Automating Navigation Sequences in AJAX Websites

Paula Montoto, Alberto Pan, Juan Raposo, Fernando Bellas, and Javier López

Department of Information and Communication Technologies, University of A Coruña
Facultad de Informática, Campus de Elviña s/n 15071 A Coruña, Spain
{pmontoto,apan,jrs,fbellas,jmato}@udc.es

Abstract. Web automation applications are widely used for different purposes such as B2B integration, automated testing of web applications or technology and business watch. One crucial part in web automation applications is to allow easily generating and reproducing navigation sequences. Previous proposals in the literature assumed a navigation model today turned obsolete by the new breed of AJAX-based websites. Although some open-source and commercial tools have also addressed the problem, they show significant limitations either in usability or their ability to deal with complex websites. In this paper, we propose a set of new techniques to deal with this problem. Our main contributions are a new method for recording navigation sequences supporting a wider range of events, and a novel method to detect when the effects caused by a user action have finished. We have evaluated our approach with more than 100 web applications, obtaining very good results.

Keywords: Web automation, web integration, web wrappers.

1 Introduction

Web automation applications are widely used for different purposes such as B2B integration, web mashups, automated testing of web applications or business watch. One crucial part in web automation applications is to allow easily generating and reproducing navigation sequences. We can distinguish two stages in this process:

- Generation phase. In this stage, the user specifies the navigation sequence to reproduce. The most common approach, cf. [1,9,11], is using the 'recorder' metaphor: the user performs one example of the navigation sequence using a modified web browser, and the tool generates a specification which can be run by the execution component. The generation environment also allows specifying the input parameters to the navigation sequence.
- Execution phase. In this stage, the sequence generated in the previous stage and the input parameters are provided as input to an automatic navigation component which is able to reproduce the sequence. The automatic navigation component can be developed by using the APIs of popular browsers (e.g. [9]). Other systems like [1] use simplified custom browsers specially built for the task.

Most existing previous proposals for automatic web navigation systems (e.g. [1,9,11]) assume a navigation model which is now obsolete to a big extent: on one hand, the user actions that could be recorded were very restrictive (mainly clicking on elements and filling in form fields) and, on the other hand, it was assumed that almost every user action caused a request to the server for a new page.

Nevertheless, this is not enough for dealing with modern AJAX-based websites, which try to replicate the behavior of desktop applications. These sites can respond to a much wider set of user actions (mouse over, keyboard strokes, drag and drop…) and they can respond to those actions executing scripting code that manipulates the page at will (for instance, by creating new graphical interface elements on the fly). In addition, AJAX technology allows requesting information from the server and repainting only certain parts of the page in response.

In this paper, we propose a set of new techniques to build an automatic web navigation system able to deal with all this complexity. In the generation phase, we also use the 'recorder' metaphor, but substantially modified to support recording a wider range of events; we also present new methods for identifying the elements participating in a navigation sequence in a change-resilient manner.

In the execution phase, we use the APIs of commercial web browsers to implement the automatic web navigation components (the techniques proposed for the recording phase have been implemented using Microsoft Internet Explorer (MSIE) and the execution phase has been implemented using both MSIE and Firefox); we take this option because the approach of creating a custom browser supporting technologies such as scripting code and AJAX requests is effort-intensive and very vulnerable to small implementation differences that can make a web page to behave differently when accessed with the custom browser. In the execution phase, we also introduce a method to detect when the effects caused by a user action have finished. This is needed because one navigation step may require the effects of the previous ones to be completed before being executed.

2 Models

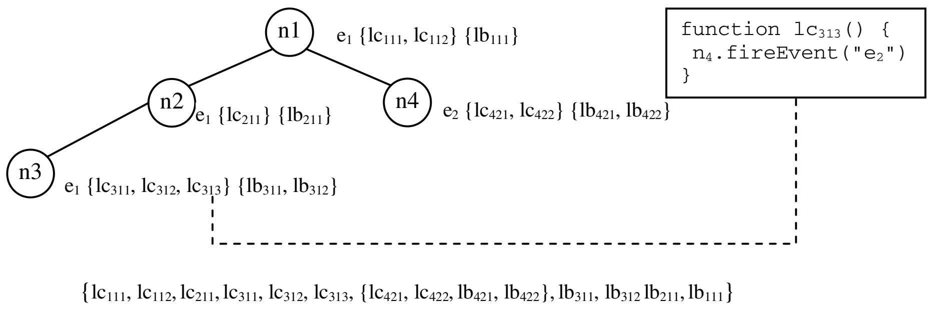
In this section we describe the models we use to characterize the components used for automated browsing. The main model we rely on is DOM Level 3 Events Model [3]. This model describes how browsers respond to user-performed actions on an HTML page currently loaded in the browser. Although the degree of implementation of this standard by real browsers is variable, the key assumptions our techniques rely on are verified in the most popular browsers (MSIE and Firefox). Therefore, section 2.1 summarizes the main characteristics of this standard that are relevant to our objectives. Secondly, section 2.2 states additional assumptions about the execution model employed by the browser in what regards to scripting code, including the kind of asynchronous calls required by AJAX requests. These assumptions are also verified by current major browsers.

