# Functional Testing of Web Applications

Stefan Gamerith

*Linzerstrasse 429 4215,*
*1140 Wien*
*Student ID: 0925081*

February 2, 2014

**Abstract**

Web applications are more and more replacing traditional Desktop applications, making Web engineering one of the most important disciplines in the IT industry. Thus functional tests, verifying the functioning of an application, are crucial to the success of a Web project. This paper first gives an introduction to Software testing in general and outlines two different Web Test Automation concepts in particular. Although there exist some distinct approaches to overcome the difficulties of regular changes in Web applications, non of them have been applied in context of functional Web application testing.

# Contents

# Introduction

Since the first proposal of HTTP [8] Web sites evolved from text pages, implementing the Request/Response Pattern [17], to complex Web applications. Whereas the former draws a clear distinction between a client who performs a request and a server who sends the response, the latter does not show this clear differentiation. Even recently a new communication protocol, denoted WebSocket protocol [25], has been published in which the browser acts like a server, listening for client-requests.

Tilley and Huang [64] proposed a new classification scheme for Web applications. They partition Web applications into three different categories: Class 1 are mainly static Web applications containing static content. Usually these are implemented as plain HTML sites. Class 2 are Web applications with some sort of dynamic behaviour. Dynamic content is realized with technologies like Flash [36], embedded Java Applets [37], Javascript or recently CSS3 [16]. Class 3 represents the most complex Web applications. In addition to Class 1 and 2 these provide dynamically generated content by server-side technologies like JSP, PHP, ASP and others. Nowadays most Web applications represent Class 3. Unfortunately developing these requires much more expertise, increasing the likelihood of defects and often requiring much higher maintenance costs.

Probably one of the most challenging aspects are constant changes. Warren et al. [67] presented a study which analyzed six websites. Their study shows a correlation between the complexity and the changes of a website. In other words, the more complex a website gets the more likely they will change in the future. There are structural and behavioural changes. While the former means changes in the document structure (e.g. Document Object Model (DOM) [18] or HTML), the latter includes all changes affecting how Web applications respond when an action is performed. This requires adaptive testing techniques. Due to the importance of this characteristic, a whole section in this thesis outlines the challenges faced with regular changes in Web applications.

Another important aspect is huge user population. While in the beginnings of the World Wide Web only universities had access, nowadays the vast majority of the population can view content distributed across all over the world. The typical Web user ranges from a mother who does an online shopping tour over a teenager updating the current relationship status on a social network platform to a student doing scientific research. Understanding these different social backgrounds of users is crucial, requiring the integration of designers in the Software Development Lifecycle (SDLC). All of the above characteristics of Web applications make functional testing a difficult task, though not impossible.

This thesis first gives an overview of what is actually meant by Software testing and then outlines the different testing approaches. The remainder of the first section discusses the three fundamental testing levels. In the second section general traps and pitfalls of Software Automation are outlined and two classes of Test Automation Frameworks are presented and compared. The third section briefly introduces some basic approaches trying to overcome the difficulties of regular changes in Web applications. Finally, the last section draws a conclusion and provides an outlook of future research directions.

# 1 Software Testing

Glenford [31] define software testing as "the process of executing a program with the intent of finding errors". According to that one might conclude that finding bugs is the only purpose of software testing. Since Test-Driven Development (TDD) [66] became popular another definition coexist: "Software testing is the process of verifying the programs output against predefined values". Besides from these IEEE [52] defines software testing as: "The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component".

## 1.1 Testing Techniques

This section outlines the different software testing techniques [41] and briefly describes testing methods representing each of the mentioned techniques in context of Web application testing.

**Black-Box Testing**
This testing technique looks at the code which needs to be tested as a whole. It does not require no internal knowledge, though test cases are usually derived from functional requirements. Therefore some sort of input data is verified against generated outcome. It is impossible to test all possible input combinations. Desikan and Ramesh [22] name it *Dijkstra's Doctrine* where for a program, accepting a six-character code, ensuring that the first character is numeric and the rest are alphanumeric, an input sequence of 9161328320 combinations needs to be generated. According to [23] one representative implementing this approach is proposed by Di Lucca et al. [24] which uses a decision table as its essential part.

**White-Box Testing**
In contrast, this technique examines the internals such as code, code structure and control flow. White-Box Testing can be further classified in static and structural testing. The former does not require executing the code, thus making the source code sufficient for examination. The source code is analyzed either by humans through a code review or by static analysis tools which check for unreachable code, unused variables, memory leaks, the

use of deprecated libraries and other metrics. Structural tests execute the program and use the control structure for coverage examination [51].

**Grey-Box Testing**
To clarify the naming conflict, the two terms Grey-Box Testing and Gray-Box Testing are used synonymously in the literature [3]. This new testing technique [47] arises in the context of Web applications. Grey-Box Testing can be described as a mixture of Black-Box and White-Box Testing (Black Box + White Box = Grey Box).

One candidate, representing this testing technique is *User-Session based testing*. Though the behaviour of Web application is examined like in Black-Box Testing it requires some internal knowledge (e.g. links to other pages).

