Automatic Ajax application testing

Sébastien Salva
LIMOS - UMR CNRS 6158
Université d'Auvergne, Campus des Cézeaux,
Aubière, France
Email: salva@iut.u-clermont1.fr

Patrice Laurençot
LIMOS - UMR CNRS 6158
Université Blaise Pascal, Campus des Cézeaux,
Aubière, France
Email: laurencot@isima.fr

***Abstract*—Asynchronous javascript and XML (AJAX) is a recent group of technologies used to develop dynamic web pages. Ajax applications are wisely used nowadays and need to be tested to ensure their reliability. This paper introduces a method and an architecture for automatic AJAX application testing. We use STS automata for describing the application and for generating test cases. We perform an improved random testing using some predefined values and also test purpose based testing for verifying specific properties. The testing framework is composed of several testers which control and monitor the test execution to give a test verdict. The Google map search application is used as an example to illustrate the method.**

Key-words: conformance testing, Ajax application, test architecture

I. Introduction

Anyone, surfing the Internet, has already encountered and used Ajax (Asynchronous JavaScript and XML) applications. Ajax gathers a group of technologies which have radically improved web document/interface sight and features. Instead of reloading the web document after each user event, Ajax offers the advantage to perform server calls and document upgrades in the background, without reloading. This improvement makes possible the addition of rich client applications inside the web document. Ajax applications are usually composed of Javascript code and DOM (Document Object Model) which splits the web document into a hierarchical tree of objects (forms, maps, html panel,...). Interoperability, which is an essential capability in web development, is ensured by the XML serialization of messages passing through the network (x of Ajax).

Ajax is a major breakthrough in the web development area even though it is rather complex to set it up. Indeed, Ajax applications are distributed, handle new kind of objects (DOM) and achieve the interoperability capability with XML. Due to this melting pot of techniques, Ajax applications require a concrete software life cycle and especially an intense testing step. This one is absolutely required to trust final Ajax applications, and is now well integrated into software development compagnies. Although testing distributed or object oriented softwares is not new, Ajax testing poses new challenges, mainly on account of the use of DOM, of the serialization using XML patterns and of the application allocation on both server and client sides. This is why Ajax applications are often tested by hand. And of course, this is difficult, heavy and costly in time.

So, testing automatically Ajax applications rises some issues that we try to solve in this paper. In a first part, we propose to conceive Ajax applications like grey boxes where most of the interactions (events, XML messages, DOM modifications) are observable. To specify them, we use the STS (Symbolic Transition System) model and theory [1] since STS are easier to handle for generating test cases and well-adapted for modelling communicating systems.. We propose that Ajax specifications are composed of several STS, one denoted STS_{DEF}, for describing its whole behavior and others for illustrating specific executions. The use of STS_{DEF} brings the advantage to perform a random testing approach that we have improved with the use of predefined values. The other STS are seen as test purposes which complete the testing process by executing the Ajax application with specific values in the aim of testing specific properties of the application. Then, we describe the test case generation over the $ioco$ implementation relation, based on the STS theory. In the last part, we introduce a new test platform which makes possible automatic Ajax testing without the need of any user interaction.

While defining this testing method, we have developed a prototype composed of several parts: a first tool *UMLtoSTS* is dedicated to the graphical modelling of Ajax application into STS. The second tool *ATP* (Ajax Test Platform) corresponds to the tester which generates test cases on the fly and executes them.

This paper is structured as follows: section II provides an overview of Ajax technologies and of some related works about Ajax testing. In section III, we express the Ajax application modelling with STS. Section IV describes the testing method: we detail how are generated the test cases and introduce a testing framework. Finally, section V gives some perspectives and conclusions.

II. Ajax Overview

A. Ajax application functioning

Ajax (asynchronous JavaScript and XML) is a group of web development techniques used for creating interactive web applications. The "classical" web application model is synchronous, which means that a user activity implies the call of the web server synchronously, the modification and the reloading of the overall content displayed in the navigator. With Ajax technologies, an event triggered by a user leads to one or more calls to the web server in the background (the

user is not aware of these calls) in order to update a web content part, without reloading it. So, while using the same web page, a user may provoke several calls and modifications of the same page.

