Efficient Execution of Web Navigation Sequences

José Losada, Juan Raposo, Alberto Pan, Paula Montoto

Information and Communications Technology Department, University of A Coruña. Facultad de Informática, Campus de Elviña, s/n, 15071, A Coruña (Spain)

{jlosada, jrs, apan, pmontoto}@udc.es

Abstract. Web automation applications are widely used for different purposes such as B2B integration and automated testing of web applications. Most current systems build the automatic web navigation component by using the APIs of conventional browsers. While this approach has its advantages, it suffers performance problems for intensive web automation tasks which require real time responses and/or a high degree of parallelism. In this paper, we outline a set of techniques to build a web navigation component able to efficiently execute web navigation sequences. These techniques detect what elements and scripts of the pages accessed during the navigation sequence are needed for the correct execution of the sequence (and, therefore, must be loaded and executed), and what parts of the pages can be discarded. The tests executed with real web sources show that the optimized navigation sequences run significantly faster and consume significantly less resources.

Keywords: Web Automation, Navigation Sequence, Optimization, Efficient Execution.

1 Introduction

Most today's web sources do not provide suitable interfaces for software programs. That is why a growing interest has arisen in so-called web automation applications that are able to automatically navigate through websites simulating the behavior of a human user. For example, a flight meta-search application can use web automation to automatically search flights in the websites of different airlines or travel agencies. Web automation applications are widely used for different purposes such as B2B integration, web mashups, automated testing of web applications, Internet meta-search or technology and business watch.

A crucial part of web automation technologies is the ability to execute automatic web navigation sequences. An automatic web navigation sequence consists in a

sequence of steps representing the actions to be performed by a human user over a web browser to reach a target web page. Figure 1 illustrates an example of a web navigation sequence to access to the content of the first message in the Inbox folder of a Gmail account.

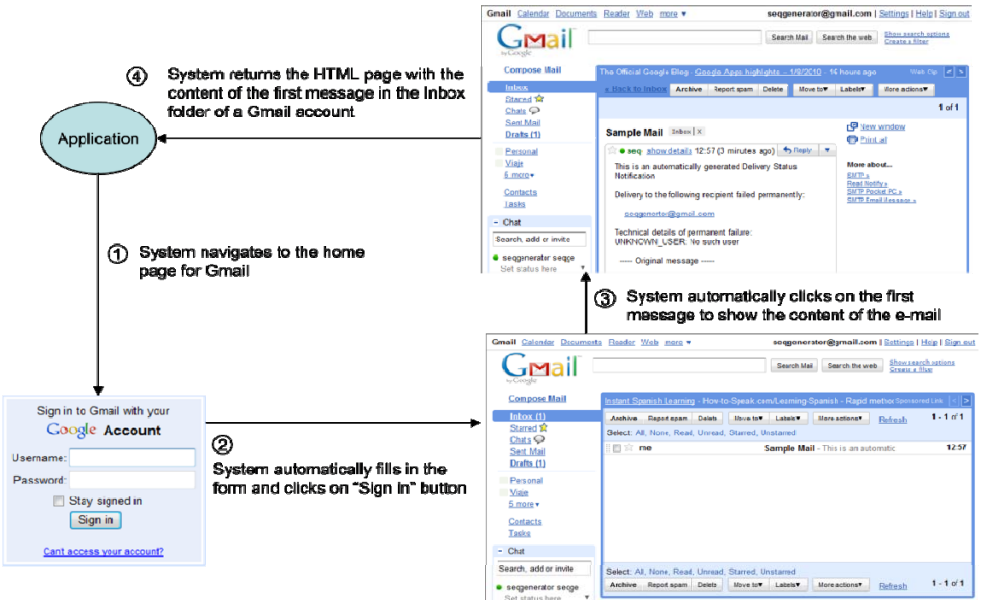

Fig. 1. Navigation Sequence Example

This work is focused in improving the performance of the execution of automatic web navigation sequences. The approach followed by most of the current web automation systems [6] [11] [12] [14] [15] consists in using the APIs of conventional web browsers to automate them. This approach does not require to develop a custom navigation component, and guarantees that the accessed web pages will behave the same as when they are accessed by a regular user.

While this approach is adequate to some web automation applications, it presents performance problems for intensive web automation tasks which require real time responses and/or to execute a significant number of navigation sequences in parallel. This is because commercial web browsers are designed to be client-side applications and, therefore, they consume a significant amount of resources, both memory and CPU. In this work we address this problem by using a custom browser specially built for web automation tasks. This browser is able to improve the response times and save a significant amount of resources (memory and CPU). We present a set of techniques and algorithms to automatically optimize the

navigation sequences, detecting which parts of the accessed pages can be discarded (not loaded), and which of the automatic events that are fired each time a new page is loaded can be omitted (not fired) without affecting to the correct execution of the navigation sequence.

There exist other systems which use the approach of creating custom browsers to execute web navigation sequences [5] [8]. Since they are not oriented to be used by humans, they can avoid some of the tasks of conventional browsers (e.g. rendering). Nevertheless, they work like conventional browsers when loading and building the internal representation of the web pages. Since this is the most important part in terms of the use of computational resources, their performance enhancements are much smaller than the ones achieved with our approach.

The rest of the paper is organized as follows. Section 2 briefly describes the models our approach relies on. Section 3 presents an overview of the solution. Section 4 explains the designed techniques in detail. Section 5 describes the experimental evaluation of the approach. Section 6 discusses related work. Finally, section 7 summarizes our conclusions.

2 Background

The main model we rely on is the Document Object Model (DOM) [4]. This model describes how browsers internally represent the HTML web page currently loaded in the browser and how they respond to user-performed actions on it. An HTML page is modelled as a tree, where each HTML element is represented by an appropriate type of node. An important type of nodes are the script nodes, used to place and execute a script code within the document (typically written in a script language such as JavaScript). The script nodes can contain the code directly or can reference an external file containing it. Those scripts are processed when the page is loaded and they can contain element declarations (e.g. a function or a variable) that are used from other script nodes or event listeners, or other script sentences that are executed at that moment.

Every 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 to indicate that a new page has just been loaded (*load*), to name but a few. Each node can register a set of listeners for different types of events. An event listener executes arbitrary script code, which normally calls a function declared in script nodes. The scripting code has the entire page DOM 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 somewhat of a compromise between the approaches historically used in major browsers (Microsoft Internet Explorer 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 reentrant, meaning that the execution of a listener can cause new events to be generated. Those new events will be processed in a synchronous way; that is, if l_i, l_{i+1} are two listeners registered to a certain event type in a given node in a 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.

In addition to the events caused by the user actions on the page, there are also some events that are automatically generated by the browser when a new page is

loaded. The most typical example is the *load* event, which is fired by the browser over the *body* element of the HTML page when the page has just been loaded. We will name these events as "automatic events".

3 Overview

This section presents an overview of our proposal.

The input for the automatic web navigation component is a navigation sequence specification. In most systems, this specification is created by example: the user performs the desired sequence manually and her actions are recorded by some plugin in the browser. The exact format used to specify navigation sequences is different in each web automation system but all of them basically consist in a list of events which must be generated on certain elements of the website pages. Between executing one event and the next, it is needed to wait for the effects of the previous event to take place (e.g. wait for a new page to be loaded in the browser). See [10] for a discussion of the different approaches for recording and executing web navigation sequences.
The basic idea of our approach consists in detecting which parts of the accessed pages can be discarded (not loaded) and which events can be omitted (not fired) without affecting the execution of the desired navigation sequence. Our approach works in two phases:

- In the optimization phase the navigation sequence is executed once, and, in the meantime, the navigation component automatically calculates which nodes of the HTML DOM [4] tree of each loaded page are needed to execute the sequence, and which ones can be discarded. Then, it stores some information to be able to detect those nodes in subsequent sequence executions (the information to identify the nodes should be resilient to small changes in the page, because in real web sites there are usually small differences between the DOM tree of the same page loaded at different moments). At the same time, the navigation component calculates which of the automatic events fired each time a page is loaded are necessary to execute the sequence.

- In the execution phase the navigation component executes the sequence using the optimization information previously calculated. When each page is loaded, a reduced HTML DOM tree is built, containing only the relevant nodes needed to execute the sequence, and only the necessary automatic events are fired.