## 1.2 Testing Levels

Testing Levels in the context of Web applications are no different than in traditional software applications. This section introduces three fundamental testing approaches, namely *Unit Testing*, *Integration Testing* and *System Testing*, beginning from fine-grained to coarse-grained.

**Unit Testing**
IEEE defines Unit Testing [52] as "testing of individual units or groups of related units". In [4] a unit "is the smallest possible testable software component". Another, yet more practical definition is [43]: A unit test is "a test, executed by the developer in a laboratory environment, that should demonstrate that the program meets the requirements set in the design specification". Though these definition clearly clarifies what unit testing is in general, often there are misunderstandings. Runeson [58] concludes, before specifying Unit tests a company should determine the granularity of units and what needs to be tested. These questions help agreeing on Unit testing principles in the development process:

- **How is a unit test conducted?**
  Nowadays most developers agree on conducting unit tests in the from of TDD [7]. In the earlier days there did not even exist code verifying other code. The only meta language to describe the functionality of a software component was either code comments or plain English text.

- **When are unit tests executed?**
  Unit tests should have no external dependencies which cause long running Unit tests. In TDD fashion non dependable tests are executed during each iteration, usually several times a day. Long running Integration tests typically run as nightly builds, ideally executed by an automated build environment like Maven [27] or Ant [26].

- **Who decides how unit test shall be conducted?** As mentioned above fine grained tests are executed by a developer during a single iteration. It is his/her own responsibility whether a chunk of code needs to be tested or not.

As we know what a Unit test is, one important question arises: How to select Unit test cases? At one extreme, test cases with one hundred percent statement coverage are surely too much. On the other hand no testing at all leads to undesirable behaviour at least or even worst, unrecoverable damage. As a rule of thumb only code causing state changes should be tested. For example testing a method, returning only the state of an object or variable (e.g. getter/setter methods) is unnecessary, though they can be used in context with other computations. Unit testing frameworks like JUnit [63] offer a special method dedicated especially for setting up the environment.

Writing testable code affects the whole development process. It changes the design of classes fundamentally. One class should serve exactly one and only one purpose. Breaking this principle means high coupling. Dependencies to other classes are not desirable because they introduce new side effects. In addition, third party dependencies may lead to unpredictable or erroneous behaviour because of blindly trusting foreign code. As a best practice, third party dependencies should be kept in one single place. This has the advantage of testing third party code in isolation and one single point of failure instead of many ones. However sometimes class level dependencies can not be removed. One solution Mackinnon et al. [45] proposed is using mock objects instead of real ones. In the literature the terms *mock object* and *stub implementation* are often used synonymously, though M. Fowler clearly differentiate [28] (together with similar confusing terms): "A *Mock object* is an object having the same interface as the real implementation." It can be initialized with state for further verification. *Stubs* "provide canned answers to calls made during the test." It usually does provide some predefined behaviour.

It can be summarized that Unit testing is an important technique ensuring software quality. Due to the fine grained test cases Unit tests should run in isolation.

**Integration Testing**
As Web applications consist of multiple components there is a need for strategies to combine them. Integration testing [4] is defined as "testing the interaction between the modules and interaction with other systems externally". IEEE defines *integration* [52] as "the process of combining software components, hardware components, or both into an overall system" and *Integration testing* [52] as "Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them".

Integration testing is different depending on the chosen integration strategy. The list below summarizes the main software integration strategies [59]:

- **Big-Bang integration**
  IEEE [52] defines it as "A type of integration [testing] in which software elements, hardware elements, or both are combined all at once into an overall system, rather than in stages". These strategy is the simplest and the most obvious one, though software systems are rarely integrated following this approach. It assumes that all integrated software components work together well and all possibilities are taken into account from the beginning. This is of course unrealistic in today's software applications as dozens of developers work on the same application showing up more and more dependencies to legacy code.

- **Continuous integration**
  This is probably one of the most popular strategies nowadays. Even small software pieces are integrated continuously. Depending on the size and complexity components are integrated daily (*daily build*), weekly (*weekly build*) or monthly (*monthly build*). Sooner or later, as the project progresses, one problem arises: What to do if a component can not be integrated because it depends on an unimplemented piece of code? A solution would be to simulate the unimplemented component. This is similar to the concept of *mock objects* mentioned earlier except that the object needs to provide at least a minimal implementation or, even better, a fake implementation. If this is not possible, another type of integration strategy needs to be chosen locally, even though the global continuous integration strategy still remains.

- **Top-down integration**
  Software components are integrated from top to bottom. A typical (Web) application consists of three or more layers. Integration, starting at the topmost layer, provide an overview of the whole system, even early in the development process. The drawback of this strategy is the simulation of bottom layers with the risk of false expectations. Fortunately not all components depend on pieces of code in lower layers. These components are often called *sub-system* [4]. Sub-systems work isolated from the rest of the components.

- **Bottom-up integration**
  It is basically the opposite of the above. Components, residing at the lower layer, are integrated before those at higher layers. It does not require any simulation because all components are defined from ground up. Users interact with an application through the graphical user interface, residing at the topmost layer which will be implemented later. Thus users can interact with the application at the end of the development phase. We recommend developing a demo application (e.g. *prototype*), providing a first impression of the whole system.

