

University of Alberta

REVERSE ENGINEERING AND TESTING DYNAMIC WEB APPLICATIONS

by

Natalia Negara

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

©Natalia Negara
Spring 2013
Edmonton, Alberta

Permission is hereby granted to the University of Alberta Libraries to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only. Where the thesis is converted to, or otherwise made available in digital form, the University of Alberta will advise potential users of the thesis of these terms.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

To my parents

Abstract

A new generation of complex interactive dynamic web applications has emerged with the introduction of Web 2.0 technologies and development frameworks. The characteristics of dynamic web applications such as runtime DOM structure and content updates introduced new challenges in the understanding, maintenance and testing of this type of web applications. In this work we address two important challenges in the field of web application maintenance. The first challenge is that of modelling web application behaviour. To solve this task we develop an automatic method for reverse engineering the features of dynamic web applications by applying a hierarchical clustering algorithm based on a novel composite-tree-edits-aware distance metric between DOM tree instances of a web application. The proposed distance metric recognizes simple and composite structural changes in a DOM tree. We have evaluated our method on three real-world web applications. The evaluation results demonstrated that the proposed distance metric produces a number of clusters that is close to the actual number of features and, also, classifies DOM trees into feature clusters more accurately than other traditional distance metrics. The second challenge is that of systematic acceptance (and regression) testing at the user-interface level, which we address by developing a tool, *CrawlScripter*, for performing automated acceptance testing of JavaScript web applications. *CrawlScripter* allows to create easy-to-understand acceptance tests using the provided library of high-level instructions. The ability of *CrawlScripter* to create automated acceptance tests for different test scenarios was evaluated on both pedagogical and real-world dynamic web applications.

Acknowledgements

I would like to thank my adviser Dr. Eleni Stroulia for her great supervision, helpful advice, and patience as I worked on my research. Also, I would like to thank Dr. Nikolaos Tsantalis for sharing his knowledge with me and for his support and encouragement while we were working on this project. Additionally, I would like to thank Rimon Mikhael, Marios Fokaefs, David Churchill, Fabio Rocha and Aaron Luchko for their useful advice and constructive suggestions.

Table of Contents

1	Introduction	1
1.1	Feature Extraction of Ajax-based Web Applications	2
1.2	Acceptance Testing of Ajax-based Web Applications	3
1.3	Contributions	3
1.4	Outline	5
2	Related Work	6
2.1	Reverse-Engineering of Web Applications	6
2.1.1	Web clustering	6
2.1.2	Other methods applied to reverse engineer web applications	9
2.2	Testing of Web Applications	10
2.2.1	Exhaustive analysis of dynamic DOM states	10
2.2.2	Approaches limited to test dynamic web applications	12
2.2.3	Frameworks and tools for acceptance testing of web applications	13
2.2.4	Remarks	14
3	Feature Extraction	16
3.1	DOM state collection	17
3.2	DOM state clean-up	19
3.3	VTracker	22
3.3.1	Affine cost computation	22
3.3.2	Simplicity heuristics	23
3.3.3	Cross-referencing	23
3.4	The Distance Metric	24
3.5	The Clustering Algorithm	28
3.5.1	Partitioning clustering algorithms	28
3.5.2	Density-based clustering algorithms	29
3.5.3	Agglomerative hierarchical clustering algorithms	29
3.5.4	Determining the number of clusters	30
3.6	Experiments and Results	33
3.6.1	Experimental setup	33
3.6.2	Evaluation Methodology	35
3.6.3	Experimental results	36
3.6.4	Discussion	38
3.6.5	Conclusions	41
4	Automated Acceptance Testing of Ajax-based Web Applications	46
4.1	The Architecture of CrawlScripter	46
4.1.1	The Web Client	46
4.1.2	The Test Repository	49
4.1.3	The Testing Engine	50
4.1.4	Implementation Challenges	53
4.2	The <i>CrawlScripter</i> Scripting Language	53
4.3	Evaluation	54
4.3.1	JPetStore	54
4.3.2	GRAND Forum	56
4.3.3	GRAND Forum: Test-scripts migration	58
4.3.4	Conclusions	60
5	Conclusions and Future Work	61

Bibliography	63
A	67
A.1 Grand Forum User Stories	67

List of Tables

2.1	Related literature on web applications testing approaches	15
3.1	Number of states, DOM size and depth for the examined web applications.	34
3.2	Jaccard similarity coefficient for the examined web applications	36
3.3	Manually extracted features	36
3.4	L method: Predicted number of clusters, F-measure, precision, and recall for Google Maps	38
3.5	Silhouette Coefficient: Predicted number of clusters, F-measure, precision, and recall for Google Maps	38
3.6	L method: Predicted number of clusters, F-measure, precision, and recall for Garage	38
3.7	Silhouette Coefficient: Predicted number of clusters, F-measure, precision, and recall for Garage	39
3.8	L method: Predicted number of clusters, F-measure, precision, and recall for Kayak	39
3.9	Silhouette coefficient: Predicted number of clusters, F-measure, precision, and recall for Kayak	39

List of Figures

3.1	<i>Crawljax</i> configuration for JPetStore	18
3.2	Before cleaning up the “Menu” part in the DOM structure of JPetStore “Search item” page	20
3.3	After cleaning up the “Menu” part in the DOM structure of JPetStore “Search item” page	21
3.4	A sample section of the DOM structure from JPetStore “Search page” (before query submission)	26
3.5	A sample section of the DOM structure from JPetStore “Search page” (after query submission)	27
3.6	JPetStore evaluation graph	31
3.7	An example of a string sequence of HTML tags used to compute the Levenshtein edit distance	35
3.8	Evaluation Graph for Google Maps with d_S	40
3.9	Evaluation Graph for Google Maps with d_C	41
3.10	Evaluation Graph for Google Maps with d_L	42
3.11	Evaluation Graph for Garage with d_S	43
3.12	Evaluation Graph for Garage with d_C	43
3.13	Evaluation Graph for Garage with d_L	44
3.14	Evaluation Graph for Kayak with d_S	44
3.15	Evaluation Graph for Kayak with d_C	45
3.16	Evaluation Graph for Kayak with d_L	45
4.1	Overview of <i>CrawlScripter</i> design time and run-time architecture	47
4.2	Graphic User Interface of the <i>Web Client</i> : User Stories and Test Scripts tab	48
4.3	Graphic User Interface of the <i>Web Client</i> : Test Reports tab	48
4.4	List of available instructions in <i>CrawlScripter</i>	49
4.5	Creating test scripts with <i>CrawlScripter</i>	49
4.6	Database schema of <i>CrawlScripter</i>	50
4.7	The <i>CrawlScripter</i> Language	54
4.8	JPetStore: “Search item (by keyword)” test script	55
4.9	JPetStore: “Search item (by keyword)” test report	56
4.10	An acceptance test script for GRAND Forum web application	58
4.11	A test report for GRAND Forum	59

Chapter 1

Introduction

Ever since its inception, the World Wide Web has been evolving with new technologies and frameworks designed to satisfy an increasing demand for developing complex and highly interactive web applications. The number of people connected to the World Web Web is dramatically growing every year [1]. The World Wide Web offers a new convenient way to deploy and access software, and as a result, today it plays a significant role in business, telecommunication, electronic commerce, education, and other information services. Web applications have become more popular and more powerful over the last decade and offer a convenient alternative to desktop software [2]. The simple applications with static HTML pages have been superseded by complex, highly interactive applications, with sophisticated graphic user interfaces (GUIs) whose content and structure is generated dynamically during run-time. As a result, new challenges in understanding, maintenance and testing of dynamic web applications have emerged. This thesis focuses exactly on two of the challenges in maintaining dynamic web applications.

Changes in web applications occur more often than in other software [3,4], and this makes web applications evolve faster. Such frequent changes and the dynamic nature of such web applications make them harder to comprehend, maintain and test. Therefore, reverse-engineering approaches are necessary for automatically extracting models of the underlying behaviour of dynamic web applications. Furthermore, new testing techniques and methods must be developed to support the quality control of such web applications.

We suggest a method for automatically extracting features from Ajax-based¹ web applications. Also, we introduce an approach for conducting automated acceptance testing of JavaScript web applications. In the following sections we will discuss the high-level details on each of these methods.

¹In this thesis we use the terms “Ajax-based” and “JavaScript” interchangeably to refer to dynamic web applications that extensively use JavaScript as a client-side scripting language. Ajax (Asynchronous Javascript XML) is a technique that uses JavaScript to create asynchronous web applications.

1.1 Feature Extraction of Ajax-based Web Applications

A *feature* of a web application represents a functionality that is defined by requirements and is accessible to users [5]. For example, the web application of an air-travel company can provide features such as searching for flights, booking flights, online check-in, checking flight status, etc. The feature “Searching for flight”, for example, may involve (a) interaction with the user (e.g., providing input about trip details such as “from” and “to” cities, “departure” and “return” dates), (b) data validation (e.g., the return date can not be earlier than the departure date), and (c) computation (e.g., auto complete of “from” and “to” fields). Thorough understanding and documentation of the behavioural features of a web application is necessary both to ensure proper use of the system by its users, and to facilitate maintenance, testing, and evolution of the system by future developers, as well as potentially programmatically to access the services of the web application through mashups.

A standard method [6, 7] for extracting features (services) of a web application consists of collecting web page instances by exhaustive exploration of a web application, and grouping these web pages into clusters of similar pages. Each cluster corresponds to a functionality that the web application supports, which is available to its users through the web pages included in the group. Grouping pages into feature clusters provides a high-level view of a web application and describes the offered services as clusters of similar web pages and shows the relationships between them.

The task of collecting web pages through an exhaustive exploration of a dynamic web application is a challenging one, since the number of web-application states that can be produced is potentially unlimited. With the introduction of tools for automatically crawling and testing modern Ajax-based web applications such as Crawljax [8], the problem of collecting web page instances from a complex dynamic web application can be mitigated. Nevertheless, the problem of feature extraction of such web applications still remains a challenge, since modern web applications can have web pages with complex HTML document structures with script and style code segments included in addition to their HTML content. This can significantly complicate the problem of page-similarity assessment on which the web clustering is based. Distance metrics, i.e., metrics for computing the similarity among instances, play an important role in clustering of data objects, and different distance metrics can produce different clustering results. A number of simple string-differencing approaches have been previously applied with some success to perform web page similarity [9–13]. Also, the selection of a distance metric for web page similarity assessment deeply impacts the result of the reengineering process [10].

The extracted behavioural model represents web-application features (services) as groups of similar pages and provides an overview of a web-application structure and a navigational model of a web application from the user’s point of view. This behavioural model can be used in different software-development tasks, such as user documentation, maintenance and re-engineering, as well as acceptance testing of web applications.

1.2 Acceptance Testing of Ajax-based Web Applications

Acceptance testing is an important activity in any software-development lifecycle and its objective is to confirm that the software meets the client requirements. To that end, acceptance tests are driven by, and correspond to, user stories, which are a high-level description of requirements. Automated acceptance tests can also be included as a part of regression tests to verify that the client requirements are not violated by the software's evolution.

Software testing, an admittedly difficult task, is even more challenging in the context of modern web applications, because of their dynamic behaviour based on the user input, the run-time updates of the client-side components, and the differences in the rendering of their GUI by the various [14]. The introduction of Web 2.0 technologies and frameworks resulted in a prevailing development of complex web applications with dynamic structure and content [15]. A considerable amount of application logic moved to the browser via such web technologies as JavaScript. JavaScript is a dynamic programming language that is widely used for creating modern web applications and is responsible for client-side computations and dynamic updates of web pages. A number of approaches [16, 17] have been suggested for automated testing of dynamic web applications by "exploring" their behaviors. These methods are not suitable for the acceptance-testing task, as they automatically generate test cases based on the exhaustive analysis of dynamic DOM states of a web application, without being aware of its intended behavior. Other frameworks and methods have been proposed that can be adopted for acceptance testing, however they either require substantial programming expertise [18] or do not have complete testing solutions for Ajax-based (Asynchronous JavaScript and XML) web applications [19, 20].

1.3 Contributions

In the first part of the thesis we develop a method for automatically extracting features of complex dynamic web applications. The method is based on collecting DOM trees through the systematic exploration of a web application and clustering them into groups corresponding to a distinct features. In the second part of the thesis we propose a method to perform automatic acceptance testing of dynamic web applications, more specifically JavaScript-enabled web applications. Using intuitive high level scripting language our method allows users to translate user stories in high level declarative test scripts and to then execute these test scripts on a web application using an automated website crawler.

The contributions of the thesis are as follows:

1. We develop a novel distance metric for assessing structural similarity of web pages. The proposed novel distance recognizes structural changes in the DOM tree of a web page as simple or composite: changes applied to a set of nodes located under the same path to the root of the tree are considered as a single composite change. An example scenario can be a

query submitted by a user on a search page. In response to the submitted query, a new page is generated, which contains a new rich table component with query results. The rich table component may provide dynamic features, such as sorting, mouse hover and pop-up menus, which usually increase the number of generated HTML nodes under the table component. The definition of the composite-change-aware tree-edit distance is based on the assumption that complex substructures are frequently replaced by other substructures. We compare the composite tree-edits based distance metric with the simplified distance metric that treats all changes uniformly, and with the “flat” Levenshtein [21] string-distance metric, which has been widely used in the literature for computing the structural similarity of web pages. We evaluate the ability of each distance metric to produce the correct number of features from a set of DOM trees extracted from three real-world Ajax-enabled web applications using two different clustering evaluation methods: the L method [22] and the Silhouette coefficient [23]. These methods can automatically determine the number of clusters to be returned from the application of the hierarchical clustering algorithm. The results demonstrate that the proposed composite tree-edits based distance metric is more appropriate for automatically determining the number of clusters that is close to the actual number of features.

2. We automate the process of feature extraction through the use of the proposed distance metric in hierarchical clustering of DOM states. From the resulted dendrogram we deduct the cut-off threshold by applying clustering evaluation methods. Previous approaches used either a predefined cut-off threshold [9] or a user-defined threshold to determine the number of clusters from a dendrogram resulted from a hierarchical clustering algorithm application [10, 11, 24].
3. We develop a tool, *CrawlScripter*, for supporting automated acceptance testing of Ajax-enabled web applications. The tool provides a high level scripting language that assists with the creation of easy-to-understand automated acceptance tests for users that do not have necessary programming expertise. The scripting language also supports assertions, which can verify the presence or absence of an element in the DOM structure of a web page, thus enabling the comparison of the actual behaviour with the expected behaviour of the application. To be able to exercise and record the behaviour of Ajax-enabled web applications we employ an open-source tool Crawljax [8]. It is designed to exercise Ajax-based web applications at the client-side, i.e., through the browser, and, in the process, it constructs a flow-graph of the application’s behavioural states, corresponding to changes in the DOM (Document Object Model) tree underlying the web page.
4. We evaluate the applicability and usefulness of our method in supporting acceptance testing with two case studies. Also, we examine to what degree we can maintain the implemented executable acceptance test scripts during the evolution of the system.

1.4 Outline

The result of this thesis is organized as follows.

Chapter 2 presents an overview of related work. We discuss the methods proposed to reverse-engineer web applications as well as the distance metrics used to group similar web pages. Additionally, we give an overview of the methods suggested to tackle the problem of dynamic web applications testing.

In Chapter 3 we discuss our approach for automatic feature extraction from dynamic web applications. We describe our composite tree-edit distance metric and provide an overview of the tree-differencing algorithm used to compute distances between extracted DOM trees. We provide details of the clustering techniques used to group similar DOM trees and list the advantages with respect to our problem of the selected clustering technique over other clustering algorithms. We also describe two methods for automatically inferring the cut-off threshold for the dendrogram resulting from the application of a hierarchical clustering algorithm. Then, we give the details of the evaluation process with a description of the experimental setup and discuss the obtained results.

Chapter 4 discusses our suggested method for automated acceptance testing of JavaScript web applications. We give an overview of Crawljax, on which our method relies and we describe *CrawlScripter*, our tool for performing automated acceptance testing of JavaScript web applications. We also describe *CrawlScripter*'s scripting language, give the details of its architecture, namely the components, tools and libraries used in its implementation. Next, we present the results of the case studies performed as an evaluation of *CrawlScripter*.

Finally, we conclude in Chapter 5, summarizing the main points of the work and evaluation results, and pointing out some directions for future work.

Chapter 2

Related Work

In this Chapter we review the research literature related to this thesis, which we organize in two groups. First, we review reverse-engineering approaches that aim to retrieve comprehensible views or behavioural models of web applications. Then, we discuss the work on automatic testing of dynamic web applications.

2.1 Reverse-Engineering of Web Applications

A large number of reverse-engineering techniques have been proposed to support the comprehension, maintenance and evolution of web applications. A general approach to this problem involves the decomposition of the system into related groups of elements. The related work discussed in this section covers the techniques for extracting the architecture or functional behaviour of a web application [9, 11–13, 24, 25, 25–27]. A common method to achieve this is to cluster web pages according to the structure of their DOM trees into meaningful groups, using the Levenshtein distance metric [9–11, 28] or a composite distance metric [12, 13, 25]. Other approaches [24, 26, 27] employ different techniques such as finding shortest paths in a directed graph, or generating facts to construct the structural model of web applications.

2.1.1 Web clustering

First, we review approaches that employ simple string-differencing distance metrics in web-page clustering. Then, we discuss the related work that applies composite metrics to group web pages and review other approaches proposed to recover the structure of web applications.

2.1.1.1 The Levenshtein distance metric

Ricca and Tonella [9] proposed a semi-automatic approach to identify static web pages with a similar structure that can be migrated to dynamic web pages. To identify the groups of web pages with a common structure, they applied an agglomerative clustering algorithm with the Levenshtein edit distance [21] assessing the structural similarity between web pages. The Levenshtein edit distance

between two string sequences is defined as the minimum number of insert, delete, and replace operations required to transform the first sequence into the second. To compute the Levenshtein distance metric between web pages, their HTML tags are represented as string sequences constructed during the traversal of the DOM tree of the web page. Clusters of similar pages are then selected by cutting the resulting dendrogram (a hierarchy of clusters produced by a hierarchical clustering algorithm), using a user-defined threshold of the required intra-cluster similarity. Next, for all pages that are in the same cluster, a common candidate template is extracted to generate dynamic web pages. The common candidate template contains all unchangeable information and common HTML tags and is extracted by applying the Longest Common Subsequence algorithm. The variable information, which is produced by comparing the original pages with the template, is inserted into a database. Next, the pages are dynamically generated from the template and database records by a server-side script. Manual refinement is required for the constructed template and database.

A similar approach based on the computation of the Levenshtein edit distance between both static and dynamic web pages was proposed by De Lucia et al. [10] for the generalization of cloned patterns. This work focuses on detecting cloned navigational patterns which can be used for reengineering of web applications, as well as for recognizing common information that can be stored in the database [10]. The Levenshtein edit distance between two web pages is separately computed at the structural (using string representations of the HTML tags of web pages constructed during DOM trees traversal) and content levels (using the textual information associated with HTML tags constructed during DOM trees traversal). For dynamic pages the similarity degree is also computed for the server-side scripts. Two pages are considered as clones if their edit distance is lower than a given threshold.

Lucca et al. [11] encoded sequences of tags from HTML and ASP pages into a string representation by traversing web pages DOM trees and computed the Levenshtein edit distance between them in order to detect clone pages that have the same structure and differ only in their content. The authors defined a separate alphabet for client and server pages, and each HTML or ASP tag is replaced with a corresponding alphabet character. This was done in order to take into account the different techniques and languages used to implement control components in these pages. For every pair of server pages both ASP distance and HTML distance are computed. Along with the Levenshtein-based technique, the authors suggested to detect duplicate client pages using a frequency based metric, which is the occurrence (frequency) of each HTML tag inside a page. For the evaluation purpose the groups of cloned pages were created through the manual analysis. The results of the evaluation performed by the authors show that both methods are comparable, but have different computational costs.

A comparative study of clustering algorithms for the detection of cloned pages at the structural level was performed by De Lucia et al. [28]. In particular, the authors considered an agglomerative hierarchical clustering algorithm, a divisive clustering algorithm, K-means partitional clustering

algorithm, and a partitional competitive clustering algorithm (Winner Takes All). As a distance between two pages, the authors used the Levenshtein edit distance applied to the string representation of the HTML tags of both static and dynamic web pages. The authors conducted the evaluation of the selected clustering algorithms over one small and two medium web applications which showed that the clustering algorithms produce comparable results.