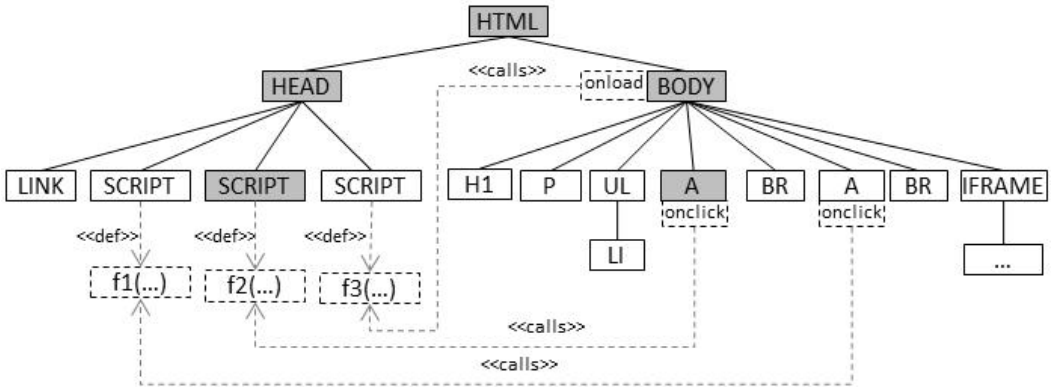

Fig. 2. DOM tree of an example page

Figure 2 shows the DOM tree of a simple example page. We use boxes to represent the nodes of the tree, and continuous lines to represent its parent-child relationship. Event listeners are represented as dashed boxes adjacent to the corresponding tree node (*onclick*, *onload*). Arrows with dashed lines are used to indicate that a script node defines a function (marked with <*def*>), and to indicate that the code of an event listener invokes a function defined in a script node (marked with <*calls*>). Suppose that the only action specified by the navigation sequence for this page is executing a *click* on the first *A* node. When the *click* event is produced, the *click* event listener (*onclick*) is executed, and the function *f2* performs a navigation to the desired page (e.g. *window.location = 'http://acme.com';*).

The shaded nodes are those that are needed to simulate the *click* action and properly perform the navigation to the next page (we call them *relevant* nodes). In this case, the relevant nodes are: the *A* node which is the target of the *click* event, the *SCRIPT* node which defines the *f2* function executed by the *click* event listener, and their respective ancestors (the exact rules to compute the relevant nodes will be described later). The rest of the nodes can be discarded (not loaded) without any problem (we call these ones *irrelevant* nodes). Besides, the automatic

load event does not need to be fired when the page is loaded, since the execution of the *onload* listener is not needed for the execution of the sequence.

This will produce significant performance and resource usage improvements:

- We will save memory, since much less nodes need to be represented.
- We will save CPU and execution time since unneeded scripts are not executed. For instance, in this case, the script nodes not shaded do not need to be executed.
- We will save bandwidth and execution time because unneeded navigations are not performed. For instance, in this case, the navigations specified by the *LINK* and *IFRAME* nodes will not be performed.

The main problem we need to address is how to calculate what we call *node dependencies*. For instance, in this example the *SCRIPT* node which defines *f2* is a dependency of the *A* node when the *click* event is fired on it (because it is needed to properly execute the *click* event listener registered in the *A* node). Notice that in the DOM model, scripts are "black boxes" and, therefore, these dependencies cannot be inferred directly. By using a custom browser, where we have full control over the script execution engine, we have a way to uncover these hidden dependencies.

Also notice that dependencies can get much more complex than in this example. For example, in the previous figure, a *click* on an anchor may produce the execution of a script that requires another script in a different node in the DOM tree to be executed previously. Another difficult example would be that the *load* event listener of the *BODY* node could generate content dynamically, including the *A* node that invokes the script that will lead us to the next page. It could even happen that the script requires another script contained in an iframe and, therefore, the iframe would need to be loaded too. We will see how to deal with these problems in the next section.

4 Proposed Techniques

In this section we begin stating some definitions and properties which will help us to model all the possible dependencies between the DOM tree nodes we are interested in (section 4.1). After that, we describe the techniques used during the optimization phase of our approach, (section 4.2). Then, we explain the method used to generate expressions to identify the irrelevant nodes at the execution phase (section 4.3). Finally we outline the operation at the execution phase (section 4.4).

4.1 Node Dependencies

Definition 1: We say that there exists a dependency between two nodes *n1* and *n2* when the node *n2* is necessary for the correct execution of the node *n1*. We say that the node *n2* is a dependency of the node *n1* and denote it as *n1*→ *n2*. The following rules define this type of dependencies:

- If the script code of a node *s1* uses an element (e.g. a function or a variable) declared in a script node *s2*, then *s1*→ *s2*. Rationale: to be able to execute the script code of the node *s1* the node *s2* must be executed previously.
- If the script code of a node *s* uses a node *n*, then *s*→ *n*. Rationale: to be able to execute the script code of the node *s*, the node *n* must be loaded previously. For instance, if *s* obtains a reference to an *anchor* node (e.g. using the JavaScript function *document.getElementById*) and navigates to the URL specified by its *href* attribute, then it will not be possible to execute *s* unless the *anchor* node is loaded.
- If the script code of a node *s* makes a modification in a node *n*, then *n*→ *s* (note that, in this scenario, the dependency *s*→ *n* also exists, applying the previous rule). Rationale: the action performed by *s* may be needed to allow *n* to be used later. For instance, if *s* modifies the *action* attribute of a *form* node to set the target URL, then it will not be possible to submit the form unless *s* is executed previously.

Definition 2: We say that there exists a dependency conditioned to the event *e* being fired over the node *n*, between two nodes *n1* and *n2*, when the node *n2* is

necessary for the correct execution of the node *n1*, when the event *e* is fired over the node *n*. We denote this as $n1 \rightarrow^{e|n} n2$. Analogous rules to the ones explained before define this type of dependencies, which, in this case, involve nodes containing event listeners:

- If the script code of an event listener *l* for the event *e* in the node *n* uses an element (e.g. a function or a variable) declared in a script node *s*, then $n \rightarrow^{e|n} s$. Rationale: if the event *e* is fired over the node *n*, then the event listener *l* is executed, and it requires the script node *s* to be executed previously.
- If the script code of an event listener *l* for the event *e* in the node *n1* uses a node *n2,* then $n1 \rightarrow^{e|n1} n2$. Rationale: if the event *e* is fired over *n1*, then the event listener *l* is executed and the node *n2* must be loaded previously.
- If the script code of an event listener *l* for the event *e* in the node *n1* makes a modification in a node *n2*, then $n2 \rightarrow^{e|n1} n1$ (note that, in this scenario, the dependency $n1 \rightarrow^{e|n1} n2$ also exists, applying the previous rule). Rationale: the action performed by *l* may be needed to allow *n2* to be used later. For instance, if *l* modifies the *action* attribute of a *form* node to set the target URL, then it will not be possible to submit the *form* unless *l* is executed previously. Since *l* will only be executed when the event *e* is fired over *n1*, then *n1* is needed.

Observe that the following transitivity properties apply to node dependencies (we will explain them through examples).

Property 1: If $n1 \rightarrow n2$ and $n2 \rightarrow n3$ then $n1 \rightarrow n3$.
The example of Figure 3.a shows a fragment of the DOM tree of a page where the script code of the node *SCRIPT1* invokes a function *f1* which is defined in the node *SCRIPT2* (*SCRIPT1*→ *SCRIPT2*), and the code of function *f1* calls a function *f2* which is defined in the node *SCRIPT3* (*SCRIPT2*→ *SCRIPT3*). For the correct execution of the script code of the node *SCRIPT1*, both the second and the third *SCRIPT* nodes are necessary, so both are dependencies of it (*SCRIPT1*→ *SCRIPT3*).

Property 2: If $n1 \rightarrow^{e|n} n2$ and $n2 \rightarrow n3$ then $n1 \rightarrow^{e|n} n3$.

The example of Figure 3.b shows a fragment of a page DOM tree where the *click* event listener of the node *A* calls a function *f1* which is defined in the *SCRIPT* node ($A \rightarrow^{click|A} SCRIPT$), and the code of the function *f1* uses the *src* attribute of the *IMG* node ($SCRIPT \rightarrow IMG$). For the correct processing of the *A* node when the *click* event is fired over it, both the *SCRIPT* and *IMG* nodes are necessary, so both are dependencies of it ($A \rightarrow^{click|A} IMG$).