- **Bi-directional integration** [4]
  This approach, also known as *Build integration* [59], combines both, Top-down and Bottom-up integration. Individual components are grouped into categories. At first, components in each category are integrated bottom-up, leaving the topmost layer. Then all remaining components are integrated top-down.

**System Testing**
IEEE defines *System testing* [52] as "testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements". As the whole application is finished, System testing begins. It aims as evaluating the system against initial requirements. Depending on the type of requirement either test cases, implementing *functional* or *non-functional* requirements, are executed as they are defined during the requirement-gathering phase.

- **Functional testing**
  IEEE defines Functional testing [52] as "Testing conducted to evaluate the compliance of a system or component with specified functional requirements". Thus functional tests ensure the correct functioning or

behaviour of the whole system and they are typically of black-box type. They focus on evaluating the outcome against predefined expectations. Functional testing of Web applications is hard, due to the reasons mentioned in the introduction. This is the reason why there is no ultimate tool, satisfying all needs and no best practice. Though there are many tools which implement different approaches.

- **Non-functional testing**
  There are several approaches implementing this kind of testing. One are performance tests. Since the well known publication of Moore [50] back in 1965, stating that the number of integrated circuits will at least double every year, computation power indeed has increased. However, performance tests are still important due to increasing complexity and feature richness of todays applications. Another non-functional testing approach is security testing. Although the Open Web Application Security Project, abbreviated OWASP [54], published the list of the top ten Web security threats, the number of exploits listed in the Metasploit Database [44] still increase. Thus security testing is crucial for the success of a software company. Apart from the above, there are other approaches which would exceed the scope of this thesis.

# 2 Test Automation

Most of the time developers spend time reading other's code. In the remaining time they write code, chat with colleagues or do other things. Therefore time consuming, long running tasks should be done manually by someone else or automated. While the former might seem outdated, in some areas there are still manual testing techniques which can not be replaced by automation. One the other hand test automation tools gain increasing importance.

This section first compares the advantages and disadvantages of test automation and compares different kinds of test automation frameworks.

## 2.1 Pitfalls of Test Automation

Possible pitfalls and misconceptions [53] of test automation are listed below.

- **Automated testing completely replaces manual testing.**
  Often people who do not understand testing or even testers and test

managers who should agree on this statement think that manual testing can be omitted if automated tests are present. Another related misconception is to automate all tests. Marick [46] differences two economic facts regarding automation versus manual testing: “If an automated test costs a certain amount of money, it will cost almost nothing to run from then on.” and “If a manual test costs a certain amount of money to run the first time, it will cost just about the same as first time to run each time thereafter.” These two statements conclude that complex automated testing should not be done manually for economic reasons. However, [53] lists some guidelines whether tests should be automated or not: 1) Tests running on every build, 2) testing on many different platforms and 3) the same setup procedures for a lot of different tests are indicators for test automation.

Another, yet more significant reason is that the only way to evaluate automated tests is by manual reviews. This is a variant of the well known chicken egg problem. In order to test the quality of automated tests, meta test cases are required which again require additional test cases and so on.

- **The Pesticide Paradox**
  Desikan and Ramesh [22] compare writing software test cases with “designing the right pesticides to catch and kill the pests”. It means that developers writing code get used to automated test cases and develop *just for* passing the test but test cases never cover one hundred percent of all possible execution paths.

- **Independence**
  Probably every developer knows that Unit test should run independently of each other. Fortunately these tests usually have low coupling, requiring minimal overhead to set up. But *all* kinds of tests should run in isolation, thus not affecting the outcome of other tests. This is extremely challenging especially for coarse grained tests like functional Web application tests. However some test cases logically depend on each other, as for instance for a simple Website which offers some online shopping functionality. One test case verifies the correct authentication behaviour for an arbitrary user and another evaluates the correct computation of the total price if a customer adds items to the shopping cart. Clearly the latter is only possible after successful authentication, thus the result of the former test case determines the

result of the latter. Nevertheless testing needs to be done independently. Therefore we recommend an isolated testing environment as close as possible to the production environment.

- **Repeatability**
  At least as important as the above aspect is if tests executed repeatedly produce the same outcome in a stable environment. One possible indicator of different outcomes are misused threading. As threads can lead to unpredictable behaviour, they should be avoided where possible. If this is undesirable, synchronization techniques help to reduce the risk of deadlocks and race conditions. Another cause may be misused random data. If some application requires initialization with random data like playing dice, either using constant data or pregenerating random data helps to predict the outcome.

Apart from the above challenges automated testing is a great technique to make testing explicit. If new developers write code, they immediately recognize the requirements formulated in test cases. Of course this is not true if management does not acquire enough resources or it is not treated as development activity. Testing is a full time job and it should be done by dedicated experts. Another important aspect, ensuring successful automated tests, is documentation in case of a more complex testing environment. Useful might be for example documenting steps necessary for running the test cases, manual test result review in case of partial automated tests and what part of the software is not tested and why.