For instance, Figure 1 illustrates the Google Map Ajax search [2] which aims to search locations in a map. If a list of locations are found, these ones are marked inside the map without reloading the web document. Otherwise, the map is cleared. Ajax applications are often composed of both client

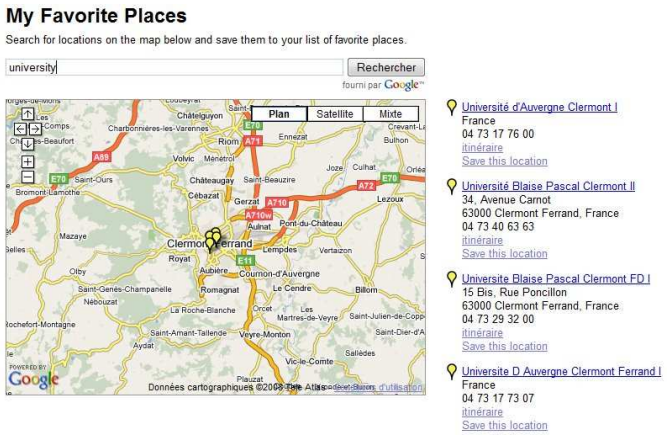

Fig. 1. Google Map Search

and server applications. The first one calls the server once an event has been triggered and then modifies the web document with received data. These calls can be synchronous (the client calls the server and waits for a response) or asynchronous (the response may be delayed). For asynchronous calls, we suppose, in this paper, that the response is correlated with the request.

The client side is usually composed of:

- DOM (Document Object Model) + javascript code to send and receive messages and to handle the displayed document, seen as a hierarchical structure of objects (forms, input labels, div,...) by mean of DOM. Usually, the code is gathered in a javascript function, launched when an event (Onmouse, Onkeyboard,...) is triggered. Data located in an initial DOM object (a form for instance) are used to construct the request sent to the server. When the client receives a response, it updates the web document by modifying one or more DOM objects,
- HTML+CSS codes to display the web document in a navigator.

The server application role is to answer to the client requests. Request and response messages are serialized into XML or JSON to ensure interoperability. The serialization pattern is given by XML schema.

Ajax application design is often done with UML sequence diagrams which show all the actors (user, client navigator, web server,...) and their interactions. An example, for the Google map search, is illustrated in Figure 2. This diagram describes the overall functioning of the application. The DOM object "location", which is a form here, gathers the input string provided by the user, the event which starts the Ajax function once triggered, and the name of the Ajax function. This one performs: the serialization of the string, the call of the web server, the receipt of the response, its deserialization, and the modification of the DOM object "map". Two responses may be received: if the location is found, the response is composed of the markers (longitude, latitude, name) which must be set on the map, otherwise the message is empty and all of the existing markers are cleared.

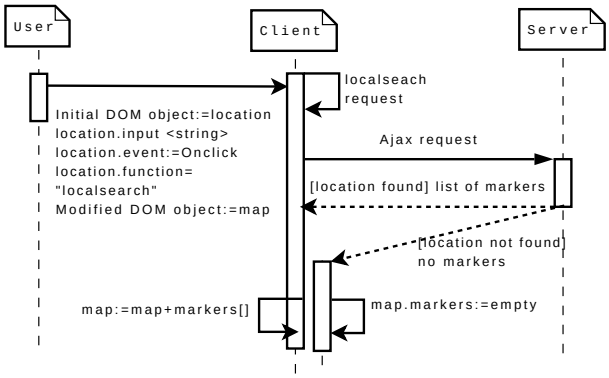

Fig. 2. UML sequence diagram

B. Related work on Ajax testing

Ajax testing rises new issues which have been tackled by some works in [3], [4], [5], [6]. Ajax applications can be tested with unit testing approaches and tools like Junit or HttpUnit. In [3], [4], solutions and tools are proposed to take into account timeouts in order to wait for the web content update while testing. A timeout based solution is implemented in [3], and a *WaiforCondition* in [4]. The testing process is still done manually. In [5] and [6], Ajax applications are modelled by state-based automata, where transitions represent callbacks modifying the web page. Test cases are made by covering the specification with the use of some criteria such as the state reachability or the path coverage.