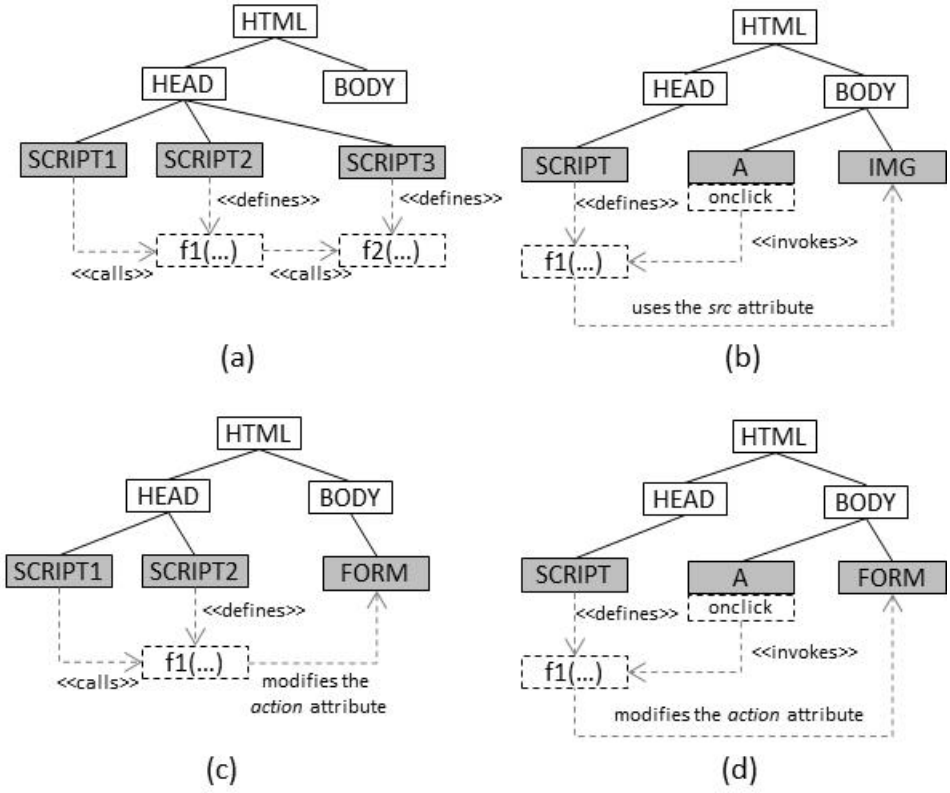

Fig. 3. Transitivity Dependency Examples

Property 3: If $n1 \rightarrow n2$, and $n3 \rightarrow n2$ because *n2* is a script node which makes a modification in *n3*, then $n3 \rightarrow n1$.

The example of Figure 3.c shows a fragment of a page DOM tree where the script code of the node *SCRIPT1* invokes a function *f1* which is defined in the node *SCRIPT2* ($SCRIPT1 \rightarrow SCRIPT2$), and the code of the function *f1* modifies the *action* attribute of the *FORM* node ($FORM \rightarrow SCRIPT2$). For the correct processing of the *FORM* node (for example to correctly submitting it), we need to ensure that *f1* is both defined (and, therefore, we need *SCRIPT2*) and executed (and, therefore, we need *SCRIPT1*). That is why both are dependencies of it ($FORM \rightarrow SCRIPT1$).

Property 4: If $n1 \rightarrow^{e|n} n2$ and $n3 \rightarrow n2$ because *n2* is a script node which makes a modification in *n3*, then $n3 \rightarrow^{e|n} n1$.
The example of Figure 3.d shows a fragment of a page DOM tree where the *click* event listener of the *A* node calls a function *f1* which is defined in the *SCRIPT* node ($A \rightarrow^{click|A} SCRIPT$), and the code of the function *f1* modifies the *action* attribute of the *FORM* node ($FORM \rightarrow SCRIPT$). For the correct processing of the *FORM* node (e.g. to correctly submitting it), when the *click* event is fired over the *A* node, both the *SCRIPT* and *A* nodes are necessary, so both are dependencies of it ($FORM \rightarrow^{click|A} A$).

4.2 Calculating the Relevant Nodes and Automatic Events

The main goal of the optimization phase is finding the set of relevant nodes for the navigation sequence. During this phase, the browser works in a similar manner to a conventional browser: the full page is loaded, generating the entire DOM tree, downloading all external elements (e.g. style sheets, script files) and executing all the script nodes defined in the page. Also, all the automatic events (recall section 2 for the definition of automatic events) are automatically fired by the browser when each new page is completely loaded (e.g. the *load* event is fired over the *body* element). After that, the browser will reproduce the desired navigation sequence by firing the necessary events on the adequate elements to emulate the user interaction with the page (e.g. clicking on elements, firing mouse events, etc.), until a navigation to a new page is started.

During all this process, the browser interacts with the script execution engine (we use Mozilla Rhino) to detect the node dependencies, according to the rules defined in the previous section. For instance, when a *script* node is executed, the browser interacts with the scripting engine to monitor what functions are called during its execution. Then, according to the first rule of Definition 1, the nodes defining those functions are marked as dependencies of the *script* node which calls them. Similarly, if the code of the *script* node creates or modifies another node, then, according to rule 3 of Definition 1, the *script* node will be a dependency of the node which is created or modified.

In a similar way, when an event (be it automatic or generated by the navigation sequence) is fired, the browser monitors which other nodes are used during the execution of the listeners associated to the event, which other events are generated and which nodes are modified by the execution of the event listeners. The appropriate dependencies according to the rules of Definition 2 will be generated.

Once the dependencies have been computed, the set of relevant nodes is built according to the following rules:

1. The nodes which are directly used in the target navigation sequence are relevant. For instance, if one step in the sequence is generating the *click* event on a *A* node, then that *A* node is relevant.
2. If a node *n* is relevant, all its ancestors are relevant. Note, that the ancestors could be needed because of the capture and bubbling phases of the event dispatching model of the DOM trees (see section 2).
3. By definition, if a node *n1* is relevant and $n1 \rightarrow n2$ then *n2* is relevant (all its dependencies are relevant too).
4. By definition, if a node *n1* is relevant, $n1 \rightarrow^{e|n} n2$, and the event *e* was fired over the node *n*, then *n2* is relevant (all its dependencies conditioned to the event *e* being fired over the node *n* are relevant too, if the event *e* was fired over *n*).
5. Some special rules apply to *form*-related nodes, to be able to properly submit forms:
   (a) If a *form* node is relevant, all the nodes corresponding to *input* and *select* elements contained in the *form* are relevant.
   (b) If an *input* or *select* node is relevant, the *form* node containing it is relevant.
   (c) If a *select* node is relevant, all its child *option* nodes are relevant.
6. A small set of nodes corresponding to some special element types are always considered relevant because they are needed to properly process other nodes of the page DOM tree. For instance, the *base* element sets the *base URL*, which means that the URLs specified by other elements are relative to it.

From the set of relevant nodes, we can easily calculate the set of irrelevant nodes which can be ignored at the execution phase. First, all the DOM tree nodes not contained in the set of relevant nodes are added to the set of irrelevant nodes.

Then, all the irrelevant nodes which have an ancestor also contained in the set are removed from it. The resulting set contains only the root nodes of the sub-trees whose descendants are all irrelevant. We call them irrelevant sub-trees.

Finally, to determine which of the automatic events are necessary for the correct execution of the sequence, the system checks, for each automatic event, if any of the relevant nodes has any dependency derived from it (i.e. it checks if a relevant node has been affected by the listeners executed as result of firing the event). If that is the case, the event is added to the list of automatic events that should be fired at execution time when the current page is loaded.

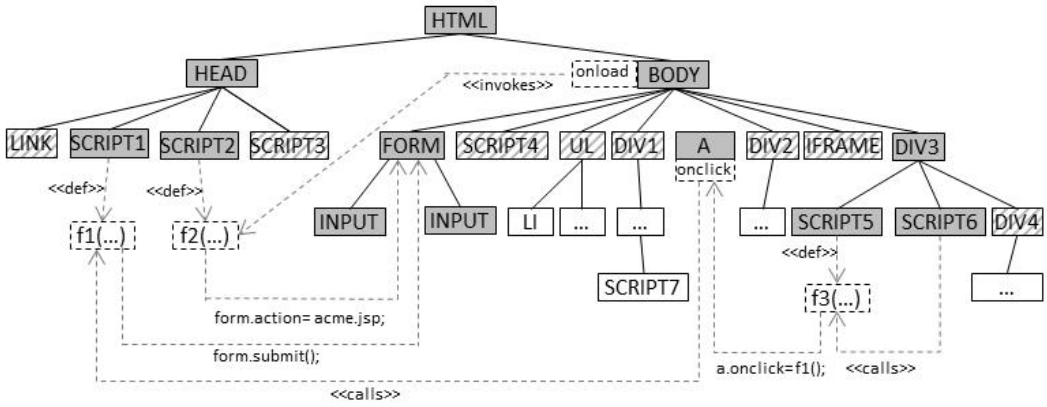

Fig. 4. Example

Let's see an example. Figure 4 shows a fragment of the DOM tree of a page. Suppose the target sequence specifies that the navigation component should execute a *click* over the *A* node. The relevant nodes for this interaction are shaded in the figure. Let's see how they are computed:

- According to rule 1, the node *A* is relevant (since it is the target of the action).
- According to rule 2, all the *A* ancestors are relevant: *BODY* and *HTML*.
- According to rule 3, all *A* dependencies are relevant: *SCRIPT5* and *SCRIPT6* (and its ancestors: *DIV3*). In this case they are needed because they execute script code which modifies the *click* event listener of the node *A* when the page is loaded.
    - The function *f3* (defined in *SCRIPT5*) modifies the *click* event listener of the node *A*, so *A* →*SCRIPT5*.

