2008 International Conference on Software Testing, Verification, and Validation

State-Based Testing of Ajax Web Applications

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

Filippo Ricca
Unità CINI at DISI
16146 Genova, Italy
filippo.ricca@disi.unige.it

Abstract

Ajax supports the development of rich-client Web applications, by providing primitives for the execution of asynchronous requests and for the dynamic update of the page structure and content. Often, Ajax Web applications consist of a single page whose elements are updated in response to callbacks activated asynchronously by the user or by a server message. These features give rise to new kinds of faults that are hardly revealed by existing Web testing approaches.

In this paper, we propose a novel state-based testing approach, specifically designed to exercise Ajax Web applications. The Document Object Model (DOM) of the page manipulated by the Ajax code is abstracted into a state model. Callback executions triggered by asynchronous messages received from the Web server are associated with state transitions. Test cases are derived from the state model based on the notion of semantically interacting events. We evaluate the approach on a case study in terms of fault revealing capability. We also measure the amount of manual interventions involved in constructing and refining the model required by this approach.

1 Introduction

The Ajax technology allows developers to realize Web applications which include advanced, sophisticated user interactions, that are not possible with the traditional Web paradigm. In fact, the synchronous request-response protocol used by traditional Web applications introduces a delay between one page and the next one, which prevents all interactions requiring a short response time. On the contrary, Ajax applications are based on asynchronous requests, which leave the user interface active and responsive. Combined with the possibility to update a page dynamically through the DOM (Document Object Model, [1]), Ajax is a powerful Web development technology.

Web testing has long been recognized as a hard task [10, 2, 8, 16]. With the advent of Ajax, novel problems add to those already known in the Web testing area. The asynchronous protocol of Ajax introduces parallelism between client and server, which requires careful code writing. The dynamic page updates through the DOM may also introduce page elements (e.g., forms and submit buttons) which turn out to be sources of faults. An Ajax client page can change in response to user interactions or asynchronous messages received from the server. Moreover, the change may depend on the current page state (i.e., the state of the DOM).

Existing Web testing techniques [10, 2, 8, 16] are not designed to address the specific features of Ajax applications. We propose a novel, state-based testing technique which targets the main sources of problems of the Ajax technology. Our technique is based on the dynamic extraction of a finite state machine for a given Ajax application and its analysis in order to identify sets of "semantically interacting" event sequences, used to generate suites of test cases. We ran a case study to understand the effectiveness of the proposed technique compared to the existing ones and to assess the related costs (manual effort involved).

After presenting the related works, we give a brief overview on typical faults that can occur in Ajax (Section 3). In Section 4, we introduce our proposal of a technique for testing Ajax applications and the *core* notion of semantically interacting events. In Section 5, we describe the experiment conducted on a case study. Finally, we draw conclusions and sketch our future work in Section 6.

2 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.

978-0-7695-3127-4/08 $25.00 © 2008 IEEE
DOI 10.1109/ICST.2008.22

When combined with a framework such as JUnit, HttpUnit permits testers to write test cases that verify the expected behavior of a Web Application [12].

Liu et al. [14] extended traditional data flow testing techniques to Web applications. Model-based testing of Web applications was proposed by Ricca and Tonella [16]. 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. Elbaum et al. [9] proposed a Web application testing approach that utilizes data captured in user sessions, stored in a modified log file, to create test cases automatically. Di Lucca et al. [8] exploited a slightly different format for the specification of test cases, based on decision tables that include input values for each execution variant and expected outputs.

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 [13], 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 [5, 17], and then developed with the aim of selecting effective state-based test sequences [3, 6]. This approach was recently applied to GUI-testing [15, 18]. Similarly to Ajax applications, GUI code is event driven and processing depends on callback execution. Hence, we share with the GUI testing techniques [15, 18] notions such as event sequences and semantically interacting events. However, an Ajax application has specific features which make it different from traditional GUI applications. Namely, the client communicates with the server asynchronously and callbacks are activated both by user events and by server messages. Moreover, the interface itself is manipulated by Ajax code through the DOM, providing a reflection mechanism on the page structure.

3 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. 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 travelling 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.