2.1.1.2 Composite distance metrics

To support the understanding of static and dynamic web pages, De Lucia et al. [12] suggested an approach that groups similar pages using the Winner-Takes-All [29] clustering algorithm, which is a simple artificial neural-network algorithm for grouping similar patterns of a neuron. The clustering is based on the correlations of data points (web pages), represented by the distance between those data points. At the structural level the distance between pages is computed using the Levenshtein edit distance. As in the aforementioned approach [11], the static/dynamic web pages are represented as string sequences of HTML tags. The HTML tags in turn are encoded into the symbols of an alphabet, which helps making the computation more precise and faster. To group pages at the content level the authors employ Latent Semantic Indexing [30]. Euclidean distance is computed as a similarity measure between the pages at the content level. The Levenshtein and Euclidean distances are then given as input to the Winner-Takes-All clustering algorithm which groups pages at the structural and content level, respectively. The number of clusters to be returned is also assumed to be provided as input to the clustering algorithm.

De Lucia et al. [13] used the Levenshtein edit distance to identify groups of web pages, similar at the content level. First, the dissimilarity between web pages at the content level is computed using the Levenshtein string edit distance and the Latent Semantic Indexing technique. In particular, the dissimilarity between web pages based on the extracted content is computed by combining a measure based on the Levenshtein edit distance and the cosine between the vectors of the page in the latent structure of content [13]. The approach uses a weighted mean to combine these dissimilarity measures. Similar pages are then grouped by iteratively applying a Graph-Theoretic clustering algorithm [31], which takes as input a graph, and then constructs a minimal spanning tree (MST). The nodes in the graph represent web page instances and the edges between nodes are assigned with weights using mean combined dissimilarity measure. All individual clusters are initially removed from the input graph, so it represents a strongly connected graph. Clusters are identified by pruning the edges of the MST that have a weight higher than a given threshold. The clustering process is iteratively performed until no single-page clusters are identified. The arithmetic mean of the edges of the constructed MST is used as a pruning threshold. However, such methods of computing thresholds could affect the quality of the results.

To support the comprehension of web applications, Di Lucca et al. [25] applied the agglomerative hierarchical clustering algorithm to decompose web applications into groups of functionally

related components. As a similarity measure the clustering algorithm takes as input the coupling measure between interconnected components, which considers the topology of connections (*link*, *redirect* and *submit* relationships). The output of the agglomerative hierarchical clustering algorithm represents a hierarchy of clusters, which is then used to understand the structure of a web application. The cutting threshold for the produced dendrogram is selected based on a “quality metric”, computed as a difference between the intra-connectivity and inter-connectivity of the clusters.

2.1.1.3 Remarks

The majority of the aforementioned methods employ the Levenshtein edit distance [21] as a similarity metric between pages. The major limitation of this metric is that it doesn’t consider the structure of the web page since the web pages are converted into sequences of HTML tags, and therefore the metrics cannot take advantage of the information embedded in the nesting of the tags. An earlier result in our research group [32] demonstrated that tree-differencing can be effective in recognizing similarities and differences between the web pages.

Also, the majority of clustering-based techniques discussed above are semi-automatic and require to provide or use a predefined threshold to partition web pages into clusters [9, 10, 12, 13]. The results produced by each threshold should be carefully analyzed in order to identify which threshold produces the best partition. This requires a huge effort from users. Our method produces a partition of clusters that closely reflect the actual features of a web application automatically, without assuming that the user provides a cut-off threshold.

2.1.2 Other methods applied to reverse engineer web applications

Some of the reverse-engineering approaches focus, not on analyzing the web-application behavior, but rather on migrating static web pages to dynamic [9, 24]. Mesbah and van Deursen [24] used a schema-based clustering technique to categorize web pages with similar structures. The authors extract the navigational model of a web application and compute the Levenshtein edit distance between the reverse engineered schemas of the web pages. The navigational path and schema-based similarities are then given as input to a clustering algorithm, which finds for a set of clone pairs (e.g. $\{(a - b), (b - c), (d - e)\}$) a transitive closure ($\{(a - b - c), (d - e)\}$). The formed clusters are analyzed to find candidate user interface components to be migrated from a multi-paged web application to a single-paged Ajax interface.

Hassan and Holt [26] presented a semi-automatic approach for reverse engineering and visualizing the architecture of web applications. The authors developed a set of tools that parse the source code of the subject web application and extract relationships between its components. The specialized extractors retrieve software artifacts such as source code, execution traces, and web pages, and then generate facts about the web application (for example, “function f uses variable a ” or “file f_1 uses file f_2 ” [26]). The generated facts are then used to produce architecture diagrams. To decom-

pose the architecture diagrams into smaller subsystems the authors apply a clustering algorithm that takes into account directory structure, file name conventions, and software metrics. The clusters are then manually refined by the developers using documentation and specific domain knowledge.

Ricca and Tonella [27] developed ReWeb, a tool for reverse-engineering web applications, which supports web applications maintenance and evolution. ReWeb also provides a graphical interface through which users can explore a high-level view of a web application as well as search or navigate the web application’s structure. The restructuring in the proposed approach is based on such techniques as reaching frames, using dominators, and finding shortest paths. ReWeb consists of three main components: a Web spider (which is responsible for downloading the subject web application), an analyzer, and a viewer. The analyzer constructs a directed graph of a web application with pages as nodes and links as edges between the nodes. The tool models the web-application structure by applying analyses techniques such as flow analyses, traversal algorithms, and pattern matching. The structural and history analysis of the web application is displayed to the user by the viewer component of the ReWeb. The proposed tool doesn’t provide complete handling of dynamic web applications, and can only be applied for such dynamic web applications which provide the FORM construct for user input.

2.2 Testing of Web Applications

Software testing, an admittedly difficult task, is even more challenging in the context of modern web applications, due to their characteristics such as dynamic DOM content and the structure of their web pages [14]. A number of approaches [16, 17, 33–35] have been proposed for automated testing of dynamic web applications by “exploring” their behaviours. These methods exhaustively explore the dynamic DOM states of a web application. Other methods provide a range of interesting solutions for web testing, however they can only partially handle testing of dynamic web applications [19, 20, 36, 37]. Table 2.1 summarizes these methods. Column *Technique* briefly describes the technique used in the proposed testing approach. The type of the supported web application (Ajax, JavaScript, Java, PHP, etc.) is shown in column *Type of supported WA*, where “WA” stands for “web application”. Column *Tool* describes the name of tool/framework in which the approach was implemented. The type of evaluation performed on the approach is described in the last column of the table. Additionally, in Section 2.2.3 we discuss the most popular testing frameworks used both in industry and research.

2.2.1 Exhaustive analysis of dynamic DOM states

Crawljax [8] automatically crawls and tests Ajax-based web applications. Inspecting the clickable elements on each page, the tool can crawl Ajax-based web applications by firing events associated with them. Also, Crawljax can fill the input fields with the random data or, if provided, with the user data. While exploring the subject web application the tool creates a state-flow graph, where

vertices are DOM states and edges are the clickable elements whose behaviour connects one DOM state to another. When Crawljax discovers a new DOM state as a result of DOM tree changes, a corresponding node and edge with an annotated event are added to the graph. The process is repeated until no new DOM states are discovered, or the stop condition is satisfied (e.g., the crawler reached the maximum number of states or maximum crawl depth). The authors of Crawljax evaluated the accuracy, scalability, and performance of Crawljax in a series of crawling tasks.

To test Ajax-based web applications Marchetto *et al.* [16] recently suggested a method to dynamically extract DOM states into a finite state machine and to analyze semantically interacting events sequences in order to automatically generate test cases. To overcome the problem of exponentially growing search space of semantically interacting event sequences, the authors later suggested to use search-based algorithms to extract maximally diverse event sequences of different length [17]. This technique reduces the size of generated test cases, since event sequences are not repeated. In contrast to our approach, the above methods focus on finding faults in a web application, instead of establishing that it delivers the desired functionalities.

Dallmeier *et al.* [38] developed WebMate, a tool designed for automatic testing of dynamic web applications. Automatically exploring a web application, WebMate constructs its usage model that captures how the user can interact with the web application through triggering JavaScript event handlers. The usage model represents a graph with the application's states as nodes and interactions between the states as edges. After triggering an action on the web page, WebMate extracts each state of the web page's DOM structure. The tool navigates to each of the detected functional states of the application until all elements are explored. This work is similar to Crawljax [8], however the difference between two tools is that in order to distinguish DOM states Crawljax uses much lower-level DOM representations than WebMate. Also, WebMate recognizes dynamically attached event handlers, and Crawljax, in turn, examines all clickable elements on the web page. To evaluate the tool, the authors applied WebMate for cross-browser testing.

To perform automated testing of JavaScript web applications Artzi *et al.* [34] suggested an approach that is based on a feedback-directed random automated test generation used in a number of prioritization functions and test input generators. The prioritization functions guide the exploration of the web application's state space. The execution of the test is analyzed in order to collect more information for test script generator and to produce additional test scripts, thus increasing the test coverage. First, the algorithm starts with the creation of *initial test input*, which consists of the URL of the initial web page, an entry state parameter, and a pseudo-event that constructs an HTML DOM structure and executes the top-level script code followed by the `unload` event handler. The execution of the generated test scripts is then iteratively repeated by creating new events until no such tests remain. The approach was implemented in a framework named Artemis. The evaluation results demonstrated that the registration of event handlers as a test execution feedback produced reasonable coverage.

Finally, Saxena *et al.* [35] proposed an automated method for testing JavaScript web applications, implemented in the Kudzu tool. Given a URL for the subject web application, the tool automatically generates tests to systematically explore the web application. The author categorized the *input space* of a web application by two components: *value space* and *event space*. To explore the web application's value state the tool applies symbolic execution. The authors define a new string constraint language for parsing JavaScript code and implement a constraint solver on which the symbolic execution engine of the Kudzu is based. To explore the event space of a web application the authors implemented a GUI explorer that examines the event sequences using some random exploring strategy. The authors evaluated the proposed approach on finding client-side injection code-vulnerables.

2.2.2 Approaches limited to test dynamic web applications

CoScripter was originally developed for automating the browsing of interesting paths through a web site, and recording user's actions by demonstrating these paths [19]. CoScripter records user actions performed in the web browser, such as clicking on links and buttons or providing input for fields, which make CoScripter well suited for web testing. The recorded actions are described in human readable and editable textual scripts that can be shared with other users and later replayed. CoScripter consists of two main components: a Firefox¹ browser plug-in that records and replays the user actions, and a repository where the users can share, edit, and rate scripts. Recording of actions in CoScripter is done by adding event listeners to the events generated in response to the user's interaction with HTML DOM elements. When a new action is detected, a step is added to the script by filling a template that corresponds to the type of the action [39]. To parse scripts, CoScripter uses an LR(1) parser, which converts the textual representation of a script into web command objects. CoScripter uses heuristics to guess labels for a large number of input elements in the DOM structure. For example, to detect a label of a textbox, the algorithm might look for the text to the left of the textbox, so the label could be found in the DOM structure between the textbox' parent element and the textbox itself. CoScripter does not support actions that rely on DHTML or Ajax updates which make it not applicable for writing automation tests for Ajax-enabled web applications. Also, being a Firefox plug-in, CoScripter is limited to one browser only. The authors performed case studies to investigate how CoScripter can support procedure-sharing practices in an enterprise.

Mahmud and Lau [20] applied CoScripter for web testing and developed CoTester, a system for automated web-application testing. CoTester extends CoScripter by implementing assertions and a machine learning algorithm for subroutine identification in the test scripts. A subroutine in CoTester is a group of test steps that represent a higher level action like "Log in to the system". Subroutines are intended to help test maintenance, and also to better reflect the structure of a test script. However, CoTester which is built upon CoScripter, is available only for the Firefox browser,

¹www.mozilla.org

and does not support Ajax requests. The authors evaluated the subroutine identification functionality of CoTester and also performed a user study to analyze the ease of use of the tool.

Similar to Marchetto *et al.* work [17], Alshahwan and Harman [33] proposed a search based algorithm for automated web testing. The algorithm starts with the static analysis of a web application and collects static information that will be used in the next steps. The authors defined a dynamically mined seed value used in the search process. This value seeds the search with constants collected from web pages during run-time. To generate the data to be used during the tests the authors employed “hill climbing” optimization method which is generally applied in searches. The authors implemented the approach in SWAT, a tool designed to test PHP web applications. The major limitation of this approach is that it doesn’t support a complete testing solution for dynamic web applications. The evaluation of the approach consisted in analysis of the test coverage, test effort, and fault detection.

Sauvé *et al.* introduced EasyAccept [36], a framework that automatically generates and executes acceptance tests for Java programs, including web applications written in Java. As an input the tool takes tests that can be specified in one or more text files. Users are required to write specific Façade classes which play roles as bridges between the EasyAccept framework and the business logic of a tested program. The commands of test scripts are defined by method signatures of Façade classes. EasyAccept also provides some built-in commands, for example, *expect* means that a string is expected to be returned. Other examples of EasyAccept commands are *expectError*, *expectTable*, *equalFiles*, and *stackTrace*. The system runs the entire test suite and compare the actual results of the tests with the expected ones.

Later, Araújo *et al.* [37] developed a web application FLOAppTest based on EasyAccept for running acceptance tests for Java programs. FLOAppTest provides a visual interface for easier test script creation and does not require the installation of any additional platforms. The web applications retrieves Java projects from collaborative environments using web services. The user can define a Façade class from a list of available Java classes within a Java project. Both EasyAccept and FLOAppTest are designed only for programs written in Java and in order to run acceptance tests require access to the source code of a tested program.

2.2.3 Frameworks and tools for acceptance testing of web applications

Selenium [40] is one of the most widely used frameworks for automated browsing of web based applications. Selenium provides a domain-specific language that allows to write automation tests in such programming language as C#, Java, Perl, PHP, Python and Ruby. Selenium tools provide flexible operations for locating GUI elements on web pages, and comparing the actual result with the expected one. Automation tests written with Selenium can be run on different browser platforms. The functionality of Selenium test scripts, as well as Selenium frameworks’ functionality, can be extended, thus making Selenium’s framework highly flexible as comparing to other automation tools.

However, Selenium users have to use workarounds to handle Ajax elements, which may not properly work in different cases. Also, Selenium tests are not guaranteed to be compatible across different browsers.

One of the most commonly used tools designed for acceptance testing is Fit (Framework for Integrated Test) [18], which is based on the JUnit [41] testing framework. Acceptance tests using Fit are jointly created by clients and developers: clients write test cases in the form of HTML tables and for each HTML table programmers write individual Java classes called “fixture” which must conform to certain Fit conventions. A fixture class reads the content of an HTML table and automatically generates acceptance tests, which are then executed by the system on a tested program. In contrast to our approach, Fit requires expertise of developers to create automation tests, since it is necessary to write fixture classes for every type of HTML table.

Other tools for web-application acceptance testing include Cucumber [42] a behaviour-driven development framework, and the Robot Framework [43], which uses a keyword-driven testing approach, with a test case represented as a plain text or HTML file. However both frameworks do not provide complete testing solutions to handle Ajax-requests in web applications.

2.2.4 Remarks

A number of approaches [16, 17, 33] have been suggested for automated testing of dynamic web applications by “exploring” their behaviours. These methods are not suitable for the acceptance-testing task, as they automatically generate test cases based on the exhaustive analysis of dynamic DOM states of a web application without being aware of its intended behavior. Other frameworks and methods have been proposed that can be adopted for acceptance testing, however they either require substantial programming expertise [18, 34, 35] or do not have complete testing solutions for Ajax-based web applications [19, 20, 36, 37].

Additionally, in contrast to the majority of aforementioned frameworks (except for FLOAppTest [37], which is suitable only for Java programs), our tool is designed as a web application which can be accessed through any web browser.

Author	Technique	Type of supported WA	Tool	Evaluation
Mesbah <i>et al.</i> [8]	construction of a state-flow graph with DOM states as nodes and events as edges	Ajax-based	Crawljax	accuracy, scalability, and performance of crawling tasks
Marchetto <i>et al.</i> [16]	automatic test case generation through dynamic extraction of DOM states into a finite state machine and semantic analysis of interacting events sequences	JavaScript	-	test cases coverage
Marchetto <i>et al.</i> [17]	extends [16] with the usage of search-based algorithm to extract diverse event sequences of different length	JavaScript	-	test cases coverage
Dallmeier <i>et al.</i> [38]	construction of a usage model represented as a graph with DOM states as nodes and interactions between the states as edges	JavaScript	WebMate	cross-browser testing
Artzi <i>et al.</i> [34]	prioritization functions and test input generators using feedback-directed random automated test generation	JavaScript	Artemis	test cases coverage
Saxena <i>et al.</i> [35]	symbolic execution method with a constraint solver; random exploring strategy to examine event sequences	JavaScript	Kudzu	finding client-side injection code-vulnerables
Leshed <i>et al.</i> [19]	recording and replaying user actions performed in the web browser	no support for DHTML or Ajax updates	CoScripter	analysis of procedure-sharing practices with CoScripter
Mahmud and Lau [20]	extends [19] with assertions and subroutine identification	no support for DHTML or Ajax updates	CoTester	subroutine identification
Alshahwan and Harman [33]	search based algorithm for run-time collection of the constants from web pages	PHP, no complete testing solution for dynamic WA	SWAT	test coverage, test effort, and analysis of fault detection
Sauvé <i>et al.</i> [36]	users write Façade classes whose method signatures define commands of test scripts	Java-based, no support for dynamic WA	EasyAccept	user studies to analyze if EasyAccept assisted in creating better quality software
Araújo <i>et al.</i> [37]	extends [36] by providing visual interface for easier test script creation	Java-based, no support for dynamic WA	FLOAppTest	N/A

Table 2.1: Related literature on web applications testing approaches

Chapter 3

Feature Extraction

Our approach for detecting features from a set of DOM states [44] of a web application consists of the following main steps. First, DOM states of the target web application are collected using *Crawljax*, which is developed for crawling and testing Ajax-based web applications. Second, the collected DOM states are cleaned up, by removing the content, scripting code, and style rules. Third, using the *VTracker* tree-differencing algorithm, the distance between all pairs of collected “cleaned-up” DOM trees is calculated. Next, the computed distances are given as input to a hierarchical clustering algorithm, which produces a hierarchy of cluster merges, based on the similarity of the DOM trees. Finally, two different clustering evaluation techniques are applied, namely the L method [22] and the Silhouette coefficient [23], to automatically determine the number of clusters that correspond to the actual features.

To assess DOM-tree similarity we propose a novel composite-tree-edits-aware distance metric that recognizes structural changes in the DOM tree as simple or composite. We compared the proposed composite distance metric with the simplified one that treats all changes uniformly, and with the Levenshtein [21] string-distance metric. The results demonstrate that the composite tree-edits based distance metric is more appropriate for automatically deducing the number of clusters that is close to the actual number of features. The experimental results also demonstrate that the L method produced more accurate results for the proposed composite tree-edits based distance metric.

Before describing the approach in detail, we would like to explain key definitions used in this work:

- **Web application page (web page):** A dynamic web page of a web application whose user interface and content are modified through event-driven changes based on the inputs provided by users. Each web page corresponds to a DOM state.
- **DOM state:** The internal structure of the DOM tree of the client-side user interface that represents the state of a web application [8].
- **Feature:** A feature of a web application represents a functionality that is defined by requirements and is accessible to users through the client-side user interface of a web application [5].

Each feature corresponds to a cluster.

- **Cluster:** In the context of our feature extraction problem, a cluster is a group of DOM states that correspond to the same feature of a web application (for example “Login to the system” or “Search for an item”).

3.1 DOM state collection

The DOM state instances of examined web applications are collected with *Crawljax* [8]. *Crawljax* can crawl any Ajax-based web application by firing events and filling in form data. It creates a *state-flow* graph of the dynamic DOM states and the transitions between them. More details regarding *Crawljax* are provided in Chapter 4. Given the URL for the examined web application, *Crawljax* starts to systematically explore the web application from its main page. We configure *Crawljax* to collect 100 different states (at maximum) starting from the main page for each web application. We also configure *Crawljax* to ignore crawl depth by setting up the maximum crawl depth value equal to 0. The crawler is configured to click on default clickable elements which include `<A>`, `<BUTTON>`, and `<INPUT>` tags and to ignore the external links. By setting the *clickOnce* value to *true* we tell *Crawljax* to exercise clickable elements on the page only once.