- *SCRIPT6,* which is executed when the page is loaded, invokes the function *f3*, so *SCRIPT6*→ *SCRIPT5,* and due to the transitivity rules explained in section 2, *A*→ *SCRIPT6.*

- According to rule 4, all *A* dependencies conditioned to the event *click* being fired over *A* are relevant too: *SCRIPT1* and *FORM* (and all its ancestors: *HEAD*). They are needed because the event listener of the node *A* invokes a function defined in *SCRIPT1* which submits the *form*.
  - The *click* event listener of the node *A* invokes the function *f1* defined in *SCRIPT1*, so $A \rightarrow^{click|A} SCRIPT1$.
  - The function *f1* uses the node *FORM*, so *SCRIPT1*→*FORM,* and due to the transitivity rules explained in section 2, $A \rightarrow^{click|A} FORM$.
- According to rule 5, if a *form* node is relevant, all the *input* nodes contained in the form are relevant: *INPUT1* and *INPUT2*. To properly submit the form all its input fields are necessary.
- According to rule 3, all *FORM* dependencies are relevant: *SCRIPT2* and *BODY* (and all its ancestors, already included in the set of relevant nodes). They are needed because the *load* event listener of the node *BODY* invokes a function defined in *SCRIPT2* which modifies the *action* attribute of the *form*.
  - The *load* event listener of the node *BODY* invokes the function *f2* defined in *SCRIPT2*, so $BODY \rightarrow^{load|body} SCRIPT2$.
  - The function *f2* (defined in *SCRIPT2*) modifies the *action* attribute of the node *FORM*, so *FORM* →*SCRIPT2*, and due to the transitivity rules explained in section 2, $FORM \rightarrow^{load|body} BODY$.

The nodes which are stripped in Figure 4 are those which are identified as the roots of the irrelevant sub-trees, which can be discarded in the following executions.

The automatic event *load*, which is fired over the *BODY*, must be added to the list of necessary automatic events, because the *FORM*, which is a relevant node, has a dependency derived from it ($FORM \rightarrow^{load|body} BODY$). Note that, to properly submit the form, the *load* event listener of the *body* element (*onload*) must have been executed, because it invokes *f2* which sets the action of the form.

4.3 Identifying the Irrelevant Subtrees at Execution Phase

Once the root nodes of the irrelevant sub-trees have been calculated, we need to generate expressions to be able to identify them at the execution phase. There are two requirements for this process. On one hand, the generated expressions should be resilient to small changes in the page because in real web sites there are usually small differences between the DOM tree of the same page loaded at different moments (e.g. new advertisement banners can appear or different data records can be shown). On the other hand, the process of testing if an expression identifies a node should be very efficient, because, at the execution phase the browser should check if each node is identified by any of those expressions before adding it to the DOM tree.

To uniquely identify a node in the DOM tree we use an XPath-like [16] expression which can contain information about the element and some of its ancestors. For our purposes, we need to ensure that the generated expression identifies a single node, but is not too specific to be affected by the aforementioned small changes in the pages. For this, we use an enhanced version of the algorithm explained in [10]. The basic idea of the algorithm consists in building an expression matching the minimum required number of nodes in the DOM tree (maximizing, this way, its resilience), using its tag name, its attributes, and its associated text.

An important concept is what we will call a "node expression". It is an XPath-like expression which only contains information about one node, and it has the following format:

//TagName[@a_1="v_1" and ... and @a_m="v_m" and text()="t"]

Where *TagName* is the tag name of the node, a_i and $v_{i\ i=\{1,\ldots,m\}}$ are names and values of attributes of the node, and *t* is the text of the node if it is a leaf node (being the *TagName* the unique element of the expression which is mandatory).

If the target node can be uniquely identified in the whole DOM tree with a node expression, then that is the result XPath-like expression to identify it. If it cannot

be uniquely identified (i.e. all the possible node expressions also match with other nodes in the DOM tree), then a node which can be uniquely identified with a node expression is searched in the path from the target node to the root of the tree. When it is found, the expression to identify the node is added to the result XPath-like expression, and the algorithm is applied again over the subtree whose root is that node. This way, the global resulting XPath-like expression would be composed by a sequence of node expressions:

$$//x_1//x_2//..//x_n$$

Where $//x_{i\ i\in\{1,...,n\}}$ are the node expressions built to uniquely identify a node in the subtree considered in each iteration of the algorithm. We define the length of an XPath-like expression as the number of node expressions compounding it.

Figure 5 gives the complete algorithm to generate the XPath-like expression to identify a node *n* contained in the DOM tree *T*. The repeat loop iterates until the target node *n* can be uniquely identified in the subtree considered in the current iteration (initially, the whole page DOM tree is considered). The while loop iterates from the target node *n* to the root of the subtree until a node which can be uniquely identified is found. When that node is found, the node expression to uniquely identify it (*x*) is added to the result expression (*result*) and the subtree considered in the next iteration of the repeat loop is set to the one which has that node as root.

The function *getNodeExp* receives as input a node and a subtree and tries to generate a node expression to uniquely identify the input node in the input subtree. If such expression uniquely identifying the node cannot be generated, it returns null.

A special case is considered at the end of the while loop, to deal with the case when there is not any node in the path from the target node to the root of the subtree which can be uniquely identified using exclusively the node data (i.e with a node expression). In that case, the function *getChildNodeExp* is called over the child node of the root of *S* (the current subtree) which is in the path to the target node. This function works in a similar way as the function *getNodeExp* but:

- It never returns null. It applies the considerations explained in the two following points to the node expression which identifies the fewer number of nodes (including the target node).
- The returned expression starts with "/" instead of "//". This means that the node must be a direct child of the last node whose information was added to the result expression (i.e. the root node of *S*, whose information was added to *result* in the previous iteration of the repeat loop). This allows differentiating this node from other nodes matching with the same node expression, which are not child nodes of the root of *S*.
- If necessary, it also uses the node position between its siblings to create an expression to uniquely identify it. This allows differentiating the node from other nodes matching with the same node expression, which are also child nodes of the root of *S*.

So, the final XPath-like expression will have the following format:

$$//x_1\,[\text{“//”} \mid \text{“/”}]\,x_2 \ldots [\text{“//”} \mid \text{“/”}]\,x_n$$

Note that the first node expression always starts with "//" because, if no other node is found before, the nodes HTML, BODY and HEAD always can be uniquely identified using only its tag name.

```
Algorithm: Generate an XPath-like expression to identify a node in a DOM tree
    - X = GenerateExpression(n,T)
Inputs:
    - n, the target node to be identified by the expression
    - T, the DOM tree where n is contained
Output:
    - result, the XPath-like expression to uniquely identify n in T.

result = "";   # Initialize the variable that will contain the result expression
S = T;         # Initialize the variable that will contain the subtree considered in each iteration
m = null;      # Auxiliary variable that will contain the node analyzed in each iteration

Repeat {   # Iterate until the target node n can be uniquely identified in S
    m = n;     # Initialize m to the target node n
    x = null;  # Initialize the variable that will contain the node expression generated to identify m

    # Iterate from n to the root of S until a node which can be uniquely identified is found or the root
    # of S is reached
    While (x==null && m!= root(S)) {
        x = getNodeExp(m, S);   # Returns an expression to uniquely identify m in S
                                # or null if such expression cannot be generated
        If (x != null) {    # The node can be uniquely identified in S
            result = result + x;
            S = <the subtree whose root is m>;
        } else {                # The node cannot be uniquely identified in S
            m = parent(m,T);  # Analyze the parent node in the tree
        }
    }
    If (m=root(S)) {        # No node can be uniquely identified in the path from n to the root of S.
        Let m' be the child of m which is in the path to the target node n;
        x = getChildNodeExp(m',S);   # Returns an expression to uniquely identify
                                     # m' as a child of m in S, using the node position if necessary
        result = result + x;
        S = <the subtree whose root is m'>;
    }
} Until (m==n);
return result;
```

Fig. 5. Algorithm to generate an XPath-like expression to identify a node

Let see now how the function *genNodeExp* tries to generate a node expression to uniquely identify a node in a subtree. As commented previously it only uses the node tag name, its attributes and its associated text (if it is a leaf node).

First, it tries to identify the element using only its tag name. If it is not enough, then it tries to use its tag name and its attributes. The algorithm considers some attributes as "more relevant" to identify a node. For example, the attribute *id*, in most of the cases, identifies a single node in the entire DOM tree by itself. Examples of other attributes considered as more relevant are *name*, *title*, *alt*, *value*, *for*, *src*, *action*, *href*, *class*, etc. The algorithm also considers some attributes as "less relevant" to identify a node. These attributes, in most of the cases, are not useful to identify the node (for example, when they only represent numeric values) and, besides, if they were used, the generated expression could be

weaker. Examples of some of these attributes are *cellpading*, *cellspacing*, *type*, *method*, *content*, *width*, *height*, *align*, *rel*, etc. Initially, the algorithm tries to generate an expression using only the more relevant attributes. If the node cannot be uniquely identified using those attributes, then it tries to generate an expression considering all the node attributes except the ones considered as less relevant. If the node cannot be uniquely identified either, then it tries to generate an expression using all the attributes.