Ajax can be used to implement on-the-fly form data validation, form-data auto completion and sophisticated GUI controls based on client-side component update (without page refreshing). The main technological novelty of Ajax is the asynchronous communication between client and server, combined with the reflection mechanism provided by the DOM, which allows updating parts of the current Web page without resending the entire page.

3.1 Ajax faults

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. Hence, Ajax testing should be directed toward revealing faults related to incorrect manipulation of the DOM, for example deriving from 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.

Asynchronous message passing is a well-known source of trouble in software development [11]. 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.

4 Ajax testing

Every 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 propose to model an Ajax Web 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 Web application, test cases can be derived according to available techniques for state-based testing [5, 17], such as state or transition coverage criteria. However, such approaches tend to generate a high number of test cases involving unrelated events. Hence, we investigate also the possibility to adapt GUI-testing approaches [15, 18] based on the notion of semantically interacting events. In the following, after clarifying how to obtain the needed model and how to determine semantic event interactions, we explain test case derivation and finally present some side-results, which take the form of asynchronism warnings.

4.1 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 is required after model extraction, to ensure that the extracted model is not under-approximating the set of admissible behaviors.

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)

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. [9, 10]. If the assumption can be made that state-based testing is part of a testing process that firstly goes through more traditional, black-box or coverage testing, we can take advantage of test cases produced by previous testing phases. Alternatively, a specific scenario execution activity should be planned.

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.

DOM element	Abstraction
`DIV | SPAN | P`	*null* | *empty*
`TEXTAREA`	*null* | *empty* | *notEmpty*
`FORM`	*null* | *notNull*
`OL | UL`	*null* | `#LI=0` | `#LI>0`
`TABLE`	*null* | `(#TD|#TR)=0` | `(#TD|#TR)>0`
`INPUT type=text`	*null* | *empty* | *notEmpty*
`INPUT type=button`	*null* | *notNull*
`A`	*null* | *notNull*
`IMG`	*null* | *notNull*
`LI`	*null* | *empty* | *notEmpty*
`INPUT type=radio`	*null* | *notNull*
`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)

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 [7]. The state abstraction function shown in Figure 2 (top) is a default, generic function that can be used as a first approximation with a given 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 *nonEmpty*, hence the default abstraction is not adequate.

With regards to transitions, we are only interested in method invocations triggered by user events or server messages and modifying 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 shows the FSM obtained from the traces in Figure 1 by means of the state abstraction function in Figure 2. We can notice that one admissible transition (dashed in Figure 3) is missing, the reason being that no execution trace gathered in this example covers it (button *empty* never

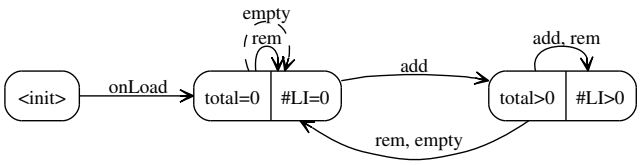

Figure 3. FSM for Cart application

clicked on already empty cart). In the general case, the FSM produced by dynamic analysis may miss some states and transitions which are never exercised in the considered scenarios. Hence, we devise a final, manual step in which the user augments the resulting FSM with missing states and transitions. The user may also drop states or transitions or redirect the latter, in case faulty executions are included in the set of considered traces. The amount of manual work required in this step depends on the number and quality of the available traces. A complementary way to improve the extracted FSM model consists of running the application under additional scenarios.

4.2 Semantic interactions

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:*

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

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.

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`). Hence, it makes a difference to execute these two events in sequence, rather then considering them independently, because they interact with each other.

The notion of pair of semantically interacting events can be easily generalized to sequences. The event sequence $\langle e_1, \ldots, e_n \rangle$ is a sequence of semantically interacting events if 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, according to Definition 1.

4.3 Test case derivation

Each sequence of semantically interacting events $\langle e_1, \ldots, e_n \rangle$ can be turned into a test case of length n (i.e., involving the execution of n callbacks). In order to make the test case executable, we need to provide input values whenever the callback execution triggered by an event uses some input coming from the user. For this purpose, 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.

