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.

adfa, p. 1, 2011.

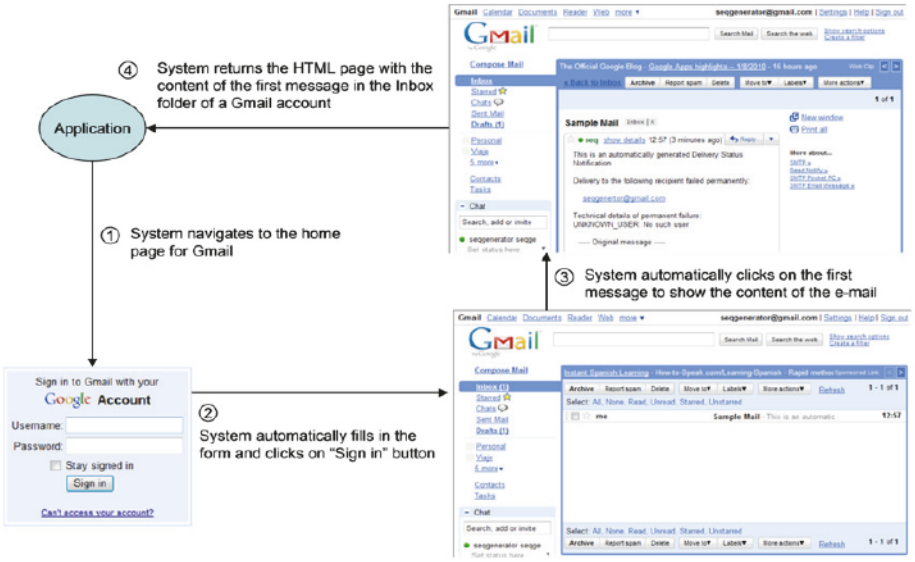

Fig. 1. Navigation Sequence Example

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 [5] [8] [9] [11] [12] 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 [4] [6]. 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 and future work.

2 Background

The main model we rely on is the Document Object Model (DOM) [3]. 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 used from other nodes).

In addition, 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. Each event is dispatched following a path from the root of the tree to the target node (capture phase) and then from the target node to the root of the tree (bubbling phase), and it can be handled locally at the target node or at any target's ancestor in the tree (at the capture or bubbling phase). 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.

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 [7] 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 [3] 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). 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.

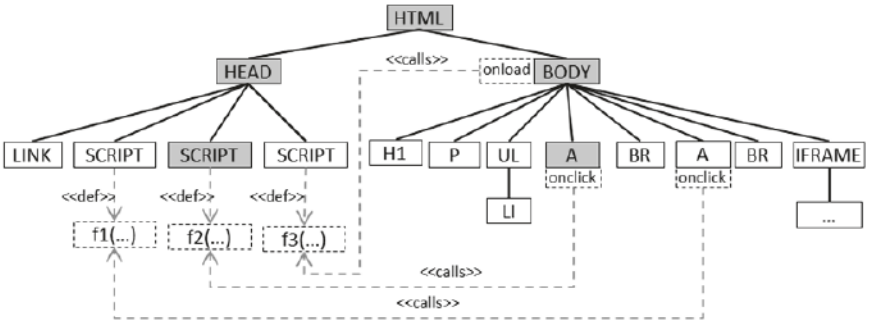

Fig. 2. DOM tree of an example page

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 the functions defined in script nodes (*<def>*), and the functions defined in script nodes which are invoked from event listeners (*<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 *onclick* event listener 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 *onclick* 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 code 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. 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 required 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 briefly 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 $n1 \rightarrow n2$. 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 $s1 \rightarrow s2$. 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 $s \rightarrow n$. 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 $n \rightarrow s$. 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 $n2 \rightarrow^{e|n1} n1$. 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 ($A \rightarrow^{click|A} IMG$).

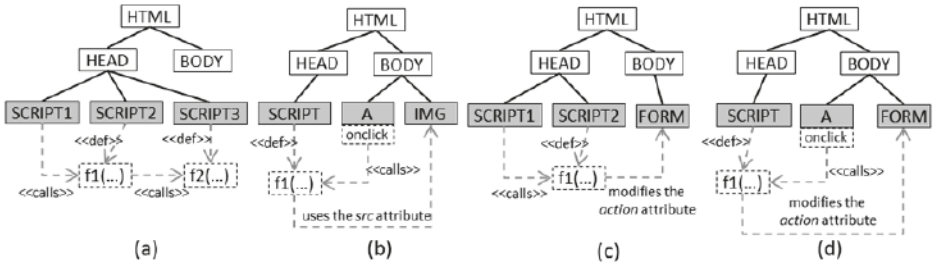

Fig. 3. Transitivity Dependency Examples

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 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. 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 changed node.

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 will be removed 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.