Our approach is rather similar to [5] in the sense that we represent Ajax applications with STS to generate test cases. However, we consider both client and server sides, we use an STS to perform random testing and other STS as test purposes. We also consider the message serialization and we propose a testing framework for automatically executing test cases. So, our approach can handle random and predefined values, can detect faults on XML messages and can generate and execute test cases automatically. The timeout notion [3], [4] is considered here by mean of the quiescence property.

III. Ajax application modelling with STS

Different kind of approaches may be developed for Ajax application testing. In black box testing, the overall application is seen as a black box which receives user events and parameters, and which produces new DOM modifications inside the web document. Although this approach simplifies

the testing process, the server side and the observability of messages passing between both the client and the server are lost. So, we suggest a grey box testing method, where the Ajax application structure is known. Moreover, this approach agrees with usual UML specifications, which illustrate all the actors of the application.

To formally write Ajax application specifications, we use automata instead of UML sequence diagrams for the reason that automata are easier to handle for generating test cases. Since Ajax applications are composed of events, parameters and XML schema, an automaton model which allows to handle a variable set is required. We have chosen the STS (Symbolic Transition System) model [1] with its theory which especially defines the $ioco_F$ implementation relation and an algorithm to construct test cases.

An STS automaton $< L, l_0, V, v_0, I, S, \rightarrow >$, mainly based on the well known LTS model (labelled transition system) is composed of an alphabet of input and output symbols S: an input begins by "?" and is given to the system, whereas an output symbol, beginning by "!", is observed from the system. It is also composed of a variable set V, initialized with v_0, which can be updated while firing a transition of $\rightarrow$. Each transition $(l_i, l_j, s, \varphi, \varrho) \in \rightarrow$ from the location l_i to l_j labelled with the symbol s, may have a variable update ϱ and a guard φ on V which needs to be satisfied to fire the transition.

In Ajax applications, messages, passing between the client and the server, must be serialized according to XML schema. To express this with STS, we define a mapping $\Delta : \chi \times \varsigma \rightarrow \{true, false\}$ where χ is the XML message set and ς is the XML schema set. $\Delta(m, s)$ returns either $true$ if $m \in \chi$ is structured as described in $s \in \varsigma$, or $false$ otherwise.

Furthermore, we denote some properties on DOM objects:

Definition III.1 *Let d be a DOM objet. We denote:*

- $\rho(d) = (param_1, ..., param_n)$ *the set of values or types of d,*
- $d.function$ *is the Ajax function which may be empty,*
- $d.event$ *is the event which starts $d.function$. $d.event$ may be empty.*

Instead of giving the STS formal definition, which can be found in [1], we prefer illustrating it with the STS of Fig. 3 and 4 which model the Google map Ajax search application. This specification, which agrees with the UML diagram of Figure 2, represents the overall application whose we can give an event and from which we can observe different kind of messages. More precisely, the first transition expresses the launch of the Ajax function "localsearch" which takes the DOM object "location" and modifies the DOM object "map". The two next ones represent the observation of XML messages which must satisfy XML schema and the last transition represents the web document update by mean of the "map" DOM object.

In this example, only one request is performed after the event callback, but it is easily possible to add other events, requests or DOM modifications. The STS model is rich enough to describe most of Ajax applications.