Event sequences augmented with input values are executable test cases. We convert them into the XML format required by the Selenium tool[1] and we run them by means of Selenium. The PASS/FAIL result of a test case execution can be 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. We have an inconsistency between the concrete sequence and the FSM whenever the event e_i brings the application from a concrete state C_i to C_{i+1}, while there is no outgoing transition labeled e_i between S_i and S_{i+1}, respectively the abstractions of C_i and C_{i+1}. For every state and state transition the user may specify a set of constraints on the expected output values, in terms of state and transition invariants, as well as transition pre- and post-conditions. Finally, a raw check that is always part of any testing process is crash detection.

Event sequences of length 1 generate test cases satisfying the event coverage criterion, i.e., each event is exercised at least once by a test case. This criterion is strictly weaker than the widely used transition coverage criterion, which implies it. Event sequences of length 2 result in a test suite that exercises each pair of semantically interacting events at least once. This criterion is strictly weaker than the coverage of paths of length 2, which implies it. The rationale behind this criterion is that we can obtain a substantial reduction of the test suite size by considering only semantically interacting events, instead of any sequence of length 2, and, at the same time, we expect not to loose too much in terms of fault revealing capability of the test suite, since we are retaining all paths involving semantic interactions between events. Generalizing to sequences of length n, we can notice that sequences of n semantically interacting events are subsumed by paths of length n, so the proposed criterion is a way to make this path length affordable, thanks to the test suite size reduction, at limited or no performance loss.

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

4.4 Asynchronism warnings

As a side effect of the analysis required to derive test cases from the FSM model of the Ajax application, we can determine whether any asynchronism problem of the following two types is there in the code: *AsyWarn1* (swapped callbacks) or *AsyWarn2* (dependent requests). Given two semantically interacting events e_1, e_2, let r_1 and r_2 be the associated requests sent to the server and let c_1 and c_2 be the corresponding callbacks. The following execution sequences may occur:

Nominal (no reordering): $\langle r_1; c_1; r_2; c_2 \rangle$
AsyWarn1 (swapped callbacks): $\langle r_1; r_2; c_2; c_1 \rangle$
AsyWarn2 (dependent request): $\langle r_1; r_2; c_1; c_2 \rangle$

Nominal is the usually expected sequence, where each request is immediately followed by the associated callback. This sequence is tested by the test suite containing sequences of length 2. *AsyWarn1* occurs when c_2 starts before c_1. It produces an incorrect final state (S_2, under the transitions in Definition 1). The reasons for c_2 starting before c_1 may be network delays, scheduling of threads on the server (second thread terminates before first one starts), scheduling of callback activation on client (second callback scheduled before first one), etc.

AsyWarn2 occurs when the second event e_2 is triggered before activating the callback for the first event e_1. The final state may be correct or not, depending on r_2. If the request r_2 depends on the results produced by the execution of c_1 (for example, the request r_2 includes information from the DOM and this information is modified by c_1), r_2 is sent to the server with incorrect information. As a consequence, the final state may be incorrect or some output values may be different from the expected ones.

We can easily detect *AsyWarn1*, in that every sequence of semantically interacting events is potentially subject to this asynchronism problem, provided the events are associated with a server request followed by callback (message handler). We can detect *AsyWarn2*, by means of a dependency analysis which determines whether any data dependency exists between c_1 and r_2. If this is the case, swapping c_1 and r_2 may produce an incorrect state or output. Since both problems are detected before and independently of testing, we report them to the tester just as warnings on the source code.

Ajax programmers tend to be unaware or not to care too much about these asynchronism problems, probably because they work under the optimistic assumption that the time between e_1 and e_2 is much higher than the time to send r_1 to the server, compute the response, send it back to the client and run c_1, so that r_2 is never ready to be sent out before c_1 is executed. However, this assumption might be wrong for example in cases where the network is slow or the server is overloaded. Programmers can protect their Ajax application from such asynchronism problems by: (*AsyWarn1*) sequentializing the requests associated with semantically interacting events through a request queue; (*AsyWarn2*) disabling any GUI widget associated with a request that depends on a callback not yet executed (the widget is disabled when r_1 is sent to the server and it is enabled upon completion of c_1).