If the attributes are not enough to uniquely identify the node, and it is a leaf node, then the algorithm tries to use the text of the node. First, it tries to generate an expression using exclusively the node tag name and its text, and if it is not enough it also uses its attributes (in the same way as commented previously).

Figure 6 shows a simple example illustrating several scenarios. It shows a fragment of a DOM tree, showing the set of attributes of each node beside it. The *SPAN* grayed node is the one to be identified. In the first iteration of the algorithm the whole DOM tree is considered (*S1*). The target *SPAN* node cannot be uniquely identified in *S1* because there are other *SPAN* nodes with the same attributes and values. So a node which can be uniquely identified is searched in the path to the root. The first one which is found is the *TABLE* node, which can be uniquely identified using its *id* attribute (note that the attribute *width* could also be used to identify the node but it is not present in the set of "more relevant" attributes, whereas the attribute *id* is). In the second iteration the target *SPAN* node cannot be uniquely identified in *S2*, and there is not any node, in the path to the root of *S2*, that can be. So, the child *TR* in the path to the target node is used to generate the node expression indicating that this node must be a direct child of the previous one (i.e. starting with "/"). In this case, the position must also be used to differentiate it from its sibling *TR* nodes. In the third iteration, the *SPAN* target node can be uniquely identified in *S3* because there is not any other *SPAN* node, in that subtree, having the value "*c2*" assigned to the attribute *class*.

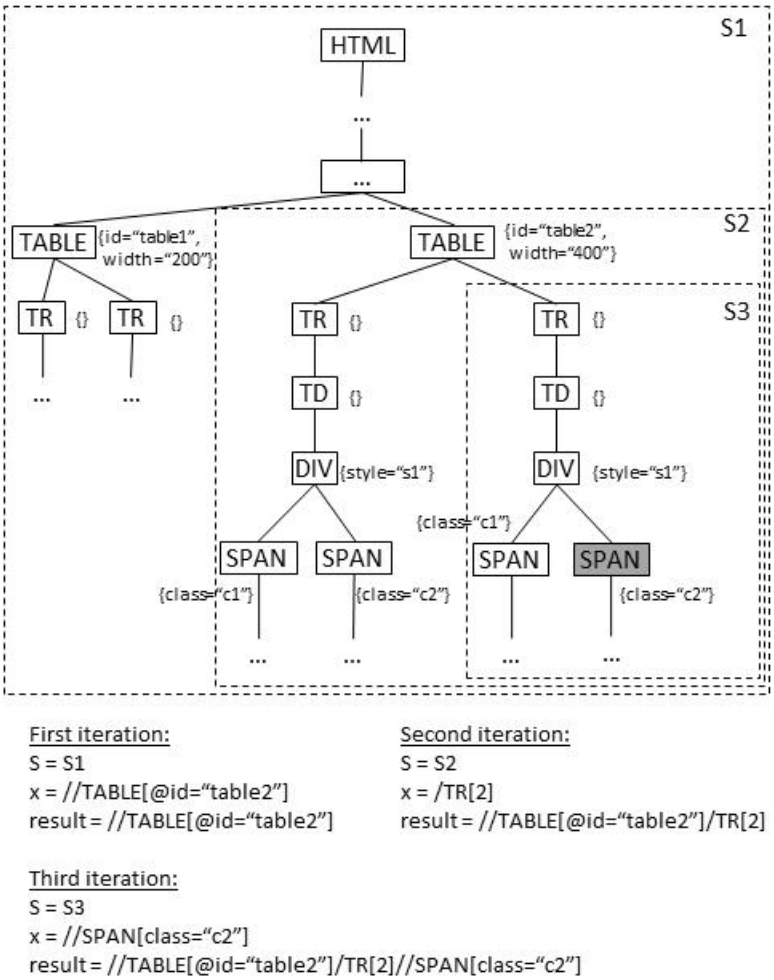

Fig. 6. XPath-like expression generation example

4.4 Execution Phase

The general functioning of the navigation component at this phase is the following one: before loading each page, it checks if it has optimization information regarding relevant nodes associated to that page, that is, a set of expressions to identify the root nodes of the irrelevant sub-trees. That information is used to build a reduced version of the HTML DOM tree, containing only the relevant nodes. Then it checks if it has optimization information related to automatic events that should be fired in that page. If that is the case, only the appropriate events are fired.

The process of checking if a node is the root of an irrelevant sub-tree should be very efficient because it is executed for all the elements present in the page to

decide if they must be added to the HTML DOM tree or not. That is why we do not use a conventional XPath matching algorithm. Instead, we leverage on the fact that the XPath-like expressions we generate use a strict subset of XPath and always verify certain restrictions. This allows us to use a faster algorithm for those particular expressions.

The main idea of the algorithm consists in checking, for each XPath-like expression, if there are nodes in the path from the analyzed node to the root of the tree which match with all the individual node expressions compounding it. Figure 7 gives the complete algorithm to check if a node is the root of an irrelevant subtree.

The external while loop iterates over the XPath-like expressions generated during the optimization phase. If any of the expressions identifies the node, then it is considered as irrelevant. To check if each XPath-like expression identifies the node, the first condition to check is if its last node expression matches with the target node. If it does not match, then that expression does not identify the node. On the contrary, if it matches and if the expression is compound by more node expressions, we need to check if there are nodes in the path to the root of the tree which match with all those node expressions. This is accomplished by the second while loop, which iterates over the individual node expressions previous to the last one. The main idea of each iteration of this loop consists in going up by the tree until a node which matches with the current node expression is found, but we need to consider the special case of the node expressions starting with "/" instead of "//" (note that, in this case, we can consider that a node matches with that node expression, only if its parent node matches with the previous node expression). So, the third while loop gets all the consecutive node expressions concatenated by "/" to create a partial XPath-like expression. Then, the fourth while loop iterates over the nodes in the path to the root of the tree, trying to find a list of consecutive nodes matching with this partial expression (i.e. each node of the list matches with the corresponding node expression contained in the partial XPath-like expression). Note that when the node expression analyzed in the second while loop does not start with "/", then the partial XPath-like expression built in the third while loop is

equal to the node expression, and the fourth while loop tries to find one node matching with it.

Algorithm: Check if a node matches with any of the XPath-like expressions which identifies the root nodes of the irrelevant subtrees

- result = CheckIfIrrelevantNode(n,X)

Inputs:

- n, the target node to check if it matches with any expression.
- X={X_1, …X_r}, where each $X_{i\ i\in\{1,...,r\}}$ is an XPath-like expression identifying the root node of an irrelevant subtree. Each X_i is an expression with the following format: //x_{i1} [“//” | “/”]x_{i2} … [“//” | “/”]x_{it} where [“//” | “/”]$x_{ik\ k\in\{1,...,t\}}$ is a node expression to identify a node using its tag name, attributes and/or text.
- T, the DOM tree built up to the moment, and where the node n will be added if it does not match with any expression in X.

Output:

- True if n matches with any $X_{i\ i\in\{1,...,r\}}$ or false in other case.

```
i = 1;     # Auxiliary expression counter
While (i<=r) {          # Process one XPath-like expression in each iteration
   m = n;               # Initialize m to the target node n
   k = length(Xi);      # Auxiliary counter, initialized to the number of node expressions in Xi
   If (matches(m,xik) {    # If the target node matches with the last node expression
      m = parent(m);    # Take the parent node
      k = k - 1;        # Point to the previous node expression
      While (k>0 && m!=null) {     # While there are node expressions left and parent nodes to match
         p = xik;                  # Partial expression initially set to the current node expression
         While (xik is preceded by “/”) {  # Add to p all the consecutive previous node expressions
            k = k - 1;                     # concatenated by “/”
            p = xik + “/” + p;
         }
         matched = false;
         While (m != null && !matched) {   # Iterates over nodes in the path to the tree root
            N = [m];          # Node list, initially containing the current node
            m’ = m;
            Repeat (length(p) -1) times {   # Add to N the same number
               m’ = parent(m’,T);           # of nodes as node expressions are in p
               append(N,m’);
            }
            if (matches(N,p) {     # If the partial expression matches with the list of nodes
               matched = true;
               m = parent(m’,T); # Continue with the parent node of the ones matched in this iteration
               k = k - 1;            # and the node expression previous to the ones matched in this iteration
            } else {
               m = parent(m,T);   # Try to match p from the parent node of the current one
            }
         }
      }
      If (k=0) {     # All the node expressions of Xi have been matched
         return true;
      } else {
         i = i+1;   # Analyze the next XPath-like expression
      }
   } else {
      i = i+1;   # Analyze the next XPath-like expression
   }
}
return false;
```