2.1 DOM Level 3 Events Model

In the DOM Level 3 Events Model, a page is modelled as a tree. Each node in the tree can receive events produced (directly or indirectly) by the user actions. Event types exist for actions such as clicking on an element (*click*), moving the mouse cursor over it (*mouseover*) or specifying the value of a form field (*change*), to name a few. Each node can register a set of event listeners for each event type. A listener executes arbitrary code (typically written in a script language such as Javascript). Listeners have the entire page tree accessible and can perform actions such as modifying existing nodes, removing them, creating new ones or even launching new events.

The event processing lifecycle can be summarized as follows: The event is dispatched following a path from the root of the tree to the target node. It can be handled locally at the target node or at any target's ancestors in the tree. The event dispatching (also called event propagation) occurs in three phases and in the following order: capture (the event is dispatched to the target's ancestors from the root of the tree to the direct parent of the target node), target (the event is dispatched to the target node) and bubbling (the event is dispatched to the target's ancestors from the direct parent of the target node to the root of the tree). The listeners in a node can register to either the capture or the bubbling phase. In the target phase, the events registered for the capture phase are executed before the events executed for the bubbling phase. This lifecycle is a compromise between the approaches historically used in major browsers (Microsoft IE using bubbling and Netscape using capture).

The order of execution between the listeners associated to an event type in the same node is registration order. The event model is re-entrant, meaning that the execution of a listener can generate new events. Those new events will be processed synchronously; that is, if l_i, l_{i+1} are two listeners registered to a certain event type in a given node in consecutive order, then all events caused by l_i execution will be processed (and, therefore, their associated listeners executed) before l_{i+1} is executed.



```
function lc313() {
 n4.fireEvent("e2")
}
```

n1
e1 {lc111, lc112} {lb111}
n2
e1 {lc211} {lb211}
n4
e2 {lc421, lc422} {lb421, lb422}
n3
e1 {lc311, lc312, lc313} {lb311, lb312}
{lc111, lc112, lc211, lc311, lc312, lc313, {lc421, lc422, lb421, lb422}, lb311, lb312 lb211, lb111}

Fig. 1. Listeners Execution Example

Example 1: Fig. 1 shows an excerpt of a DOM tree and the listeners registered to the event types e_1 and e_2. The listeners in each node for each event type are listed in registration order (the listeners registered for the capture phase appear as l_{cxyz} and the ones registered for the bubbling phase appear as l_{bxyz}). The figure also shows what listeners and in which order would be executed in the case of receiving the event-type e_1 over the node n_3, assuming that the listener on the capture phase l_{c313} causes the event-type e_2 to be executed over the node n_4.

DOM Level 3 Events Model provides an API for programmatically registering new listeners and generating new events. Nevertheless, it does not provide an introspection API to obtain the listeners registered for an event type in a certain node. As we will see in section 3.1, this will have implications in the recording process in our system.

2.2 Asynchronous Functions and Scripts Execution Model

In this section we describe the model we use to represent how the browser executes the scripting code of the listeners associated to an event. This model is verified by the major commercial browsers.

The script engine used by the browser executes scripts sequentially in single-thread mode. The scripts are added to an execution queue in invocation order; the script engine works by sequentially executing the scripts in the order specified by the queue.

When an event is triggered, the browser obtains the listeners that will be triggered by the event and invokes its associated scripts, causing them to be added to the execution queue. Once all the scripts have been added, execution begins and the listeners are executed sequentially.

The complexity of this model is slightly increased because the code of a listener can execute asynchronous functions. An asynchronous function executes an action in a non-blocking form. The action will run on the background and a callback function provided as parameter in the asynchronous function invocation will be called when the action finishes.

The most popular type of asynchronous call is the so-called AJAX requests. An AJAX request is implemented by a script function (i.e. in Javascript, a commonly used one is *XMLHTTPRequest*) that launches an HTTP request in the background. When the server response is received, the callback function is invoked to process it.

Other popular asynchronous calls establish timers and the callback function is invoked when the timer expires. In this group, we find the Javascript functions *setTimeout(ms)* (executes the callback function after ms milliseconds) and *setInterval(ms)* (executes the callback function every ms milliseconds). Both have associated cancellation functions: *clearTimeout(id)* and *clearInterval(id)*.

It is important to notice that, from the described execution model, it is inferred the following property:

Property 1: The callback functions of the asynchronous calls launched by the listeners of an event will never be executed until all other scripts associated to that event have finished.

The explanation for this property is direct from the above points: all the listeners associated to an event are added to the execution queue first, and those listeners are the ones invoking the asynchronous functions; therefore, the callback functions will always be positioned after them in the execution queue even if the background action executed by the asynchronous call is instantaneous.

3 Description of the Solution

In this section we describe the proposed techniques for automated web navigation. First, we deal with the generation phase: section 3.1 describes the process used to record a navigation sequence in our approach. Section 3.2 deals with the problem of identifying the target DOM node of a user action: this problem consists in generating a path to the node that can be used later at the execution phase to locate it in the page and section 3.3 deals with the execution phase.

