
Tudu	
1	The function for adding a todo duplicates the todo element
2	*Delete completed Todos* doesn't delete todos
3	Delete completed Todos delete also the last todo edited (if any)
4	*Advanced Add todos* doesn't work when the number of todos is greater than 3
5	*Advanced Add todos* doesn't work when the number of todos is greater than 1
6	*Delete todo* doesn't work when Delete completed Todos is clicked before add
7	*Delete current list* works only if add to do list is clicked two times
8	*Delete completed Todos* doesn't work if after the click, *quickDelete* is clicked in 5 seconds
9	*Delete todo* doesn't work correctly but, when it is selected, it performs the *edit todo* function
10	The *edit list* action works correctly but, during its execution, it deletes the same list
11	*Edit list* has an unexpected delay before to do its task. This leads to an unexpected behaviour if other operations are executed during that while
12	The todo ordering operation works correctly but if some todos *actionCheckbox* are checked, these todos are deleted
13	*Advanced Add todos* works correctly but, if a note is filled, an additional item is added to the list
14	The *Share list* add also an empty item to the todos list (when the *Hide* button is clicked)
15	*Advanced Add todos* works correctly but if a note is filled for a todo then its DOM-representation doesn't contain the *ActionDelete* button
Oryx	
1	The save function does not work correctly and reliably, when a new element is added to an already saved workflow it is not save correctly
2	The main canvas is not deactivated when a model export window is shown
3	The save function stores wrong information and properties about the saved workflow
4	The workflow stencil set does not work
5	The PNML-export function does not work, it is not correctly activated when the canvas is not empty
6	The exported file is not correctly generated, it is corrupted
7	The BPMN syntax-check does not work correctly, some kinds of element are not considered
8	Missing attributes: in the BPMN stencil set, some properties and assignments are missing
9	The export function is not action-independent
10	When export an existing workflow, its relative URLs to the stencil-sets are not correctly defined and built
11	Handle terminate end events and exception throwing in PNML-export function
12	Handling the task-flow in PNML-export function and syntax check

Table 1. Faults injected into Tudu and Oryx

2. we executed the instrumented application for a long execution time, with the aim of exercising at least each AJAX event of the application GUI. The execution has been traced in log files by *FSMInstrumentor*;

3. the recorded traces have been analyzed by *FSMExtractor* to extract the application FSM;

4. the built FSM has been analyzed by *FSMTestCaseGenerator* to extract suites of test cases based on SEM, ALT, and $HILL$ (considering all the proposed fitness functions);

5. we generated some mutated versions of the target application by injecting several faults (see below);

6. we executed each generated test suite with the aim of revealing the injected faults.

Injected Faults

Confidence in the results of the experimental study can be achieved only if the faults generated artificially resemble the real faults. In order to seed both Tudu and Oryx with faults as similar as possible to real faults, we retrieved a set of real failure descriptions from the bug tracking systems[4] of such applications and we manually analyzed them. Whenever possible, we reproduced such failures by seeding faults that we regarded as reasonable causes for such failures.

The result of this process is that 15 (state-based) failures have been injected into Tudu and 12 into Oryx. Some of them are briefly described in Table 1. For instance, the fault number 2 of Tudu introduces an error in the functionality *Delete completed Todos*, which prevents invocation of the Java function that actually deletes the completed todos from the database. This fault is a state-based fault since, after the injection, the click of the "Delete completed Todos" button does not change the initial state of the application (i.e., selected todos are not deleted). An interesting example of fault injected into Oryx is number 4. The injection of this fault implies that when the user selects the BPMN stencil set, some design elements (e.g., specific kinds of task) are not correctly loaded in the list provided to the user by the tool.

[4](Tudu) http://sourceforge.net/tracker/?group_id=131842&atid=722407 (Oryx) http://code.google.com/p/oryx-editor/issues/list