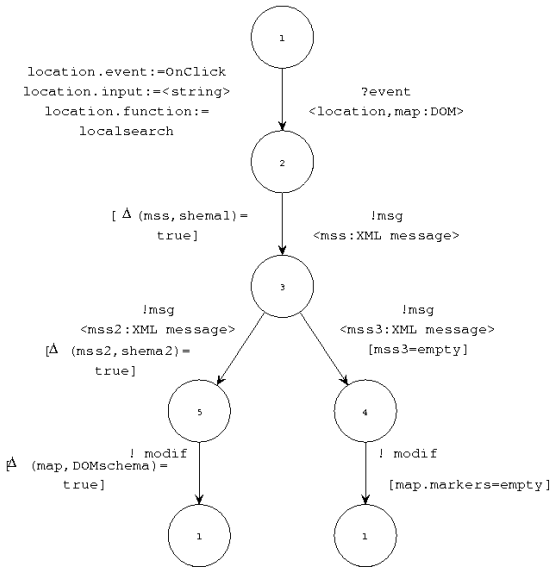

Fig. 3. STS modelling the functioning of the Google Map search

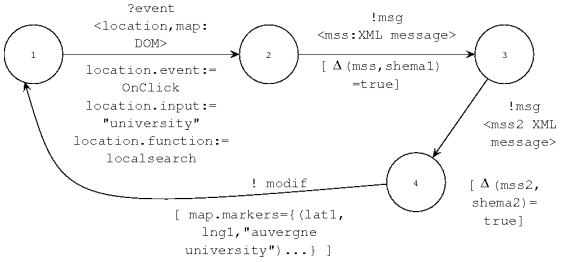

Fig. 4. STS modelling one execution of the Google Map search

With the Google map Ajax search, we have intentionally given two STS. The first one gives the Ajax application definition where the observed messages and the modified DOM object must be constructed according to existing XML schema. The second STS completes this global definition by expressing one execution and by showing the exact modification of the "map" DOM object.

This combination brings the advantage to handle the specification with different points of view: with the first STS, defining the whole application, we have the possibility to perform random testing by changing the DOM values given to the Ajax application. With the other STS, which can be seen as test purposes (sequences of properties which must be found in the specification), we have the possibility to test whether the Ajax application accepts particular DOM values and yields the correct web document updates.

The use of several STS for Ajax specification leads to the following definition:

Definition III.2 *An Ajax specification $AS =< S_{DEF}, S >$ is a tuple where:*

- S_{DEF} *is an STS defining the Ajax application,*

- *S is a set of STS where $s \in S$ is an STS modelling one execution of the Ajax application. s is composed of specific variable values,*
- *for each STS $s \in S \cup \{S_{DEF}\}$, and for each transition $t \in \rightarrow_s$ labelled by an input symbol is, $is =$ "$?event < i, o : DOM" >$ where i and o are DOM objects,with $i \neq \varnothing$ and $i.function \neq \varnothing$. o can be equal to $\varnothing$.*

In the user viewpoint, using STS is not the easiest way to write Ajax specifications. This is why we have developed the graphical tool *UMLtoSTS* which helps to specify Ajax applications with UML sequence diagrams. These ones are then translated into STS and stored into XML files. The *UMLtoSTS* interface is illustrated in Figure 5.

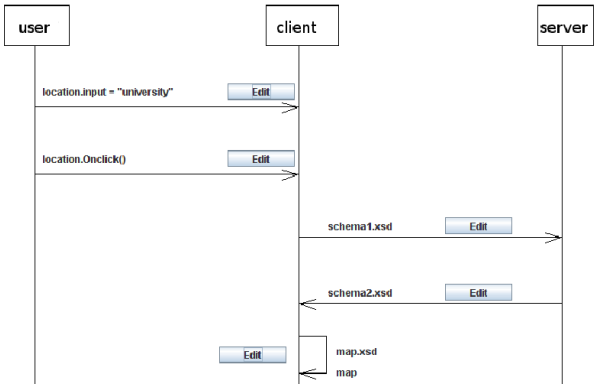

Fig. 5. UMLtoSTS editor tool

IV. AJAX TESTING METHOD

A. Test case generation

Since specifications are written with STS, we can benefit from the $ioco_F$ implementation relation and algorithms described in [1]. Test cases are defined as STS, where the final locations are labeled either by *pass* or *fail*. These ones are generated according to the $ioco_F$ relation which describes the implementation conformance. The idea behind the $ioco_F$ relation is to compare the observable traces of the specifications with the implementation under test ones. If the implementation traces belong to the specification ones, then the implementation is conformed. Traces correspond to the exhaustive and observable event suites (output symbols and variables). Quiescence is also seen as an observable event (quiescent locations are those from which no output event are observed). The notion of quiescence is important with Ajax applications because of the communication between the web server and the client. Indeed, observing quiescence means that one message has not been received. Consequently, when a quiescent location is revealed, instead of waiting infinitely a message, the test case execution is stopped and a *fail* test verdict is given.

In our case, an Ajax specification AS is composed of one STS STS_{Def}, which expresses its whole behavior, and of others STS $\{s_1, ..., s_n\}$, written with values, each one describing exactly one execution. The intuition behind our test case generation is to perform random testing with STS_{Def} and test purpose based testing with $\{s_1, ..., s_n\}$.