3.1 Recording User Events

The generation phase has the goal of recording a sequence of actions performed by the user to allow reproducing them later during the execution phase.

A user action (e.g. a click on a button) causes a list of events to be issued to the target node, triggering the invocation of the listeners registered for them in the node and its ancestors, according to the execution model described in the previous section. Notice that each user action usually generates several events. For instance, the events generated when the user clicks on a button include, among others, the mouseover event besides of the click event, since in order to click on an element it is previously required to place the mouse over it. Recording a user action consists in detecting which events are issued, and in locating the target node of those events.

In previous proposals, cf. [1,6,9], the user can record a navigation sequence by performing it in the browser in the same way as any other navigation. The method used to detect the user actions in these systems is typically as follows: the recording tool registers its own listeners for the most common events involved in navigations (mainly clicks and the events involved in filling in form fields) in anchors and form-related tags. This way, when a user action produces one of the monitored event-types *e* on one of the monitored nodes *n*, the listener for *e* in *n* is invoked, implicitly identifying the information to be recorded.

Nevertheless, the modern AJAX-based websites can respond to a much wider set of user actions (e.g. placing the mouse over an element, producing keyboard strokes, drag and drop…); in addition, virtually any HTML element, and not only traditional navigation-related elements, can respond to user actions: tables, images, texts, etc.

Extending the mentioned recording process to support AJAX-based sites would involve registering listeners for every event in every node of the DOM tree (or, alternatively, registering listeners for every event in the root node of the page, since the events execution model ensures that all events reach to the root). Registering listeners for every event has the important drawback that it would "flood" the system by recording unnecessary events (e.g. simply moving the mouse over the page would generate hundreds of *mouseover* and *mouseout* events); recall that, as mentioned in section 2, it is not possible to introspect what events a node has registered a listener for; therefore, it is not possible to use the approach of registering a listener for an event-type *e* only in the nodes that already have other listeners for *e*.

Therefore, we need a new method for recording user actions. Our proposal is letting the user explicitly specify each action by placing the mouse over the target element, clicking on the right mouse button, and choosing the desired action in the contextual menu (see Fig. 2). If the desired action involves providing input data into an input element or a selection list, then a pop-up window opens allowing the user to specify the desired value (see Fig. 2). Although in this method the navigation recording process is different from normal browsing, it is still fast and intuitive: the user simply changes the left mouse button for the right mouse button and fills in the value of certain form fields in a pop-up window instead of in the field itself.

This way, we do not need to add any new listener: we know the target element by capturing the coordinates where the mouse pointer is placed when the action is specified, and using browser APIs to know what node the coordinates correspond to. The events recorded are implicitly identified by the selected action.

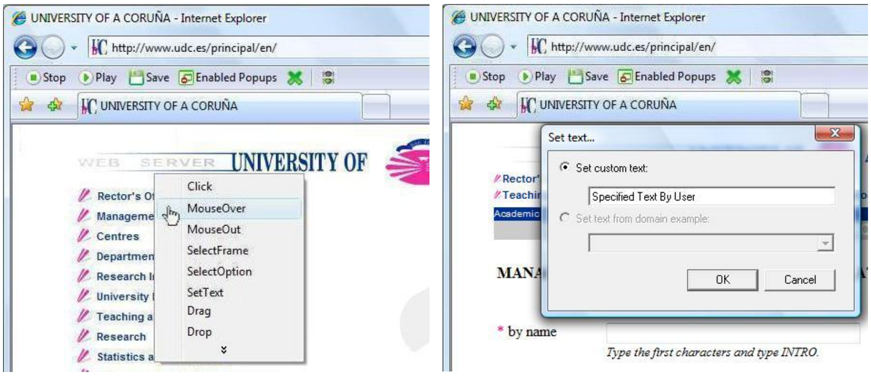


Fig. 2. Recording Method

Our prototype implementation includes actions such as *click*, *mouseover*, *mouseout*, *selectOption* (selecting values on a selection list), *setText* (providing input data into an element), *drag* and *drop*. Since each user action actually generates more than one event, each action has associated the list of events that it causes: for instance, the *click* action includes, among others, the events *mouseover*, *click* and *mouseout*; the *setText* action includes events such as *keydown* and *keyup* (issued every time a key is pressed) and *change* (issued when an element content changes).

This new method has a problem we need to deal with. By the mere process of explicitly specifying an action, the user may produce changes in the page before we want them to take place. For instance, suppose the user wishes to specify an action on a node that has a listener registered for the *mouseover* event; the listener opens a pop-up menu when the mouse is placed over the element. Since the process of specifying the action involves placing the mouse over the element; the page will change its state (i.e. the pop-up menu will open) before the user can specify the desired action. This is a problem because the process of generating a path to identify the target element at the execution phase (described in detail in section 3.2) cannot start until the action has been specified. But, since the DOM tree of the page has already changed, the process would be considering the DOM tree after the effects of the action have taken place (the element may even no longer exist because the listeners could remove it!).

