Automatic Optimization of Web Navigation Sequences

José Losada, Juan Raposo, Alberto Pan, and Javier López

Abstract. Web automation applications are widely used for different purposes such as B2B integration, automated testing of web applications or technology and business watch. In this work-in-progress paper we outline a set of techniques which constitute the basis to build a web navigation component able to analyze a web navigation sequence and automatically optimize it, detecting which parts of the loaded pages are needed, and which ones can be discarded in the following executions of the sequence. Our techniques build on the Document Object Model and the first tests executed with real web sources have found them to be very effective.

1 Introduction

Web automation applications are widely used for different purposes such as B2B integration, web mashups, automated testing of web applications, Internet metasearch or technology and business watch. One crucial part in web automation applications is how to easily generate and reproduce navigation sequences. We can identify two distinct stages in this process:

- In the *generation* stage the user specifies the navigation sequence to reproduce. The most common approach, cf. [1, 6, 7, 9], is using the 'recorder' metaphor.
- In the *execution* phase the sequence generated in the previous stage is provided as input to an automatic navigation component which is able to reproduce it.

The automatic navigation component used in the execution phase can be developed by using the APIs of popular browsers, cf. [4, 7, 8, 10, 11], or simplified custom browsers specially built for the task, cf. [1, 3, 5, 9].

José Losada · Juan Raposo · Alberto Pan · Javier López
Information and Communications Technology Department, University of A Coruña
Facultad de Informática, Campus de Elviña, s/n, 15071, A Coruña (Spain)
e-mail: {jlosada,jrs,apan,jmato}@udc.es

J.M.C. Rodríguez et al. (Eds.): Trends in PAAMS, AISC 157, pp. 125–132.
springerlink.com

The approach of using the APIs of commercial web browsers have some advantages: no effort is required to develop a navigation component, and the accessed web pages behave the same as when they are accessed by a regular user navigating with the commercial web browser. But, on the contrary, the performance of the component is limited by the commercial browser performance and the functionalities that the browser API provides. The main purpose of commercial web browsers is to be used by human users, and they consume a significant amount of resources, both memory and CPU. So, this approach is not the most appropriate to execute intensive or real time web automation tasks, which need to execute a significant number of navigation sequences in the less possible time.

The approach of creating a custom browser, supporting technologies such as scripting code and AJAX requests, is effort-intensive and can be vulnerable to implementation differences that can make a web page behave differently when accessed with the custom browser. Nevertheless, the main advantage is the performance: custom browsers can consume fewer resources (memory and CPU), and they are able to execute navigation sequences faster than commercial browsers.

Current systems which use the approach of creating custom browsers to execute navigation sequences, like [3] or [5], can avoid some steps executed by commercial web browsers (e.g. the page rendering phase), but they replicate its functioning when loading and building the internal representation of the web pages. The pages are always completely loaded (e.g. all the scripts contained in the page are executed).

In this work-in-progress paper we present the basis for a web navigation component able to analyze a web navigation sequence and automatically optimize it, detecting which parts of the loaded pages can be discarded:

- In the *optimization phase* the sequence is executed once, and in the meantime the execution component automatically calculates which nodes of the HTML DOM [2] tree of the loaded pages are needed to execute the sequence and which ones can be discarded. Then, it stores some information to be able to detect those elements in subsequent sequence executions.
- In the *execution* phase the execution component executes the sequence using the optimization information. When each page is loaded, a reduced HTML DOM tree is built, containing only the relevant nodes needed to execute the sequence.

This way, smaller HTML DOM trees are built when each page is loaded resulting in less memory usage. Besides, the script code, including AJAX requests, contained in elements not loaded in the simplified tree are not executed, and the external resources (e.g. JavaScript or CSS files) referenced from elements not loaded in the tree are not retrieved, therefore optimizing CPU time and network usage.

The step of searching which elements must be loaded in the simplified HTML DOM tree is the unique latency that the navigation component adds at execution time. As we will demonstrate in the experimental evaluation, this latency is insignificant compared to the time savings derived from building the simplified tree.

2 Models

In this section we briefly describe the model we use to characterize the component used to automatically optimize web navigation sequences.