## 2.2 Test Automation Frameworks

As the World Wide Web attracted more and more people there has been a growing need for automated evaluation of Web applications. Companies are willing to invest a lot for a powerful automation tool. All the above approaches have the term *framework* in common. R. Johnson and B. Foote define it in the context of Object Oriented Programming [39] as "a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuses at a larger granularity than classes". This definition can also be applied to automated testing where the term *Test Automation Framework* seems appropriate.

Kit [42] distinguishes three generations of automation tools:

1. The most basic tools use the *Capture and Playback* approach. It works as a one needs to manually perform all steps necessary for evaluation

which are subsequently recorded for replaying. These tools have one major drawback in common as high maintenance costs [40]. Once everything is recorded there is no guarantee that all test cases do not break in the future, though all work now. Another yet more problematic disadvantage are hard-coded data generated by scripting languages. This should be avoided in any programming language [12]. The last problem during playback are synchronization issues due to the distributed nature of Web applications. Obviously it takes some time for the browser to fetch a Website. In order to function correctly the tool implementing capture and playback needs to properly guess the delay after the server has sent the response. Of course no tool is able to predict the future thus (over-)estimating the delay or requiring the programmer to explicitly put synchronization points into the generated script. Both solutions are cumbersome and difficult.

2. Second generation Capture and Playback tools have lessons learned. Instead of a single capture phase, tools require writing scripting code. Vendors equip their tools with a full blown scripting language which brings all the advantages of a modern programming language like code-reuse, conditional execution and error recovery to name a few. However treating automation as a software engineering task requires additional domain experts.

3. Tools from the above two generations generate a single script for all test cases. Third generation Capture and Playback tools use many small scripts each responsible for a single test case. It further extracts hard-coded data out of the scripts leading to a new era of frameworks called *Data-driven Frameworks*.

**Data-driven Frameworks**

The basic concept of Data-driven Automation Frameworks is a separation between the behavioural instructions and the data used as parameter for each test case. As [40] stated, parameters are usually kept in separated files organized as a grid. There is no limitation about the actual data format, though for usability reasons, using spreadsheet readable formats is recommended. Even though the concept of Data-driven Automation Frameworks is not application specific, until today only tools testing GUI and Web applications can be found. In the remainder of this paragraph a popular Data-driven Automation tool *Selenium* [38] specifically designed for Web application testing is presented.

Selenium was originally developed by Jason R. Huggins working on a new time tracking system for ThoughtWorks back in 2004 [11]. At that time client-side Javascript code was a nightmare to test due to different browser implementations. Almost a decade later it is still one of the most challenging tasks. Basically Selenium includes two different open source tools: *Selenium IDE* and *Selenium 2*. The former provides testing novices the ability to get used to how tests are recorded and the scripting environment. It currently runs on Mozilla Firefox only, thus it ships as a Firefox extension. The recommended version for production use is Selenium 2 which is the successor of Selenium 1. Developers have spent a lot of effort in backward compatibility, however it is of advantage to upgrade old Selenium 1 tests.

As Selenium mimics human user interaction, commands are *the* essential part of interaction. All available commands can be grouped into three command-groups: *Actions*, *Accessors* and *Assertions*. While Actions typically alter the state of the application, Accessors operate on the state of the application and Assertions verify the application state. In addition Selenium provides *Element Locators* and *Patterns*. The former, as the name implies, tells Selenium where to find HTML elements. The latter, which is typically a regular expression, can be used for instance to identify the expected value of a HTML element.

Table 1 provides a short summary of the most important commands.

| **Action** | **Selenium IDE Command [5]** | **Selenium 2 Command[1] [6]** |
|---|---|---|
| Navigation to a specific URL | `open(URL)` | `driver.get(URL)` |
| clicking on an input element | `click(locator)` | `*.click()`[2] |
| locating UI elements | `dom, xpath, css, identifier`[3] | `driver.findElements(By.*)`[4] |

Table 1: A list of basic Selenium commands

[1]All commands are examples written in the Java programming language

[2]* refers to any clickable HTML object

[3]An element locator is used as a parameter of various Action-commands. Locators usually have the format of *locatorType=argument*. Further information on element locators can be found in [5].

[4]If exactly one element is expected, the singular form `driver.findElement` should be used, otherwise the plural form `driver.findElements` is suggested. For more information

One major difference between Web applications and traditional GUI applications is that the testing process and the actual execution process run asynchronously. Therefore failing test cases do not necessarily mean that a test case indeed fails. For that reason Selenium introduced *AndWait* commands. Commands having the suffix AndWait perform its intended action as its non-waiting counterparts, but in addition wait for the page to load after the action has been done. As in AJAX Web applications there is no page refresh. *WaitForCondition* commands provide an alternative to the above commands. A typical representative of this groups of commands is the *waitForElementPresent* command which stops executing until the specified element is present.

Due to the following major drawback, the use of tools following a Data-driven approach is not recommended for commercial applications. The reason for this is scalability. To understand this problem, the internals of a tool following a Data-driven approach need to be examined. Each row in a data-grid represents exactly a single test case. The implementation of the latter is done either automatically by recording user interactions or specified manually. In either way as more application code needs to be tested new test cases generated in one of the above ways are required. This leads to the conclusion that the number of lines of automation code is *proportional* to the number of tests [40] which is generally undesirable.