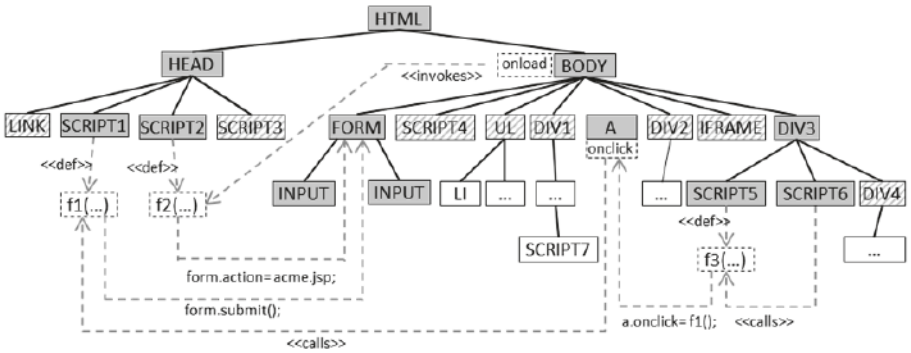

Fig. 4. Example

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 handler of the node *A* when the page is loaded.
  - The function *f3* (defined in *SCRIPT5*) modifies the *click* event handler of the node *A*, so $A \rightarrow SCRIPT5$.
  - *SCRIPT6,* which is executed when the page is loaded, invokes the function *f3*, so $SCRIPT6 \rightarrow SCRIPT5$, and due to the transitivity rules explained in section 2, $A \rightarrow SCRIPT6$.
- 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 handler of the node *A* invokes a function defined in *SCRIPT1* which submits the *form*.
  - The *onclick* event handler 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 $SCRIPT1 \rightarrow FORM$, 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 handler of the node *BODY* invokes a function defined in *SCRIPT2* which modifies the *action* attribute of the *form*.
  - The *onload* 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 $FORM \rightarrow SCRIPT2$, 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 a node matches an expression should be very efficient, because, at the execution phase the browser should check if each node matches with any of those expressions.

To uniquely identify a node in the DOM tree we use an XPath-like [13] expression. 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 [7], which is not described in detail here due to space constraints.

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 allow us to use a much faster algorithm for those particular expressions. The algorithm is not described due to space constraints.

5 Evaluation

To evaluate the validity of our approach we implemented a custom browser. This browser 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.

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 a normal execution (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 to collect the optimization information. Then, we ran one normal execution without using the optimization information, and another one using it. Table 1 shows the metrics measured for each of the two executions in the selected websites (each cell shows the result of the normal execution followed by the result of the optimized one). To prevent the problem of small variations in web pages when they are accessed in different moments, each sequence was executed 5 times and the result is the average of the 5 executions.

Measuring the resources used in all the navigation sequences, the optimized executions only require the 18% 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 27% of the scripts and download the 60% of the HTML documents, the 34% of the external scripts and CSS stylesheets, and execute the 36% of the AJAX requests.

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 [4] because it is an open source project and also supports JavaScript and CSS, and a navigation component

using the APIs of a commercial web browser, in this case Microsoft Internet Explorer 9. Table 2 shows the average execution time of 20 consecutive executions of each of our test navigation sequence, discarding those that don't fit in the range of the standard deviation. The table also shows the percentage of time in comparison with the execution time of our custom browser using its optimization capabilities.

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 125% in the worst case to the 472% in the best case. Calculating the average of the percentages, and discarding those that do not fit in range of the average ± standard deviation (the standard deviation is 38%), the execution time of the custom browser without optimization is 2.04 times slower (204%) than the execution time with optimization. The median value of the executions indicates that the custom browser without optimization is 2.08 times slower. With the same calculations, the custom browser based on HtmlUnit is 3.15 times slower (315%) than the custom browser with optimization and the median of the executions is 3.13. In the case of the navigation component based on Microsoft Internet Explorer, the average execution time is 5.03 times slower (503%) than the execution time of the custom browser with optimization and the median of the executions is 4.87.