Before using the test case generation algorithm of [1], we construct a set of STS for random testing, by injecting in STS_{Def} some values gathered in a set, denoted R. This set contains for each type, a list of values that will be used for constructing Ajax requests. Some of these values are chosen randomly whereas other values are predefined. These last ones are assumed to have a high bug-revealing rate when used as inputs.

We denote $R(t) \in R$ the set of values for the type t which can be a simple type or a complex one. Figures 6, 7 show some values used for the type "String" and for "tabular" of "simple-type". So, for a tabular composed of String elements, we use the empty tabular, tabulars with empty elements and tabulars of String constructed with $R(String)$.

We have constructed R with the following types: "String", "Integer", "Float", "Tabular" and believe that these ones should be sufficient to cover most of the Ajax applications.

```
<type id="String">
        <val value=null />
        <val value="" />
        <val value="$" />
        <val value="*" />
        <val value="hello" />
        <val value=RANDOM" /> <!-- a random
        String-->
        <val value=RANDOM(8096)" /> <!-- a random
        String of 8096 caracters-->
</type>
```

Fig. 6. R(String)

```
<type id="Tabular">
        <val value=null /><!-- an empty
        tabular-->
        <val value= null null /><!--tabular
        composed of two empty elts-->
        <val value= simple-type />
</type>
```

Fig. 7. R(tabular)

For an Ajax specification $AS =< S_{DEF}, S >$, we construct test cases with the following steps:

1) we extract the transition list $(t_1, ..., t_k) \in (\rightarrow_{S_{DEF}})^k$, labelled by the input symbol "?event<i,o:DOM>",
2) for each transition $t \in (t_1, ..., t_k)$, we construct the set of values $Values(t)$ over R. $Values(t) = \{(r_1, ...r_n) \mid t$ labelled by $?event < i, o : DOM >, (r_1, ...r_n) \in R(p_1) \times ... \times R(p_n)$ with $\rho(i) = (p_1, ...p_n), \}$. So, we obtain the sets $Values(t_1), ..., Values(t_k)$. If the parameter types are complex (tabular), we compose them with other types to obtain the final values. We use an heuristic to estimate and eventually to reduce the number of tests according to the number of tuples in $Values$,
3) we construct the set of STS TS from S_{DEF} by injecting the previous tuples of values into S_{DEF}. For

each $(v_1, ..., v_k) \in Values(t_1) \times ... \times Values(t_k)$, we derive an STS $s \in TS$ from S_{DEF} such as for each $t_i \in (t_1, ..., t_k)$ labelled by "?event<i,o:DOM>", $\rho(i) = v_i$,

4) we extend TS with the set of STS $S \in AS$ which are already constructed with values,
5) test cases are finally constructed on the fly according to the algorithm in [1].

The reader has already noticed that steps 2 and 3 may lead to a test case explosion since we use cartesian products. The first one is used to construct a set of values $Values(t)$ over R for each transition t labelled by "?event<i,o:DOM>". While constructing $Values(t)$, we estimate its cardinality and if this one is very large, we reduce it randomly. The second cartesian product is used to fill in S_{DEF} the transitions labelled by "?event" (step 3) with values. Theoretically, the larger the number of such transitions is, the larger the test case number is. Practically, the number of events used with one Ajax function is very limited (less than five). But once more, we use an heuristic to estimate the test case number and to choose randomly a limited number of tuples in $Values(t_1) \times ... \times Values(t_k)$.