5 Experimental results

The object of the experiment is the well-known Web application Tudu[2], supporting Web-based management of personal todo lists. The user can create, remove or change her/his todo lists and can share them with other, registered users. Moreover she/he can manage the events associated with each todo list. The Tudu application consists of around 10k lines of codes, written in Java/JSP and it uses several frameworks (e.g., Struts, Spring, Oro, Aspectj, Log4j, Velocity, Xalan, etc.), among which DWR, which supports Ajax programming. The application can store its data both into an external database (e.g., MySql) or into the embedded database Hsqldb.

We developed three tools to support the proposed Ajax testing technique: *FSMInstrumentor*, *FSMExtractor* and *FSMTestCaseGenerator*. *FSMinstrumentor* can instrument the Ajax code of the application under test so as to save execution traces into a log file. Saved traces include information relevant for FSM construction, such as events triggering callback execution and concrete state of the Web page before and after callback execution. *FSMExtractor* analyzes the traces in the log file and applies a set of predefined or custom abstraction functions to produce the FSM model of the application. Finally, *FSMTestCaseGenerator* extracts sequences of events according to coverage criteria such as transition coverage or k-limited semantically interacting event sequences. A database of possible input values (divided by value type) is filled in automatically with the input values collected in the traces. The user can then augment the database with additional, custom values. *FSMTestCaseGenerator* uses this database to complete the sequences of events with the input data necessary to construct a complete test case. Finally, test cases are converted into the format accepted by the tool Selenium.

5.1 Research questions

The main aim of this experiment is to study the advantages, effectiveness and effort involved in the proposed testing technique, compared to test cases generated according

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

to the FSM path coverage criteria. We also consider the test cases that are produced during functional testing, for further comparison. The experiment goal is to address to the following research questions:

RQ1: *What is the size reduction achieved by sequences based on semantic interactions, compared to FSM coverage-adequate sequences having the same length?*

RQ2: *What is the fault revealing capability of test sequences based on semantic interactions, compared to FSM coverage and functional test suites?*

RQ3: *How many traces are necessary to construct an accurate state model?*

RQ4: *What manual refinement steps are necessary to get executable test cases?*

RQ1 can be answered by computing the ratio between the number of test cases that are generated to exercise all semantically interacting sequences of a given length k and the number of test cases that are generated to cover all paths of length k in the FSM. For **RQ2** we measure the number of faults revealed by each considered testing technique (the fault seeding method used in this study is described below). To address **RQ3** we make the assumption that only a fraction of the total number of traces be available at a given time and we build the FSM from such incomplete set of traces. The resulting FSM is compared with the complete one, to determine how many states and transitions are indeed missing. The plot of the degree of completeness (coverage) of the extracted FSM vs. the number of traces considered gives an idea of the rate by which the final, accurate FSM can be obtained.

Finally, to answer **RQ4** we classify the manual refinement interventions that are potentially required into three categories:

Input values: Whenever the database of input values collected automatically from the traces is regarded not representative enough, the user can add further values.

Oracles: Each abstract state of the FSM can be automatically translated into a Selenium assertion on the expected DOM state. Another assertion that can be easily generated automatically simply checks that the application does not die unexpectedly with an error (crash test). However, it may be the case that the resulting assertions do not capture completely the expected behavior of the application when reaching a given abstract state. In such a case, the user can add custom assertions for verification by Selenium.

Relationship between DOM elements and callbacks: Knowledge of the DOM elements directly or indirectly accessed by each callback is required to produce the asynchronism warning *AsyWarn2* (dependent request). The relationship between DOM elements and callbacks can be approximated by a static or a dynamic code analysis which determines the related data flows. However, such an analysis is expected to be approximate, either because it uses partial, dynamic information, or because it is based on some over-conservative, static computation of the DOM element uses/definitions made by Ajax code. In fact, in the general case it is not decidable whether a variable depends on a given element of the DOM or not, because DOM access may be dynamic. Hence, the automatically computed relationship between DOM elements and callbacks might need to be refined manually.