The main model we rely on is the Document Object Model (DOM) [2]. 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 script 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.

Each node in the tree can receive events produced (directly or indirectly) by the user actions. Event types exist for actions such as clicking on an element (*click*), moving the mouse cursor over it (*mouseover*), or to indicate that a new page has just been loaded (*load*), to name but a few. Each node can register a set of event listeners for different types of events. Each event is dispatched following a path from the root of the tree to the target node, and it can be handled locally at the target node or at any target's ancestors in the tree (this is called "bubbling"). An event listener executes arbitrary code, which normally calls a function declared in script nodes.

In the context of the Document Object Model, we say that there exists a dependency between two nodes N_1 and N_2 when the node N_1 is necessary for the correct execution of the node N_2. We say that the node N_1 is a dependency of the node N_2. The following rules define the dependencies involving nodes which execute script code (script nodes or nodes containing event listeners):

1. If the script code of a node S_1 uses an element (e.g. a function or a variable) declared in a script node S_2, then S_2 is a dependency of S_1. To be able to execute the script code of the node S_1 the node S_2 must be loaded.
2. If the script code of a node S uses a node N (e.g. using the JavaScript function *document.getElementById*), then N is a dependency of S. To be able to execute the script code of the node S, the node N must be loaded.
3. If the script code of a node S makes a modification in a node N (e.g. it modifies the *action* attribute of a form node), then S is a dependency of N. If the node N is going to be used, then the script node S needs to be loaded to perform the modification in the node N.

Note that, node dependencies are transitive. For example, if an event listener of a node N_1 invokes a function f which is defined in a node N_2, and the implementation of f uses the node N_3, then both N_2 and N_3 are dependencies of N_1.

3 Description of the Solution

3.1 Optimization Phase

The optimization phase involves one execution of the navigation sequence where the navigation component automatically calculates which nodes of the DOM trees of the loaded pages are needed to execute the sequence (*relevant* nodes), and which ones can be discarded (*irrelevant* nodes). Then, it stores some information to be able to identify these nodes in the following executions of the sequence.

First, we will explain the techniques designed to calculate the set of *relevant* nodes and the set of *irrelevant* nodes. While executing a navigation sequence we can differentiate two steps for each page loaded:

1. The *page loading* step involves loading the page, generating the DOM tree, downloading external elements (e.g. style sheets, script files) and executing the script nodes defined in the page. Finally, some predefined events are automatically fired when the new page is completely loaded (e.g. the *load* event is fired over the body node), and some event listeners can be executed as response.
2. The *page interaction* step involves executing the pertinent actions and firing the necessary events, to execute the navigation sequence commands which emulate the user interaction with the page (e.g. clicking on elements, firing mouse movement events, etc.), until a navigation to a new page is started.

In both steps, there may be multiple interactions among nodes in the page, which must be taken in consideration to determine which nodes of the DOM tree are required for the correct execution of the navigation sequence.

During the page loading step, the navigation component can use the rules explained in section 2 to build a node dependency graph, containing the node dependencies for all the script nodes that are executed, and for all the nodes which contain event listeners that are executed as response to the events fired. In a similar way, during the page interaction step, for each event which is fired, a dependency graph is calculated for all the nodes which execute event listeners in response to the event. Finally, all these node dependency graphs are merged into a unique global dependency graph.

Then, the set of relevant nodes can be built using the following rules:

1. The target nodes of each of the actions to be executed or the events to be fired during the page interaction step are relevant.
2. If a node is relevant, all its ancestors are relevant. This is needed because of the "bubbling" stage in the DOM event execution model (see section 2).
3. By definition, if a node is relevant, all its dependencies are relevant.
4. If an input node is relevant, the form node containing it is relevant.
5. If a form node is relevant, all the input and select nodes contained in the form are relevant.
6. If a select node is relevant, all its child option nodes are relevant (this rule and the two previous ones are needed to be able to properly submit forms).
7. A small set of node types are always considered relevant (e.g. base nodes).