Criteria	Test cases length *k*											Total
	2	3	4	5	6	7	8	9	10	11	12	
Tudu												
SEM	73	385	2309	-	-	-	-	-	-	-	-	2767
ALT	92	71	63	61	46	6	-	-	-	-	-	339
$HILL_{EDiv_{min}}$	5	7	16	24	10	19	7	5	2	4	1	100
	17	57	72	36	18	0	0	0	0	0	0	200
$HILL_{EDiv_{avg}}$	2	11	32	32	19	4	0	0	0	0	0	100
	8	13	14	17	28	44	18	28	13	11	6	200
$HILL_{PDiv_{min}}$	19	42	17	6	4	1	4	4	0	2	1	100
	39	90	38	9	5	2	4	8	15	11	17	200
$HILL_{PDiv_{avg}}$	11	14	18	15	9	5	3	3	9	11	2	100
	4	16	26	39	37	13	14	8	15	11	17	200
$HILL_{TCov}$	12	16	14	25	7	16	9	0	0	1	0	100
	25	49	16	35	11	35	20	6	1	2	0	200
Oryx												
SEM	93	193	670	2081	-	-	-	-	-	-	-	3037
ALT	103	98	87	56	9	2	-	-	-	-	-	355
$HILL_{EDiv_{min}}$	2	8	21	24	14	10	11	6	4	0	0	100
	6	26	31	34	46	29	20	4	2	2	0	200
$HILL_{EDiv_{avg}}$	1	1	10	21	22	22	15	4	1	3	0	100
	56	36	44	16	14	16	9	7	1	1	0	200
$HILL_{PDiv_{min}}$	11	40	35	9	2	0	0	2	0	1	0	100
	104	66	18	3	2	0	7	0	0	0	0	200
$HILL_{PDiv_{avg}}$	28	18	25	6	10	8	5	0	0	0	0	100
	53	39	46	20	17	21	4	0	0	0	0	200
$HILL_{TCov}$	10	43	38	7	2	0	0	0	0	0	0	100
	107	70	14	9	0	0	0	0	0	0	0	200

Table 2. Test suite size

5.3 Results and Discussion

FSMExtractor analyzed the 28 and 17 execution traces used to exercise Tudu and Oryx respectively and available for producing their FSMs. The built FSMs contain 14 and 15 states and 37 and 39 transitions, respectively for Tudu and Oryx. No customization of the abstraction function was necessary for analyzing Tudu and build its FSM. A customization was needed for analyzing Oryx to model a `DIV` HTML tag filled with one or more element of the same type `DIV`. The `DIV` element is abstracted in terms of: "$null|empty(\#DIV = 0)|!empty(\#DIV > 0)$". Since the size of the initial suites considered in both applications is less than N_{max} the applied hill climbing algorithm is deterministic.

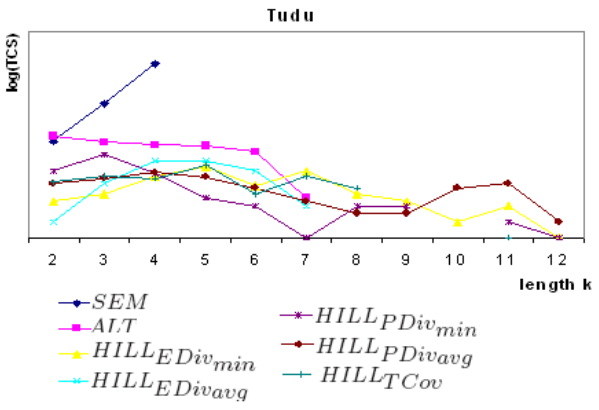

Figure 5. Test cases per k

RQ1: Testing suite size

As shown in Table 2, the test suite size for the $HILL$ criteria has been fixed (100, 200 test cases), so that a high size reduction is obtained for the $HILL$ test suites with respect to SEM (3%, 6%) and a moderate size reduction is obtained with respect to ALT (28.5%, 57%).

Table 2 shows also the number of test cases per testing criterion, divided according to their sequence length k. Figure 5 shows the plot obtained for Tudu, for the $HILL$ criteria. Only the results obtained for 100 test cases are plotted in the figure. They are plotted in a logarithmic scale to make them more readable. The test cases more evenly distributed across sequence lengths k are those generated