5.2 Experimental procedure

The main steps performed during the experiment are:

FSM extraction

In order to produce the execution traces used by *FSMExtractor* to build the FSM, we considered a number of execution scenarios corresponding to the functionalities of the Tudu application as described in its Web page. For each described functionality, the application was exercised under all alternative conditions in the documentation available online. A side effect of this phase is the construction of a set of execution cases that can be easily turned into a functional test suite for the application under test. In fact, in this phase the application is regarded as a black-box and the criterion for test case derivation is coverage of all functionalities described in its documentation. We ended up with 28 execution scenarios, turned into a functional test suite executable by Selenium.

The traces are analyzed by *FSMExtractor* and the corresponding FSM is built. Then, the user validates it. In fact, the resulting FSM is produced by a dynamic analysis, which may under-approximate the complete FSM. Hence, missing states or transitions must be added manually.

Test cases generation

Semantically interacting sequences of given length K (for short, semantic sequences) are extracted by *FSMTestCaseGenerator*, which looks for pairs of events that do not commute (in terms of resulting DOM state). For comparison, all paths of the same length K, starting at the initial node of the FSM, are also extracted by means of *FSMTestCaseGenerator*.

Before converting the extracted sequences into test cases executable by Selenium, the user may have to help the tool with additional input values and refined oracles, as described above. Once this manual step is over, test case generation can proceed automatically and semantic or path-coverage adequate sequences can be turned into Selenium test cases.

Id	Fault Description
1	The "maxLength" property is not specified for the *Description* field (of the *todos add* operation). Thus, a user should be able to enter values longer than database layer allows it to be
2	The current list cannot be shared with other users
3	*Adding todo list* does not work since the enter key does not save new todo list. The *Add new Todo List* form will disappear correctly when the enter button is clicked but the new list is not saved
4	Non-numeric priority values are silently swallowed
5	field *Description* of the add todo action is cleaned (i.e., empty) before that the add todo task is done
6	*Advanced Add todos* works correctly but, if a note is filled for a todo, its DOM-representation contains an *actionDelete* button in addition to the correct set of buttons
7	The function for adding a todo duplicates the todo element
8	*Delete completed Todos* doesn't delete todos
9	Delete completed Todos delete also the last todo edited (if any)
10	*Advanced Add todos* doesn't work when the number of todos is greater than 3
11	*Advanced Add todos* doesn't work when the number of todos is greater than 1
12	*delete todo* doesn't work when Delete completed Todos is clicked before add
13	*delete current list* works only if add to do list is clicked two times
14	*Delete completed Todos* (i.e., it is activated by the sequence of events: click into actionCheckbox and quickDelete) doesn't work if after the first click the user click the quickDelete link in less than 5 seconds)
15	*delete todo* doesn't work correctly but, when it is selected, it performs the *edit todo* function
16	the *edit list* action works correctly but, during its execution, it deletes the same list
17	*edit list* has an unexpected delay before to do its task. This leads to an unexpected behaviours if other operations are executed during that while
18	the todo ordering operation works correctly but if some todos *actionCheckbox* are checked, these todos are deleted
19	*Advanced Add todos* works correctly but, if a note is filled for a todo, an item without description is added to the todos list in addition to the current todo with the note (that is correctly created)
20	the *share list* add also an empty item to the todos list (when the *hide* button is clicked)
21	*Advanced Add todos* works correctly but if a note is filled for a todo then its DOM-representation doesn't contain the *actionDelete* button

Table 1. Faults injected into Tudu

Fault-injection
In order to seed Tudu with faults as similar as possible to real faults, we retrieved a set of real failure descriptions from the bug tracking system of TuDu 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 described in Table 1. We produced 21 different versions of Tudu (with one fault injected into each version). The faults injected are of two types: faults that modify the correct sequence of states in the FSM (*state-based faults*) and faults that don't modify it (*generic faults*). In the second category are included, for example, faults related to form construction and to input validation. The first six faults in Table 1 belong to the *generic fault* category, while the others are *state-based faults*. An example of a fault in the second category is number 8, which introduces an error during deletion of completed todo items from the database, resulting in a missing state transition.

Test case execution
The Selenium test cases produced by *FSMTestCaseGenerator* are executed automatically on all 21 faulty versions of Tudu. Any failed test case (violated assertion) is counted as a fault revealed by the executed test case.