We solve this problem by deactivating the reception of user events in the page during the recording process. This way, we can be sure that no event alters the state of the page before the action is specified. Once the user has specified an action, we use the browser APIs to generate on the target element the list of events associated to the action; this way, the effects of the specified action take place in the same way as if the user would have performed the action, and the recording process can continue.

Another important issue we need to deal with is ensuring that a user does not specify a new action until the effects caused by the previous one have completely finished. This is needed to ensure that the process for generating a path to identify at the execution phase the target element of the new action has into account all the changes in the DOM tree that the previous action provokes. Detecting the end of the effects of an action is far from a trivial problem; since it is one of the crucial issues at the execution phase, we will describe how to do it in section 3.3.

3.2 Identifying Elements

During the generation phase, the system records a list of user actions, each one performed on a certain node of the DOM tree of the page. Therefore, we need to generate an expression to uniquely identify the node involved in each action, so the user action can be automatically reproduced at the execution phase.

An important consideration is that the generated expression should be resilient to small changes in the page (such as the apparition in the page of new advertisement banners, new data records in dynamically generated sections or new options in a menu), so it is still valid at the execution stage.

To uniquely identify a node in the DOM tree we can use an XPath [15] expression. XPath expressions allow identifying a node in a DOM tree by considering information such as the text associated to the node, the value of its attributes and its ancestors. For our purposes, we need to ensure that the generated expression on one hand identifies a single node, and on the other hand it is not too specific to be affected by the formerly mentioned small changes. Therefore, our proposal tries to generate the less specific XPath expression possible that still uniquely identifies the target node. The algorithm we use for this purpose is presented in section 3.2.1.

In addition, the generated expressions should not be sensible to the use of session identifiers, to ensure that they will still work in any later session. Section 3.2.2 presents a mechanism to remove session identifiers from the generated expressions.

3.2.1 Algorithm for Identifying Target Elements

This section describes the algorithm for generating the expression to identify the target element of a user action.

As it has already been said, the algorithm tries to generate the less specific XPath expression possible that still uniquely identifies the target node. More precisely, the algorithm first tries to identify the element according to its associated text (if it is a leaf node) and the value of its attributes. If this is not enough to uniquely identify the node, its ancestors (and the value of their attributes) will be recursively used. The algorithm to generate the expression for a node n consists of the following steps:

1. Initialize $X_{\{n\}}$ (the variable that will contain the generated expression) to the empty string. Initialize the variable n_i to the target node n.
2. Let m be the number of attributes of n_i, T_{ni} be the tag associated to n_i and t_{ni} be its associated text. Try to find a minimum set of r attributes $\{a_{ni1},\ldots,a_{nir}\}_{r<=m}$, of $_{ni}$ such that the following expression ('+' represents the concatenation of two strings):

"//" + T_{ni} [@a_{ni1}=v_{ni1} and... and @a_{nir}=v_{nir} and @text()=t_{ni}] + $X_{\{n\}}$+ "/"

uniquely identify n. (NOTE:The fragment and *text()=*t_{ni} of the expression would only be added if n_i is a leaf node, since only leaf nodes have associated text).

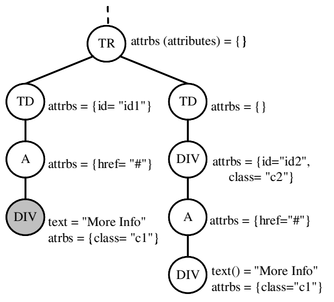
3. If the set is found then

    3.1) return the expression from step 1.

   else

    3.2) Let $\{a_{ni1},\ldots,a_{nim}\}$ be the set of all attributes of n_i. Set $X_{\{n\}}$ = *"/"+*T_{ni} *[@*a_{ni1}=v_{ni1} *and... and @*a_{nim}=v_{nim} *and @text()=*t_{ni}*]* + $X_{\{n\}}$; that is, we add conditions by all the attributes of n_i to the expression.

4. If n_i is not the root of the DOM tree then
   4.1) Set n_i=*parent*(n_i) and go to step 1
   else
   4.2) Obtain the relative position *j* of *n* in the page with respect to all the nodes verifying the current expression $X_{\{n\}}$. Return *"/"+ $X_{\{n\}}$+ [j] + "/"*.



First Iteration:
$X_{\{n\}}$ = /DIV[@class="c1" and text()="More Info"]
(*/$X_{\{n\}}$/* does not identify the grayed element)

Second Iteration:
$X_{\{n\}}$ = /A[@href="#"]/DIV[@class="c1" and text()="More Info"]
(*/$X_{\{n\}}$/* does not identify the grayed element)

Third Iteration:
$X_{\{e\}}$ = /TD[@id="id1"]/A[@href="#"]/DIV[@class="c1" and text()="More Info"]
(*/$X_{\{n\}}$/* identifies the grayed element)

Result:
//TD[@id="id1"]/A[@href="#"]/DIV[@class="c1" and text()="More Info"]/

Fig. 3. Algorithm for Identifying Target Elements Example

Fig. 3 shows an example sub-tree and the $X_{\{n\}}$ value in each iteration of the algorithm to generate the XPath expression to identify the grayed *DIV* node.