Additionally, we configure *Crawljax* to provide meaningful input to simulate user input rather than generate random input strings. For example, knowing the identifier of the “Submit” field on the JPetStore search page, we can set up a value for this field which will be used by *Crawljax* whenever it will encounter the input field with the specified *id* attribute. Figure 3.1 shows an example of *Crawljax*’s configuration for JPetStore web application.

After the crawling of the subject web application is complete, we traverse the constructed state-flow graph and collect the discovered DOM states, which are then cleaned up and are given to a tree-differencing algorithm for pairwise comparison.

```
private static final String URL =
    "http://http://localhost:8080/petstore";

private static final int MAX_CRAWL_DEPTH = 0;
private static final int MAX_STATES = 100;

private static CrawlSpecification getCrawlSpecification() {
    CrawlSpecification crawler = new CrawlSpecification(URL);

    crawler.clickDefaultElements();
    crawler.setRandomInputInForms(false);
    crawler.setClickOnce(true);

    crawler.setMaximumStates(MAX_STATES);
    crawler.setDepth(MAX_CRAWL_DEPTH);

    crawler.setInputSpecification(getInputSpecification());

    return crawler;
}

private static InputSpecification getInputSpecification() {
    InputSpecification input = new InputSpecification();
    input.field("searchForm:searchString").setValue("cat");
    return input;
}
```

Figure 3.1: *Crawljax* configuration for JPetStore

3.2 DOM state clean-up

Before being given as input to the tree-differencing algorithm, the DOM trees are traversed and “cleaned up” from the text content, `<SCRIPT>` and `<STYLE>` tags and associated with them code. More specifically, the clean up process involves the following steps.

- The content (text information associated with HTML tags) of the page is removed.
- Tags `<SCRIPT>` and `<STYLE>` (and the content between them) are removed, to exclude scripts and CSS styling rules.
- Special attributes with script values are removed. For example, we noticed that attributes such as `jsprops`, `jsdisplay`, `jstcache`, `jsattrs`, `href`, `onclick` of the `<DIV>` tags contain scripts as values that are slightly changing between subsequent DOM states and affect the similarity of the DOM states:

```
<DIV class="mv-secondary" jsprops="activityId:8" style="height: 26px;">  
<DIV class="mv-secondary" jsprops="activityId:9" style="height: 26px;">
```

Figures 3.2 and 3.3 illustrate the excerpts of the “Menu” part in the DOM structure of the JPet-Store “Search item” page before and after the clean up process respectively. As seen in Figure 3.3 text between opening and closing `<TITLE>`, `<DIV>`, `<A>` tags is removed. All `<SCRIPT>` and `<STYLE>` tags with the corresponding code are completely removed as well.

```

<HTML>
<HEAD>
  <TITLE>Search Page</TITLE> <SCRIPT src="common.js"
    type="text/javascript"></SCRIPT>
</HEAD>
<BODY>
  <STYLE type="text/css">#rss-bar { margin: 0 auto 0px;}#rss-bar
  table
    td#rss-channel { background-repeat: no-repeat;
      background-position:
        top left; font-size: 14px; font-weight: bold;
        vertical-align: top;
        text-align: center; width: 254px;}#rss-bar table a { color:
        white;
        text-decoration: none;}#rss-bar table a:hover { color:
        #ffff00;}</STYLE>
  <SCRIPT type="text/javascript">var rss = new bpui.RSS();
    dojo.addOnLoad(function(){rss.getRssInJson('
    /petstore/faces/dynamic/bpui_rssfeedhandler/getRssfeed',
    'https://blueprints.dev.java.net/servlets/ProjectRSS?type=news',
    '4', '4000', 'News from BluePrints', 'news.jsp');});</SCRIPT>
  <TABLE bgcolor="white" border="0" bordercolor="gray"
    cellpadding="0" cellspacing="0" width="100%">
    <TBODY>
      <TR id="injectionPoint">
        <TD width="100">
          <A class="menuLink" href="/petstore/faces/index.jsp">
            <IMG border="0" height="70"
              src="/petstore/images/banner_logo.gif"
              width="70"></A>
          </TD>
          <TD align="left">
            <DIV class="banner">Java Pet Store</DIV>
          </TD>
          <TD align="right" id="bannerRight">
            <A class="menuLink"
              href="/petstore/faces/fileupload.jsp"
              onmouseout="this.className='menuLink';"
              onmouseover="this.className='menuLinkHover';">
              Seller</A>
            <SPAN class="menuItem">|</SPAN>
            <A class="menuLink" href="/petstore/faces/search.jsp"
              onmouseout="this.className='menuLink';"
              onmouseover="this.className='menuLinkHover';">
              Search</A>
            <SPAN class="menuItem">|</SPAN>
          </TD>
          ...
        </TR>
        ...
      </TBODY></TABLE>
  ...
</BODY>
</HTML>

```

Figure 3.2: Before cleaning up the “Menu” part in the DOM structure of JPetStore “Search item” page

```

<html>
<head>
<title></title>
</head>
<body>
  <table bgcolor="white" border="0" bordercolor="gray"
    cellpadding="0" cellspacing="0" width="100%">
    <tbody>
      <tr id="injectionPoint">
        <td width="100">
          <a class="menuLink">
            
          </a>
        </td>
        <td align="left">
          <div class="banner"></div>
        </td>
        <td align="right" id="bannerRight">
          <a class="menuLink"
            onmouseout="this.className='menuLink';"
            onmouseover="this.className='menuLinkHover';">
          </a>
          <span class="menuItem"></span>
          <a class="menuLink"
            onmouseout="this.className='menuLink';"
            onmouseover="this.className='menuLinkHover';">
          </a>
          <span class="menuItem"></span>
        </td>
        ...
      </tr>
      ...
    </tbody>...</table>
    ...
  </body>
</html>

```

Figure 3.3: After cleaning up the “Menu” part in the DOM structure of JPetStore “Search item” page

3.3 VTracker

To compute the distance between two DOM states we employ the generic tree-differencing algorithm *VTracker* [32], which extends the *Zhang-Shasha* tree-edit distance algorithm [45]. The *Zhang-Shasha* algorithm takes as input two ordered-labeled trees and computes the minimum edit distance between them based on a given cost-function for each of the edit operation types (insert, delete, change).

VTracker takes as input a pair of XML documents, each one considered as a *labeled ordered trees* and produces as output a *tree-edit sequence* of insert, delete, change, and move edit operations that can be applied to transform the first tree into the second one. The order of elements in the XML trees for *VTracker* is important as it resembles the order of the content of XML documents [32, 46].

As a result of comparing trees T_1 and T_2 the algorithm produces an edit script, which is a sequence M of mappings $map(i, j)$, where i is the node from T_1 and j is the node from T_2 such that for all (i_1, j_1) and $(i_2, j_2) \in M$ [32, 45]:

- $i_1 = i_2$ iff $j_1 = j_2$ (each node can be maximum involved in one edit operation);
- $T_1[i_1]$ is on the left of $T_1[i_2]$ iff $T_2[j_1]$ is on the left of $T_2[j_2]$ (the siblings order is preserved during the mapping);
- $T_1[i_1]$ is an ancestor of $T_1[i_2]$ iff $T_2[j_1]$ is an ancestor of $T_2[j_2]$ (the ancestor-child order is preserved during the mapping).

VTracker extends the *Zhang-Shasha* algorithm by providing additional properties which are discussed below.

3.3.1 Affine cost computation

The *Zhang-Shasha* algorithm considers each insert/delete operation on a node as independent regardless of the edit operations applied to the node's children or ancestors, and always assigns the same cost to each operation. As a result, two different edit scripts can have the same type and number of edit operations and equal minimum edit distances, regardless of the fact that one edit script may contain changes in a small neighbourhood of nodes where the second script includes changes scattered across the whole tree. According to [32] such behaviour is unintuitive, since a set of changes of the same type are likely to occur within the same sub-tree, rather than be scattered across the whole tree.

VTracker extends the original algorithm by adjusting the cost assigned to edit operations depending on other related operations. The cost of each operation, which in *VTracker* is considered as *context sensitive*, is regulated by an affine-cost function. The logic of this affinity function is as follows. If all children of a node are candidates for deletion, then it is more likely that the node itself will be deleted as well and the cost operation for deletion of this node is considered to be less

than any other regular deletion of a node. The same assumption is preserved for an insertion of a node. To reflect this affinity the cost of such insert/delete edit operations in *VTracker* is reduced by 50% [32, 46].

3.3.2 Simplicity heuristics

There are cases when *VTracker* can produce several edit scripts with the same cost. In such situations the algorithm should decide which edit script to return as output of XML trees comparison. To solve this problem *VTracker* applies a number of simplicity filters. The fundamental assumption in applying the simplicity heuristics in *VTracker* is that more complex edit scripts with the same cost are less likely to be produced than less complex edit scripts [32].

When the set of solutions is produced, *VTracker* applies the following simplicity heuristics to discard the most unintuitive solutions.

1. The algorithm looks for the minimal edit sequence path: if there are several paths with the same minimum cost, the algorithm selects the one that has the least number of addition/deletion operations;
2. The algorithm looks for paths with uninterrupted sequences of similar edit operations. It considers continuous edit operations of the same type as a single edit operation. If there are several different paths with the same minimum cost and the same number of edit operations, the algorithm selects the path that has the least number of changes in operations' types along the tree branch.
3. The algorithm considers that the same edit operations should be applied to sibling nodes. Therefore, *VTracker* maximizes the number of nodes along the branch tree to which the same edit operation is applied.

Essentially, the aforementioned heuristics and the usage of the affine-cost function are based on the assumption that similar edit operations are likely to be applied to the set of related nodes rather than to the set of independent nodes.

3.3.3 Cross-referencing

VTracker extends the *Zhang-Shasha* algorithm by introducing the use of cross-references between nodes of the compared trees, by considering the elements being referenced as a part of the structure of the referring elements. Also, while matching nodes, *VTracker* takes into account the context in which the element is used (i.e. the elements from which this element is being referenced). This process is called as *context-aware matching* [32, 46].

3.4 The Distance Metric

Different distance metrics lead to different clustering results, and it is essential to choose the right distance metric for clustering purposes because it affects the quality of the resulting clusters. After the comparison of two DOM trees is done, *VTracker* produces the edit script, which represents a sequence of primary edit operations for a pair of nodes that are

- matched (are exactly the same) in both trees;
- changed from the first tree to the second;
- moved from one location in the first tree to another location in the second tree;
- removed from the first tree; or
- inserted in the second tree.

The distance between two compared DOM trees T_1 and T_2 is defined as follows :

$$d_S(T_1, T_2) = \frac{|diff(T_1, T_2)|}{|diff(T_1, T_2)| + |match(T_1, T_2)|} \quad (1)$$

where:

- $diff(T_1, T_2)$ is the set of change, move, remove and insert edit operations in the tree-edit sequence for T_1 and T_2 ; and
- $match(T_1, T_2)$ is the set of match edit operations in the tree-edit sequence of T_1 and T_2 .

The distance metric defined in (1) ranges over the interval $[0, 1]$. It is minimized when $|diff(T_1, T_2)|$ is equal to zero (two DOM trees are completely similar) and is maximized when $|match(T_1, T_2)|$ is equal to zero (two DOM states are completely different).

We introduce a new distance metric, which recognizes structural changes in the DOM tree as simple or *composite*. A single *composite edit operation* identifies a subset of primary edit operations in $diff(T_1, T_2)$ applied to a set of corresponding nodes (in the first tree in the case of remove edit operations, in the second tree in the case of insert edit operations, or in both trees in the case of move/change edit operations) located under the same path to the root in the DOM tree. A composite edit operation may consist of different types of primary edit operations (for example, node inserts along with node changes), as long as the involved nodes are nested under the same path to the root. Two or more primary edit operations belong to the same composite edit operation if they are applied to nodes on the same (sub)path to the root. More specifically, two individual edit operations on

nodes A , B belong to the same composite edit operation, if the path to node A (starting from the root) is part of the path to node B .

Having the extracted set of composite edit operations we can define a new distance metric between two trees T_1 and T_2 as following:

$$d_C(T_1, T_2) = \frac{|diff'(T_1, T_2)| + |comp(T_1, T_2)|}{|diff'(T_1, T_2)| + |comp(T_1, T_2)| + |match(T_1, T_2)|} \quad (2)$$

where:

- $comp(T_1, T_2)$ is the set of composite edit operations extracted from the tree-edit sequence of T_1 and T_2
- $diff'(T_1, T_2) = diff(T_1, T_2) \setminus editOps(T_1, T_2)$
- $editOps(T_1, T_2) = \bigcup_{x \in comp(T_1, T_2)} x$ is the union of all edit operations in the set of composite edit operations.

Figures 3.4 and 3.5 illustrate a sample section of a DOM state of the JPetStore [47] search page before submitting a query (Figure 3.4) and after obtaining the query results (Figure 3.5). In response to the query, the DOM structure of the page is dynamically updated to contain a new table component with the query results (HTML element `FORM` with `name="resultsForm"` Figure 3.5). The simplified distance metric d_S handles each inserted node under the `<FORM HTML` element with `id="resultsForm"` as a separate primary edit operation. In turn, the composite distance measure d_C considers the insertion of the new `FORM HTML` node along with its nested nodes as a single composite edit operation.

In other words, in the proposed distance metric defined in (2), the number of primary edit operations that belong to composite edit operations is replaced by the number of composite edit operations. This distance metric is more robust to composite changes by reducing the weight of composite operations in comparison to non-composite edit operations.

```
<HTML>
<HEAD>
<TITLE>Search Page</TITLE>
</HEAD>
<BODY>
  <H1>Search Page</H1>
  <FORM action="/faces/search.jsp" id="searchForm" method="post"
    name="searchForm">
    <INPUT name="searchForm" type="hidden" value="searchForm">
    <TABLE>
      <TBODY>
        <TR>
          <TH>Search String</TH>
          <TD><INPUT id="searchForm:searchString"
            value="cat"></TD>
        </TR>
        <TR>
          <TD><INPUT id="searchForm:searchSubmit"
            type="submit"
            value="Submit"> <INPUT
            id="searchForm:searchReset"
            type="reset" value="Reset"></TD>
        </TR>
      </TBODY>
    </TABLE>
  </FORM>
</BODY>
</HTML>
```

Figure 3.4: A sample section of the DOM structure from JPetStore “Search page” (before query submission)

```

<HTML>
<HEAD>
<TITLE>Search Page</TITLE>
</HEAD>
<BODY>
  <H1>Search Page</H1>
  <FORM action="/faces/search.jsp" id="searchForm" method="post"
    name="searchForm">
    <INPUT name="searchForm" type="hidden" value="searchForm">
    <TABLE>
      <TBODY>
        <TR>
          <TH>Search String</TH>
          <TD><INPUT id="searchForm:searchString"
            value="random string">
          </TD>
        </TR>
        <TR>
          <TD><INPUT id="searchForm:searchSubmit"
            type="submit"
            value="Submit"> <INPUT
            id="searchForm:searchReset"
            type="reset" value="Reset"></TD>
        </TR>
      </TBODY>
    </TABLE>
  </FORM>

  <FORM action="/faces/search.jsp" id="resultsForm" method="post"
    name="resultsForm">
    <INPUT name="resultsForm" type="hidden" value="resultsForm">
    <TABLE>
      <TBODY>
        <TR>
          <TH>Map</TH>
          <TH>Name</TH>
          <TH>Description</TH>
          <TH>Tags</TH>
          <TH>Price</TH>
        </TR>
        <TR>
        </TR>
      </TBODY>
    </TABLE>
  </FORM>
</BODY>
</HTML>

```

Figure 3.5: A sample section of the DOM structure from JPetStore “Search page” (after query submission)

3.5 The Clustering Algorithm

In this section we discuss the most important clustering techniques as applied to the problem of feature extraction.

3.5.1 Partitioning clustering algorithms

Partitioning clustering algorithms divide data objects into a set of k clusters, where k is a predetermined number [48], [49]. Such algorithms usually start with a random partitioning of data objects into k groups, which are then iteratively refined until a termination condition is satisfied. The reassignment of clusters is based on the analysis of a fitness function's value, which usually corresponds to the cohesion of the clusters. A stopping criteria for the partitioning algorithm can be, for example, a fixed number of iterations, or when the partition of data objects remains unchanged.

K-means [49] is one the most widely used partitioning clustering algorithms. The algorithm clusters data objects based on their "neighbourhood". As input, in addition to data objects, the algorithm also requires the number of clusters k to be identified, as well as a random initial guess of k centres (centroids). During the refinement process the algorithm finds for each data object the closest centre to which the data object will belong. The algorithm then recalculates centres of the groups based on the data objects they include. These steps are repeated until the centres remain the same.

The manual specification of k clusters makes the K-means clustering algorithm unsuitable for our problem, since we don't know in advance the number of features that exist in a web application. A number of heuristic approaches are introduced to tackle the problem of automatically determining the number of clusters k and an initial assignment of centroids [50]. A common approach to address these two problems is to run K-means algorithm multiple times using different values of k and then to choose a set of resulting clusters with a smallest *sum of the squared error*. However, since the number of clusters k should range from 2 to the number of input data objects, these approaches may introduce a significant computational overhead if the number of data objects given as input (in our case, DOM trees) is large. Additionally, if the initial guess of centres was not good enough, the clusters produced by K-means algorithm can be of a poor quality.

Another problem related to K-means is that the algorithm can produce empty clusters if no data objects are allocated during the random initialization of clusters [51]. Also, in contrast to the hierarchical clustering algorithm, K-means is non-deterministic because the random selection of centroids can produce different final results [52]. Additionally, K-means does not perform well with input data that contains outliers [53] (detection and removal of outliers can improve the clustering result, but this requires examination of the input data). Finally, the algorithm can not be applied to any type of input data, since it is restricted to the data objects that can have specific coordinates in a feature space (e.g., Cartesian plane) and a notion of centroid [51]. In our problem, we give as an input the distances between pairs of DOM trees.

3.5.2 Density-based clustering algorithms

Density-based clustering algorithms define clusters as regions of data objects with high density separated by regions of a lower density [54]. The outliers and the data objects that separate clusters are considered as noise and border data points respectively. One of the most popular density-based clustering algorithms is DBSCAN [55], which uses the model of *density reachability* and *density connectivity*. DBSCAN groups the data objects based on a certain density criteria (for example, the minimum number of data objects in the density area) within a predefined radius. However, in this case the density depends on the selected radius. Unlike partitioning clustering algorithms, DBSCAN doesn't require to provide the number of clusters. However, manual specification of a density criteria also makes this algorithm not suitable for our problem since it is not clear how to map the density and the number of DOM trees corresponding to a feature in a web application. Also, the existence of so called *bridges* (i.e, data objects which are located at the equal distance from two dense regions) can potentially cause the merging of clusters of DOM trees that belong to different features.

3.5.3 Agglomerative hierarchical clustering algorithms

To group DOM trees that correspond to the same feature of a web application we employ a hierarchical clustering algorithm. The hierarchical clustering algorithm is more suitable for our problem since it has a number of advantages in the context of our problem as compared to other clustering algorithms such as partitioning or density-based.

There are two basic hierarchical clustering methods: agglomerative and divisive. Agglomerative hierarchical clustering algorithm starts with setting each data object as an individual cluster. In each iteration the algorithm merges the closest pair of clusters based on the provided distance measure for each pair of data objects. The process is repeated until all data objects are grouped into one cluster. In contrast to K-means, the hierarchical clustering algorithm is executed only once and doesn't require providing initial random guesses of clustering. The produced hierarchy of clusters with a single cluster with all data objects at the root of the hierarchy is called a *dendrogram*.

Divisive hierarchical clustering algorithms start with all data objects placed in a single cluster. In each iteration, the clusters are split into smaller clusters. The algorithm stops when each data object is placed in a separate cluster. In contrast to agglomerative hierarchical clustering algorithm, the divisive one needs to know which cluster should be divided and how to do the division [51].

In our approach to group DOM states we employ the agglomerative hierarchical clustering algorithm, which is deterministic. Also, in contrast to partitioning and density-based clustering algorithm, it does not require to manually specify input parameters. In general, the naive implementation of the hierarchical clustering algorithm has the complexity of $O(N^3)$, which makes the algorithm too slow when the size of the input data is large.