Criteria	Revealed Fault	rFDC (%)
Tudu		
SEM	10 (66.6%)	1.5
ALT	2 (13.3%)	0.2
$HILL_{EDiv_{min}}$	10 (66.6%)	3
	9 (60%)	1.9
$HILL_{EDiv_{avg}}$	6 (40%)	3.3
	6 (40%)	1.8
$HILL_{PDiv_{min}}$	5 (33.3%)	1.6
	8 (53.3%)	1.9
$HILL_{PDiv_{avg}}$	8 (53.3%)	2.4
	8 (53.3%)	2.2
$HILL_{TCov}$	9 (60%)	2.1
	9 (60%)	1.8
Oryx		
SEM	11 (91%)	1.7
ALT	7 (58.3%)	2.1
$HILL_{EDiv_{min}}$	5 (41.6%)	2.7
	9 (75%)	1.6
$HILL_{EDiv_{avg}}$	9 (75%)	3.1
	9 (75%)	1.8
$HILL_{PDiv_{min}}$	6 (50%)	2.5
	8 (66.6%)	1.9
$HILL_{PDiv_{avg}}$	6 (50%)	2.4
	8 (66.6%)	1.6
$HILL_{TCov}$	6 (50%)	3
	8 (66.6%)	2

Table 3. Revealed faults

by $HILL_{EDiv_{min}}$, $HILL_{PDiv_{min}}$ and $HILL_{PDiv_{avg}}$ for Tudu; $HILL_{EDiv_{min}}$ and $HILL_{EDiv_{avg}}$ for Oryx. The other criteria generate suites of test cases that are more concentrated in a specific subset of the considered *k*. For instance, for Tudu, $HILL_{EDiv_{avg}}$ generates test cases from length $k = 2$ to 7. Sometimes a smaller test suite size limit has the effect of removing shorter sequences early. This forces the selection of alternative (sub-optimal) sequences eventually resulting in longer test cases (e.g., $HILL_{EDiv_{min}}$ for Tudu 100 vs 200).

The use of the hill climbing algorithm seems to be effective in increasing the length of the test cases of the generated suite, while keeping the test suite size small. In our case studies, $HILL_{EDiv_{min}}$ produced better distributed test cases across lengths.

RQ2: Faults revealed

Table 3 shows the faults revealed by each testing suite. The number of faults revealed by $HILL_{EDiv_{min}}(100)$ is the same as those revealed by SEM for Tudu (66%). A comparable, but slightly worse performance (60%) is exhibited on Tudu by $HILL_{EDiv_{min}}(200)$ and $HILL_{TCov}$. On Oryx, all search based methods perform worse than SEM, with $HILL_{EDiv_{min}}(200)$ and $HILL_{EDiv_{avg}}$ getting quite close to it (75% faults revealed by $HILL$ vs. 91% faults revealed by SEM). On the other hand, the fault detection ratio (see column $rFDC$) is generally improved by search based methods, i.e., search based methods produce test suites containing a higher ratio of fault revealing test cases. This is especially true when the test suite size is 100. For example, $HILL_{EDiv_{min}}(100)$ produces suites with 3% of test cases revealing each fault (on average) for Tudu and 2.7% for Oryx, compared to 1.5% and 1.75% produced by *SEM* in the two respective cases. The fault detection capability is approximately doubled. On this metrics, $HILL_{EDiv_{avg}}(100)$ performs even better (3.3% and 3.1% respectively).

Overall Considerations

According to the overall result of the experiment, we conclude that $HILL_{EDiv}$ (both *min* and *avg* versions) is the most effective and adequate testing technique for generating test suites based on semantic sequences of events. Using $HILL_{EDiv_{min}}$ the generated test suites achieve a comparable (actually, slightly lower) number of revealed faults, with respect to *SEM*, with a substantial reduction of the test suite size (3% and 6% of the *SEM* suite size for the 100 or 200 test suites, respectively). $HILL_{EDiv_{avg}}$ is the one obtaining very high fault detection ratio in Table 3 for both AJAX applications. The reduced test suite size comes with an increased fault detection ratio, which makes it easier for the tester to reveal a fault early during the testing process.

Our experiment confirms the intuitions [8, 13] that: (a) length *k* and distribution over *k* of the event sequences taken into account for generating test cases deeply affect their fault revealing capability; and, (b) increasing the suite diversity and the number of long interaction sequences is a suitable strategy to improve the effectiveness of the automatically generated test suites.

Threats to validity

We considered only two applications, so our results cannot be easily generalized to arbitrary Ajax Web applications. However, the tested applications (Tudu and Oryx) are quite typical Ajax applications, in terms of their size, technology and the implemented functionalities. This means that the considered bugs, the recovered FSMs and the generated test suites have probably features that can be found in other similar applications. As happening with any experimental study in software engineering, it is only by replication that we can build a sound body of knowledge on this topic.

6 Related works

Several techniques and a few tools have been presented in the literature to support testing of Web applications.