	Test Cases		
Test Criterion	**Sequences**	**Semantic Sequences**	**Ratio**
Functionality coverage	27	*13*	*0.48*
FSM coverage	56	*13*	*0.23*
k=2	431	95	0.22
k=3	3616	623	0.17
k=4	29076	4428	0.15
k=5	223500	29641	0.13

Table 2. Semantic sequences and reduction

5.3 Results

FSMExtractor analyzed the 28 execution traces available and produced an FSM for Tudu containing 14 states and 59 transitions, using the default abstraction function (no customization was necessary for Tudu). *FSMTestCaseGenerator* produced test sequences with length between 2 and 5. After manual refinement, these test cases, as well as the functional ones, were executed by Selenium.

RQ1: Size reduction

Table 2 shows the number of test cases per test criterion and the ratio between semantic and non-semantic sequences. The column "Sequences" in this table shows that the number of path-coverage adequate test cases grows exponen-

tially with the length k. Such a number becomes unmanageable after k=5. Functional test cases are associated with sequences scattered among quite different lengths, in the range 1-11. Specifically, 13 functional test cases have length greater than k=5. The "FSM coverage" criterion indicates transition coverage (i.e., k=1). Column "Semantic Sequences" shows the number of sequences that have been recognized as semantic ones. The size reduction (i.e., *1-Ratio*) is in the range 78% - 87% across the different test criteria.

RQ2: Faults revealed

Table 3 shows the faults revealed by each technique, at varying sequence length, up to 4 (due to time constraints, it was not possible to run the test cases generated for k=5).

	Coverage		Semantic Sequences		
Id	**Funct.**	**FSM**	**k=2**	**k=3**	**k=4**
Generic Faults					
1	y	y	y	y	y
2	y	n	n	n	n
3	y	y	y	y	y
4	y	n	n	n	n
5	y	n	n	n	n
6	n	n	n	y	y
Tot	5	2	2	3	3
State-based Faults					
7	y	n	n	n	n
8	y	n	y	y	y
9	n	n	n	n	n
10	n	n	n	n	n
11	y	n	n	n	y
12	y	n	n	n	y
13	y	y	y	y	y
14	y	n	y	y	y
15	y	y	y	y	y
16	y	y	n	y	y
17	y	n	n	n	n
18	n	n	n	n	n
19	n	n	n	n	y
20	n	y	n	y	y
21	y	n	n	y	y
Tot	10	4	4	7	10

Table 3. Revealed faults

Let us consider the generic faults first. There is no big difference between test cases extracted according to the FSM coverage criterion and the semantic sequences based on k=2,3 and 4. In fact only one fault (number 6) has been found using k=3,4 rather than k=2 and FSM coverage. The reason is that this specific bug involves at least a sequences of three different GUI-events (e.g., click on the "Advanced add" link, fill-in the note, confirm the todo creation). Hence, this bug cannot be revealed with k=2 or FSM coverage. Bugs number 2, 4 and 5 have not been found by test cases based on FSM coverage and semantic sequences. The reason is that these bugs do not affect the state of the DOM. Since the automatically generated test oracles are based on the FSM states that abstract the DOM state and no further assertion was added by the user, the test oracles were unable to reveal these faults. To detect them, a more fine-grained set of oracles should be used. For instance, bug number 5 creates a todo with an empty description, but the used FSM abstraction function skips the todo item description altogether (it is abstracted away completed).

If we consider the state-based faults, we can notice that an increasing length of the test sequence is associated with an increased capability of exposing faults. Going from k=2 to k=4 we observe an additional number of revealed faults ranging from 1 to 6, compared to the faults revealed by FSM coverage. Note that functional coverage testing and k=4 have been able to detect the same number of bugs. Indeed, they found different sets of bugs. Bugs number 7 and 17 have been found only by functional coverage tests, while bugs number 19 and 20 only by test cases based on semantic sequences of length k=4 and k=3 respectively. Hence, functional coverage and semantic interactions appear to be complementary testing techniques, that could be effectively combined to achieve improved testing performance.

All semantically interacting sequences generated by *FSMTestCaseGenerator* (see Table 2) give raise to an asynchronism warning of type *AsyWarn1*. Among such sequences, only those associated with a data flow from one event (callback) to the next request-event give raise to an asynchronism warning of type *AsyWarn2*. They are respectively 66 sequences for k=2, 497 for k=3 and 3861 for k=4.