Table 1. Metrics comparing normal and optimized executions in some websites

	HTML DOM Nodes created	Scripts Executed	Frames and Windows	HTML pages Downloaded	External objects Downloaded	AJAX Requests
Alexa	1377 / 91	46 / 17	1 / 1	2 / 2	24 / 13	0 / 0
Amazon	8604 / 4496	223 / 77	23 / 3	22 / 5	43 / 7	5 / 2
Apple	2900 / 395	51 / 15	1 / 1	3 / 3	14 / 10	1 / 1
Barnes	8814 / 6021	73 / 41	3 / 1	6 / 3	45 / 26	0 / 0
Bloomberg	5880 / 397	204 / 46	4 / 4	5 / 5	82 / 17	1 / 0
CNET	3650 / 104	110 / 45	5 / 4	7 / 6	53 / 26	0 / 0
CNN	4167 / 97	138 / 14	9 / 1	12 / 3	46 / 7	2 / 0
EBay	4051 / 1134	80 / 51	3 / 1	6 / 4	25 / 19	0 / 0
Flickr	1519 / 98	140 / 20	1 / 1	4 / 4	97 / 8	0 / 1
GoogleNews	6408 / 102	45 / 8	1 / 1	4 / 4	6 / 1	0 / 0
Imdb	4250 / 820	213 / 47	36 / 1	23 / 4	65 / 14	7 / 4
Linkedin	2025 / 152	53 / 6	3 / 1	5 / 3	17 / 4	3 / 0
Reference	2260 / 285	112 / 20	6 / 1	7 / 2	28 / 3	0 / 0
Reuters	2446 / 277	230 / 37	6 / 2	7 / 3	143 / 31	4 / 1
Softonic	3928 / 241	89 / 22	9 / 1	11 / 3	30 / 17	0 / 0
Spiegel	3517 / 302	89 / 19	6 / 1	7 / 4	20 / 13	0 / 0
StackOverflow	4486 / 322	43 / 13	1 / 1	3 / 3	20 / 5	0 / 0
Taringa	2668 / 1251	178 / 59	9 / 1	10 / 5	58 / 26	0 / 0
TheGuardian	3924 / 332	236 / 84	4 / 2	4 / 3	82 / 35	0 / 0
TripAdvisor	9949 / 202	372 / 39	1 / 1	5 / 5	24 / 9	0 / 0
W3CSchools	2078 / 42	66 / 20	4 / 1	4 / 3	31 / 11	0 / 0
Walmart	1627 / 124	57 / 8	2 / 1	1 / 1	14 / 3	3 / 0
Wikipedia	5246 / 152	49 / 24	1 / 1	4 / 4	34 / 21	3 / 3
Wordpress	694 / 96	68 / 20	1 / 1	2 / 2	29 / 8	1 / 0
WSJournal	6416 / 1125	85 / 65	28 / 24	49 / 46	44 / 36	3 / 0
Yahoo	2755 / 954	162 / 78	7 / 2	9 / 4	31 / 17	0 / 0
Yelp	1655 / 24	47 / 2	5 / 1	2 / 2	7 / 0	0 / 0
TOTAL	107294 / 19636 (18%)	3259 / 897 (27%)	180 / 61 (33%)	224 / 136 (60%)	1112/ 387 (34%)	33 / 12 (36%)

Table 2. Average execution times in milliseconds

	Custom browser with optimization	Custom browser without otimization	HtmlUnit	Internet Explorer
Alexa	3431	5152 (150%)	6292 (183%)	14678 (427%)
Amazon	6488	14552 (224%)	15718 (242%)	23441 (361%)
Apple	2502	4425 (176%)	8339 (333%)	16350 (653%)
Barnes	11593	16405 (141%)	20513 (176%)	33622 (290%)
Bloomberg	3879	18335 (472%)	20750 (534%)	25196 (649%)
CNET	5844	11587 (198%)	23907 (409%)	28500 (487%)
CNN	3355	15083 (449%)	26190 (780%)	22149 (660%)
EBay	6332	8448 (133%)	11583 (182%)	20388 (321%)
Flickr	8662	22572 (260%)	17879 (206%)	22962 (265%)
GoogleNews	2497	4931 (197%)	11614 (465%)	22651 (907%)
Imdb	8314	20468 (246%)	19049 (229%)	25130 (302%)
Linkedin	2901	5942 (204%)	6760 (233%)	16102 (555%)
Reference	3555	11640 (327%)	22893 (643%)	19514 (548%)
Reuters	3647	13393 (367%)	14828 (406%)	19004 (521%)
Softonic	4450	9372 (210%)	19797 (444%)	38279 (860%)
Spiegel	2492	6540 (262%)	7812 (313%)	21022 (843%)
StackOverflow	3684	7551 (204%)	16186 (439%)	17350 (470%)
Taringa	10239	21339 (208%)	21075 (205%)	28025 (273%)
TheGuardian	5631	13043 (231%)	19358 (343%)	25104 (445%)
TripAdvisor	6102	9047 (148%)	15139 (248%)	24994 (409%)
W3CSchools	3142	6746 (214%)	13205 (420%)	20541 (653%)
Walmart	1350	4594 (340%)	7276 (538%)	12280 (909%)
Wikipedia	4850	7791 (160%)	10054 (207%)	19505 (402%)
Wordpress	2081	4655 (223%)	5031 (241%)	13890 (667%)
WSJournal	13896	17415 (125%)	18629 (134%)	20941 (150%)
Yahoo	7815	13755 (176%)	21501 (275%)	25359 (324%)
Yelp	2344	4354 (185%)	7692 (328%)	18875 (805%)

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 [5], Wargo [8], QEngine [9], Sahi [11], Selenium [12], 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.

Other systems use the approach of creating simplified custom browsers specially built for the task. WebVCR [2] and WebMacros [10] 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 [4] and Kapow [6] 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.

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. According to our experiments these 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.

Acknowledgments This research was partially supported by the Spanish Ministry of Science and Innovation under project 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. Document Object Model (DOM). http://www.w3.org/DOM/
4. HtmlUnit, http://htmlunit.sourceforge.net/
5. 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)
6. Kapow, http://www.openkapow.com
7. Montoto P., Pan A., Raposo J., Bellas F, López J.: Automated browsing in AJAX websites. Data Knowl. Eng. 70(3), 269-283 (2011)
8. 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)
9. QEngine, http://www.adventnet.com/products/qengine/index.html
10. 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
11. Sahi, http://sahi.co.in/w/
12. Selenium, http://seleniumhq.org/
13. XML Path Language (XPath), http://www.w3.org/TR/xpath