Now, we provide further detail about some of the steps. The step 1 of the algorithm tries to identify the minimum set of attributes of the currently considered node n_i, that allow completing the identification of *n*. To do this, we add attributes one by one until either *n* is uniquely identified or all the attributes of n_i, have been added. To decide the order in which we add the attributes, we have defined an order for the attributes of each HTML tag based on its estimated selectivity (that is, how much they contribute to narrow the selection). For instance, we consider the *id* and *name* attributes highly selective for all HTML tags and the *href* attribute highly selective for the *A* tag, while we consider the class attribute as of low selectivity.

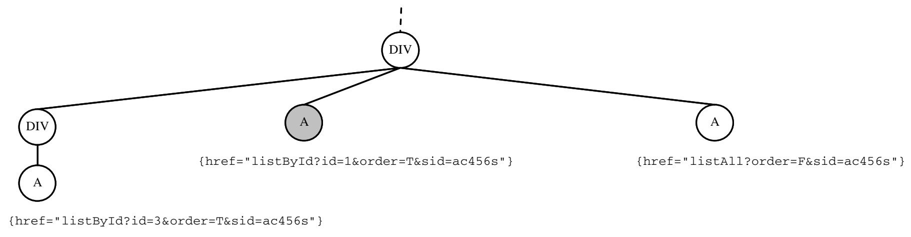
Step 3.2 considers the case when the algorithm reaches the root, and the generated expression still does not uniquely identify *n*. In that case, the algorithm adds to the XPath expression the relative position in the page of *n* with respect to the rest of elements identified by the expression.

3.2.2 Removing Session IDs

Many websites use session identifiers in URL attributes to track user sessions. In these sites, the values of attributes containing URLs may vary in each session. Since our method to identify target elements at the execution phase relies on attribute values, this causes a problem for our approach.

Our prototype implementation recognizes the main standard formats for including session identifiers in URLs. Unfortunately, many websites do not use any standard, but include the session identifier using arbitrary query parameters.

Therefore, we propose an algorithm to generalize the value of attributes containing URLs. The algorithm is based on two observations: 1) a query parameter acting as session identifier must take the same value in all the URLs of the page in which it appears; 2) if a query parameter takes the same value in all URLs with the same host


DIV
DIV
A
A
A
{href="listById?id=1&order=T&sid=ac456s"}
{href="listAll?order=F&sid=ac456s"}
{href="listById?id=3&order=T&sid=ac456s"}

```
//A[matches(@href,"listById?id=1&order=[^&=]+&sid=[^&=]+")]/
```

Fig. 4. Removing Session IDs Example

and query parts, then it is irrelevant for the purpose of identifying an element in the DOM tree by the value of its attributes.

The basic idea of the algorithm derives directly from the above observations: find all the query parameters that take the same value in all the URLs in which they appear and ignore their values for identification purposes. Although some of the identified query parameters may not be session identifiers, according to observation 2 it is safe to ignore their values anyway.

Fig 4 shows a simple example of the algorithm where *n* is the grayed node in the figure. The query parameters named *order* and *sid* take the same value in all the URLs with the same path (in the example the page does not contain other URLs with the same path). Therefore, they are considered irrelevant for node identification purposes. (NOTE: *matches()* is XPath function for applying regular expressions).

3.3 Execution Phase

The generation phase generates a program capturing the navigation sequence recorded by the user. The execution phase runs the program in the automatic navigation component.

A first consideration is that we opt to use the APIs of commercial web browsers to implement the automatic web navigation components instead of building a simplified custom-browser. The main reason for taking this option is that web 2.0 sites make an intensive use of scripting languages and support a complex event model. Creating a custom browser supporting those technologies in the same way as commercial browsers is very effort-intensive and, in addition, is extremely vulnerable to small implementation differences that can make a web page to behave differently when accessed with the custom browser than when accessed with a “real” browser. Our techniques for the execution phase have been implemented in both MSIE and Firefox.

To reproduce an action in the navigation sequence, there are three steps involved:

1. Locating the target node in the DOM tree of the page.
2. Generating the recorded event (or list of events) on the identified node.
3. Wait for the effects of the events to finish. This is needed because the following action can need the effects of the previous ones to be completed (e.g. the action *n+1* can generate an event on a node created in the action *n*).

The implementation of 1) and 2) is quite straightforward using browser APIs and given the output of the recording process. Step 1) uses the XPath expression produced by the process described in section 3.2, and step 2) uses the events recorded in the process described in section 3.1.

In turn, step 3) is difficult because browser APIs do not provide any way of detecting when the effects on the page of issuing a particular event have finished. These effects can include dynamically creating or removing elements in the DOM tree, maybe also having into account the response to one or several AJAX requests to the server. Previous works have addressed this problem by establishing a timer after the execution of an event before continuing execution. This solution has the usual drawbacks associated to a fixed timeout in a network environment: if the specified timeout is short, then when the response to an asynchronous AJAX request is slower than usual (or even if the machine is very heavily loaded), the sequence may fail. If, in turn, we use a higher timeout valid even in those circumstances, then we are introducing an unnecessary delay when the server is responding normally.