**Model-driven Frameworks**

Model-driven Frameworks take a fundamentally different approach. They check the correctness of a system using a technique called *model checking.* Clarke et al. [15] defines model checking as "an automatic technique for verifying finite state [concurrent] systems". The term *finite state* typically refers to the concept of Finite State Machines (FSM) [30] which are graphically represented by a directed graph where nodes make up the states. Changes from one state to another are called transitions. A FSM must have by definition exactly one start/entry state and one or more end states. In Model Checking a finite state model $M$ is limited by a constraint expressed as a logical formula $f$ where following condition is checked: $M$ satisfies $f$. A logical formula $f$ is composed of atomic propositions and boolean expressions [15]. Researchers have spent a lot of effort in several temporal logic formalisms. The two most important ones are *computation tree logic (CTL)* and *linear temporal logic (LTL)*, though for the purpose of model checking

on element locators refer to [6].

of Web applications, simpler ones seem sufficient [62].

Ricca and Tonella proposed in their paper [55] one of the first model checking approaches. They introduced a meta model (*navigational model*), describing the general Web application structure which is given in the Unified Modeling Language (UML) [57]. A navigational model shows the connections (*links*) between Webpages. Navigation constraints are realized as key-value pairs of static and dynamic Webpages. Whereas the content of the former never changes, the content of the latter is computed dynamically at runtime using input parameters. Then a test case is "a sequence of pages to be visited plus the input values to be provided to pages containing forms" [55]. Although the primary research focus lies on *path coverage* (e.g. verifying that a path is visited), their approach promises additional testing criterias: All-paths testing, All-uses testing, Hyperlink testing, Page testing and Defintion-use testing. As an automatic test case generation scheme they proposed an algorithm which generates test cases, though it requires manual involvement for path coverage. Due to the latter factor and the growing number of rich client applications, more dynamic approaches have been explored.

Tanida et al. [62] extended the above approach to AJAX Web applications. In order to build a navigational model all application states need to be explored. Although they used the tool CRAWLAJAX [49] for exploring applications states, manual crawling (*guided crawling*) is still necessary to explore all application states. Automatic model generation is a time consuming and inefficient task. Therefore different approaches such as hashcode computation, state compression, recursive indexing [33], delta update or Sweep Line [14] have been taken into account in order to reduce the state space [34]. However [62] implemented a *state abstraction* technique which simplifies removing all redundant states. A tool named GOLIATH accepts the generated navigation model and evaluates a set of expressions. Hence it operates on the DOM, it accepts XPath expressions like `doc.xpath('//a[@id="login"]').any?`.

Although Testing Ajax applications through model checking seems most promising, van Deursen and Mesbah [65] proposed a model checker [48] which is based on *invariants*. That are different constraints concerning the user interface. However a number of open research questions remain:

- What is the most promising approach for expressing user interface constraints?
- Is it possible to construct a model of *every* Ajax application?

- Hence Javascript code is a script language, how does Javascript faults affect the model generation process?

All in all, as in [10] Model based Testing Frameworks are the newest generation of Test Automation Frameworks, hence it can be seen as the *fourth generation*. While the first and second generation mainly focuses on test execution, neglecting test design, the third generation separates test case definition from test-scripting. Although researches have investigated in extending the traditional capture-playback approach creating action-based, keyword-based, object-based or class-based approaches, every single one does require some sort of human interaction. Though, as motivated in [49], fully automated model generation is not possible mainly due to DOM state changes during script execution.

# 3 Resilience and Adaptiveness

One of the main challenges in Web application testing are changing Webpages. There are even categories of Websites which were build with the intention of regular changes such as News sites. Whereas on the frontend side updates seem to be in real time, news sites are usually realized as a Content Management System (CMS). CMS systems ease the development of high adaptive Web applications, though requiring more sophisticated testing approaches. This section gives a brief overview of different approaches satisfying this requirement.

## 3.1 Robust XPath

Abe and Hori proposed an approach [2] of stable XPath [21] expressions which resist structural changes. XPath expressions ease the navigation through hierarchical content such as XML [19], (X)HTML [20], DOM [18] and any other tree-like structures. XPath defines several navigation axes though they all can be qualified as forward axes and reverse axes. While the former navigates downwards, the latter navigates upwards. Web application test tools use XPath expressions for the identification of nodes. While in principle these tools can deal with HTML nodes and DOM nodes, we recommend using tools working with the latter, thus offering the possibility to navigate through dynamic content. However, tools often produce XPath expressions which break after a slightly changing document structure.

Abe and Hori identified three types of XPath expressions: *single-node pointing*, *alternative predicate* and *relative addressing*. Single-node pointing expressions identify, as the name suggests, exactly one single node. It will not point to any other nodes. Alternative predicate expressions use a predicate expression to limit the number of result nodes. They do not necessarily have to return a single node, instead returning a set of nodes. Relative addressing expressions use a fixed anchor node which will not change frequently.