To calculate the set of irrelevant nodes, first, all the DOM tree nodes not contained in the set of relevant nodes are added to it. Then, all the irrelevant nodes which have an ancestor also contained in the set of irrelevant nodes are removed from the set. The resulting set contains only the root nodes of the sub-trees whose descendants are all irrelevant (we call them irrelevant sub-trees).

Now, we will briefly explain the techniques used to generate expressions to identify the root nodes of the irrelevant sub-trees at execution time. 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. different data records can be shown in dynamically generated sections). 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 that expressions before creating and adding it to the HTML DOM tree.

To uniquely identify a node in the DOM tree we use an XPath-like expression. XPath [12] expressions allow identifying a node in a DOM tree by considering information such as the node type, the text associated to the node, the value of its attributes and its ancestors. Our proposal starts from a very simple XPath-like expression using only the type, text and attributes associated to the target node and, if it does not uniquely identify the node, the expression is progressively augmented including information from some appropriate ancestors, until it does. We build the least restrictive expression that still uniquely identifies the target node. Besides, these type of expressions can be evaluated efficiently at the execution phase. The algorithm is not deeply described due to space constraints.

3.2 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 associated to that page, that is, a set of expressions to identify the root nodes of the page irrelevant sub-trees. That information is used during the parsing stage to build a reduced version of the page HTML DOM tree, containing only the relevant fragments. If a node matches with any expression of the set, the node is not added to the tree and the entire page fragment below that node is completely discarded.

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. Due to the method used to create the XPath-like expressions that identify the root nodes of the irrelevant sub-trees, an efficient algorithm has been designed to check if a node matches with an expression. The algorithm is not described due to space constraints.

Another important issue to deal with during the execution phase, is the identification of the pages where the optimizations should be applied. In most cases, the order in which the pages are loaded when the navigation sequence is executed could be used to identify them, but further investigation is required to design a more robust method for identifying the pages. This task is currently in progress.

4 Evaluation

To evaluate the validity of our approach a custom browser was implemented. This browser emulates Microsoft Internet Explorer (MSIE) version 8 and was fully implemented 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 custom browser has proved very efficient. It works in most of the navigation sequences and it is faster than MSIE and other commercial browsers (like Firefox, Chrome, etc.) is most of the cases. To evaluate the techniques and algorithms proposed in this paper they have been implemented in the core of this custom browser.

This section explains the preliminary set of experiments that we have designed and executed, which can be divided in two different types. For the first type, we selected a set of popular websites of different domains. In each website we recorded a navigation sequence involving several pages. We ran a first execution of the navigation sequence to collect the optimization information. Then, we ran two more executions, the first one without using the optimization information, and the second one using it. Table 1 shows the metrics measured for each of this two executions in a representative subset of the selected websites. (each cell shows the result of the normal execution followed by the result of the optimized one). Note that one execution is enough to calculate these metrics because they will have the same values in all the executions while the website pages remain without changes.

The results of the first type of experiments show that in almost every source more than the 50% of the nodes are identified as irrelevant. In half the sources, the nodes identified as irrelevant are more than the 75% (up to 96,5%) of the total nodes. Those irrelevant nodes include scripts, style sheets and frames. Discarding those nodes, the browser also avoids unnecessary downloads and the execution of unnecessary scripts, so the memory and CPU usage required to execute the navigation sequence is highly minimized when the optimization information is used.

The second type of experiments consists of a benchmark using 5 instances of our custom browser running in parallel, executing the same navigation sequence during a fixed amount of time (10 minutes). To avoid the latency of the network, some of the websites used in the first type of experiments were replicated in a local web server, simulating the original website. Table 2 shows the number of executions completed, using and without using the optimization information.

The results of the second type of experiments show that the executions using the optimization information are, in average, 41,7% (it varies from 21,7% to 70%) faster than normal executions. Note that these experiments use the sources replicated locally, so they do not include the time savings derived from downloading fewer resource files (CSS, JavaScript, etc.) from remote servers.