Fig. 7. Algorithm to check if a node is the root of an irrelevant subtree

Suppose we are building the DOM tree of the figure 5 and we have the expression generated to discard the grayed node (recall section 4.3). For each node which is added to the DOM tree we need to check if that expression identifies it:

- All the nodes which do not have the tag name *SPAN*, or have it but they do not have the attribute *class* equals to "*c2*", do not match with the last node expression (*//SPAN[class="c2"]*), so they are not identified by the expression.
- The first *SPAN* node with attribute *class* equals to "*c2*" matches with the last node expression. Then, the previous node expression is analyzed. In this case, the partial expression *//TABLE[@id="table2"]/TR[2]* (because they are concatenated by "/"). A list of two consecutive nodes matching this expression cannot be find in the path to the tree root (note that when analyzing the *TABLE* node and its first *TR* child, the expression does not match because of the position of the *TR* between the children of the *TABLE*), so the node is not identified by the expression.
- The second *SPAN* node with attribute *class* equals to "*c2*" matches with the last node expression. Besides, we are able to find two consecutive nodes in the path to the tree root matching the partial expression *//TABLE[@id="table2"]/TR[2]* (the *TABLE* node and its second *TR* child). At this point, all the node expressions compounding the XPath-like expression have been matched, so the expression identifies the node, and it is considered as the root of an irrelevant subtree. As a consequence, the node and all its descendants would be discarded, and not added to the DOM tree.

5 Evaluation

To evaluate the validity of our approach we implemented a custom browser. This browser emulates Microsoft Internet Explorer (MSIE) version 9 and was fully developed in Java using open-source libraries including Apache Commons-Httpclient to handle HTTP requests, Neko HTML parser to build DOM structures, and Mozilla Rhino as JavaScript engine. The browser neither has user interface nor renderization capabilities, but is able to simulate them, and it also supports CSS, cookies and Java Applets. Most of the JavaScript objects and functions

implemented in MSIE are also implemented in the custom browser including support for AJAX and some built-in ActiveX objects. Some MSIE advanced features are not implemented, including support for proprietary scripting languages (e.g. VBScript) or support for embedded objects (e.g. Adobe Flash). There are also some MSIE proprietary non-standard JavaScript functionalities not implemented in the custom browser.

This section explains the set of experiments that we have performed. We selected a set of websites of different domains included in the top 500 sites on the web according to Alexa [1]. In each website we recorded a navigation sequence representative of its main function (e.g. a product search in an e-commerce website). Every sequence executes events to fill and submit forms, to navigate through hyperlinks and, in some cases, to display content collected with AJAX requests.

In the first experiment, we compared the resources consumed by our custom browser when it uses its optimization capabilities, with the resources consumed in its normal operation mode (which emulates the behavior of the commercial browsers, loading the accessed pages entirely and firing all the automatic events). We ran a first execution of the navigation sequence, in each of the selected websites, to collect the optimization information. Then, we compared a normal execution of each sequence, without using the optimization information, and another one using it. To prevent the problem of small variations in web pages when they are accessed in different moments, each sequence was executed 10 times and the results shown in this section are the averages of the 10 executions.

Table 1 shows the following metrics for each web site:

- Mean number of XPath-like expressions generated per page. That is, the mean number of irrelevant subtrees identified per page.
- Mean length of the generated XPath-like expressions. That is, the mean number of “node expressions” per XPath-like expression.
- Total time consumed to calculate node dependencies (and the percentage it represents regarding the time consumed by the normal execution of the sequence).

- Total time consumed to calculate the necessary automatic events and the irrelevant nodes from the node dependencies, and to generate the XPath-like expressions identifying the root nodes of the irrelevant subtrees (and the percentage it represents regarding the time consumed by the normal execution of the sequence).
- Total time consumed by the normal execution of the sequence.

As we will demonstrate later, the number of XPath-like expressions (between 29.5 and 159.25 per page, with a global mean of 72.79 per page) is relatively small compared to the number of nodes which they allow discarding. The mean length of the expressions is always greater than 1 which implies that, all the sources contains nodes that cannot unambiguously be identified using only their text and/or their attributes. On the other hand, the mean length of the expressions is always fewer than 2, so the generated expressions contain information about a small number of nodes, having a high resilience to small changes. Finally, it can be observed that the time consumed to calculate node dependencies and generate the XPath-like expressions is quite small (globally, they represent, respectively, the 0.69% and the 2.51% of the time consumed by a normal execution), so, we can conclude that the process of calculating and colleting the optimization information is very efficient, and it could be executed frequently, if desired, to prevent the invalidation of the collected optimization information due to major changes in the websites pages.

Table 2 shows the following metrics comparing the normal and the optimized executions (each cell shows the result of the normal execution followed by the results of the optimized execution):

- Total number of HTML DOM tree nodes created.
- Total number of script nodes created and executed.
- Total number of frame and window objects created.
- Total number of HTML pages downloaded. Note that the number of frames and windows created can be greater than the number of HTML pages downloaded because some frames only execute JavaScript code without needing to download an HTML page.

- Total number of external objects downloaded (including JavaScript and CSS files).
- Total number of AJAX requests executed.

Measuring the resources used in all the navigation sequences, the optimized executions only require the 12.41% of the nodes. Discarding those nodes, the browser also avoids unnecessary downloads and the execution of unnecessary scripts, so the memory and CPU usage, is highly minimized. The optimized executions only execute the 24.85%% of the scripts, create the 31.11% of the frames and windows, download the 50.81% of the HTML documents and the 33.23% of the external objects, and execute the 29.03% of the AJAX requests.

The first five columns of Table 3 show the times consumed by the browser to perform the main tasks necessary to execute each navigation sequence (again, each cell shows the result of the normal execution followed by the results of the optimized execution). These tasks are:

- Build the DOM tree (this task include creating frames and windows when needed).
- Execute scripts.
- Download HTML pages.
- Download external objects (including JavaScript and CSS files).
- Execute AJAX requests.

The sixth column shows the time consumed, in the optimized execution, to check if the nodes are the root of an irrelevant subtree according to the optimization information (this task corresponds to the execution of the algorithm explained in the section 4.4, to decide if each node should be added to the DOM tree). Note that this time is part of the time consumed in the optimized execution to build the DOM tree, and which is shown in the first column.

Finally, the seventh column shows the total time consumed to execute the sequence (note that this time is not exactly the sum of the first five columns because the browser needs to execute other internal tasks to execute the navigation sequences).

Measuring the mean time consumed in all the navigation sequences, the optimized executions, compared to the normal ones, consume the 45.68%. By tasks, to build

the DOM tree they consume the 37.36%, to execute scripts the 37.69%, to download HTML pages the 67.37%, to download external objects the 33.3%, and to execute AJAX requests the 32.38%.
The last row shows the total time which the optimized executions save in each task, and the percentage which it represents regarding the total time of the normal executions. As can be seen, checking if the nodes should be added to the DOM tree only adds a penalization of the 0.18%, which is insignificant compared to the time savings in all the tasks. Even if we consider only the task of building the DOM tree, which in the optimization execution includes the time to check if the nodes should be added to the DOM tree, a 1.02% of the time is saved (this is explained because creating objects is a much more expensive operation than comparing strings). Executing scripts it is saved a 16.42% of the time, downloading HTML pages a 10.08%, downloading external objects a 25.27%, and executing AJAX requests a 1.65%. Globally a 54.32% of the time is saved.

In the second experiment we compared the execution time of our custom browser using and without using its optimization capabilities, with the execution time of other representative navigation components. We used a navigation component based on another custom browser, in this case, we chose HtmlUnit [5] because it is an open source project and also supports JavaScript and CSS, and a navigation component using the APIs of two commercial web browsers, in this case Microsoft Internet Explorer 9 and Mozilla Firefox 19.0. Table 4 shows the average execution time of 20 consecutive executions of each of our test navigation sequences, discarding those that don't fit in the range of the standard deviation. The table 4 also shows, between brackets, the percentage they represent in comparison with the execution time of our custom browser using its optimization capabilities. The last four rows show, respectively, the following aggregate metrics about the time percentages: the average, the standard deviation, the average discarding those results that do not fit in range of the average ± standard deviation, and the median.

The execution time of the custom browser using its optimization capabilities always got better results. Compared with the executions without optimization, the execution time varies from 141% in the worst case to the 651% in the best case.

Calculating the average of the percentages, the execution time of the custom browser without optimization is 2.44 times slower (244%) than the execution time with optimization. Discarding the results that do not fit in range of the average ± the standard deviation (the standard deviation is 45%), the execution time of the custom browser without optimization is 2.01 times slower (201%). The median value of the executions indicates that the custom browser without optimization is 2.19 times slower (219%).