An example of a simple HTML code is given in Listing 1.

```
<html>
    <body>
        <div>
            <label>Audi</label>
            <a href="http://www.audi.at"></a>
            <div>Perfect!</div>
        </div>
        <div>
            <label>VW</label>
            <a href="http://www.volkswagen.at"></a>
            <div>Perfect!</div>
        </div>
        <div>
            <label>Seat</label>
            <a href="http://www.seat.at"></a>
        </div>
    </body>
</html>
```

Listing 1: HTML code listing used as source for XPath expressions

Table 2 shows an example for each type of XPath expression.

| Type | Expression |
|---|---|
| single-node positioning | `/html[1]/body[1]/div[1]/label` |
| single-node positioning | `/descendant::label[1]` |
| alternative predicate | `//a[@href='http://www.volkswagen.at']` |
| relative addressing | `//a[@href='http://www.volkswagen.at']/ancestor::label[1]` |

Table 2: Illustration of three different types of XPath expressions

The most promising approach is relative addressing because M. Abe and M. Hori observed that links are unlikely to change. Though some Websites heavily rely on numeric links such as `http://orf.at/stories/2192632/` which are likely to change after a short period of time. Here URL rewriting is used to create human readable, bookmarkable links. Another problem are links with different link text. Sometimes links are written with leading `http://` and other times not. However a slightly modified XPath expression solves the problem, e.g. `//a[contains(./@href, www.volkswagen.at)]` instead of `//a[@href='http://www.volkswagen.at']`.

Robust XPath expressions are a simple, yet effective way to survive structural changes. However, the authors noted that their empirical study was to small and the XPath expressions were tightly coupled to the provided HTML document to generalize their conclusions to all kinds of HTML documents.

## 3.2 Tree Similarity

Until recently researchers have spent a lot of effort in algorithms all addressing Tree Similarity due to different areas of application such as string similarity for spell checkers, XML document analysis and text search algorithms to name just a few.

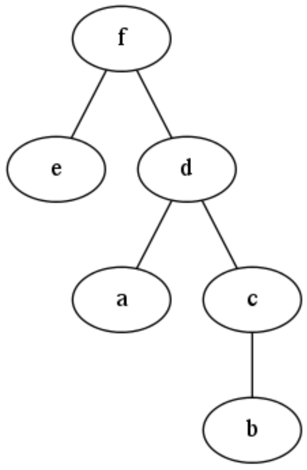
**Tree edit distance**
Back in 1977 Seijcow presented in his paper [60] an approach of measuring tree similarity. He observed that a tree can be transformed into another, yet similar tree using three operations:

- **Insert**
  Inserts a new child node at a certain position, substitute former child nodes with the node to be inserted and insert former child nodes as children of the new inserted node.

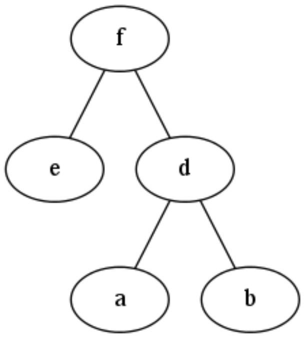
- **Delete**
  Removes a node from a tree and connects its children directly to its parent node.

- **Rename (Update)**
  Changes the label of a node, preserving the overall structure of the tree.

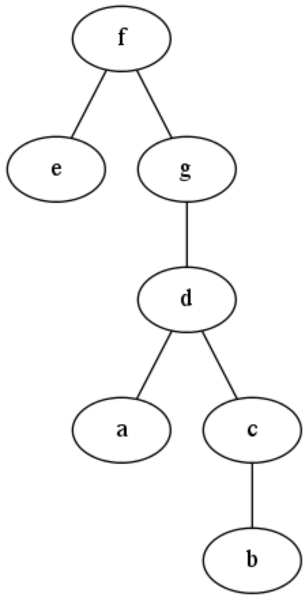
The above informal definition implies that insert and delete are inverse edit operations (i.e. insert undoes delete and vice versa). Figure 1 illustrates all three operations (adapted from [9]).



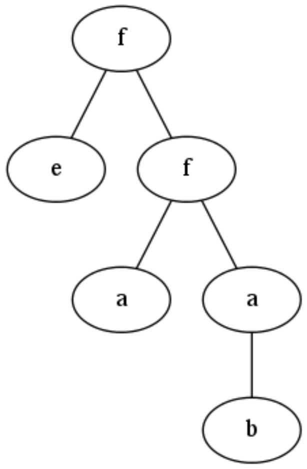
(a) Source tree for all operations



(b) Tree after deletion of node $c$



(c) Tree after insertion of node $g$



(d) Tree after relabelling $c$ to $a$ and $d$ to $f$

Figure 1: Illustration of tree-edit operations

The tree edit distance, formally defined as $\delta(T_1, T_2)$ whereas $T_1$ and $T_2$ are labelled trees, is the minimal number of operations necessary for transforming $T_1$ into $T_2$: $\min\{\gamma(S)|S$ is a sequence of operations transforming $T_1$ into $T_2\}$. $\gamma(S)$ is a function, measuring the cost of a sequence of operations

$S = s_1, ..., s_k$.

Although a large number of algorithms have been developed in the past (e.g. Tai [61] introduced the first algorithm with non exponential time complexity), there are faster, yet more complex solutions to the domain of tree similarity.