The remaining of this section explains the method we propose to detect when the effects caused directly or indirectly by a certain event have finished. This way, the system waits the exact time required. The correctness of the method derives from the assumptions stated in section 2, which are verified by the major commercial browsers.

The method we use to detect when the effects of an event-type e generated on a node n have finished consists of the following steps:

1. We register a new listener *l* to capture the event *e* in *n*. The code of the listener *l* invokes an asynchronous function specifying the callback function *cf.* What asynchronous function is actually invoked in l is mainly irrelevant; for instance, in Javascript, we can simply invoke *setTimeout(cf,0).* Notice that as consequence of property 1 in section 2, it is guaranteed that *cf* will be executed after all the listeners triggered by the execution of *e* have finished. Therefore, if the listeners had not made any other asynchronous call, then the control arriving to *cf* would indicate that the effects of *e* had finished and the navigation sequence execution could continue. Nevertheless, since the listeners can actually execute other asynchronous calls, this is not enough.
2. To be notified of every asynchronous call executed by the listeners triggered by *e*, we redefine those asynchronous functions providing our own implementation of them (for instance, in Javascript we need to redefine *setTimeout*, *setInterval* and the functions used to execute AJAX requests such as *XMLHTTPRequest*). The template of our implementation of each function is shown in Fig 5. The function maintains a counter that is increased every time the function is invoked (the counter is maintained as a global variable initialized to zero for every emitted event). After increasing the counter, the function calls the former standard implementation of the asynchronous function provided by the browser but substituting the received callback function by a modified one (the *new_cf* function created in Figure 5). This new callback function invokes the original callback function and then decreases the counter. This way, the counter always takes the value of the number of currently active calls.
3. When the callback function *cf* from step 1 is executed, it polls the counters associated to the asynchronous functions. When they are all 0, we know the asynchronous calls have finished and execution can proceed.

4. There may be some cases where the effects of *e* actually never finish. This is for instance the case when the *setInterval* function is used. This function executes the callback function at specified time intervals and, therefore, its effects last indefinitely unless the function *clearInterval* is used. In the generation-phase, if the *setInterval* calls are not cleared after a certain timeout, the system notifies it to the user so she/he can specify the desired action, which can be to wait a fixed time or wait for a certain number of intervals to complete.

```
old_asyncFunction = standardAsyncFunction;
new_asyncFunction = new function(param1,param2,…,paramn,cf) {
    counter++;      //counter is a global variable
    new_cf = new function() {
      result = cf();
      counter--;
      if (counter==0) {
          notifyEndAsyncFunctions();
      }
      return result;
    };
    old_asyncFunction(param1,param2,…,paramn,new_cf);
  };
standardAsyncFunction = new_asyncFunction;
```

Fig. 5. Asynchronous Function Redefinition

In addition of the possible effects of an event in the current page, the event can also make the browser (or a frame inside the page) navigate to a new page. When the new page/frame is loaded (this can be detected using browser APIs), the load event is generated; this event has as target the body element of the page. Before continuing the execution of the navigation sequence, we need to wait until the end of the effects of the load event have finished, using the same technique used for the rest of events.

4 Evaluation

To evaluate the validity of our approach, we tested the implementation of our techniques with a wide range of AJAX-based web applications. We performed two kinds of experiments:

1. We selected a set of 75 real websites making extensive use of scripting code and AJAX technology. We used the prototype to record and reproduce one navigation sequence on each site. The navigation sequences automated the main purpose of the site. For instance, in electronic shops we automated the process of searching products; in webmail sites we automated the process required to access e-mails.
2. Some of the main APIs for generating AJAX-based applications such as Yahoo! User Interface Library (YUI) [16] and Google Web Toolkit (GWT) [4] include a set of example websites. At the time of testing, GWT included 5 web applications and YUI included 300 examples. We recorded and executed 12 navigation sequences in the web applications from GWT ensuring that every interface element from the applications was used at least once. In the case of YUI, we recorded 40 sequences in selected examples (choosing the more complex examples). This second group of tests is useful because many real websites use those toolkits.