Table 1 Metrics comparing normal and optimized executions in some websites

Website	HTML DOM Nodes created	Scripts Executed	Frames and Windows	HTML pages Downloaded	External objects Downloaded	AJAX Requests
Reuters	3264 / 1463	303 / 168	6 / 3	9 / 7	176 / 103	4 / 4
Pixmania	3808 / 1527	156 / 96	2 / 1	5 / 5	75 / 46	1 / 0
Optize	2699 / 734	102 / 53	1 / 1	3 / 3	36 / 25	0 / 0
Wikipedia	4742 / 1168	69 / 58	5 / 1	5 / 5	47 / 43	4 / 4
Amazon	8319 / 5046	295 / 201	28 / 13	30 / 15	63 / 37	9 / 7
Ebay	5474 / 2603	119 / 111	10 / 8	14 / 13	33 / 31	0 / 0
Vueling	37865 / 7237	3504 / 894	178 / 31	78 / 28	1115 / 340	123 / 3
Bloomberg	6585 / 1160	351 / 212	8 / 3	12 / 7	162 / 103	2 / 0
Fnac	6993 / 1309	232 / 107	13 / 7	23 / 15	104 / 54	0 / 0
AppleStore	1914 / 67	40 / 17	1 / 1	3 / 3	12 / 11	0 / 0
NYTimes	6911 / 2016	279 / 223	8 / 5	14 / 11	192 / 155	4 / 4
Imdb	5033 / 1633	285 / 166	41 / 6	47 / 11	112 / 75	6 / 6
CNet	6117 / 1067	247 / 168	10 / 6	15 / 11	116 / 90	0 / 0
AbeBooks	3553 / 730	172 / 104	6 / 2	8 / 5	82 / 71	2 / 2
AllBooks4Less	2959 / 545	69 / 30	6 / 2	9 / 6	20 / 11	9 / 3

Table 2 Benchmarking normal versus optimized executions

□	Pixmanía	Optize	Wikipedia	Amazon	Ebay	Vueling
Normal	111	390	244	130	173	18
Optimized	195	635	323	277	218	60

5 Related Work

Currently, web automation applications are widely used for different purposes. The automatic navigation component used by these applications is developed by using the APIs of popular browsers or simplified custom browsers specially built for the task. WebVCR [1] and WebMacros [9] rely on simple HTTP clients that lack the ability to execute complex scripting code or to support AJAX requests. Wargo [7], Smart Bookmarks [4], Sahi [10], Selenium [11] and QEngine [8] use a commercial browser as execution engine. Therefore, the performance of the component is limited by the commercial browser performance, which consumes a significant amount of resources. HtmlUnit [3] and Kapow [5] use their own custom browser with support for many JavaScript and AJAX functionalities. They are more efficient than commercial web browsers, but they replicate its functioning when loading pages and building its internal representation, without allowing any type of extra optimizations.

6 Conclusions and Future Work

In this paper, we have presented a novel set of techniques and algorithms to optimize automatic 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. According to a preliminary set of experiments they seem to be very effective, but further experimentation is required. We also plan to refine some of the algorithms like, for example, the method used to identify the pages that are loaded in order to apply them the correct optimizations.

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.

References

[1] Anupam, V., Freire, J., Kumar, B., Lieuwen, D.: Automating web navigation with the WebVCR. In: WWW 2000, pp. 503–517 (2000)
[2] Document Object Model (DOM), http://www.w3.org/DOM/
[3] HtmlUnit, http://htmlunit.sourceforge.net/
[4] Hupp, D., Miller, R.C.: Smart Bookmarks: automatic retroactive macro recording on the web. In: ACM Symposium on User Interface Software and Technology (UIST), pp. 81–90 (2007)
[5] Kapow, http://www.openkapow.com
[6] Little, G., Lau, T., Cypher, A., Lin, J., Haber, E., Kandogan, E.: Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In: SIGCHI Conference on Human Factors in Computing Systems, pp. 943–946 (2007)
[7] 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 (2002)
[8] QEngine, http://www.adventnet.com/products/qengine/index.html
[9] Safonov, A., Konstan, J., Carlis, J.: Beyond Hard-to-Reach Pages: Interactive, Parametric Web Macros. In: 7th Conference on Human Factors & the Web (2001)
[10] Sahi, http://sahi.co.in/w/
[11] Selenium, http://seleniumhq.org/
[12] XML Path Language (XPath), http://www.w3.org/TR/xpath