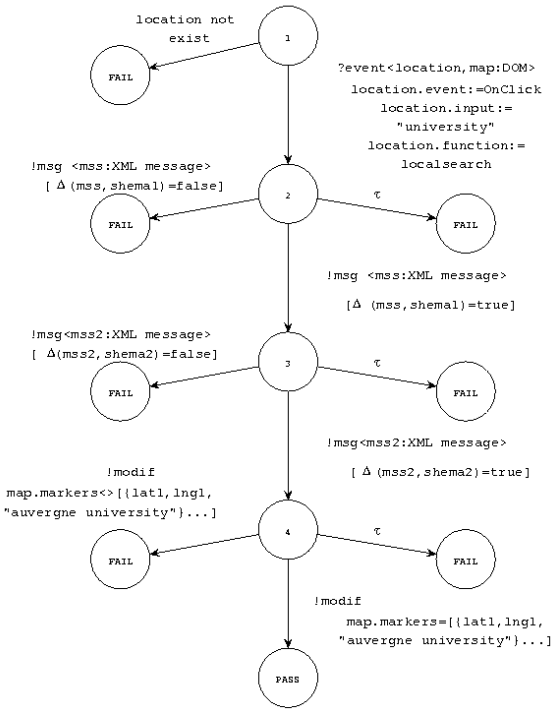

Fig. 8. A test case example

The final test case set, denoted TC, is constructed from TS. Well known testing tools like TORX [7] or TGV [8] may be used to generate them with respect of the $ioco$ implementation relation. However, we have implemented the algorithm of [1] in our own testing tool to facilitate its integration. Test case generation is executed on the fly while following one path, in order to prevent from an eventual state space explosion problem which may occur while constructing all the test case branches, according to the infinite domain of the variables used.

We illustrate a test case example in Figure 8, which is obtained from the STS of Figure 4.

B. Test case execution

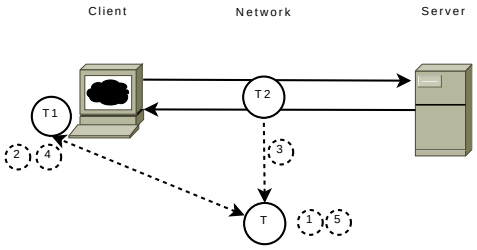

Fig. 9. Test architecture

Test cases are executed within the testing framework, illustrated in Figure 9, which has been implemented in a prototype. On the one hand, we have of course an Ajax application under test AUT, deployed on both client and server sides. On the other hand, the tester is itself composed of several actors T, $T1$ and $T2$. The tester $T1$ launches Ajax functions and sends the response to T. $T2$ retrieves the XML messages passing through the network and T constructs on the fly test cases while receiving messages from $T1$ and $T2$. More precisely:

```
1  while a transition t labelled by
   ?event < init, last : DOM > is received do
2      check whether init and its properties (init.event,
       ρ(init), init.function) exists inside the web
       document;
3      if not then
4          return a Fail verdict to T;
5      else
6          fill out the web document DOM init with
           ρ(init);
7          call directly the Ajax function init.function;
8          if last ≠ null then
9              serialize the web document DOM last;
10             send it to T;
11         end
12     end
13 end
```

Algorithm 1: Tester $T1$

- **Tester T1:** this tester simulates a user which handles the web document. This tester, written in Javascript language is added to the web document and is run as soon as the web document is loaded. Its algorithm is given in Algorithm 1. $T1$ takes a transition of the test case, labelled by an input symbol "?event" (1). $T1$ checks whether the DOM object $init$ and its properties (function, event values,...) exist. If $init$ (or of course one of its properties) does not exist, the test cannot be performed so

the fail verdit is returned to T (4). Otherwise, $T1$ fills out the web document DOM object $init$ with values. $T1$ calls $init.function$ (6,7). Once this one ends, a web document DOM object $last$ has eventually been modified. In this case ($last \neq null$) this one is sent to T (9,10). Note that it is not required to manually trigger each event. $T1$ checks that the event (Onclick, Onkeypressed,...) exits in the web document and directly calls Ajax functions.

- **Tester T2:** this tester corresponds to an autonomous XML sniffer which is installed between the client and the server. As soon as a complete XML message is read, this one is sent to the tester T.
- **Tester T:** this one orchestrates the test case execution. T takes a list of STS $TS = \{s_1, ...s_m\}$ constructed as described in section IV-A. For each STS s_k, T constructs on the fly one test case and covers it, according to the algorithm of [1]. When it reaches a transition labelled by "?event", this one is given to $T1$ in order to launch an Ajax function. Otherwise, T receives passively observable messages from $T1$ or $T2$ and checks that the guards of the test case transitions are satisfied. This is done again until T reaches a final test case location. Quiescence is taken into consideration while the execution: we set that quiescence is observed after 60s. If quiescence is observed whereas a message ought to be received, then a final location, labelled by "fail" is reached. In any case, once the test case is completely executed, the tester comes into a location either labelled by "pass" or "fail", which indicates the local verdict of the test case.