Table 1. Experimental Results

Website	Played	Website	Played	Website	Played
www.a9.com/java	✓	www.fidelityasap.com	✓	www.optize.es	✓
www.abebooks.com	✓	www.fnac.es	✓	www.paginasamarillas.es	✓
www.accorhotels.com	✓	www.gmail.com	✓	www.penguin.co.uk	✓
www.addall.com	✓	www.gongdiscos.com	✓	people.yahoo.com	✓
www.voyages-sncf.com	✓	www.hotelopia.es	✓	code.jalenack.com/periodic	✓
www.alitalia.com/ES_ES/	✓	www.hotelsearch.com	✓	www.pixmania.com	✓
www.allbooks4less.com	✓	www.iberia.com	✓	www.planethome.de	✓
www.amadeus.net	✓	www.iit.edu	✓	www.priceline.com	✓
www.amazon.com	✓	www.imdb.com/search	✓	www.renault.es	✓
store.apple.com	✓	www.infojobs.net	✓	www.renfe.es	✓
www.atrapalo.com	✓	www.jet4you.com	✓	www.reuters.com	✓
autos.aol.com	✓	www.laborman.es	✓	www.rumbo.es	✓
www.balumba.es	✓	www.landrover.com	✓	www.shop-com.co.uk	✓
www.barnesandnoble.com	✓	www.es.lastminute.com	✓	www.sparkassen-immo.de	✓
www.bookdepository.co.uk	✓	www.marsans.es	✓	www.sterling.dk	✓
www.booking.com	✓	www.meridiana.it	✓	www.ticketmaster.com	✓
www.carbroker.com.au	✓	www.msnbc.msn.com	✓	tudulist.com	✓
www.casadellibro.com	✓	www.muchoviaje.com	✓	www.tuifly.com/es	✓
www.cervantesvirtual.com	✓	www.musicstore.com	✓	es.venere.com	✓
www.cia.gov	✓	www.myair.com	✓	www.viajar.com	✓
controlp.com	✓	www.mymusic.com	✓	www.vuelosbaratos.es	✓
www.digitalcamerareview.com	✓	www.es.octopustravel.com	✓	www.webpagesthatsuck.com	✓
www.ebay.es	✓	www.ofertondelibros.com	✓	news.search.yahoo.com/news/advanced	✓
www.edreams.es	✓	www.okipi.com	✓	news.yahoo.com	×
www.elcorteingles.es	✓	vols.opodo.fr	✓	mail.yahoo.com	✓

The techniques proposed for the recording phase have been implemented using MSIE and the execution phase has been implemented using both MSIE and Firefox. In each group of experiments, we recorded the navigation sequences on MSIE and executed them using both MSIE and Firefox. The execution on MSIE allows us to measure the effectiveness of our techniques in both the recording and execution phases. We execute the sequences in Firefox to check that the algorithm presented in section 3.3 is valid in both browsers. Since MSIE and Firefox usually build different DOM trees for the same pages, in some cases the XPath expression generated by the recording in MSIE were manually modified to fit the DOM tree in Firefox. Notice that this is not a limitation of our approach: it only highlights the issue that the browser used for the recording and execution phase should be the same.

The results of the evaluation were encouraging (see Table 1). In the first set of experiments (real websites), 74 of 75 sequences were recorded and executed fine.

In the case of *news.yahoo.com*, the XPath expression generated to identify an element used an URL with a query parameter which changed every time the page was reloaded. This parameter is not a session identifier since it changes its value during the same session. If the recorded XPath expression is modified manually to ignore the value of this parameter, then the sequence works correctly. To solve problems like this, we could include redundant localization information; this way, if an element cannot be identified using the “minimal” expression, then we can still use the other information to search the nearest match in the page ([1] uses a similar idea that could be extended to deal with these cases, although they do not use other necessary information, such as hierarchical information). Another option is allowing the user to provide several examples of the same sequence for detecting those parameters.

The second group of tests was completely successful in GWT applications, while in the YUI case only one sequence could not be recorded. The problem was that the *blur* event was not being generated with the *setText* action. Once this was corrected, the sequence could be recorded.

5 Related Work

WebVCR [1] and WebMacros [11] were pioneer systems for web navigation sequences automation using the "recorder metaphor". Both systems were only able to record a reduced set of events (clicks and filling in form fields) on a reduced set of elements (anchors and form-related elements). In the execution phase they relied on HTTP clients that lacked the ability to execute scripting code or to support AJAX requests. Furthermore, the techniques they used for identifying the target elements of user actions were based on the text associated to the elements and the value of some specific pre-configured attributes (e.g. *href* for *A* tags and *src* for *FORM* tags).

Wargo [9] introduced using a commercial browser as execution component, thus supporting websites using scripting languages and guaranteeing that the websites will behave in the execution phase in the same way as when a human user accesses it. Nevertheless, it still showed the remaining previously mentioned problems.

Instead of using the "recorder" metaphor, in SmartBookmarks [6] the macros are generated retroactively; when the user reaches a page and bookmarks it, the system tries to automatically find the starting point of the macro. In order to do this, SmartBookmarks permanently monitors the user actions. As it was explained in section 3.1, recording user actions in the browser as the user navigates forces to either restrict the set of monitored events or suffering from an "event-flooding" problem. SmartBookmarks only supports the events click, load and change. Another drawback is that it relies on timeouts to determine when to continue executing the sequence. HtmlUnit [5] is an open-source tool for web applications unit testing. HtmlUnit does not provide a recording tool; instead, the user needs to manually create the navigation sequences using Java coding. In addition HtmlUnit uses its own custom browser instead of relying on conventional browsers. Although their browser has support for many Javascript and AJAX functionalities, this is vulnerable to small implementation differences that can make a web page to behave differently when accessed with the custom browser.

Selenium [13] is a suite of tools to automate web applications testing. Selenium uses the recorder metaphor through a toolbar installed in Firefox. It is only able to record a reduced set of events. To identify elements, Selenium uses a system based on the text or generates an XPath expression that does not try to be resilient to small changes. Another drawback is that Selenium does not detect properly the end of the effects caused by a user action in the recording process.