The hierarchical clustering algorithm requires a *linkage criteria* upon which the selection of the clusters to be merged is being decided. The most commonly used versions of agglomerative

clustering algorithm are:

- single-linkage or MIN (the minimum distance between any two data objects in two different clusters);
- complete-linkage or MAX (the maximum distance between any two data objects in two different clusters); or
- group average (the average distance among all pairs of data objects in two different clusters).
This linkage criteria is in-between of the single-linkage and complete-linkage versions.

The single-linkage approach tends to form highly separated clusters whereas complete-linkage version of the hierarchical clustering algorithm tends to form more tightly centered clusters [31]. In the context of grouping similar DOM trees, we assume that most of the web pages of a web application can have common parts, for example header, footer, and menu. This implies that the distances between DOM trees are relatively small by default, and in the context of the hierarchical clustering application, there are no highly distant clusters of DOM trees. Therefore, the single-linkage approach is not suitable in our case, since it tends to form highly separated clusters. The complete-linkage version of the algorithm is more appropriate for our problem, since we are interested in grouping the most similar DOM trees rather than just similar DOM trees (to form tight clusters).

3.5.4 Determining the number of clusters

Given the dendrogram produced by the hierarchical clustering algorithm, a cut-off threshold is required to determine the actual cluster partition; the threshold essentially decides how many clusters the final partition will include, eliminating the partitions “higher” in the dendrogram. The usage of a fixed cut-off threshold can result in a set of clusters of different quality varying over the set of input data objects. A fixed low cut-off threshold can produce clusters with similar data objects scattered across different clusters, while a high cut-off threshold can produce a partition of clusters that include dissimilar data objects. Therefore, the cut-off threshold is, in most cases, subjectively determined, assuming some knowledge of the input data. Previous approaches used either a predefined threshold [9] or required a user-defined threshold [10, 11, 24].

In our approach, we apply two clustering analysis methods to determine the number of clusters in a partition and thus infer the cut-off threshold automatically. More specifically, we apply two different methods, namely the L-method [22] and Silhouette coefficient [23] to automatically determine the number of clusters to be returned by the hierarchical agglomerative clustering algorithm without providing ground truth for cluster evaluation or using predefined input parameters.

3.5.4.1 The L method

The L-method is a technique for identifying an appropriate number of clusters to be returned by hierarchical clustering or segmentation algorithms, and it doesn't require any input parameters or

constants. Using clustering evaluation metrics such as merge distances produced by the hierarchical clustering algorithm the L method constructs an *evaluation graph* (Figure 3.6) where the x -axis is the number of clusters and the y -axis is the value of the merge distances at x clusters. The *knee* (the region of the maximum curve) is used to determine the number of correct clusters to be returned as a result of the clustering algorithm. The knee on the evaluation graph is estimated as a region between two lines that most closely fit the curve. To estimate a reasonable number of clusters to be returned by clustering algorithm the L method should be run only once. The running time of the L method algorithm is $O(N^2)$ with respect to the number of data points on the evaluation graph [22].

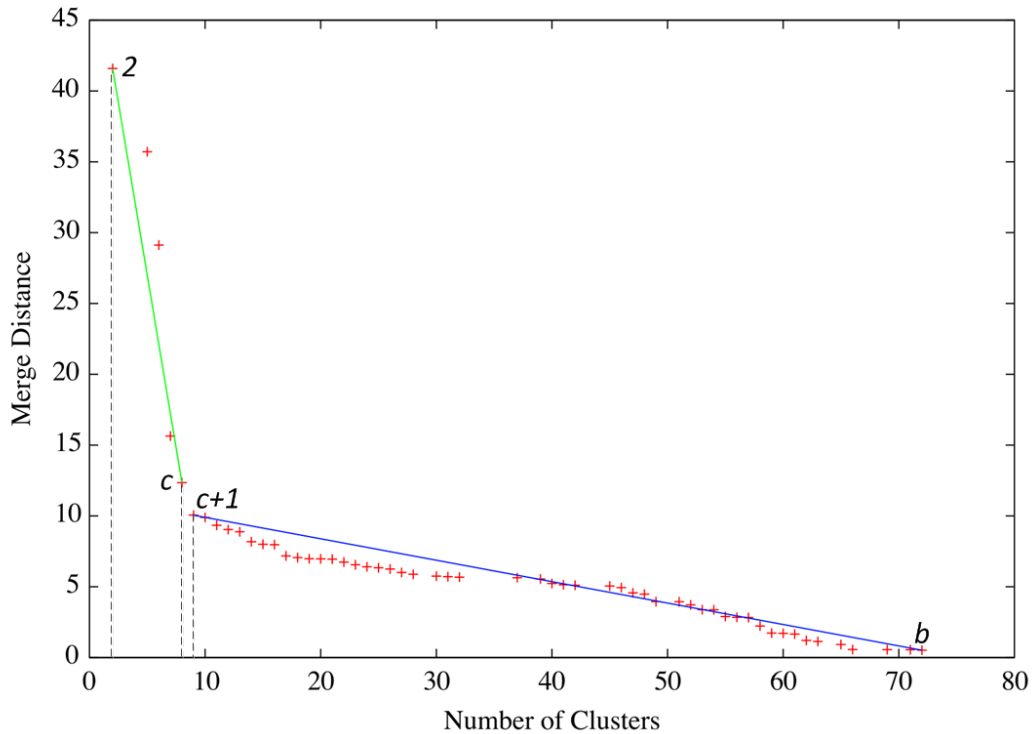


Figure 3.6: JPetStore evaluation graph

An example of an evaluation graph constructed using the merge distances produced by hierarchical clustering algorithm using the complete-linkage criteria for the JPetStore web application is shown on Figure 3.6. There are three recognizable regions on the graph:

1. an inclined region of data points on the left of the graph;
2. a flat region of data points that form almost a straight path on the right side of the graph; and
3. a curved region of data points between them.

The nearly straight segment that is formed by the data points on the right side of the evaluation graph indicates that there are a lot of similar clusters that are merged together. The abruptly increasing region (moving from the right) on the left of the evaluation graph means that dissimilar clusters

are being merged (the merge distances grow very quickly), hence the quality of the clusters at this point is being decreased. A reasonable number of clusters is located between these two regions in the curved area, or *knee* of the evaluation graph [22]. The region of data points that form an inclined line on the evaluation graph contains clusters that are dissimilar to each other, whereas the flat region of data points contains many clusters that are similar to each other. Therefore, the knee region contains clusters that are both highly homogeneous and dissimilar to each other [22].

The L method aims to find a pair of lines that most closely fit the data points on the graph. Each line must start at either end of the data points and must include at least two points. Both lines together should cover all data points on the graph. The method does not take into account the data point that corresponds to the final cluster (when all clusters are merged into one), therefore the x -values range from 2 to b (which is equal to the total number of elements given as input to the clustering algorithm), and the total number of data points on the graph is $b - 1$. If we consider that the data points are partitioned at $x = c$ (see Figure 3.6), then L_c and R_c are the left and right sequences of data points, respectively. Left sequence (L_c includes data points $x = 2..c$ and right sequence R_c comprises data points $x = c + 1..b$). The total root mean squared error $RMSE_c$ is defined as follows:

$$RMSE_c = \frac{c-1}{b-1} \times RMSE(L_c) + \frac{b-c}{b-1} \times RMSE(R_c) \quad (3)$$

where $RMSE(L_c)$ is the root mean squared error of the best fit line for the left sequence of data points in L_c and $RMSE(R_c)$ is the root mean squared error of the best fit line for the right sequence of data points in R_c [22]. The weights are proportional to the lengths of L_c ($c - 1$) and R_c ($b - c$), respectively. The L method seeks the value of c , \hat{c} , such that $RMSE_c$ is minimized. The location of the knee at $x = \hat{c}$ is used as the number of clusters to return.

In order to compute \hat{c} , the L method iterates over the values of $c = 3..b - 2$ and forms a pair of lines (the first line by joining the points corresponding to x -values 2 and c and the second one by joining the points corresponding to x -values $c + 1$ and b) in order to compute the $RMSE_c$ value for each possible value of c .

3.5.4.2 The Silhouette coefficient

The Silhouette coefficient [23] is a clustering evaluation metric that describes how well each data point is located within a set of clusters. The technique that computes the Silhouette coefficient requires as input dissimilarities (i.e., distances) between pairs of data point and a set of clusters produced by any clustering algorithm. For every data point i in a set of data points I , it computes $a(i)$ as an average dissimilarity between i and any other data point within the same cluster A . The

value $a(i)$ describes how well i fits within its cluster (the smaller the value of $a(i)$, the better the data point is fitted). Next, for every cluster C which doesn't contain i , it computes the average dissimilarity of i with all data points from C . The cluster with the lowest average dissimilarity is called the *nearest neighbour* and the average dissimilarity between i and this cluster is denoted as $b(i)$.

The Silhouette $s(i)$ for data point i , and the average Silhouette coefficient \bar{s}_k over all data points in I in a partition of k clusters are computed as follows:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))} \quad (4)$$

$$\bar{s}_k = \frac{1}{|I|} \sum_{i=1}^{|I|} s(i) \quad (5)$$

If cluster A contains only one data point, $s(i)$ in this case is equal to zero. The value of $s(i)$ ranges over the interval $[-1, 1]$; the closer the value of $s(i)$ is to one, the more appropriately the data point i is clustered within the set of clusters. The larger the value of \bar{s}_k , the better the quality of the clustering is. The complexity of the computation of the Silhouette coefficient is $O(N^2)$.

3.6 Experiments and Results

We evaluated the proposed distance metric d_C in terms of its capabilities to determine the correct number of features in a web applications using two different methods: the L method [22] and Silhouette coefficient [23]. We compared the produced results with the results obtained when using the Levenshtein edit distance as a distance metric for DOM trees comparison. As discussed in Chapter 2, Levenshtein edit distance has been widely used to compare the *structural* similarity of web pages. Additionally, we compared with the results produced when using the simplified tree-edit-operation based distance metric d_S that does not take into account composite changes.

3.6.1 Experimental setup

We conducted the experiments using three different web applications¹:

- an online shopping application (garage.ca)
- a map-navigation application (googlemaps.com) and
- an online travel booking application (kayak.com).

¹From Google Maps, Kayak, and Garage web applications the only information collected was their DOM structure. To collect this information, *Crawljax* parsed the DOM trees located at <http://googlemaps.com>, <http://kayak.com>, and <http://garage.ca> and then stripped them of all content which was visible on the HTML page. No data besides the DOM tree itself was collected. The Google Maps API was not used for any data retrieval in the case of Google Maps.

All three web applications are implemented using Ajax, which allows to dynamically update their DOM trees at runtime. The DOM trees instances of examined web applications were collected with *Crawljax* [8].

Web application	# of states	Average # of nodes	Median (nodes)	Standard deviation (nodes)	Average DOM depth	Median (depth)	Standard deviation (depth)
Google Maps	45	968	1080	231.83	20	20	0.59
Garage	66	547	540	111.37	12	12	1.60
Kayak	78	894	887.5	541.65	26	29	6.36

Table 3.1: Number of states, DOM size and depth for the examined web applications.

Table 3.1 reports the number of DOM states collected by *Crawljax*, the average, median and standard deviation of the number of nodes in DOM states, and the depth of DOM structures for each of the examined web applications. Analyzing the standard deviation values we can see that Garage and especially Google Maps have little variation in the size and depth of their DOM states. This indicates that the DOM states within Garage and Google Maps web applications have similar structure. For example, Google Maps has a consistent presentation layout (the search panel and the map area) among different features that the web application offers. Therefore, we expect to have a set of clusters for these two web applications with a lower accuracy. In contrast, standard deviation values for Kayak web application show higher variations in size and depth of DOM states. This can be explained by the nature of the Kayak web application which is a search engine for travel offers and deals: search queries submitted to Kayak can produce web pages with different search results that may vary in length, size and structural complexity of the DOM tree of the web page in general. This results in DOM states with different structures, therefore we expect the accuracy of the produced set of clusters for Kayak to be higher. Clearly, the different nature of the examined web applications explains the difference in the variation of size and depth between the generated DOM trees.

To perform a pairwise comparison of DOM trees using the Levenshtein edit distance as a similarity metric we follow the same approach used in the literature [9–11]. We extract a sequence of HTML tags from the DOM trees and remove the content (text between corresponding tags) and tags’ attributes. An example of such a string for DOM structure from Figure 3.4 is shown on Figure 3.7. Considering each HTML tag as a token, the Levenshtein edit distance computes the minimum number of edit operations (insertion, deletion, and replacement) required to transform the first string of HTML tags into the second one. We normalize the Levenshtein distance within the $[0, 1]$ range by dividing it with the maximum achievable number of edit operations which is equal to the length of the largest string of HTML tags (out of two input strings). The normalized Levenshtein distance metric used for comparison of two strings sequences of HTML tags S_1 and S_2 is defined in (6):

$$d_L(S_1, S_2) = \frac{Ld(S_1, S_2)}{\max(\text{length}(S_1), \text{length}(S_2))} \quad (6)$$

where $Ld(S_1, S_2)$ is the Levenshtein distance between sequences S_1 and S_2 and $\text{length}(S)$ is the number of tags in sequence S .

```
<HTML><HEAD><TITLE></TITLE></HEAD><BODY><H1></H1><FORM><INPUT><TABLE>
<TBODY><TR><TH></TH><TD><INPUT></TD></TR><TR><TD><INPUT></TD></TR>
</TBODY></TABLE></FORM></BODY></HTML>
```

Figure 3.7: An example of a string sequence of HTML tags used to compute the Levenshtein edit distance

3.6.2 Evaluation Methodology

To evaluate the accuracy of clustering results produced by each distance metric, a set of actual clusters of similar pages (*reference*) was manually constructed as follows. We captured a screenshot of the corresponding web page as shown in the browser for each discovered DOM state. Two people (participants) were involved in the manual clustering process: the author, as well as a fellow member of the research group. Participants independently examined the screenshots of collected web pages and independently manually grouped them into clusters of features according to their intuitive visual perception of the feature that each web page offers. More specifically, both participants agreed to follow the following rule during the independent clustering: “look for web pages that belong to the same functionality”. The screenshots corresponding to the same feature (for example “search for flights”) were grouped into one cluster. In effect the two participants created clusters based on the collected screenshots, by judging whether two screenshots reflected the same feature. The automated method then creates clusters by analyzing the DOM states corresponding to the screenshots. Next, during the meeting, the two participants merged the clusters obtained by each of them, and where the disagreement of a feature interpretation happened, the participants reached a common consensus by discussing the decisions between them. The disagreement happened in cases where one participant grouped sub-clusters of features into a single cluster of a higher-level feature, while the other participant grouped each of these lower-level features into separate clusters. To eliminate the dissimilarities in the set of clusters obtained by each participant the two individuals decided to adopt the approach of fine-grained decomposition of features (where lower-level features are grouped into separate clusters).

The initial agreement between the two experimenters (before merging) based on the Jaccard similarity coefficient [51] (which is used to compare the similarity of data sets and measure the level of agreement) for the examined three web applications is shown in Table 3.2. For two clusterings C and C' the Jaccard similarity coefficient is defined as $J(C, C') = \frac{N_{11}}{N_{11} + N_{10} + N_{01}}$, where N_{11} is the

Web app.	Jaccard similarity coefficient
Google Maps	0.890
Garage	0.997
Kayak	0.993

Table 3.2: Jaccard similarity coefficient for the examined web applications

number of data-point pairs in the same cluster under both C and C' , N_{10} is the number of data-point pairs in the same cluster under C but not under C' , and N_{01} the number of data point pairs in the same cluster under C' but not under C . The computed Jaccard similarity coefficients for all three web applications show a generally high agreement ($\geq 89\%$ based on Jaccard Index) between the categories of DOM states obtained from the project participants.

The merged results have been considered as actual clusters of similar pages (*reference*) based on which the accuracy of the examined distance metrics was evaluated. Table 3.3 lists the features extracted by the project participants. The labels assigned to the features were constructed by the participants based on their shared understanding of the cluster screenshots.

Web app.	Features
Google Maps	(1) Main search page, (2) Input origin and destination for directions, (3) Display directions by car, (4) Display directions by public transport
Garage	(1) Main page, (2) Gift cards and certificates, (3) Sign-in page, (4) Purchase gift card, (5) Locate store on GoogleMaps, (6) Online store FAQ, (7) Contact Garage, (8) Tops, (9) Coats, (10) Tanks, (11) Long sleeves tees, (12) Shirts, (13) Jeans, (14) Accessories, (15) Sleepwear, (16) Sale items, (17) Item detailed view, (18) Technical help, (19) Payment help
Kayak	(1) Search for flights, (2) Login page, (3) Password reminder, (4) Help, (5) Search for hotels, (6) Hotel search results, (7) Search for cars, (8) Car search results, (9) Car advanced search, (10) Search for deals, (11) Deals search results, (12) Flight search results, (13) Trip planner, (14) Find a Kayak booking, (15) "More" page, (16) Hotel advanced search

Table 3.3: Manually extracted features

3.6.3 Experimental results

On the set of DOM states collected by Crawljax from the examined web applications we have applied the hierarchical agglomerative clustering algorithm (using the complete-linkage criterion) with each of the distance metrics: simple tree-edits based distance metric d_S , composite tree-edits based distance metric d_C , and Levenshtein distance metric d_L .

Using the partitions of clusters and merge distances obtained from the resulting dendrogram for each examined web application and distance metric:

1. We applied the L method [22] to the evaluation graphs (Figures 3.8, 3.9, 3.10, 3.11, 3.12, 3.13, 3.14, 3.15, 3.16) created using the distances produced by d_S , d_C , and d_L distance met-

rics as input to the hierarchical clustering algorithm. For each evaluation graph we computed the value of $x = \hat{c}$ that minimizes the $RMSE_c$ value and represents a reasonable number of clusters to be returned according to the L method.

2. For each partition P_k with k number of clusters, where k ranges from 2 to $n - 1$ (n is the total number of extracted DOM states for a web application) we computed \bar{s}_k . A reasonable number of clusters k_α to be returned corresponds to the maximum value α over all Silhouette coefficients \bar{s}_k :

$$\alpha = \max(\{\bar{s}_k : k = 2, \dots, n - 1\}) \quad (7)$$

The predicted number of clusters returned by both methods for each distance metric and web application were used to produce partitions with \hat{c} and k_α number of clusters, respectively. For each obtained partition to evaluate its quality we computed precision and recall.

Let the set of actual clusters be defined as *reference* and the set of clusters returned by a hierarchical clustering algorithm with a distance metric for the same set of elements be defined as *response*. A given pair of elements (a, b) is considered as:

- *True Positive*, if a and b belong to the same cluster both in reference and response
- *False Negative*, if a and b belong to the same cluster in reference, but to different clusters in response
- *False Positive*, if a and b belong to different clusters in reference, but the same cluster in response.

By applying this process to every pair of elements we can obtain the total number of *True Positives* (TP), *False Negatives* (FN) and *False Positives* (FP), based on which the *precision* and *recall* measures can be calculated as follows:

$$precision = \frac{TP}{TP + FP} \quad recall = \frac{TP}{TP + FN}$$

We can combine both metrics by computing *F-measure*, which is defined as follows:

$$F - measure = 2 * \frac{precision * recall}{precision + recall}$$

The predicted number of clusters \hat{c} returned by the L Method along with the corresponding merge distance, F-measure, precision and recall values for each examined web application are shown in Tables 3.4, 3.6, and 3.8. The predicted number of clusters k_α determined by the Silhouette coefficient, as well as the corresponding merge distance, F-measure, precision and recall values for each examined web application are shown in Tables 3.5, 3.7, and 3.9, where column α is the maximum obtained value over all Silhouette coefficients. The actual number of clusters is obtained from manual clustering performed by project participants.

distance	min $RMSE_c$	merge dist.	\hat{c}	F- measure	Precision	Recall
d_S	0.41	1.33	13	0.448	0.963	0.292
d_C	0.18	3.33	6	0.832	0.985	0.721
d_L	4.41	2.07	15	0.396	0.957	0.250
Actual number of clusters: 4						

Table 3.4: L method: Predicted number of clusters, F-measure, precision, and recall for Google Maps

distance	α	merge dist.	k_α	F- measure	Precision	Recall
d_S	0.86	2.91	12	0.458	0.964	0.300
d_C	0.79	4.41	5	0.930	0.937	0.924
d_L	0.88	2.07	15	0.396	0.957	0.250
Actual number of clusters: 4						

Table 3.5: Silhouette Coefficient: Predicted number of clusters, F-measure, precision, and recall for Google Maps

3.6.4 Discussion

The results shown in Tables 3.4, 3.6, and 3.8 demonstrate that a different number of clusters was produced for each distance metric using the L method. However, we can see that the best results for each web application were obtained when using the composite distance d_C , which returned a \hat{c} value that is closer to the actual number of clusters in all examined web applications (and in the case of Garage the predicted number of clusters is exactly the same as the actual one). More specifically, when using the d_C distance metric, the predicted number of clusters returned by the L method is 6 for Google Maps (the actual number of clusters is 4), 19 for Garage (the actual number of clusters is 19), and 17 for Kayak (the actual number of clusters is 16). In turn, for distance metrics d_S and d_L the predicted number of clusters \hat{c} is notably different (larger or smaller) compared to the actual number of clusters.