Regarding the other browsers, the HtmlUnit custom browser is the one that got better results. In average it is 3.55 times slower than our custom browser with optimization (2.48 times if we discard the results that do not fit in range of the average ± the standard deviation), and the median of the executions indicates that it is 3.01 times slower. In the case of the navigation components based on Microsoft Internet Explorer and Mozilla Firefox, the average execution times are 6.34 and 5.19 times slower than the execution time of the custom browser with optimization (4.49 and 3.85 times if we discard the results that do not fit in range of the average ± the standard deviation), and the median of the executions indicates that they are 5.07 and 4.63 times slower, respectively.

The website where the optimized execution got better results was W3CSchools. As can be seen in Tables 2 and 3 it is because in the normal execution it downloaded 33 external objects and executed 89 scripts, but none of them were necessary in the optimized execution. This allows saving a lot of time in the corresponding tasks. The worst result was obtained in the website Barnes&Noble. As can be seen in Table 2, in this website the optimized exeuction could build a smaller DOM tree, but it needed to download the same external objects and HTML pages, and executed the same scripts. In Table 3, it can be observed that the optimized execution saves time building the DOM tree and also executing scripts, although the same ones are executed. The scripts are executed faster in the optimized execution because some of them contain operations which are executed faster when applied to a reduced DOM tree (for example if they access to the collection which contains all the nodes of the tree).

6 Related Work

Currently, web automation applications are widely used for different purposes. The approach followed by most of the current web automation systems, like Smart Bookmarks [6], Wargo [11], QEngine [12], Sahi [14], Selenium [15], and Montoto et al. [7] consists in using the APIs of conventional web browsers to automate them. This approach has two important advantages: it does not require to develop a new browser (which is costly), and it is guaranteed that the page will behave in the same way as when a human user access the page with her browser. Nevertheless, it presents performance problems for intensive web automation tasks which require real time responses and/or to execute a significant number of navigation sequences in parallel. This is because commercial web browsers are designed to be client-side applications and, therefore, they consume a significant amount of resources and time, as we have demonstrated in the evaluation section.

Other systems use the approach of creating simplified custom browsers specially built for the task. WebVCR [2] and WebMacros [13] rely on simple HTTP clients that lack the ability to execute complex scripting code or to support AJAX requests. Our custom browser supports all those complexities.

HtmlUnit [5] and Kapow [8] use their own custom browser with support for many JavaScript and AJAX functionalities. They are more efficient than commercial web browsers, because they are not oriented to be used by humans and can avoid some tasks (e.g. rendering). Nevertheless, HtmlUnit works like conventional browsers when loading and building the internal representation of the web pages. The last versions of Kapow are not downloadable, but to the best of our knowledge it also works like conventional browsers regarding this issue. Since this is the most important part in terms of the use of computational resources, their performance enhancements are much smaller than the ones achieved with our approach, as we have demonstrated in the evaluation section.

Related to the problem of identifying elements in web pages, some systems [2] [7] [11] [12] [13] [14] use the text associated to the elements and the value of some specific pre-configured attributes (e.g. href for A tags). In complex websites it is frequent that some elements cannot unambiguously be identified by their text

and/or the value of their attributes (as our experiments have demonstrated). Smart Bookmarks [6] can also generate full XPath expressions pointing to the target element when the above strategy does not uniquely identify it. But these expressions are not resilient to small changes on the page loaded at different moments. Selenium [15] generates XPath expressions to identify the target element trying to make them resilient to changes but they consider only some pre-defined attributes (e.g. *id*, *href*). Kapow [6] generates an XPath-like expression that tries to be resilient to small changes, although the details of the algorithm they use have not been published. Works like [3] [9] have also addressed the problem of generating change-resilient XPath expressions, but in those approaches, the user have to provide several example pages identifying the target element.

7 Conclusions

In this paper, we have presented a novel set of techniques and algorithms to efficiently execute web navigation sequences. Our approach is based on executing the navigation sequence once, to automatically collect information about the elements of the loaded pages that are irrelevant for that navigation sequence. Then, that information is used in the next executions of the sequence, to load only the required elements and fire only the required events.

To evaluate the proposed techniques and algorithms, they have been implemented in the core of a custom browser, developed for this purpose. According to our experiments the techniques are very effective: smaller DOM tree nodes are built, unneeded scripts are not executed and unneeded navigations are not performed. This way, the techniques allow to save bandwidth, memory and CPU usage, and to execute the navigation sequences faster compared with the same custom browser without using its optimization capabilities, and with other representative navigation components.

Acknowledgments. This research was partially supported by the Spanish Ministry of Science and Innovation under projects TIN2009-14203 and TIN2010-09988-E, and the European Commission under project FP7-SEC-2007-01 Proposal Nº 218223.

8 References

[1] Alexa, The Web Infomration Company. http://www.alexa.com

[2] Anupam V., Freire J., Kumar B., Lieuwen D., Automating web navigation with the WebVCR, Computer Networks 33(1-6), 503-517 (2000)

[3] Davulcu H., Yang G., Kifer M. and Ramakrishnan I.V, Computational Aspects of Resilient Data Extraction from Semistructured Sources, ACM Symposium on Principles of Database Systems (PODS) 2000, pp. 136-144.

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

[5] HtmlUnit, http://htmlunit.sourceforge.net/

[6] Hupp D., Miller R.C.: Smart Bookmarks: automatic retroactive macro recording on the web. In: Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology, pp. 81-90. ACM New York, Newport (2007)

[7] iOpus, http://www.iopus.com

[8] Kapow, http://www.openkapow.com

[9] Lingam S., Elbaum S., Supporting End-Users in the Creation of Dependable Web Clips. WWW 2007, 953-962.

[10] Montoto P., Pan A., Raposo J., Bellas F, López J.: Automated browsing in AJAX websites. Data Knowl. Eng. 70(3), 269-283 (2011)

[11] Pan A., Raposo J., Álvarez M., Hidalgo J., Viña A.: Semi automatic wrapper-generation for commercial web sources. In: IFIP WG8.1 Working Conference on Engineering Information Systems in the Internet Context, pp. 265-283. Kluwer, B.V. Deventer, Japan (2002)

[12] QEngine, http://www.adventnet.com/products/qengine/index.html

[13] Safonov A., Konstan J., Carlis J.: Beyond Hard-to-Reach Pages: Interactive, Parametric Web Macros. In: 7th Conference on Human Factors & the Web. Madison 2001

[14] Sahi, http://sahi.co.in/w/

[15] Selenium, http://seleniumhq.org/

[16] XML Path Language (XPath), http://www.w3.org/TR/xpath

Figure Legends

Tables

Table 1. Metrics about the optimization phase

	Irrelevant nodes per page	XPATH-like expressions length	Time calculating node dependencies	Time generating XPATH-like expressions	Total execution time
Alexa	43,5	1,2	36 (0,81%)	95 (2,15%)	4426
Amazon	78,25	1,14	46 (0,54%)	364 (4,26%)	8549
AppleStore	57,67	1,6	61 (1,44%)	130 (3,07%)	4228
Barnes&Noble	49	1,19	54 (0,75%)	149 (2,07%)	7187
Bloomberg	66,67	1,29	56 (0,71%)	322 (4,07%)	7908
CNET	71,67	1,23	92 (0,8%)	177 (1,53%)	11563
CNN	37,33	1,46	72 (0,77%)	218 (2,35%)	9294
Ebay	159,25	1,38	39 (0,47%)	693 (8,27%)	8377
Flickr	30,25	1,39	70 (0,75%)	67 (0,72%)	9338
GoogleNews	47,25	1,38	42 (0,72%)	187 (3,22%)	5810
Imdb	120,33	1,35	79 (0,84%)	268 (2,86%)	9361
Linkedin	86,33	1,92	43 (0,69%)	252 (4,04%)	6230
Reference	152,5	1,28	95 (0,75%)	189 (1,5%)	12639
Reuters	52,75	1,62	99 (0,51%)	212 (1,1%)	19341
Softonic	59,75	1,27	38 (0,58%)	184 (2,8%)	6579
Spiegel	123,5	1,48	48 (0,5%)	376 (3,93%)	9570
StackOverflow	34,67	1,3	56 (0,83%)	121 (1,79%)	6770
Taringa	72,67	1,25	65 (0,47%)	126 (0,92%)	13746
Theguardian	106,33	1,18	124 (1,01%)	383 (3,13%)	12219
Tripadvisor	41,75	1,18	15 (0,3%)	307 (6,24%)	4921
W3CSchools	53	1,36	51 (0,63%)	89 (1,09%)	8143
Walmart	97	1,26	73 (0,58%)	531 (4,23%)	12554
Wikipedia	51	1,67	81 (1,13%)	249 (3,46%)	7192
Wordpress	29,5	1,17	37 (0,64%)	41 (0,71%)	5776
WSJournal	90	1,55	149 (0,71%)	229 (1,09%)	21028
Yahoo	77,5	1,06	45 (0,51%)	107 (1,21%)	8875
Yelp	76	1,13	41 (0,61%)	176 (2,62%)	6706
Global	**72,79**	**1,34**	**1707 (0,69%)**	**6242 (2,51%)**	**248330**