Functional testing tools of Web applications (e.g., LogiTest, Maxq, Badboy, Selenium) are based on capture/replay facilities: they record the interactions that a user has with the graphical interface and repeat them during regression testing. Another approach to functional testing is based on HttpUnit. HttpUnit is a Java API providing all the building blocks necessary to emulate the behavior of a browser. When combined with a framework such as JUnit, HttpUnit permits testers to write test cases that verify the expected behavior of a Web application.

Model-based testing of Web applications was proposed by Ricca and Tonella [10]. Coverage criteria (e.g. page and link coverage) are defined with reference to the navigational model, i.e. a model containing Web pages, links and forms. Another proposal of model-based testing of Web applications was made by Andrews et al. [2]. In this case, the navigational model is a finite state machine with constraints recovered by hand by the test engineers directly from the Web application.

Testing of Web applications employing new technologies (e.g., Ajax, Flash, ActiveX plug-in components, Struts, Ruby on rails, etc.) is an area that has not been investigated thoroughly so far. A first step in this direction has been made in the Selenium TestRunner tool [6], which supports the "waitForTextPresent" condition, necessary whenever asynchronous messages received by the client determine the next execution step of a test case.

The work presented in this paper is based on the state-based testing approach, originally defined for object-oriented programs [11]. The approach was recently applied to GUI [13] and Ajax [8] testing. In fact, similarly to Ajax applications, GUI code is event driven and processing depends on callback execution. Hence, we share with the GUI testing techniques notions such as event sequences and semantically interacting events. However, Ajax applications have specific features (asynchronous communications and DOM manipulation) which make them different from GUI applications.

7 Conclusions and future work

We have proposed an improvement of the original state-based testing technique designed to address the features of Ajax applications. In the proposed technique, the DOM manipulated by Ajax code is abstracted into an FSM and sequences of semantically interacting events are extended by a hill climbing algorithm to generate testing suites of reasonably small size, comprising long interaction sequences and maximizing a specific objective function, focused on the test suite diversity.

The experiment that we have conducted confirms the intuition that longer interaction sequences have higher fault exposing potential. By selecting long sequences that maximize the test suite diversity, it is possible to keep their number small, with a limited degradation in number of revealed faults and a substantial increase of the fault detection ratio.

Our future work will be devoted to the improvement of the FSM recovery step, in order to automatically infer proper abstraction functions. We will experiment with alternative search based algorithms and we will apply them to a larger benchmark of Ajax applications. We will also investigate the role of input selection and infeasible paths in the FSM during test case generation.

References

[1] Document Object Model (DOM). *http://www.w3.org/DOM*.

[2] A. Andrews, J. Offutt, and R. Alexander. Testing Web Applications by Modeling with FSMs. *Software and System Modeling, Vol 4, n. 3*, July 2005.

[3] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *Proc. of the Fourth International Workshop on Dynamic Analysis (WODA)*, Shangai, China, May 2006.

[4] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher. Leveraging user session data to support web application testing. *IEEE Transactions of Software Engineering*, 31(3):187–202, March 2005.

[5] K. Gatlin. Trials and tribulations of debugging concurrency. *ACM Queue*, 2(7), October 2004.

[6] J. Larson. Testing ajax applications with selenium. InfoQ magazine, 2006.

[7] A. Marchetto, P. Tonella, and F. Ricca. A case study-based comparison of web testing techniques applied to ajax web applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(6):477–492, December 2008.

[8] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of ajax web applications. In *Proc. of IEEE International Conference on Software Testing (ICST)*, Lillehammer, Norway, April 2008.

[9] B. Nikolik. Test diversity. *Information and Software Technology*, 48:1083–1094, April 2006.

[10] F. Ricca and P. Tonella. Analysis and testing of Web applications. In *Proc. of ICSE 2001, International Conference on Software Engineering, Toronto, Ontario, Canada, May 12-19*, pages 25–34, 2001.

[11] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. *IEEE Conference on Software Maintenance (ICSM)*, September 1993.

[12] X. Yuan and A. Memon. Alternating gui test generation and execution. In *Proc. of IEEE Testing: Academic and Industrial Conference (TAIC PART)*, Washington, DC, USA, 2008.

[13] X. Yuan and A. M. Memon. Using GUI run-time state as feedback to generate test cases. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 396–405, Washington, DC, USA, May 23–25, 2007. IEEE Computer Society.