Analyzing the evaluation graphs produced for each distance metric (Figures 3.8, 3.9, 3.10, 3.11, 3.12, 3.13, 3.14, 3.15, 3.16), we observe that on the graphs produced with d_C as a distance metric we can clearly identify regions required by the L method to effectively detect a knee. On Figures 3.9,

distance	min $RMSE_c$	merge dist.	\hat{c}	F- measure	Precision	Recall
d_S	4.90	8.52	25	0.861	0.972	0.774
d_C	2.26	7.03	19	0.960	0.973	0.947
d_L	2.23	11.47	14	0.856	0.840	0.872
Actual number of clusters: 19						

Table 3.6: L method: Predicted number of clusters, F-measure, precision, and recall for Garage

distance	α	merge dist.	k_α	F-measure	Precision	Recall
d_S	0.67	20.90	15	0.852	0.834	0.872
d_C	0.55	7.93	17	0.937	0.920	0.955
d_L	0.78	14.59	13	0.854	0.837	0.872
Actual number of clusters: 19						

Table 3.7: Silhouette Coefficient: Predicted number of clusters, F-measure, precision, and recall for Garage

distance	min $RMSE_c$	merge dist.	\hat{c}	F-measure	Precision	Recall
d_S	2.74	7.79	24	0.832	1.00	0.713
d_C	1.24	4.58	17	0.996	1.00	0.992
d_L	2.72	4.86	29	0.770	1.00	0.626
Actual number of clusters: 16						

Table 3.8: L method: Predicted number of clusters, F-measure, precision, and recall for Kayak

3.12, and 3.15 we can see the following distinct regions: a flat region of data points (on the right) followed by a sharply-increasing region of data points (on the left) which indicates that the merge distances started to grow very rapidly. According to [22] this abrupt increase occurs when highly dissimilar clusters are merged by the clustering algorithm. The reasonable number of clusters to be returned can be discovered on the evaluation graph right before the sharp increase (when moving from right to left). The knee region on these graphs (Figures 3.9, 3.12, and 3.15) is well-defined, therefore the returned \hat{c} value is more accurate and precise for distance measure d_C , than for the other two distances. These results demonstrate that the hierarchical clustering algorithm with the composite distance metric d_C produced well-separated clusters for which the L method was able to identify a reasonable number of clusters very close to the actual number of clusters defined according to human perception. Additionally, as it can be observed from Tables 3.4, 3.6, and 3.8 distance d_C , which makes a distinction between composite and simple changes, obtained the highest F-measure value in all three examined web applications (0.832 for Google Maps, 0.960 for Garage, and 0.996 for Kayak) using the L method.

distance	α	merge dist.	k_α	F-measure	Precision	Recall
d_S	0.81	51.88	15	0.994	0.996	0.992
d_C	0.88	15.61	12	0.992	0.984	1.00
d_L	0.80	20.64	21	0.820	1.00	0.694
Actual number of clusters: 16						

Table 3.9: Silhouette coefficient: Predicted number of clusters, F-measure, precision, and recall for Kayak

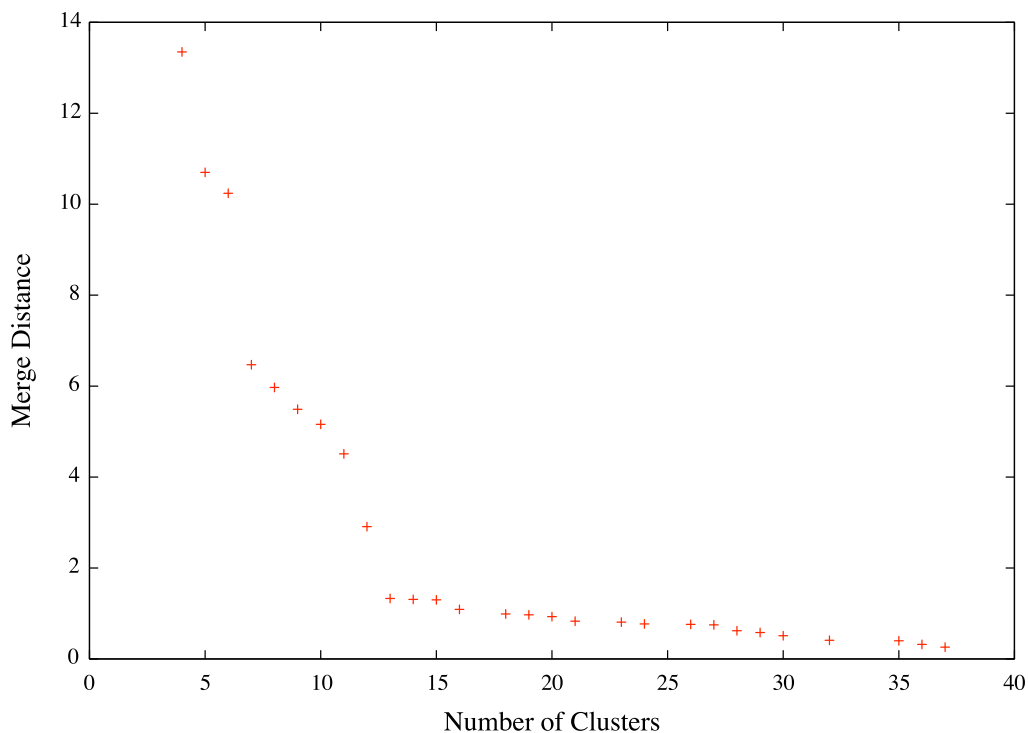


Figure 3.8: Evaluation Graph for Google Maps with d_S

The evaluation graphs constructed by using merge distances produced by hierarchical clustering algorithm with distance measure d_S (Figures 3.8, 3.11, and 3.14) have a smoother transition between the flat and increasing paths of data points, and thus the returned by the L method number of clusters for d_S is not so precise.

On the contrary, most of the evaluation graphs in Figures 3.10, 3.13 and 3.16, do not contain any obvious sharp transition between the flat and the increasing paths of data points, and therefore for the Levenshtein distance measure d_L the L method could not identify an acceptable number of clusters. The knees on these evaluation graphs are ambiguous. This implies that the distance metric used is not well-defined, therefore the clusters produced by the hierarchical clustering algorithm with d_L as a distance metric are not clearly separated. The L method could not determine an acceptable number of clusters for Google Maps web application, and as a result the corresponding partitions have poor clustering quality which is demonstrated by low F-measure values (0.448 for d_S and 0.396 for d_L). For Garage and Kayak web applications the F-measure values for distances d_S and d_L are notably lower than the F-measure value for distance d_C .

The number of clusters determined by the computation of the average Silhouette coefficient also varies for each distance metric and web application (Tables 3.5, 3.7, and 3.9). However, analyzing the computed F-measure (0.930 for Google Maps, 0.937 for Garage, and 0.992 for Kayak), we can see that the proposed distance metric d_C produced sets of clusters of high quality for all three web

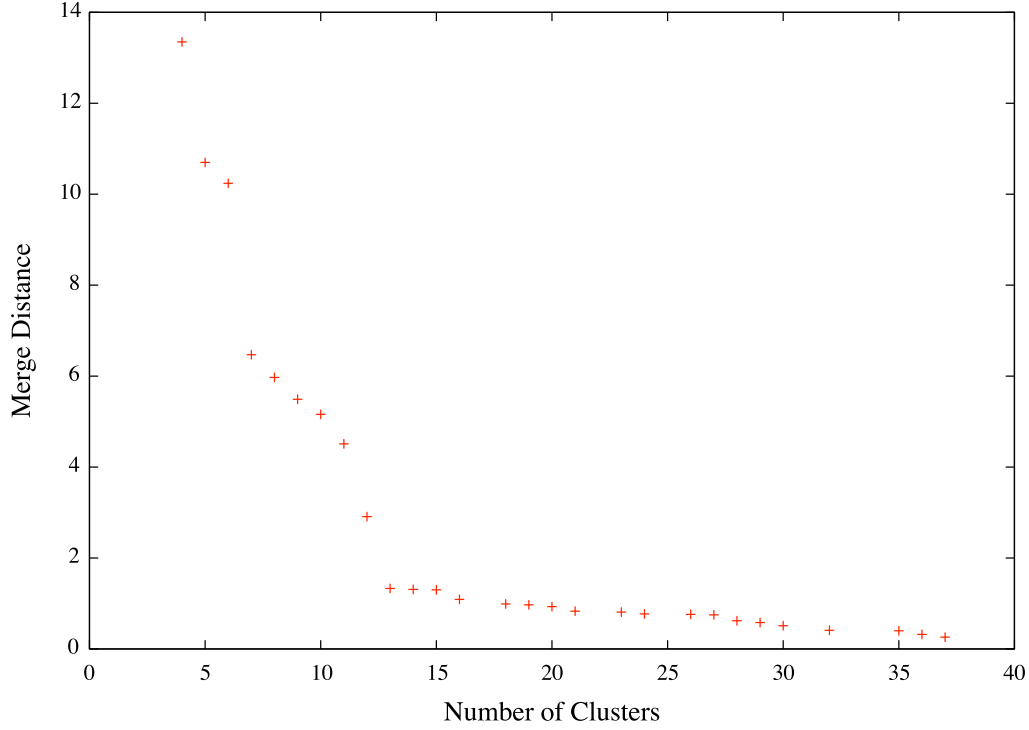


Figure 3.9: Evaluation Graph for Google Maps with d_C

applications, whereas the clustering quality of the partitions for distance metrics d_S and d_L is not stable and varies for each examined web application. More specifically, the F-measure values (0.458 for d_S and to 0.396 for d_L) for Google Maps web application indicate that the predicted number of clusters for distance metrics d_S and d_L produced partitions with poor clustering results. For other two web applications, Garage and Kayak, the F-measure values for distance d_L are lower than the F-measure value obtained by distance d_C (with the exception of d_S for Kayak).

Obviously, the distance metric affects the quality of the produced sets of clusters. In general, we obtained good results for our distance metric d_C using both clustering evaluation methods for all three web applications (in contrast to other examined distances). Also, by observing that our distance metric d_C produced best results for lower cut-off thresholds we can imply that d_C is a metric that in general produces smaller distances between the compared DOM trees.

3.6.5 Conclusions

The results of the evaluation on three different real-world web applications have shown that the usage of the composite tree-edits based distance metric improved the accuracy of the clustering results over the clustering results obtained by previous approaches that treat web pages as sequences of HTML tags (i.e., ignoring completely the tree structure of the web pages) and a simplified tree-edit-operation based distance metric that does not take into account composite changes. Also, these two

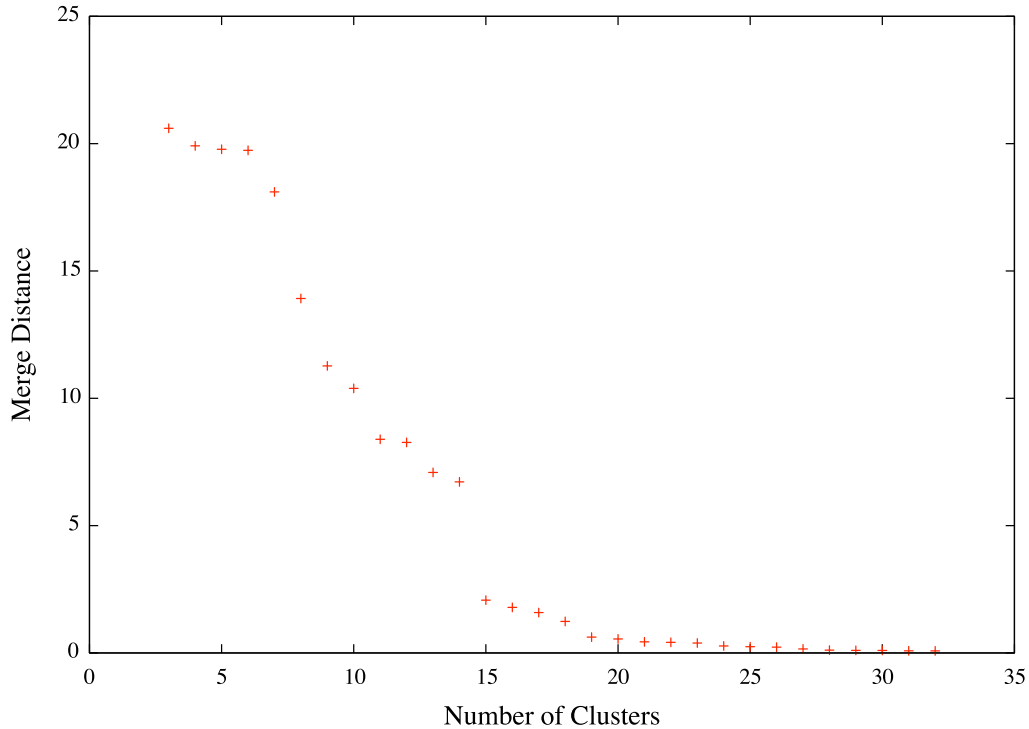


Figure 3.10: Evaluation Graph for Google Maps with d_L

distance metrics did not work as well with the L method and Silhouette coefficient, producing clustering results with lower accuracy as comparing with the clustering results obtained by our proposed distance d_C . Moreover, in this research we demonstrated that the process of extracting the features for a dynamic web application can be fully automated by employing hierarchical clustering algorithms and deducing the reasonable number of clusters in a partition using the clustering evaluation techniques. Finally, the evaluation results showed that the proposed distance metric d_C produced partitions of clusters with high accuracy for all web applications.

The extraction of web-application features plays an important role in reverse-engineering. The extracted features can provide an overview of a web application’s structure which can assist in documenting and understanding the overall functionality of the web application. Also, the retrieved features of the web application can be used in constructing the web application’s navigational model from the user’s point of view.