Table 2. Metrics comparing normal and optimized executions

	HTML DOM nodes created	Scripts executed	Frames and Windows	HTML pages downloaded	External objects downloaded	AJAX requests
Alexa	1176/144	48/20	1/1	2/2	27/16	0/0
Amazon	7965/4047	176/77	6/2	9/4	13/5	2/1
AppleStore	2611/79	69/1	1/1	3/3	15/11	2/0
Barnes&Noble	3989/136	26/26	1/1	4/4	14/14	0/0
Bloomberg	6281/187	243/28	14/11	8/7	53/6	0/0
CNET	3395/157	113/56	7/4	9/6	52/24	0/0
CNN	4539/40	103/8	6/1	7/3	30/5	0/0
Ebay	4932/3175	80/37	4/1	8/4	25/9	0/0
Flickr	1332/61	61/9	2/1	5/4	19/1	0/0
GoogleNews	7460/114	48/11	2/1	4/4	9/3	0/0
Imdb	2608/485	183/56	28/1	8/3	34/10	4/3
Linkedin	2095/167	52/12	3/1	5/3	20/5	3/0
Reference	2709/579	152/29	7/2	9/3	33/11	0/0
Reuters	2797/298	265/50	11/2	12/3	156/41	4/1
Softonic	4932/250	79/6	12/1	15/4	17/3	0/0
Spiegel	3361/139	92/25	20/3	21/4	22/7	1/0
StackOverflow	3950/153	43/9	1/1	3/3	21/5	4/1
Taringa	2530/256	209/15	10/1	13/3	47/8	7/0
Theguardian	4519/248	257/70	5/1	4/3	76/28	0/0
Tripadvisor	6769/88	92/14	1/1	4/4	6/0	0/0
W3CSchools	2380/32	89/0	8/1	8/3	33/0	0/0
Walmart	6926/385	208/29	15/3	4/3	42/13	4/3
Wikipedia	5078/143	52/24	1/1	4/4	37/21	0/0
Wordpress	472/37	56/21	6/1	7/2	18/10	0/0
WSJournal	6303/1148	204/118	39/24	60/35	78/50	0/0
Yahoo	1946/85	127/33	7/1	8/2	16/2	0/0
Yelp	2815/508	52/6	7/1	2/2	14/0	0/0
Total	**105870/13141 (12,41%)**	**3179/790 (24,85%)**	**225/70 (31,11%)**	**246/125 (50,81%)**	**927/308 (33,23%)**	**31/9 (29,03%)**

Table 3. Times comparing normal and optimized executions

	Building DOM Tree	Executing Scripts	Downloading HTML pages	Downloading external objects	Executing AJAX requests	Checking irrelevant nodes	Total time
Alexa	108/25	1733/952	820/825	1716/947	0/0	3	4426/2782
Amazon	170/76	1792/873	4700/3544	849/190	940/244	5	8549/5019
AppleStore	152/27	2881/1305	193/212	529/391	261/0	4	4228/2003
Barnes&Noble	122/42	4171/2173	1910/1912	934/912	0/0	5	7187/5094
Bloomberg	118/62	3443/281	889/656	3392/525	0/0	9	7908/1593
CNET	108/75	2575/1063	4710/3949	4097/1839	0/0	42	11563/7065
CNN	100/47	1988/652	3410/1562	3747/468	0/0	2	9294/2779
Ebay	135/106	3102/1442	3716/2580	1318/1005	0/0	47	8377/5274
Flickr	111/39	1965/530	2999/2569	4221/480	0/0	9	9338/4055
GoogleNews	263/50	2393/584	1345/1387	1757/328	0/0	5	5810/2414
Imdb	193/65	3095/1501	2513/1335	2737/676	756/614	23	9361/4279
Linkedin	94/42	1085/499	2634/1752	1686/473	639/0	24	6230/2839
Reference	147/53	3103/1252	3035/1514	6300/1754	0/0	24	12639/4694
Reuters	179/40	3105/749	2817/620	12485/3983	686/190	17	19341/5621
Softonic	114/52	1614/647	2258/1654	2542/870	0/0	10	6579/3272
Spiegel	202/62	1102/567	3432/1101	4791/2519	15/0	15	9570/4297
StackOverflow	81/41	1471/570	654/758	3255/703	1279/244	6	6770/2341
Taringa	192/32	3798/980	4054/2244	5083/1241	567/0	8	13746/4546
Theguardian	126/56	4090/1985	438/397	7486/3023	0/0	15	12219/5604
Tripadvisor	217/41	768/53	3225/3207	617/0	0/0	7	4921/3353
W3CSchools	176/30	2219/0	1927/1108	3787/0	0/0	13	8143/1251
Walmart	171/63	5028/1717	3315/2343	3067/783	910/668	15	12554/5633
Wikipedia	77/50	1977/1261	824/850	4148/2100	0/0	7	7192/4309
Wordpress	96/46	568/277	2822/1064	2266/1371	0/0	34	5776/2792
WSJournal	184/231	3894/2402	11041/6984	5800/3877	0/0	72	21028/13621
Yahoo	101/21	926/333	4195/2736	3597/875	0/0	6	8875/4013
Yelp	305/36	1538/11	2823/2806	1874/0	0/0	14	6706/2906
Total	**4042/1510 (37,36%)**	**65424/24659 (37,69%)**	**76699/51669 (67,37%)**	**94081/31333 (33,3%)**	**6053/1960 (32,38%)**	**0/441**	**248330/113449 (45,68%)**
Time Savings	**2532 (1,02%)**	**40765 (16,42%)**	**25030 (10,08%)**	**62748 (25,27%)**	**4093 (1,65%)**	**-441 (-0,18%)**	**134881 (54,32%)**

Table 4. Average execution times in milliseconds

	Custom browser with optimization	Custom browser without optimization	HtmlUnit	Internet Explorer	Mozilla Firefox
Alexa	2782	4426 (159%)	5329 (192%)	13902 (500%)	13152 (473%)
Amazon	5019	8549 (170%)	10927 (218%)	22320 (445%)	18584 (370%)
AppleStore	2009	4228 (210%)	5043 (251%)	16953 (844%)	16253 (809%)
Barnes&Noble	5094	7187 (141%)	6422 (126%)	27578 (541%)	26390 (518%)
Bloomberg	1593	7908 (496%)	18081 (1135%)	34744 (2181%)	26710 (1677%)
CNET	7065	11563 (164%)	17537 (248%)	26615 (377%)	21389 (303%)
CNN	2779	9294 (334%)	21763 (783%)	20392 (734%)	14649 (527%)
Ebay	5274	8377 (159%)	12286 (233%)	22993 (436%)	17894 (339%)
Flickr	4055	9338 (230%)	11813 (291%)	21277 (525%)	14124 (348%)
GoogleNews	2414	5810 (241%)	8599 (356%)	27337 (1132%)	16783 (695%)
Imdb	4279	9361 (219%)	11429 (267%)	21530 (503%)	16629 (389%)
Linkedin	2839	6230 (219%)	5839 (206%)	17941 (632%)	13135 (463%)
Reference	4694	12639 (269%)	19650 (419%)	17849 (380%)	17364 (370%)
Reuters	5621	19341 (344%)	22261 (396%)	20323 (362%)	19562 (348%)
Softonic	3272	6579 (201%)	14048 (429%)	16600 (507%)	18893 (577%)
Spiegel	4297	9570 (223%)	12948 (301%)	14562 (339%)	14513 (338%)
StackOverflow	2341	6770 (289%)	6377 (272%)	19113 (816%)	13681 (584%)
Taringa	4546	13746 (302%)	14614 (321%)	18690 (411%)	17569 (386%)
Theguardian	5604	12219 (218%)	18490 (330%)	23730 (423%)	27909 (498%)
Tripadvisor	3353	4921 (147%)	14942 (446%)	24896 (742%)	18772 (560%)
W3CSchools	1251	8143 (651%)	8793 (703%)	19049 (1523%)	12407 (992%)
Walmart	5633	12554 (223%)	20183 (358%)	20998 (373%)	20896 (371%)
Wikipedia	4309	7192 (167%)	10711 (249%)	18742 (435%)	14524 (337%)
Wordpress	2792	5776 (207%)	6373 (228%)	16020 (574%)	14177 (508%)
WSJournal	13621	21028 (154%)	19201 (141%)	21719 (159%)	19087 (140%)
Yahoo	4013	8875 (221%)	13496 (336%)	21816 (544%)	16639 (415%)
Yelp	2906	6706 (231%)	10228 (352%)	20035 (689%)	19828 (682%)
Average		244%	355%	634%	519%
Standard Dev.		109 (45%)	210 (59%)	404 (64%)	284 (55%)
Average ± Stdev.		201%	248%	449%	385%
Median		219%	301%	507%	463%