**Longest common subsequence**
Instead of viewing tree similarity as a sequence of operations, these paragraph introduces algorithms which are based on finding the longest common subsequence of linked nodes. Yang [69] introduced an algorithm addressing the similarity between two code fragments. Hua and Yang [35] followed a similar approach in context of Web data extraction.

Algorithm 1 shows the function *SimpleTreeMatching(A,B)* which takes two trees $A$,$B$ and returns the number of the longest continuous sequence of connected nodes.

**Algorithm 1:** SimpleTreeMatching(A,B)

```
if the roots of the two trees $A$ and $B$ contain distinct symbols then
    return 0;
end
$m$ := the number of first-level subtrees of $A$;
$n$ := the number of first-level subtrees of $B$;
$M[i,0] := 0$ for $i = 0, \dots, m$;
$M[0,j] := 0$ for $j = 0, \dots, n$;
for $i := 1$ to $m$ do
    for $j := 1$ to $n$ do
        $M[i,j]:=max(M[i,j-1], M[i-1,j], M[i-1,j-1] + W[i,j])$
        where $W[i,j] = SimpleTreeMatching(A,B)$
        where $A$ and $B$ are the i-th and j-th first-level subtrees of $A$
        and $B$ respectively
    end
end
return $M[m,n] + 1$;
```

As [69] mentioned, two nodes are considered identical if and only if they contain identical symbols. For simplicity node symbols are limited to letters of the English alphabet. However every other content is imaginable, for instance HTML-tags as illustrated in [35]. A more appropriate equality criteria is tag *and* attribute equality. In some cases the above criteria might be too restrictive, thus opening a new area of research for extending the above algorithm. One approach would be an extension, taking attribute similarity and tag similarity into account because existing ones [35] consider the latter one only.

Another drawback of the above algorithm is that two trees do not match even if they differ only in the root node. Therefore Yang [69] proposed an extension of the *SimpleTreeMatching* algorithm, loosening the constraint of matching nodes. Two nodes match if they are *comparable* or *identical*: Every identical node is also comparable but not the other way round.

Due to the wide area of application a lot of effort has been spent into algorithms addressing tree similarity. It would go far beyond the scope of this thesis to discuss all aspects of tree similarity.

## 3.3 Other Approaches

There are plenty of other research work addressing different areas, though some ideas seem also appropriate for adaptive functional Web application testing. This subsection outlines some of them and briefly discusses further research approaches using ideas from below.

As pag [1] motivated in their paper tools relying on the structural organization of a Website are not feasible due to the common misuses of the HTML markup language. Thus the W3C Consortium suggests Cascading Style Sheets (CSS [16]) to facilitate a clear separating between content and appearance. Event though the Document Object Model (DOM [18]) seems a better choice, combining both in a consistent way, there are similar looking, but differently organized Websites. As a remedy to this Wenyin et al. [68] first proposed an approach, measuring the visual similarity of a Website. They measure visual similarity in three metrics: *block level similarity*, *layout similarity* and *style similarity*. To calculate these metrics, a Webpage is divided into salient blocks. Whereas block level similarity is defined as "the weighted average of all pairs of matched blocks", layout similarity is defined as "the ratio of the weighted number of matched blocks to the number of total blocks". The latter metric can be calculated using the histogram of the style feature. Fu et al. [29] proposed a different approach measuring similarity based on visual cues. A Webpage is first converted into images bringing the advantage of structural independence, meaning that their approach is not limited to HTML Webpages. Then the *Earth Mover's Distance* [56] is calculated, providing a universal, yet comparable metric.

Gu et al. [32] proposed a fundamentally different approach facilitating Web content adaption. Their paper is based on the knowledge of the objective of Web publishers, inferring that Web authors first decides *what* to present and *then* decides *how* to present the information. Unlike former content adaption technologies [13, 70] an automatic top-down, tag-tree independent approach is presented, removing the limitation of a physical realization format. The essential part is the *Web Content Structure* which is formally defined as a triple $\Omega = (O, \Phi, \delta)$. Whereas $O$ is a finite set of identifiable objects, $\Phi$ describes possible separators and $\delta$ defines the relationship of every object pair. In other words $\Omega$ is a tag-tree independent abstraction of the visual representation of a Webpage. In addition Gu et al. [32] provide an algorithm to automatically infer the Web Content Structure from any Webpage. Therefore the detection process recursively divides a Webpage into smaller blocks and then merges similar ones via visual similarity.

To sum up, even though there are many different approaches addressing changing Webpages in isolation, every one has its strength and weaknesses. Yet, not a single approach has been applied to functional Web testing, offering a wide area of research.

## 4 Conclusion

Since the growing trend towards more complex Web applications replacing traditional Desktop applications, functional tests checking the functioning of an application became even more important. Although functional tests deserve their existence in Web applications, the majority of todays Websites do not include them. This emphasizes the fact that no best practice for Web application testing has been established yet. This is especially true for AJAX Web applications where application states can not be predicted due to DOM changes during script execution. Even functional testing for static Websites is challenging because of, amongst other reasons, regular changes to the Website.