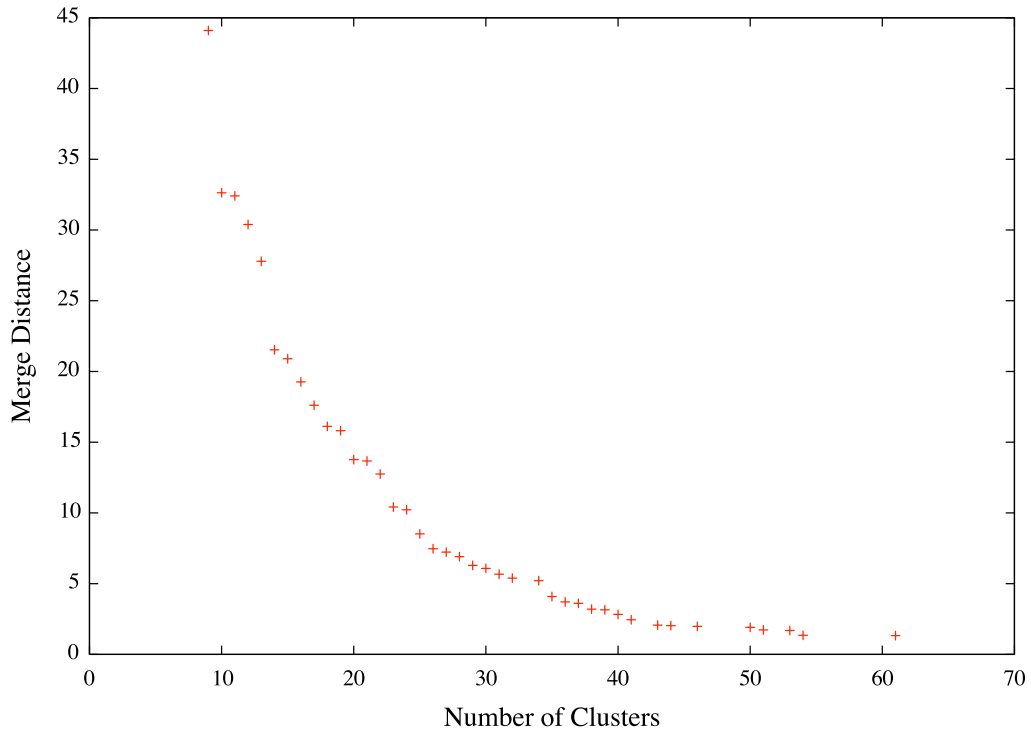


Figure 3.11: Evaluation Graph for Garage with d_S

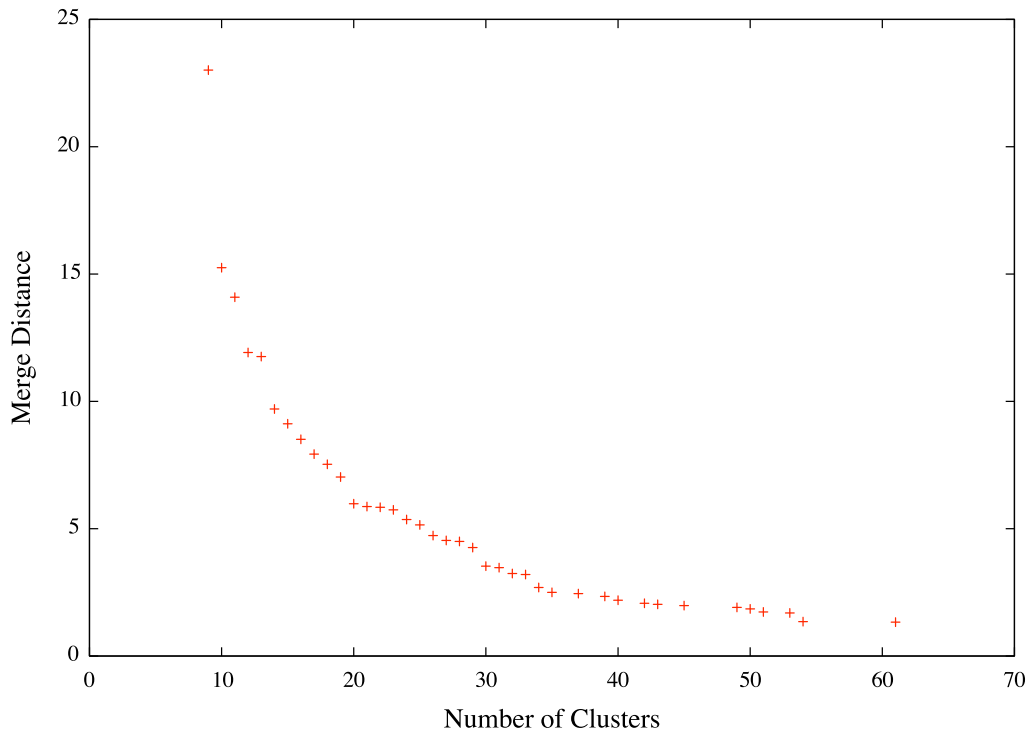


Figure 3.12: Evaluation Graph for Garage with d_C

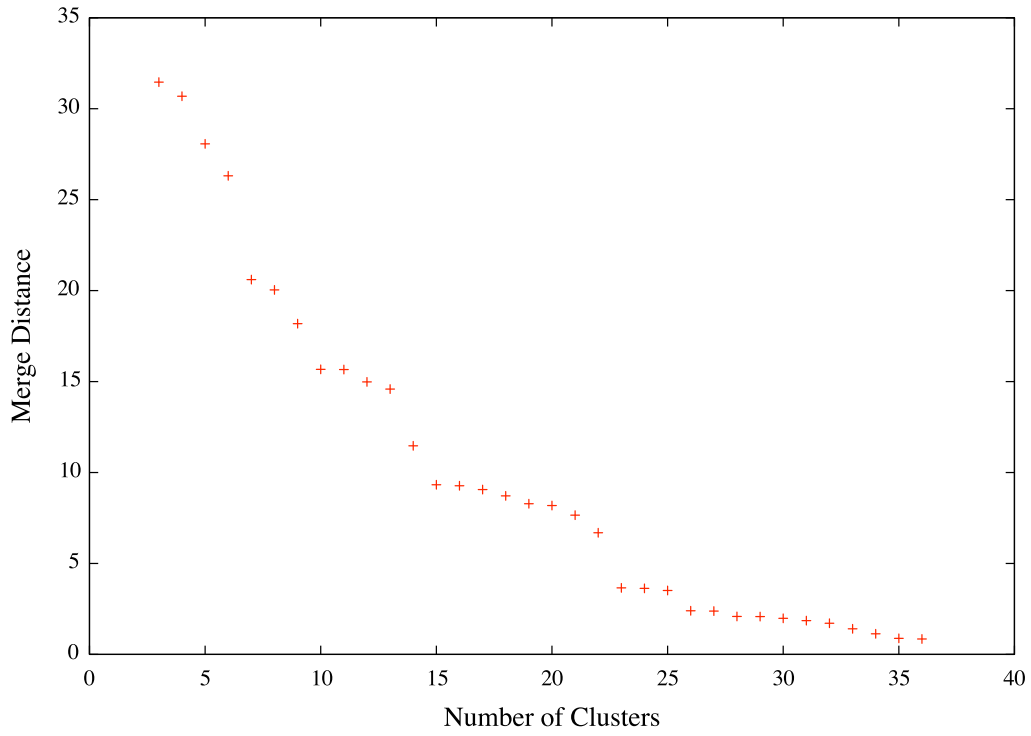


Figure 3.13: Evaluation Graph for Garage with d_L

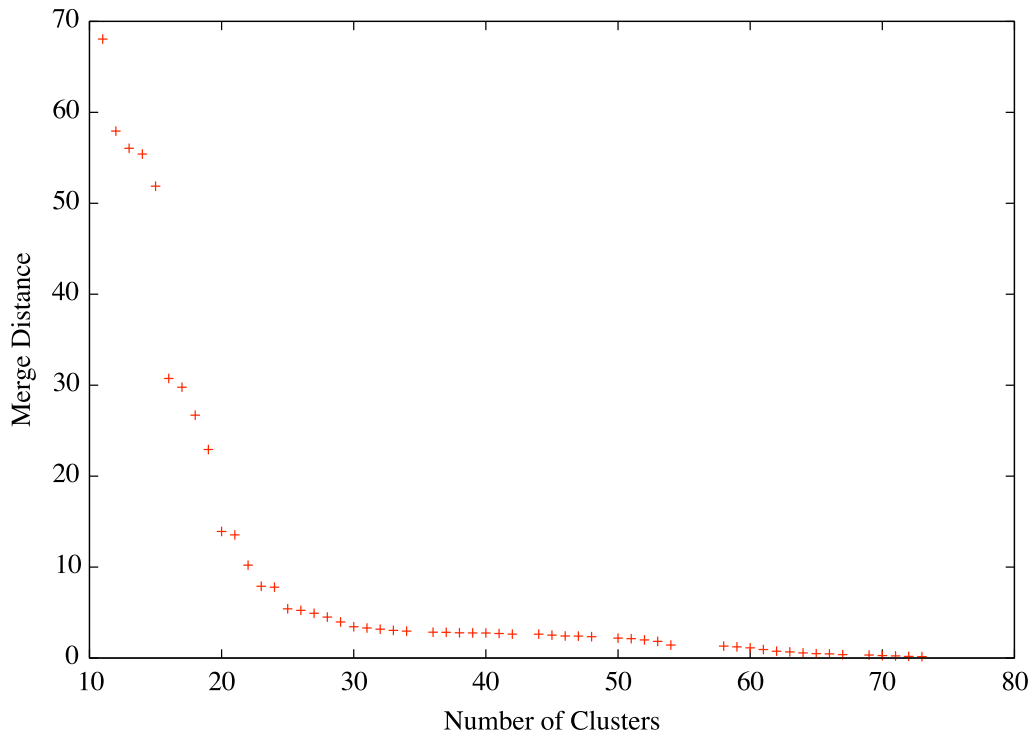


Figure 3.14: Evaluation Graph for Kayak with d_S

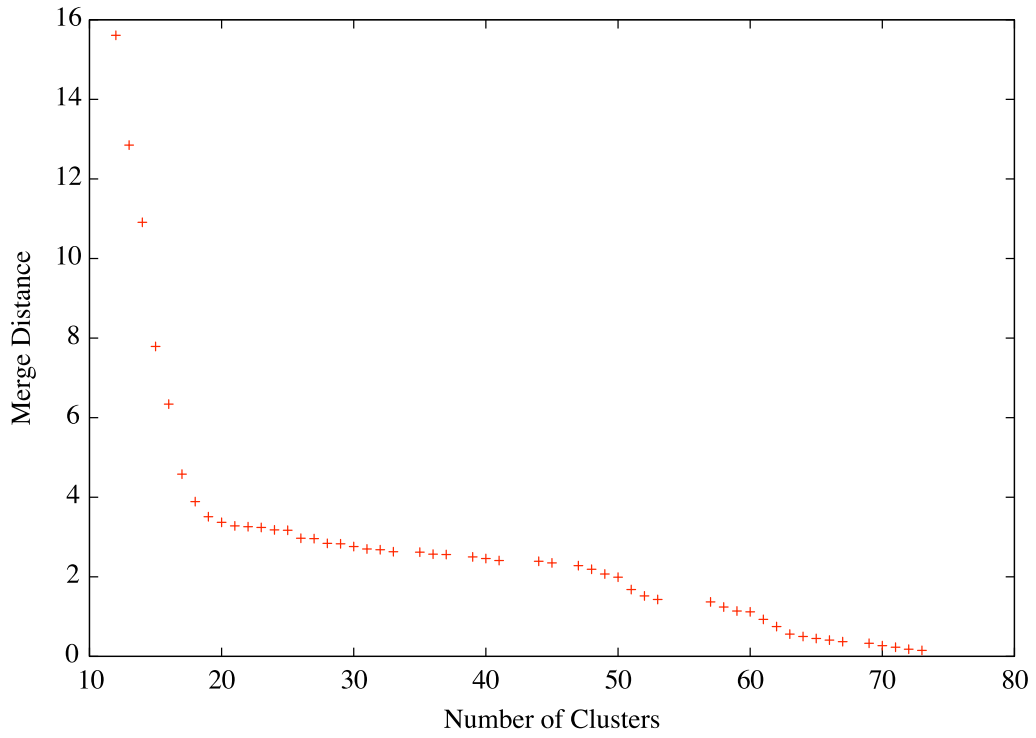


Figure 3.15: Evaluation Graph for Kayak with d_C

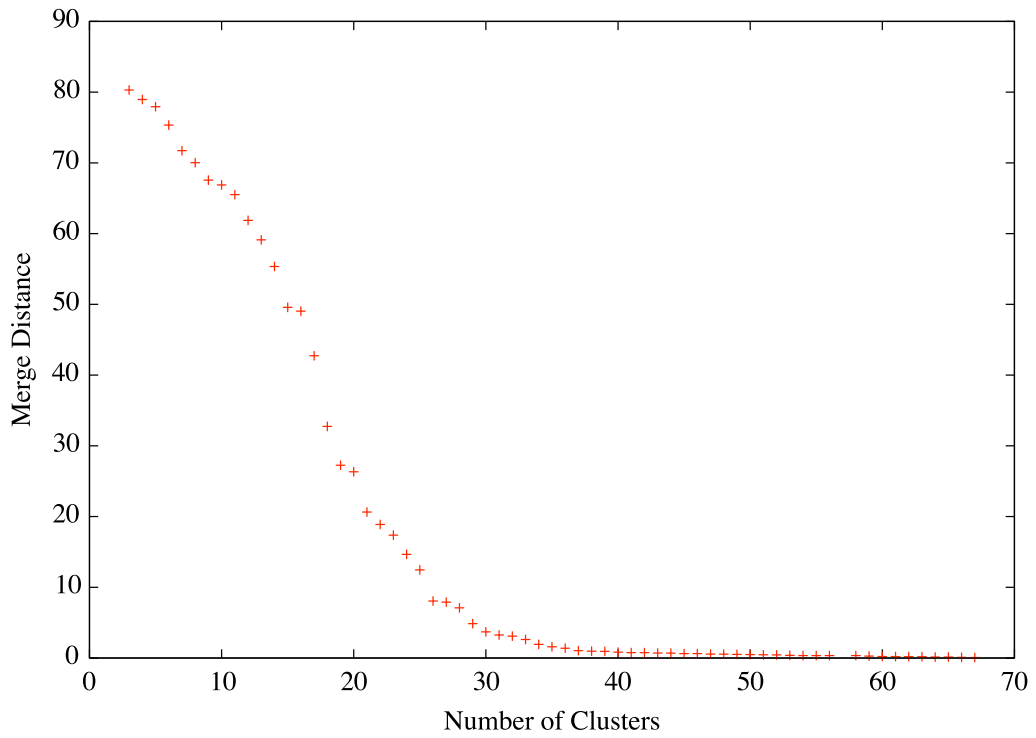


Figure 3.16: Evaluation Graph for Kayak with d_L

Chapter 4

Automated Acceptance Testing of Ajax-based Web Applications

In this Chapter, we discuss *CrawlScripter* which is an application for creating easy-to-understand automated acceptance tests for Ajax-based (JavaScript) web applications [56]. The application provides a high-level scripting language that eases the creation of automated acceptance tests for users that do not have necessary programming expertise. The scripting language also supports assertions, which can be used to verify the presence (or absence) of an element in the DOM structure of a web page, thus enabling the comparison of the actual behaviour of a web application with the expected one.

4.1 The Architecture of CrawlScripter

Figure 4.1 illustrates the architecture of *CrawlScripter*, which consists of three modules: the “*Testing Engine*”, a “*Test Repository*”, and a “*Web Client*”. Each of the modules and their functionalities are explained in the following sections. We also provide a brief description of the technologies used in each of the modules, and give an overview of *Crawljax*, which is a component of the “*Testing Engine*” used for executing test scripts on a target web applications. The initial version of the *Testing Engine* and *Test Repository* modules were based on a design and implementation of Kristofer Mitchell as a part of a project for the CMPUT 402 course at the University of Alberta.

4.1.1 The Web Client

The *Web Client* is implemented using the JSF [57] (Java Server Faces) framework, which allows the building of rich web user interfaces with Ajax-enabled components and offers a desktop-like experience to the user through a web browser. In *CrawlScripter*, the client is responsible for communication with users through the user interface, for database data query and retrieval, and for interaction with the *Testing Engine*. It provides a web-based user interface to create new acceptance tests or to edit existing ones for a selected application, to submit requests to run test scripts, and

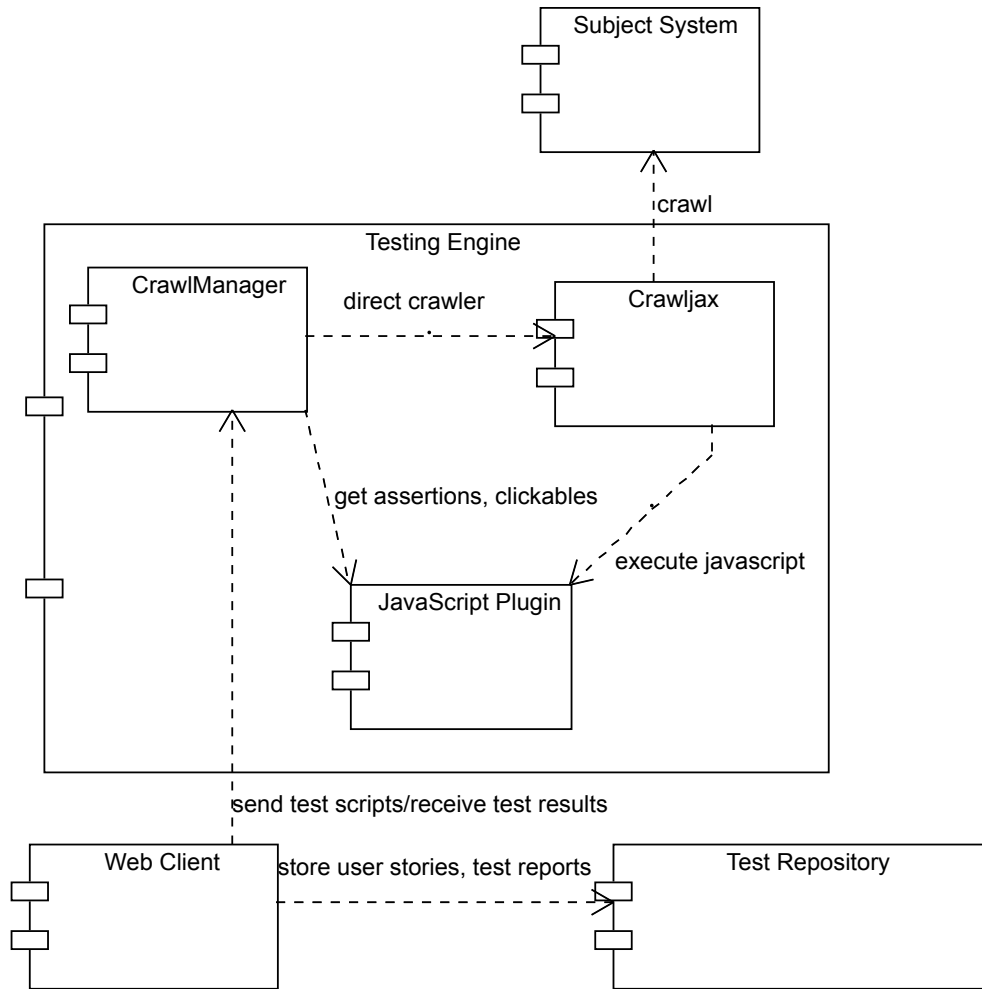


Figure 4.1: Overview of *CrawlScripter* design time and run-time architecture

to review test reports. Figures 4.2 and 4.3 illustrate snapshots of user stories/test scripts and test reports tabs of the *Web Clients*' user interface.

4.1.1.1 Typical usage of the *Web Client*

Users can implement test scripts in *CrawlScripter* for multiple web applications. They can add a new application in *CrawlScripter* by providing its details such as the application URL and application name. New user stories and corresponding acceptance test scripts are created for a selected target application by clicking on the "New User Story" button on the user stories tab. After filling in all the necessary details of a user story such as title, primary actor, goal, and benefit, the user can add test steps for the test script by clicking on the "Add command" button, which adds a new empty line. The user can select instructions (Figure 4.4) from the drop down list and specify the arguments of the instruction in the text box next to the drop down list. Figure 4.5 demonstrates the user interface for a test script creation.

To execute the implemented acceptance test scripts, the user submits a request to run a (number

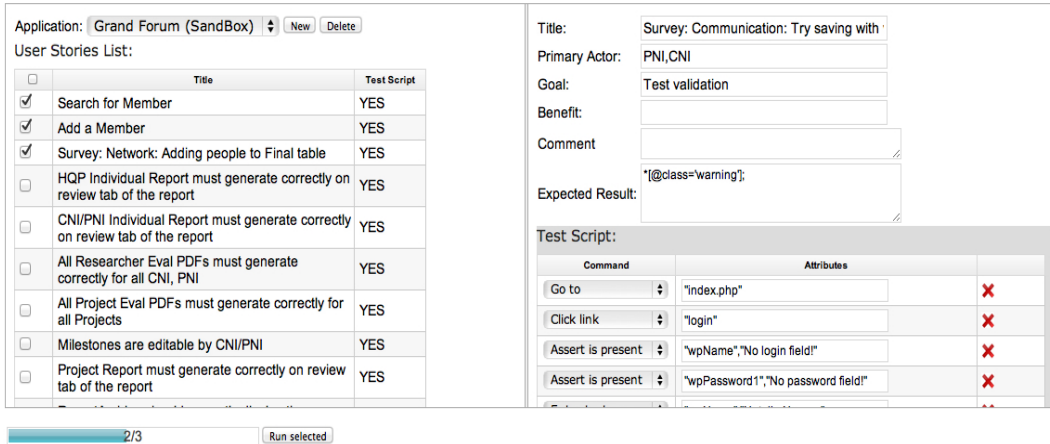


Figure 4.2: Graphic User Interface of the *Web Client*: User Stories and Test Scripts tab

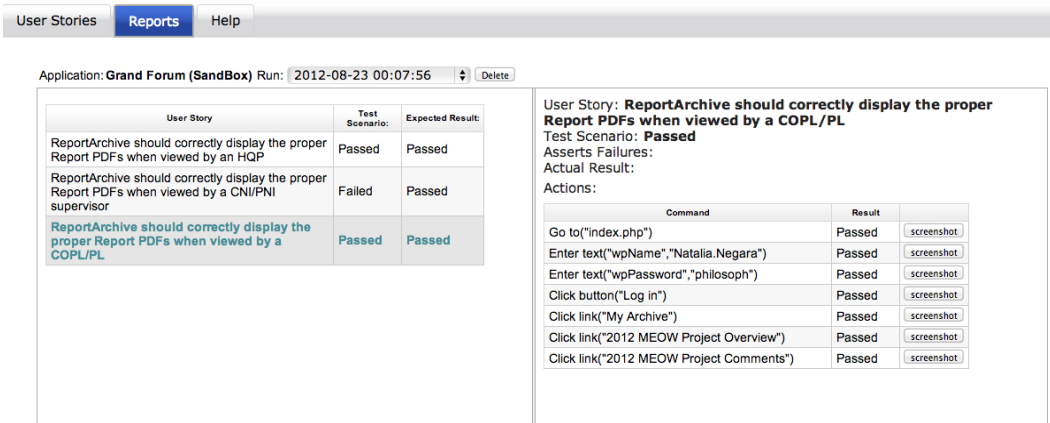


Figure 4.3: Graphic User Interface of the *Web Client*: Test Reports tab

of selected test script(s) by clicking on the “Run selected” button (Figure 4.2). The user is informed about the test execution progress through a progress bar, which displays which test script out of the total number of submitted test scripts is currently being executed. The user can interrupt the test execution process at any time.

The test results can be reviewed under the “Reports” (Figure 4.3) screen in the *Web Client*. Test reports for a specific test run can be accessed by selecting a corresponding timestamp of the test run from the “Test runs” drop-down list. The test reports are designed to provide the user with information about the results of the executed test cases. The the test reports can be useful for making further decisions by project managers or other member of the project’s team. Each test report covers the status (passed or failed) of instructions in a test script. It also gives details about any failed assertions and the actual result of the tested behaviour. Finally, the test reports include the screenshots corresponding to each executed test step, which may provide more information about

the state of the tested web application, beyond what is captured in the state's DOM, when the result was obtained.