Let us consider the 95 sequences of length k=2. Since they are semantically interacting, swapping the order may produce a wrong output. Programmers should be aware that Ajax does not ensure proper ordering automatically. Moreover, the 66 sequences with a data flow between one event and the next one may produce incorrect requests if the two events are swapped, since the DOM elements shared by the two events may contain values not yet updated by the first callback but used in the second request.

RQ3: Traces needed for FSM construction

Fig. 4 shows the states and transitions added to the FSM by *FSMExtractor* as a function of the number of traces considered. We can notice that the two curves reach a plateau around trace number 22. No further state is added later; only a few transitions are missing and can be detected when running the last few traces.

The shape of these curves gives us an idea of the manual effort involved when the number of traces is insufficient to build a complete FSM. If, for example, only 14 traces (half of the total) are available, the resulting FSM will have 10

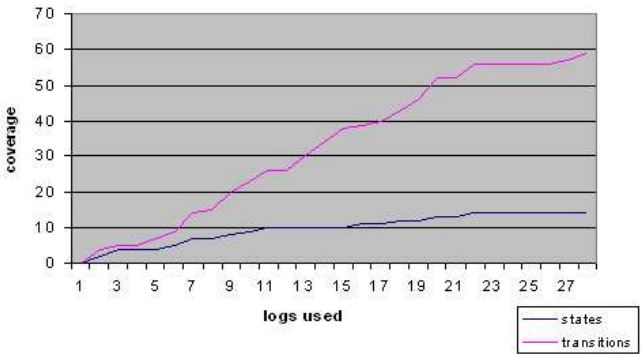

Figure 4. States and transitions added to FSM by number of traces

DOM-List element	Condition	Assertion
todoListsMenuBody	empty	waitForNotText
todoListsMenuBody	not-empty	waitForText
listOfTudu	empty	waitForElementNotPresent
listOfTudu	not-empty	waitForText

Table 4. Examples of rules used to convert FSM states into Selenium assertions

states instead of 14, hence 4 must be added manually, and 30 transitions instead of 59, hence 29 are missing. Usually, it is relatively easy to discover states, while transitions may be associated with exceptional or rare conditions. Correspondingly, most of the FSM refinement effort is related to adding missing transitions. In our case study, when all 28 traces are used no further refinement is necessary.

RQ4: Manual refinement steps

The manual interventions potentially required to turn event sequences into executable test cases include: (1) extend input values collected from traces with additional ones; (2) extend assertions on expected states with assertions on expected behavior; (3) associate callbacks to DOM elements, to enable generation of the asynchronism warning *AsyWarn2* (dependent request). In our case study, no manual intervention was necessary for (1) and (2), since automatically collected values and automatically computed assertions were judged satisfactory. (3) required some manual work, because none of the static or dynamic analyses potentially useful to approximate the association between callbacks and DOM elements is yet available, hence such an information was produced entirely by hand.

Input values were obtained from the execution traces,

Callback	DOM elements
menuAddList	listName
menuShareList	userName
menuEditList	listName
advancedAdd	listName, priority, note, date
actionEdit	listName, priority, note, date
quickAdd	listName

Table 5. Associations between callbacks and DOM elements

which were considered representative enough of the various running conditions, so that no additional value was manually inserted into the database. Assertions were derived from the states reached in the FSM whenever a given sequence of events is applied. Abstract states were turned into conditions on the concrete states by means of the same abstraction functions used during FSM extraction. The resulting assertions could be easily mapped onto Selenium assertions, as shown in Table 4 for some examples. Crash test assertions were also generated automatically.

Knowledge about the dependency between callbacks and DOM elements is currently not extracted automatically by our tools, although it would be possible to get an approximation of such information by means of static or dynamic code analysis. Consequently, we defined manually the DOM elements accessed by each callback. We ended up with the associations shown in Table 5. The manual effort involved in the construction of this table was quite limited, as apparent from the size of the table.