Let us show how this architecture runs chronologically by using the STS of Figure 4 which produces on the fly the test case of Figure 8. For greater readability, we assume that the Ajax application under test AUT behaves as the specification AS, illustrated in Figure 3. The successive steps are depicted in Figure 9. (1) $T1$ is added to the web document and T loads the web document with a navigator. The tester T constructs the test case on the fly. It retrieves the unique transition labelled by "?event" and sends it to $T1$. (2) $T1$ checks if $location$ exists and modifies it by adding the string "university" to the input label. Then it starts the Ajax function $location.localsearch$. (3) While constructing the test case, the tester T waits successively for two XML messages, provided each by $T2$ in less than 60s. The first message mss satisfies the XML schema $schema1$ and $mss2$ satisfies $schema2$, thus T continues the test case execution and now waits for a message composed of the "map" DOM objet. (4) The Ajax function $localsearch$ ends with the update of map. $T1$ serializes and sends it to T. (5) Finally, T checks whether map is composed of the marker tab [lng,lat,"Auvergne University"...] and reaches a final test case location labelled with "pass", which is the local verdict.

After having executed each test case tc_i of $TC = \{tc_1, ..., tc_n\}$, generated from the specification AS, the tester reaches a final location, labelled by the local verdict "pass" or "fail". For tc_i, we denote this local verdict $lv(tc_i)$. Once the test case of TC have been executed on the Ajax application under test AUT, we can conclude whether the $ioco_F$ relation is satisfied or not, i.e $AUT ioco_F AS$.

Definition IV.1 *Let AS be an Ajax specification, TC be the test case set generated from AS according to the definition of the $ioco_F$ relation and AUT be the Ajax application under test.*
$AUT ioco_F AS$, iff $\forall tc_i \in TC, lv(tc_i) = "pass"$,
$\neg(AUT ioco_F AS)$, iff $\exists tc_i \in TC \mid lv(tc_i) = "fail"$.

V. Conclusion

Conceiving a framework for testing Ajax applications automatically rises new issues on account of their unusual nature. These ones are distributed over heterogeneous actors (client and web server), handle web documents with DOM and must be interoperable by mean of XML serialization. We have first focused on the Ajax modelling with STS and have chosen a grey box representation to observe all the client/server interactions. Then, we have proposed a test case generation based on the $ioco$ implementation relation. We have also conceived a test architecture which has the capability to execute automatically test cases without triggering manually user events.

Ajax testing and this paper may lead to some perspectives. First, Ajax applications can be more complex than the ones considered here: they may be implemented with object oriented patterns and may perform several simultaneous web server calls. So, further research is required to provide solutions for modelling and testing parallel Ajax calls.

Random testing is a mere approach that we have improved with the R set, composed of predefined values. However, this does not guarantee the coverage of all the specification paths. A better solution would be a preliminary analysis of the data and of the transition guards to ensure the complete path coverage.

References

[1] L. Frantzen, J. Tretmans, and T. Willemse, "Test Generation Based on Symbolic Specifications," in *Formal Approaches to Software Testing – FATES 2004*, ser. Lecture Notes in Computer Science, J. Grabowski and B. Nielsen, Eds., no. 3395. Springer, 2005, pp. 1–15. [Online]. Available: http://www.cs.ru.nl/ lf/publications/FTW05.pdf

[2] Google, "Google map search," http://maps.google.fr/, http://code.google.com/apis/maps/documentation/.

[3] J. M. Caffrey, "Automatisation de test ajax," in *MSDN magazin*, Feb. 2007.

[4] J. Larson, "Testing ajax applications with selenium," in *InfoQ magazine*, 2006.

[5] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of ajax web applications," in *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*. IEEE Computer Society, 2008, pp. 121–130.

[6] A. Marchetto, P. Tonella, and F. Ricca, "A case study-based comparison of web testing techniques applied to ajax web applications," in *Int J Software Tools Technologies Transfer*, no. 10. Springer-Verlag, 2008.

[7] G. J. Tretmans and H. Brinksma, "Torx: Automated model-based testing," in *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, A. Hartman and K. Dussa-Ziegler, Eds., December 2003, pp. 31–43.

[8] C. Jard and T. Jeron, "Tgv: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 4, pp. 297–315, 2005.