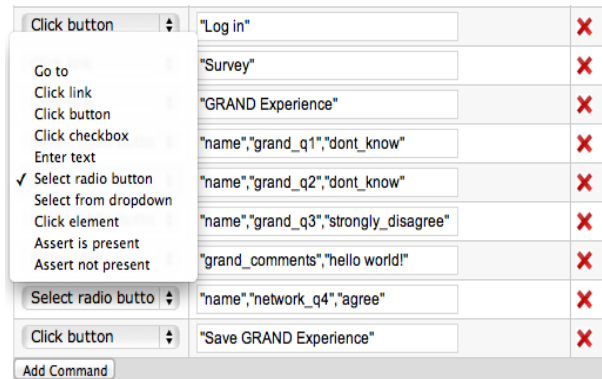


Figure 4.4: List of available instructions in *CrawlScripter*

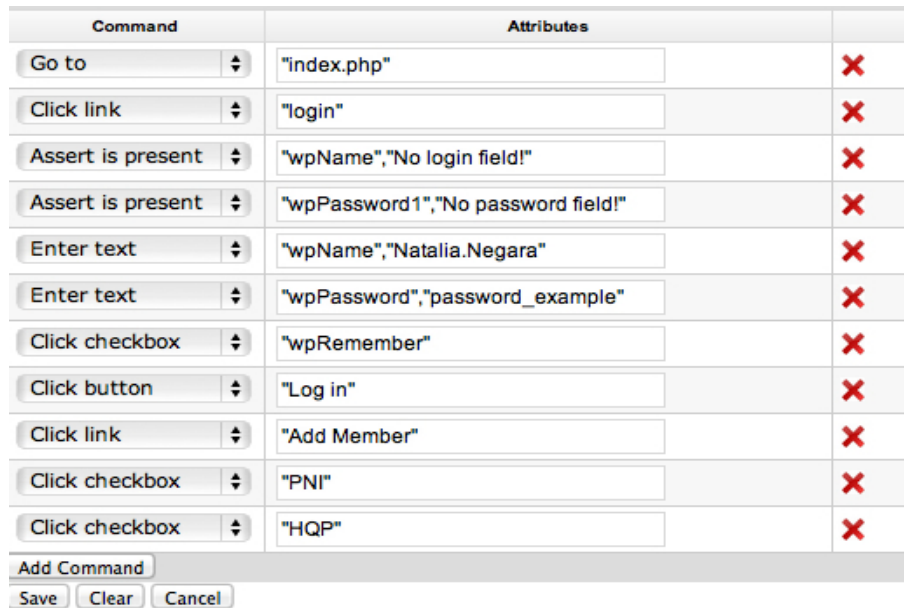


Figure 4.5: Creating test scripts with *CrawlScripter*

4.1.2 The Test Repository

The *Test Repository* is implemented in MySQL and is used for storage and retrieval of test scripts, test reports, and configuration information. The database schema of the application is shown in Figure 4.6.

The interaction of the *Web Client* with the repository for data query and retrieval is supported by the Hibernate [58] library, which provides mappings from Java classes to database tables, and Java types to SQL types. The *CrawlScripter* database includes the following main tables: *user_story*,

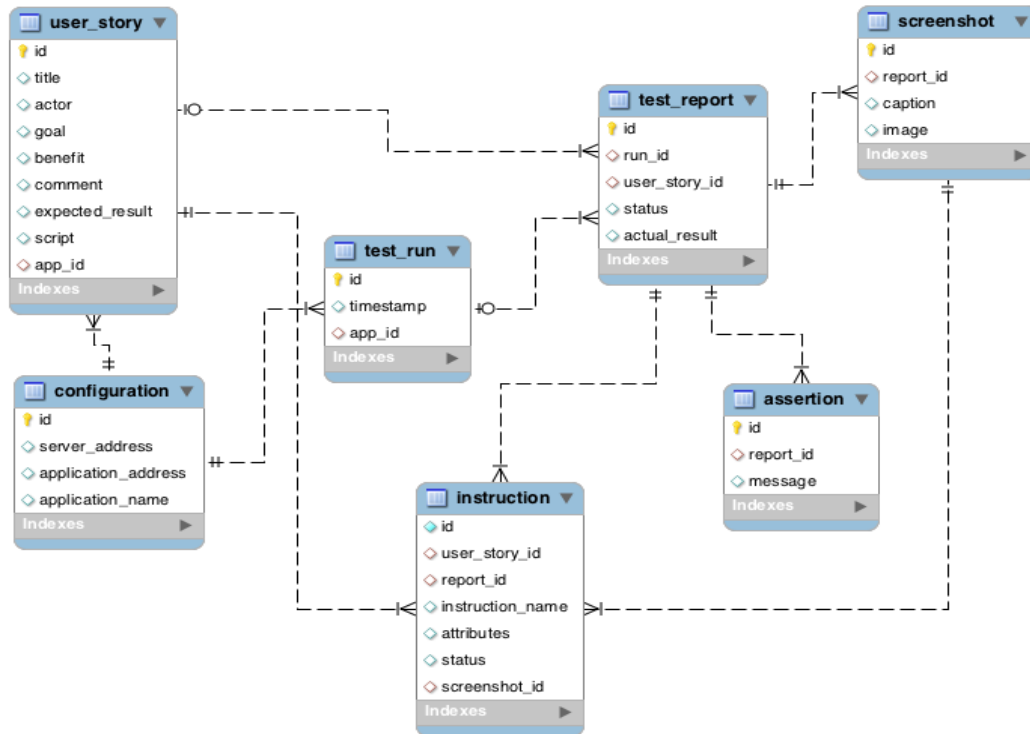


Figure 4.6: Database schema of *CrawlScripter*

configuration, *test_run*, *test_report*, *assertion*, *instruction*, and *screenshot*. The *user_story* table is used to store the details of a user story such as title, goal, actor, benefit, and comments. Additionally, each user story is associated with a web application to be tested, whose details (the application address, and the application name) are stored in the corresponding *configuration* table. The results of test runs are stored in *test_run*, *test_report*, *instruction*, *assertion*, and *screenshot* tables. The *test_run* table stores timestamps of the tests runs. The *test_report* table is used to store the obtained actual result and the overall status (passed or failed) of the test script. Any message of assertions that failed during the test script run are stored in the *assertion* table. The *instruction* table is used to store the instruction name, instruction attributes, instruction’s execution status (passed or failed), and a reference to the associated screenshot. All captured screenshots are stored in the *screenshot* table as a BLOB datatype, which is used to store data in a binary format in a database as a single entity. The *screenshots* table also stores captions of the captured screenshot, which corresponds to the instruction name.

4.1.3 The Testing Engine

The *Testing Engine* is responsible for running test scripts on the pages of the tested JavaScript-enabled web application. The basic components of the *Testing Engine* are the *CrawlManager*, the *JavaScript Plugin*, and *Crawljax*. The *Testing Engine* automatically iterates over each instruction in

the test script and performs the associated action in the web application being tested.

4.1.3.1 Crawljax

Crawljax was developed by Mesbah *et al.* [8] for automatically crawling and testing Ajax-based web applications. *Crawljax* can crawl any Ajax web application by identifying “clickable” elements on a web page rendered at the client’s browser, firing events associated to these clickable elements, and filling the input fields with data. Through *Crawljax*’s API users can configure which elements should be examined as clickable candidates, or which elements should be ignored during the crawling process. These capabilities make *Crawljax* a powerful testing tool. The main components of *Crawljax* are:

- the “Embedded Browser”, which executes JavaScript and supports the technologies required by Ajax;
- the “Robot”, which is responsible for clicks and filling input data on the embedded browser;
- the “Controller”, which detects state changes and updates the State Machine; and
- the “State Machine”, which is responsible for maintaining the state-flow graph and a pointer to the current state.

The “Embedded Browser” is implemented on top of Selenium (WebDriver) APIs [40] and currently supports Internet Explorer, Firefox, and Chrome. Selenium allows to create automated browsing test scripts in a number of popular programming languages such as Java, C#, PHP, Python and Ruby. These test scripts are manually written and require a significant amount of manual effort from users. Using a programmatically remote-controlled instance of a web browser through Selenium WebDriver, *Crawljax* examines Ajax web applications in an automatic manner. Additionally, during the automated crawling process *Crawljax* takes care of properties that are specific to Ajax-based web application and which make browsing of such web application challenging. These properties are [8]:

- client-side execution;
- state changes and navigation within the web application;
- runtime dynamic DOM;
- delta communication style of interaction between client and server;
- internal state changes of clickable elements.

Crawljax creates a *state-flow* graph that models possible navigational paths of a web application as a sequence of states (corresponding to the DOM states of the application user interface) and transitions between them (corresponding to the clickable elements whose behaviour links one state to the

next). The state-flow graph is built incrementally. In the beginning, it is initialized to contain only a single state, which is the DOM state of the starting web page of the web application. *Crawljax* starts to crawl over the application and as a result of DOM-tree changes new DOM states are produced and added to the graph. These changes are caused either by server-side changes sent to the client, or by client-side events. *Crawljax* defines a set of *candidate clickable elements* which fire events of different types, e.g `click`, `mouseover`, `submit`. Each HTML element that corresponds to the labelling requirements (based on the tag name, attributes and their values, if specified) is added by *Crawljax* to the set of candidate clickable elements. For each element in the set, the Robot component fills the input data and fires an event. After the event is fired on a candidate clickable element, *Crawljax* compares the DOM tree as it was before the event against the DOM tree resulting after the event. *Crawljax* supports two ways of DOM trees comparison. One way is based on the Levenshtein [21] distance computation between the compared DOM trees. Another way to compare two DOM trees is to pass them through a chain of *Comparators*, where each one is responsible for comparing specific parts of a DOM tree and passing the output to the next comparator filter. A new DOM state is created and added to the state-flow graph if there are changes detected in a resulting DOM tree. To identify an already discovered DOM state and to eliminate duplicate states, the tool computes hash-codes for each DOM tree and compares every new state with any other state in the state-flow graph. Also, a new edge annotated with the event, whose firing led to the new DOM state, is added to the graph. Finally, the current pointer of the state machine is set to the newly added DOM state from which the crawling process recursively tries to find other possible reachable states. The process is repeated until no new states can be found.

4.1.3.2 Test-scripts Execution

To execute an acceptance-testing procedure, the user submits a request to run a (number of) selected test script(s). The *Web Client* retrieves from the *Test Repository* the test scripts written by the user and sends them to the *Testing Engine* via a REST API.

In addition to the test scripts, the API also specifies a callback URL and the URL of the web application to be tested. On the *Testing Engine*, the test scripts are parsed into a Test Suite object. For each test script in the Test Suite, the *CrawlManager* sets up a *Crawljax* crawler with a *JavaScript Plugin*. During the test execution, when the DOM structure of the web page is changed or the browser moves to a new page, the *JavaScript Plugin*'s `onNewState` or `onUrlLoad` method is called, and the JavaScript code of the current test step is executed as a body of an anonymous function. Clickable elements for the current test step and any assertions that were evaluated are returned to the *CrawlManager*, which then directs *Crawljax* 'actions' by specifying *Crawljax*'s candidate clickable elements for the current web page. If *Crawljax* doesn't find a candidate clickable element, a "Failed" status is returned to *CrawlManager* for the current test step. For every new DOM state (which implies that the candidate clickable element was found and the event was fired) a screenshot

is captured. The test script is executed to its completion, or until an element specified in the instruction is not found and *Crawljax* cannot proceed. The *CrawlManager* sends the results of the test scripts' execution along with the screenshots back to the *CrawlScripter* client using the provided callback URL. The client saves the test execution results in the *Test Repository*.

4.1.4 Implementation Challenges

During the implementation of *CrawlScripter* we run into a number of technical challenges. At first, the use of *Crawljax* in our approach seemed to be straightforward. However, since *Crawljax* crawls the application by clicking on the clickable elements in an ad-hoc order, the hardest part in the implementation consisted in guiding *Crawljax* actions via instructions specified in the test scripts (click on the specific element on the current web page). In our approach test instructions are converted into JavaScript functions, which allow it to execute assertions, to push elements to be clicked provided by users into the list of candidate clickable elements, and to set up values in the input fields on the page. The solution to the problem of guiding *Crawljax* was to develop the *JavaScript* plugin that would be able to execute the JavaScript part of the test script whenever the DOM state of the page is changed or a new URL is loaded, and the *CrawlManager* that will be able to direct *Crawljax* on each web page by specifying clickable elements for that web page.

4.2 The *CrawlScripter* Scripting Language

The scripting language of *CrawlScripter* consists of a library of high-level instructions (input by the user) that can be used to write test scripts that would be executed by the crawler on the web application to be tested. These instructions are close to natural language, therefore users of *CrawlScripter* are not required to have specific programming knowledge to write automated acceptance tests. The grammar of *CrawlScripter*'s scripting language is shown in 4.7.

CrawlScripter supports three types of instructions: firing events, specifying inputs, and evaluating assertions. Most of the actions performed by users in web applications can be represented as web commands that consist of a single verb that corresponds to the action performed, and one or two nouns (arguments upon which the action will be performed) [59]. Each instruction in *CrawlScripter* begins with the verb phrase that expresses the action to be performed, followed by the nouns or, in case of an assertion's failure message, a phrase. Event instructions represent *Crawljax*'s *click* event types on the elements of the web page and as an argument require specifying an element's label to be clicked on (for example, `Click link ``Search```). Input-specification instructions require as arguments the name (as an attribute in DOM structure) of an input field element and a value to be filled in the field (for example, `Enter text ``searchField``, ``dog```). Assertions are used to verify the presence or absence of a DOM element on a web page. Assertion instructions take as arguments the element's identifier (or name) to be checked for presence (or absence), which is followed by the failure message (for example, `Assert is present ``searchField``,`

```

< Instruction >
→< Event > | < Input_spec > | < Assertion >
< Event >→ Click < Element >< Label >
< Input_spec >→< Input_type >< Label >< Value >
< Assertion >
→< Assert_type >< Label >< Failure_message >
< Tag >→< Tag_name >< Attribute >< Value >
< Element >→ link | button | checkbox | < Tag >
< Input_type >→ enter text | select from drop – down
< Assert.Type >→ present | not present
< Label >→ "String"
< Tag_name >→ "String"
< Tag >→ "String"
< Attribute >→ "String"
< Value >→ "String"
< Failure_message >→ "String"

```

Figure 4.7: The *CrawlScripter* Language

``The search field is not present''). In case of the assertion failure the provided message is displayed to the user.

CrawlScripter parses the test scripts and extracts from each instruction the type of the action to be performed on the web page and its arguments. This information is then compiled into JavaScript functions.

4.3 Evaluation

We have conducted two studies to empirically assess the applicability of our method in supporting acceptance testing and translation of functional requirements of dynamic web applications into executable test scripts. We evaluated the ease of use of *CrawlScripter* in creating test scripts and also tried to find if *CrawlScripter* can be applied in a variety of test scenarios. Also, we analyzed the extent to which we can maintain the implemented acceptance test scripts during the software evolution being tested. From these studies we discovered opportunities for future extensions of *CrawlScripter*.

4.3.1 JPetStore

The subject of the first study is the Java Pet Store (JPetStore) [47] sample Ajax-enabled application. The study consisted of translating user stories for JPetStore into executable automated acceptance tests and running them on the target web application.

JPetStore is a well-known educational application designed to demonstrate the use of Java Enterprise Edition Platform in developing “Ajaxified” web applications. JPetStore is an internet pet store where users can perform different actions typical to most e-stores. Users can browse the catalog of products, search for a specific item in the catalog, add a new items for sale, search for items using their location or tags, rate items, or flag an item as an appropriate. The JPetStore application consists

of a single page where (based on user actions) different areas of the page are dynamically updated using Ajax technology.

We constructed the following user stories (and corresponding test scripts) for JPetStore from the functionalities described in the “About” section which is shipped together with the JPetStore application:

- Browsing the Catalog
- Searching the Catalog (by keyword)
- Rate an Item
- Flag as an appropriate
- Add new Item for Sale
- Reviewing an Order
- Search Item by Location
- Search Item by Tag

The above 8 user stories were implemented as *CrawlScripter* acceptance test scripts, and were successfully executed over the JPetStore web application. Figures 4.8 and 4.9 illustrate snapshots for an acceptance test script of a JPetStore “Search item (by keyword)” user story and its corresponding test report as shown to users in the *CrawlScripter*. For brevity and presentation purposes we show only the test script and its test report for one user story created for JPetStore web application.

Test Script:		
Command	Attributes	
Go to	"index.jsp"	✘
Click link	"Search"	✘
Enter text	"searchForm:searchString","dog"	✘
Click checkbox	"searchForm:searchTags"	✘
Click button	"Submit"	✘
Assert is present	"resultsForm:mapSubmit","Map checked"	✘

Add Command

Save Clear Cancel

Figure 4.8: JPetStore: “Search item (by keyword)” test script

User Story: **Search the Catalog**
Test Scenario: **Passed**
Asserts Failures:
Actual Result:

Actions:

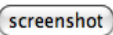
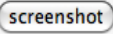
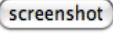
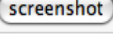
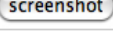
Command	Result	
Go to("index.jsp")	Passed	
Click link("Search")	Passed	
Enter text("searchForm:searchString","dog")	Passed	
Click checkbox("searchForm:searchTags")	Passed	
Click button("Submit")	Passed	

Figure 4.9: JPetStore: “Search item (by keyword)” test report

4.3.2 GRAND Forum

The subject of our second study is a JavaScript-enabled development version of the GRAND Forum [60], which is a platform for the GRAND NCE [61] community, where its users can participate in events of interest, and in different forum activities such as collaborating and reporting. The forum also provides tools for analyzing the evolution of the community network.

4.3.2.1 Evaluation Methodology

CrawlScripter was deployed on one of our lab’s servers and was available to be accessed through any web browser¹. We introduced *CrawlScripter* to one of the developers of GRAND forum, who is also a member of our research group. We gave him a one-hour training session, providing details on how to use *CrawlScripter* to implement test scripts and to execute them over the target application. The instructions described the following:

- How to access *CrawlScripter* through the browser.
- How to create a new user story and the corresponding test script.
- How to add new commands to the test script.
- Types of available commands (events, input, and assertions) and the list of available commands for each type.
- Required arguments for each command, and examples for each of them.

¹129.128.184.112:8080/CrawlScripter

- How to obtain element's attributes (such as *id* or *name* and their values from the web page to include as arguments of a test command.
- Demonstration of two test script examples for user stories: "Add a member", "Search for a member".
- How to save a test script.
- How to submit single/several test scripts for test execution.
- How to review and understand the produced test reports.
- How to view results for previous test runs.

The participant used the tool every day for one hour for the period of two weeks to create acceptance test scripts for GRAND forum user stories, which were written by other member of the GRAND forum development team. The participant independently created the test scripts and submitted them for execution. Once the test reports were obtained, the participant reviewed them to verify the results. While the participant was using *CrawlScripter*, the author was observing the participant's actions to ensure proper functionality of the system.

4.3.2.2 Evaluation Results

Using *CrawlScripter*'s library, the developer successfully implemented 23 acceptance test scripts (out of 23 user stories) which simulated various usage scenarios over GRAND Forum. The GRAND Forum user stories that were implemented by the participant in automated acceptance test scripts are listed in Appendix A. The covered scenarios are of different complexity for a variety of important functionalities of the collaborative platform such as reporting, notifications, activities, and surveys. The implemented acceptance test scripts have been successfully executed with *CrawlScripter* on the GRAND Forum application. The participant also had the opportunity to review the produced test reports.

The developer was intrigued with the ability of *CrawlScripter* to test different parts and scenarios of a dynamic web application by creating easy-to-understand automated test scripts that are close to natural language. He reported that "*the use of the tool was intuitive for a first time user, and in general it was fairly easy to apply the CrawlScripter for writing automated acceptance test scripts*". He also reported that "*the translation of a user story into a single executable acceptance test script didn't require as much time and effort as he has expected*". Figures 4.10 and 4.11 illustrate snapshots for a test script of a GRAND Forum user story created by the GRAND Forum developer and the corresponding test report produced by *CrawlScripter*.

The implemented test scripts exercised a variety of clickable elements such as buttons, links, radio buttons, drop down lists, and checkboxes. The GRAND Forum user stories included complex scenarios such as generation of reports, filling surveys, or receiving notifications, which involved

Application: Grand Forum (SandBox) [New] [Delete]		Test Script:	
User Stories List:		Command	Attributes
<input type="checkbox"/>	Title	Go to	"index.php" X
<input type="checkbox"/>	Search for Member	Enter text	"wpName","Natalia.Negara" X
<input type="checkbox"/>	Add a Member	Enter text	"wpPassword","philosoph" X
<input type="checkbox"/>	Survey: Network: Adding people to Final table	Click button	"Log in" X
<input type="checkbox"/>	HQP Individual Report must generate correctly on review tab of the report	Click link	"Survey" X
<input type="checkbox"/>	CNI/PNI Individual Report must generate correctly on review tab of the report	Click link	"Communication" X
<input type="checkbox"/>	All Researcher Eval PDFs must generate correctly for all CNI, PNI	Click checkbox	"class","all_person_checkbox" X
<input type="checkbox"/>	All Project Eval PDFs must generate correctly for all Projects	Click checkbox	"id","all_email_checkbox" X
<input type="checkbox"/>	Milestones are editable by CNI/PNI	Click checkbox	"id","all_forum_checkbox" X
<input type="checkbox"/>	Project Report must generate correctly on review tab of the report	Select radio butto	"name","Wolfgang_Heidrich_inperson" X
		Select radio butto	"name","Wolfgang_Heidrich_email" X
		Select radio butto	"name","Wolfgang_Heidrich_forum" X

Figure 4.10: An acceptance test script for GRAND Forum web application

excessive JavaScript calls and dynamic updates of the DOM structure of GRAND Forum’s web pages. The implemented test scripts varied in the number of steps. The test scripts were evaluated by analyzing the application’s output using the specified by the user assertions. During the execution of the test scripts the participant discovered a few issues that were not noticed during the earlier testing of the GRAND Forum application during development. None of the issues which were discovered were attributed to *CrawlScripter*. One issue was related to a typo in a name of a link, another issue was triggered by a MySQL exception on a database insert.