Sahi [12] is another open-source tool for automated testing of web applications. Sahi includes a navigation recording system and it allows the sequences to be executed in commercial browsers. To use Sahi, the user configures its navigator to use a proxy. Every time the browser requests a new page, the proxy retrieves it, adds listeners for monitoring user actions, and returns the modified page. Using a proxy makes the recording system independent of the web browser used. Nevertheless, using a proxy does not allow using approaches where the user explicitly indicates the actions to record; therefore, as discussed previously, it forces to choose between either monitoring only a reduced set of events or suffering from “event flooding”. Sahi only supports recording events such as click and change. Other events such as *mouseover* can be used at the execution phase if the user manually codes the navigation scripts. Another drawback is that they do not detect the end of the effects caused by a user action, using timeouts instead.

In the commercial software arena, QEngine [10] is a load and functional testing tool for web applications. QEngine also uses the recorder metaphor through a toolbar installed in MSIE (also used as execution component). In addition of the most typical events supported by the previously mentioned systems, QEngine also supports a form of explicitly specifying *mouseover* events on certain elements, consisting in placing the mouse over the target element for more than a certain timeout (avoiding this way the “flooding” problem). Nevertheless, they do not capture other events such as *mouseout* or *mousemove*. To identify elements, they use a simple system based on the text, attributes and relative position of the element. While this may be enough for application-testing purposes where changes are controlled, it is not enough to deal with autonomous web sources. In addition, as previous systems, QEngine does not detect the end of the effects of an action. iOpus [7] is another web automation tool that uses the recorder metaphor. Their drawbacks with respect to our proposal are almost identical to those mentioned for QEngine.

Kapow [8] is yet another web automation tool oriented to the creation of mashups and web integration applications. Kapow uses its own custom browser. Therefore, in our evaluation it showed to be vulnerable to the formerly mentioned drawback: small implementation differences can make a web page to behave differently. For instance, from the set of 12 sequences from Google Web Toolkit we used in our tests, the Kapow browser could only successfully reproduce 1 of them. To identify the target elements, Kapow generates an XPath expression that tries to be resilient to small changes, although the details of the algorithm they use have not been published.

With respect to the algorithm to identify target elements, [2,14] have also addressed the problem of generating change-resilient XPath expressions. In those approaches, the user provides several example pages identifying the target element; and the system generalizes the expression by examining the differences between them. In our case, that would force the user to record the navigation sequence several times. We believe that process would be much more cumbersome to the user.

6 Conclusions

We have presented a set of new techniques to record and execute web navigation sequences in AJAX-based websites. Previous proposals show important limitations in the range of user actions that they can record and execute, the methods they use for identifying the target elements of user actions and/or how they wait for the effects of a user action to finish. Our techniques have been successfully implemented using both MSIE and Firefox. Our main contributions are a new method for recording navigation sequences able to scale to a wider range of events and a novel method to detect when the effects caused by a user action (including the effects of scripting code and AJAX requests) have finished, without needing to use inefficient timeouts. We have also evaluated our approach with more than 100 web applications, obtaining a high degree of effectiveness.

References

1. Anupam, V., Freire, J., Kumar, B., Lieuwen, D.: Automating web navigation with the WebVCR. In: Proceedings of WWW 2000, pp. 503–517 (2000)
2. Davulcu, H., Yang, G., Kifer, M., Ramakrishnan, I.V.: Computational Aspects of Resilient Data Extraction from Semistructured Sources. In: Proc. of ACM Symposium on Principles of Database Systems (PODS), pp. 136–144 (2000)
3. Document Object Model (DOM) Level 3 Events Specification, `http://www.w3.org/TR/DOM-Level-3-Events/`
4. Google Web Toolkit, `http://code.google.com/webtoolkit/`
5. HtmlUnit, `http://htmlunit.sourceforge.net/`
6. Hupp, D., Miller, R.C.: Smart Bookmarks: automatic retroactive macro recording on the web. In: Proc. of ACM Symposium on User Interface Software and Technology (UIST 2007) (2007)
7. iOpus, `http://www.iopus.com`
8. Kapow, `http://www.openkapow.com`
9. Pan, A., Raposo, J., Álvarez, M., Hidalgo, J., Viña, A.: Semi automatic wrapper-generation for commercial web sources. In: Proc. of IFIP WG8.1 Working Conference on Engineering Information Systems in the Internet Context 2002, pp. 265–283 (2002)
10. QEngine, `http://www.adventnet.com/products/qengine/index.html`
11. Safonov, A., Konstan, J., Carlis, J.: Beyond Hard-to-Reach Pages: Interactive, Parametric Web Macros. In: Proc. of the 7th Conference on Human Factors & the Web (2001)
12. Sahi, `http://sahi.co.in/w/`
13. Selenium, `http://seleniumhq.org/`
14. Lingam, S., Elbaum S.: Supporting End-Users in the Creation of Dependable Web Clips. In: Proc. of WWW 2007, pp. 953–962 (2007)
15. XML Path Language (XPath), `http://www.w3.org/TR/xpath`
16. Yahoo! User Interface Library (YUI), `http://developer.yahoo.com/yui`