5.4 Discussion

In summary, the results obtained on the Tudu case study show that:

RQ1: Substantial test suite size reduction is achieved by considering semantically interacting sequences instead of all sequences of a given length k. In particular, with k=2 semantic sequences are just 22% ("Ratio") of all sequences of the same length.

RQ2: Semantic sequences have high fault exposing power. They overcome the results obtained by the commonly used state transition coverage criterion. Moreover, with k=4 they give results comparable to those obtained from functional testing, which is known to be extremely effective [4].

RQ3: A quite accurate FSM model may require as many as 22 traces for an application comparable to Tudu, for which the complete FSM was built out of 28 traces. If only a fraction (say, half) of them is available, most states are anyway identified, while a comparable fraction (say, half) of transitions is discovered, the rest requiring manual addition.

RQ4: Most refinement steps necessary to turn event se-

quences into executable test cases can be automated partially or completely. The remaining manual effort was negligible in our case study.

Of course, it is hard to generalize these findings to other Ajax applications, especially if they depart from the characteristics of Tudu. However, the preliminary results we obtained on Tudu are very encouraging and indicate a strong potential for semantic sequences to reveal a substantial amount of faults in Ajax code. Moreover, the level of automation that can be reached in this kind of testing is extremely high, so that we think that our approach can be integrated into the testing phase with only minor impact on the testing effort. One key factor in the construction of an accurate FSM and in the creation of a representative database of input values is the quantity and quality of the traces available. One can easily imagine how a large number of traces could be produced, either by capturing real user sessions (assuming the client runs instrumented Ajax code), or by replying user sessions on an instrumented client in the testing environment. Such a configuration should allow testers to gather enough data for accurate FSM construction and test case generation.

6 Conclusions and future work

We proposed a state-based testing technique specifically designed to address the features of Ajax Web applications. In this technique, the DOM manipulated by Ajax code is abstracted into an FSM and sequences of semantically interacting events are used to automatically generate test cases.

Our case study shows that semantic sequences are a fraction of all sequences of the same length, which makes it possible to exercise the application with long interaction sequences. The fault exposing power of semantic sequences grows with their length. In our case study it overcame state transition coverage at sequence length k=2 and it became comparable to functional testing at k=4. Indeed, sequences of length 4 revealed different sets of faults, thus suggesting that our technique should be used as a complement to functional testing. The effort involved in FSM construction and test case generation was pretty small in our case study. Actually, a lot of steps can be partially or completely automated, leaving just some minor work to be done manually.

In our future work, we will investigate techniques to support the generation of even longer sequences. Our intuition is that longer interaction sequences have higher fault exposing potential. We will also proceed with the implementation of the tools, which still lack some data flow analysis required for asynchronism warning 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] T. Ball, D. Hoffman, F. Ruskey, R. Webber, and L. White. State generation and automated class testing. *Software Testing, Verification and Reliability (STVR)*, 10(3):149–170, July-Aug 2000.

[4] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, 13(12):1278–1296, December 1987.

[5] R. Binder. State-based testing. *Object Magazine*, July-Aug 1995.

[6] K. Bogdanov and M. Holcombe. Statechart testing method for aircraft control systems. *Software Testing, Verification and Reliability (STVR)*, 11(1):39 –54, July-Aug 2001.

[7] 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.

[8] G. A. Di Lucca, A. R. Fasolino, F. Faralli, and U. D. Carlini. Testing Web applications. In *Proc. of the International Conference on Software Maintenance (ICSM)*, Montreal, Canada, October 2002. IEEE Computer Society.

[9] S. Elbaum, S. Karre, and G. Rothermel. Improving Web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 49–59, Portland, USA, May 2003. IEEE Computer Society.

[10] 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.

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

[12] E. Hieatt, R. Mee, and G. Faster. Testing the web application engineering internet. *IEEE Software*, 19(2):60–65, March/April 2002.

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

[14] C.-H. Liu, D. C. Kung, P. Hsia, and C.-T. Hsu. An object-based data flow testing approach for web applications. *International Journal of Software Engineering and Knowledge Engineering*, 11(2):157–179, April 2001.

[15] A. M. Memon. An event-flow model of gui-based applications for testing. *Software Testing, Verification and Reliability*, 2007.

[16] 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.

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

[18] 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.