Currently, *CrawlScripter* does not support the cross-referencing of test scripts, and the GRAND Forum developer found it cumbersome to write the same test steps for the “Login to GRAND Forum” functionality in all 23 test scripts. He suggested to extend *CrawlScripter* with the ability to reference tests steps for repetitive functionalities with already implemented test scripts rather than have to create the test steps again from scratch (e.g. test steps in other test scripts related to the Login functionality can be replaced with test script already implemented to test the Login functionality). Additionally, he suggested to enhance the *CrawlScripter* with the functionality that will allow to export test reports to a file. Another reported limitation of *CrawlScripter* was related to the current version of the tool’s user interface, which was the inability to reorder test steps in a test script.

Overall, the developer had a positive experience with *CrawlScripter* in performing acceptance testing of the GRAND Forum application. He also reported that he “*would definitely continue using it, when user interface is polished up*”.

4.3.3 GRAND Forum: Test-scripts migration

Software systems constantly evolve due to various reasons, and particularly due to the modification of functionalities. In this case, the acceptance tests, if not being maintained, could either produce

User Story: **Survey: Communication: Try saving with validation fail2**
 Test Scenario: **Passed**
 Asserts Failures:
 Actual Result:
 Actions:











Command	Result	
Go to("index.php")	Passed	
Enter text("wpName","Natalia.Negara")	Passed	
Enter text("wpPassword","philosoph")	Passed	
Click button("Log in")	Passed	
Click link("Survey")	Passed	
Click link("Communication")	Passed	
Click checkbox("class","all_person_checkbox")	Passed	
Click checkbox("id","all_email_checkbox")	Passed	
Click checkbox("id","all_forum_checkbox")	Passed	
Select radio	Passed	

Figure 4.11: A test report for GRAND Forum

incorrect results or become non-executable and unusable. We were interested in analyzing the impact of changes at the user-interface level over implemented test scripts and to what extent we can maintain these test scripts which are implemented with *CrawlScripter* during software evolution.

Our target application, GRAND Forum was also subjected to evolution since a number of user-interface modifications related to the *Reporting* functionality were applied to the system due to the changes in the requirements. The inspection of the modifications and implemented user stories demonstrated that 8 out of 9 user stories related to the *Reporting* functionality were affected by these modifications and clearly failed during the repeated execution of these test scripts on GRAND Forum. The acceptance test for the user story “ReportArchive should correctly display the proper Report PDFs when viewed by a CNI/PNI supervisor” wasn’t affected by the changes made on the GRAND Forum, and was successfully executed without any corrections.

Six of the nine test scripts have suffered from changes applied to the labels of clickable elements. For example, only the renaming of the button “*Generate report*” in “*Generate report PDF*” affected three test scripts. In addition to this modification, one test script also had a test step that involved clicking on a button “*Download report as PDF*” which was renamed to “*NI Report PDF*”. Another example of the changes in the labels of clickable elements that affected two steps of the user story titled as “Project Report must generate correctly on review tab of the report” was the renaming of two buttons (on the project’s “Review & Submit” page) “*Download Overview as PDF*” and “*Download Comments as PDF*” to “*Project Report PDF*” and “*Project Comments PDF*” respectively.

After the corrections were applied to the test steps, the corresponding test scripts were successfully executed on GRAND Forum.

The two test scripts related to “Reporting” functionality were not evaluated, since at the moment of writing this thesis and conducting the case studies, the changes to the functionality that

corresponds to their user stories were not applied on GRAND Forum.

4.3.4 Conclusions

To evaluate *CrawlScripter*, we implemented test scripts corresponding to the user stories of pedagogical and real-world web applications (JPetStore and GRAND Forum). In the case of GRAND Forum, the evaluation of *CrawlScripter* was performed by a participant who carried out the acceptance testing of the subject web application. The conducted case studies demonstrated the ease of use of *CrawlScripter* in creating automating acceptance tests. The implemented tests scripts were successfully executed in both case studies. Also, the evaluation of the test script migration demonstrated that updates necessary to the test scripts in order to maintain their functionality were all fairly simple. The only changes necessary were updating the clickable labels whose names had changed in the web application, and that no logic changes were necessary.

Chapter 5

Conclusions and Future Work

In this work, we proposed solutions for two problems in the area of maintaining web applications, namely reverse engineering and testing of dynamic web applications. To better comprehend Ajax-based web applications we developed a method for automatically extracting features from such web applications. Also, we suggested an approach for performing automated acceptance testing of JavaScript web applications.

The proposed method for feature extraction consists of collecting DOM trees of a web application and clustering these DOM trees into groups, each one corresponding to a distinct feature. To group DOM trees we apply a hierarchical clustering algorithm with a novel composite-tree-edit distance metric as a similarity measure. In the context of this research activity, we made two main contributions. The first contribution is the introduction of a distance metric that makes a distinction between composite and simple changes in the tree structure of a DOM state, and thus reduces the impact of composite changes on the computation of structural similarity. The second contribution is the automatization of the identification of feature cluster partition, which is accomplished by using the proposed similarity measure as a distance metric within the hierarchical clustering algorithm. We apply clustering-evaluation techniques, namely the L method [22] and the Silhouette coefficient [23], to automatically deduce from a resulted dendrogram the number of clusters in a partition that is close to the actual number of features in a web application.

We evaluated the new distance metric on three different Ajax-based real-world web applications. The results demonstrated that by using the proposed distance metric we could obtain higher accuracy of clustering in comparison to related approaches that used string differencing to assess the web page similarity. Also, the proposed distance metric produced in general better results than the simplified tree-edit distance metric that didn't consider composite changes to assess web page similarity.

Additionally, we evaluated the ability of all three examined distance metrics to automatically determine a number of clusters that correspond to the actual number of features using the L method and the Silhouette coefficient clustering evaluation techniques. The results of the evaluation over the three web applications demonstrated that the proposed distance metric produces a number of clusters that more closely corresponds to the actual number of features, thereby showing that the process of

features extraction can be fully automated when employing a hierarchical clustering algorithm. On the other hand, the evaluation results for the other two distance metrics did not work well with the L method and the Silhouette coefficient, producing lower accuracy results, which demonstrate that they are not suitable for automatically inferring the cut-off threshold for a resulting dendrogram.

In the second part of our thesis we proposed an approach for supporting automated acceptance testing of JavaScript web applications. The approach allows the specification of the intended behaviours of a target web application and to compare it with the actual behaviour, relying on Crawljax crawling tool. We implemented the approach in *CrawlScripter*, a tool that provides a library of high-level instruction for creating automated acceptance test scripts and to execute them on a web application using Crawljax. The scripting language of *CrawlScripter* includes assertions that allow the verification of the absence or presence of elements on a web page.

We evaluated the ease of use of *CrawlScripter* and its capability in creating acceptance test scripts for different test scenarios on pedagogical and real-world JavaScript web applications. In the second case study (real-world application) the tool was used by the developer of a complex web application, under development in our lab. During the evaluation, automated acceptance test scripts were implemented for different scenarios, which were then successfully executed on subject web applications.

As future work for the feature extraction we would like to automatically derive labels for the produced clusters that could also serve as names for the discovered features by applying the Latent Dirichlet allocation (LDA) [62] technique on the content of the web pages included in the clusters. Also, as a future idea we would also like to design and implement a tool that will allow us to visualize and browse the behavioural model of a web application constructed from the extracted features. The tool will provide a better comprehension of the application's behavioural model as well as as will visualize the available features in the web application and interactions which happen between them.

Future work for *CrawlScripter* will include more case studies using more real-world JavaScript web applications to evaluate usefulness, applicability, and scalability of the proposed approach. Other ideas for future work are related to the enhancement of *CrawlScripter*, as follows.

- We would like to extend the tool by introducing the ability to reference already implemented user stories which will allow to reuse the test scripts (for example, Login functionality), and also will make the test scripts shorter.
- Another possible future extension of *CrawlScripter* could be the integration of the tool with the Firefox browser, which will allow the user to add elements to the test script by simple clicking on them on the web page.
- Finally, *CrawlScripter* can be extended by providing the functionality of validating web pages style by comparing it with the provided expected standards. This can be achieved by examining the DOM structure of the web page, which we can access with the help of Crawljax.

Bibliography

- [1] Internet growth statistics. [Online]. Available: <http://www.internetworldstats.com/emarketing.htm>
- [2] L. Kats, R. G. Vogelij, K. T. Kalleberg, and E. Visser, "Software development environments on the web: A research agenda," in *Proceedings of the 11th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software (Onward 2012)*. ACM Press, 2012.
- [3] Y. Deshpande, S. Murugesan, A. Ginige, S. Hansen, D. Schwabe, M. Gaedke, and B. White, "Web engineering," *Journal of Web Engineering*, vol. 1, no. 1, pp. 3–17, Oct. 2002.
- [4] G. Costagliola, F. Ferrucci, and R. Francese, "Web Engineering: Models and Methodologies for the Design of Hypermedia Applications," in *Handbook of Software Engineering and Knowledge Engineering, volume 2, Emerging Technologies, pages 181–199*. World Scientific. World Scientific Pub. Co, 2002, pp. 181–200.
- [5] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," *Journal of Software Maintenance and Evolution: Research and Practice*, 2012.
- [6] Y. Zou and M. Hung, "An approach for Extracting Workflows from E-Commerce Applications," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ser. ICPC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 127–136.
- [7] Q. Zhang, R. Chen, and Y. Zou, "Reengineering User Interfaces of E-Commerce Applications Using Business Processes," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, ser. ICSM '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 428–437.
- [8] A. Mesbah, A. van Deursen, and S. Lenseslink, "Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes," *ACM Transactions on the Web*, vol. 6, no. 1, pp. 3:1–3:30, 2012.
- [9] F. Ricca and P. Tonella, "Using Clustering to Support the Migration from Static to Dynamic Web Pages," in *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, ser. IWPC '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 207–216.
- [10] A. De Lucia, R. Francese, G. Scanniello, and G. Tortora, "Reengineering Web Applications Based on Cloned Pattern Analysis," in *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, ser. IWPC '04. Los Alamitos, CA, USA: IEEE Computer Society, 2004, pp. 132–141.
- [11] G. A. Di Lucca, M. Di Penta, and A. R. Fasolino, "An Approach to Identify Duplicated Web Pages," in *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, ser. COMPSAC '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 481–486.
- [12] A. De Lucia, G. Scanniello, and G. Tortora, "Using a Competitive Clustering Algorithm to Comprehend Web Applications," in *Proceedings of the Eighth IEEE International Symposium on Web Site Evolution*, ser. WSE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 33–40.
- [13] A. De Lucia, M. Risi, G. Tortora, and G. Scanniello, "Towards automatic clustering of similar pages in web applications," in *Proceedings of the 11th IEEE International Symposium on Web Systems Evolution*, ser. WSE '09. IEEE Computer Society, 2009, pp. 99–108.

- [14] G. A. Di Lucca and A. R. Fasolino, "Testing Web-based applications: The state of the art and future trends," *Information and Software Technology*, vol. 48, no. 12, pp. 1172–1186, Dec. 2006.
- [15] M. Jazayeri, "Some Trends in Web Application Development," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 199–213.
- [16] A. Marchetto, P. Tonella, and F. Ricca, "State-Based Testing of Ajax Web Applications," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ser. ICST '08, Washington, DC, USA, 2008, pp. 121–130.
- [17] A. Marchetto and P. Tonella, "Using search-based algorithms for Ajax event sequence generation during testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 103–140, 2011.
- [18] R. Mugridge and W. Cunningham, *Fit for Developing Software: Framework for Integrated Tests*. Prentice Hall PTR, 2005.
- [19] G. Leshed, E. M. Haber, T. Matthews, and T. Lau, "CoScripter: automating & sharing how-to knowledge in the enterprise," in *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, 2008, pp. 1719–1728.
- [20] J. Mahmud and T. Lau, "Lowering the barriers to website testing with CoTester," in *Proceedings of the 15th international conference on Intelligent user interfaces*, New York, NY, USA, 2010, pp. 169–178.
- [21] V. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, 1966.
- [22] S. Salvador and P. Chan, "Determining the Number of Clusters/Segments in Hierarchical Clustering/Segmentation Algorithms," in *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence*, ser. ICTAI '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 576–584.
- [23] P. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis," *Journal of Computational and Applied Mathematics*, vol. 20, no. 1, pp. 53–65, Nov. 1987.
- [24] A. Mesbah and A. van Deursen, "Migrating Multi-page Web Applications to Single-page AJAX interfaces," in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, ser. CSMR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 181–190.
- [25] G. A. Di Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. De Carlini, "Comprehending Web Applications by a Clustering Based Approach," in *Proceedings of the 10th International Workshop on Program Comprehension*, ser. IWPC '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 261–.
- [26] A. E. Hassan and R. C. Holt, "Towards a Better Understanding of Web Applications," *Web Site Evolution, IEEE International Workshop on*, vol. 0, p. 112, 2001.
- [27] F. Ricca and P. Tonella, "Understanding and Restructuring Web Sites with ReWeb," *IEEE MultiMedia*, vol. 8, no. 2, pp. 40–51, Apr. 2001.
- [28] A. De Lucia, M. Risi, G. Scanniello, and G. Tortora, "Comparing clustering algorithms for the identification of similar pages in web applications," in *Proceedings of the 7th international conference on Web engineering*, ser. ICWE'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 415–420.
- [29] D. Harman, "Information retrieval." Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992, ch. Ranking algorithms, pp. 363–392.
- [30] S. T. Dumais, "Latent Semantic Indexing (LSI): Trec-3 report," in *Overview of the Third Text REtrieval Conference*, 1995, pp. 219–230.
- [31] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM Computing Surveys*, vol. 31, no. 3, pp. 264–323, Sep. 1999.

- [32] R. Mikhael and E. Stroulia, "Accurate and Efficient HTML Differencing," in *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*, ser. STEP '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 163–172.
- [33] N. Alshahwan and M. Harman, "Automated web application testing using search based software engineering," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 3–12.
- [34] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip, "A framework for automated testing of javascript web applications," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. ACM, 2011, pp. 571–580.
- [35] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A Symbolic Execution Framework for JavaScript," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 513–528.
- [36] J. P. Sauvé, O. L. Abath Neto, and W. Cirne, "EasyAccept: a tool to easily create, run and drive development with automated acceptance tests," in *Proceedings of the 2006 international workshop on Automation of software test*, 2006, pp. 111–117.
- [37] B. C. Araújo, A. C. Rocha, A. Xavier, A. I. Muniz, and F. P. Garcia, "Web-based tool for automatic acceptance test execution and scripting for programmers and customers," in *Proceedings of the 2007 Euro American conference on Telematics and information systems*, 2007, pp. 56:1–56:4.
- [38] V. Dallmeier, M. Burger, T. Orth, and A. Zeller, "WebMate: a tool for testing web 2.0 applications," in *Proceedings of the Workshop on JavaScript Tools*, ser. JSTools '12. ACM, 2012, pp. 11–15.
- [39] A. Cypher, C. Drews, E. Haber, E. Kandogan, J. Lin, T. Lau, G. Leshed, T. Matthews, and E. Wilcox, *No Code required: giving users tools to transform the web*. Burlington, MA: Morgan Kaufmann, 2010, ch. Collaborative Scripting for the Web., pp. 85–104.
- [40] (2004) Selenium project home page. [Online]. Available: <http://seleniumhq.org/>
- [41] K. Beck, E. Gamma, D. Saff, and M. Clark. JUnit home page. [Online]. Available: <http://www.junit.org/>
- [42] A. Helleøy. Cucumber home page. [Online]. Available: <http://cukes.info/>
- [43] P. Klärck and J. Härkönen. (2005) Robot Framework home page. [Online]. Available: <http://code.google.com/p/robotframework/>
- [44] N. Negara, N. Tsantalis, and E. Stroulia, "Feature detection in ajax-enabled web applications," in *Accepted to 17th European Conference on Software Maintenance and Reengineering*. IEEE.
- [45] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM Journal on Computing*, vol. 18, no. 6, pp. 1245–1262, Dec. 1989.
- [46] M. Fokaefs, R. Mikhael, N. Tsantalis, E. Stroulia, and A. Lau, "An Empirical Study on Web Service Evolution," in *Proceedings of the 2011 IEEE International Conference on Web Services*, ser. ICWS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 49–56.
- [47] Sun Microsystems. (2007) Java Pet Store Reference Application 2.0. [Online]. Available: <http://java.sun.com/developer/technicalArticles/J2EE/petstore/>
- [48] E. W. Forgy, "Cluster analysis of multivariate data: efficiency vs interpretability of classifications," *Biometrics*, vol. 21, pp. 768–769, 1965.
- [49] J. B. MacQueen, "Some Methods for Classification and Analysis of MultiVariate Observations," in *Proceedings of the fifth Berkeley Symposium on Mathematical Statistics and Probability*. University of California Press, 1967, pp. 281–297.
- [50] S. Dudoit and J. Fridlyand, "A prediction-based resampling method for estimating the number of clusters in a dataset." *Genome biology*, vol. 3, no. 7, Jun. 2002.

- [51] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining, (First Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.
- [52] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.
- [53] S. A. Elavarasi, J. Akilandeswari, and B. Sathiyabhama, “A survey on partition clustering algorithms,” *International Journal of Enterprise Computing and Business Systems*, vol. 1, no. 1, pp. 1–14, 2011.
- [54] H.-P. Kriegel, P. Kröger, J. Sander, and A. Zimek, “Density-based clustering,” *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 1, no. 3, pp. 231–240, 2011.
- [55] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise,” in *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining*, 1996, pp. 226–231.
- [56] N. Negara and E. Stroulia, “Automated acceptance testing of javascript web applications,” in *Proceedings of the 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 318–322.
- [57] (2010) The project page for JavaServer Faces. [Online]. Available: <http://www.java-serverfaces.org/>
- [58] Red Hat. Hibernate official website. [Online]. Available: <http://www.hibernate.org/>
- [59] G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan, “Koala: capture, share, automate, personalize business processes on the web,” in *Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, 2007, pp. 943–946.
- [60] Development version of the GRAND forum. [Online]. Available: http://grand.cs.ualberta.ca/~dgolovan/grand/_forum/_giga/index.php
- [61] GRAND Network of Centres of Excellence. Official website of GRAND. [Online]. Available: <http://grand-nce.ca/>
- [62] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent dirichlet allocation,” *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, Mar. 2003.

Appendix A

A.1 Grand Forum User Stories

1. All Researcher Eval PDFs must generate correctly for all CNI, PNI
2. All Project Eval PDFs must generate correctly for all Projects
3. HQP Individual Report must generate correctly on review tab of the report
4. CNI/PNI Individual Report must generate correctly on review tab of the report
5. Project Report must generate correctly on review tab of the report
6. ReportArchive should correctly display the proper Report PDFs when viewed by an HQP
7. ReportArchive should correctly display the proper Report PDFs when viewed by a CNI/PNI supervisor
8. ReportArchive should correctly display the proper Report PDFs when viewed by a COPL/PL
9. Milestones are editable by CNI/PNI
10. Milestones create Notifications to everybody involved, and to the COPL/PL
11. Milestones create Notifications to NI involved
12. Editing own Profile, but not other people
13. Trying to edit other person's profile
14. Survey: can proceed with consent
15. Survey: Complete About section
16. Survey: Relationships: Save all friends
17. Survey: Communication: Try saving with validation fail
18. Survey: Communication: Try saving with validation fail2

19. Survey: Projects: Test validation
20. Survey: GRAND: Validation Fail
21. Survey: Network: Adding people to Final table
22. Adding Activity
23. Check Dashboard values