Nevertheless a few approaches aiming to solve the problem of regular changes have emerged, though they all were proposed in a different context than functional Web application testing. In our opinion functional testing through model checking, inferring an abstract model of the behaviour of a Web application, seems most promising for commercial use, though a complete mapping will not be possible.

# References

[1] HTML Page Analysis Based on Visual Cues. In *Proceedings of the Sixth International Conference on Document Analysis and Recognition*, ICDAR '01, pages 859–, Washington, DC, USA, 2001. IEEE Computer Society.

[2] M. Abe and M. Hori. Robust pointing by XPath language: authoring support and empirical evaluation. In *Applications and the Internet, 2003. Proceedings. 2003 Symposium on*, pages 156–165, 2003.

[3] S. Acharya and V. Pandya. Bridge between Black Box and White Box Gray Box Testing Technique. *International Journal of Electronics and Computer Science Engineering*, 2(1), 2012.

[4] B.S. Ainapure. *Software Testing And Quality Assurance*. Technical Publications, 2008. ISBN 9788184315011.

[5] SeleniumHQ Browser Automation. `http://release.seleniumhq.org/selenium-core/1.0.1/reference.html`, 2013. accessed 19-September-2013.

[6] SeleniumHQ Browser Automation. `http://docs.seleniumhq.org/docs/03_webdriver.jsp#selenium-webdriver-api-commands-and-operations`, 2013. accessed 19-September-2013.

[7] K. Beck. *Test Driven Development: By Example*. Addison-Wesley, Pearson Education, 2003. ISBN 9780321146533.

[8] T. Berners-Lee. Information Management: A Proposal. `http://www.w3.org/History/1989/proposal.html`, 1989. accessed 18-July-2013.

[9] P. Bille. A survey on tree edit distance and related problems. *Theor. Computer Science*, 337(1-3):217–239, June 2005.

[10] M. Blackburn, R. Busser, and A. Nauman. Why model-based test automation is different and what you should know to get started. In *International conference on practical software quality and testing*, pages 212–232, 2004.

[11] C. T. Brown, G. Gheorghiu, and J. Huggins. *An introduction to testing web applications with twill and selenium*. O'Reilly Media, Inc., 2007.

[12] C. Caner. `http://www.kaner.com/pdfs/autosqa.pdf`, 1997. accessed 26-July-2013.

[13] J. Chen, B. Zhou, J. Shi, H. Zhang, and Q. Fengwu. Function-based object model towards website adaptation. In *Proceedings of the 10th international conference on World Wide Web*, WWW '01, pages 587–596, New York, NY, USA, 2001. ACM. ISBN 1-58113-348-0.

[14] S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 450–464, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-41865-2.

[15] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999. ISBN 9780262032704.

[16] The World Wide Web Consortium. `http://www.w3.org/TR/2000/WD-css3-roadmap-20000414`, 2000. accessed 22-July-2013.

[17] The World Wide Web Consortium. `http://www.w3.org/2000/xp/Group/1/10/11/2001-10-11-SRR-Transport_MEP`, 2001. accessed 18-July-2013.

[18] The World Wide Web Consortium. `http://www.w3.org/DOM/`, 2005. accessed 19-July-2013.

[19] The World Wide Web Consortium. `http://www.w3.org/TR/2008/REC-xml-20081126/`, 2008. accessed 28-July-2013.

[20] The World Wide Web Consortium. `http://www.w3.org/TR/2010/REC-xhtml-basic-20101123/`, 2010. accessed 28-July-2013.

[21] The World Wide Web Consortium. `http://www.w3.org/TR/xpath20/`, 2010. accessed 28-July-2013.

[22] S. Desikan and G. Ramesh. *Software Testing: Principles and Practices*. Dorling Kindersley/Pearson Education, 2006. ISBN 9788177581218.

[23] G.A. Di Lucca. Testing Web-based applications: the state of the art and future trends. In *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, volume 2, pages 65–69, 2005.

[24] G.A. Di Lucca, A.R. Fasolino, F. Faralli, and U. De Carlini. Testing Web applications. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 310–319, 2002.

[25] I. Fette and A. Melnikov. The RFC Series and RFC Editor. *Internet RFC 6455*, 2011.

[26] The Apache Software Foundation. `http://ant.apache.org`, 2013. accessed 22-July-2013.

[27] The Apache Software Foundation. `http://maven.apache.org`, 2013. accessed 22-July-2013.

[28] M. Fowler. Mocks Aren't Stubs. `http://martinfowler.com/articles/mocksArentStubs.html`, 2007. accessed 28-July-2013.

[29] A.Y. Fu, L. Wenyin, and X. Deng. Detecting Phishing Web Pages with Visual Similarity Assessment Based on Earth Mover's Distance (EMD). *IEEE Trans. Dependable Secur. Comput.*, 3(4):301–311, October 2006.

[30] A. Gill. *Introduction to the theory of finite-state machines*. McGraw-Hill electronic sciences series. McGraw-Hill, 1962.

[31] J. Myers Glenford. *The Art of Software Testing*. Wiley, J, 2nd edition, 2004. ISBN 0471469122.

[32] X. Gu, J. Chen, W. Ma, and G. Chen. Visual Based Content Understanding towards Web Adaptation. In *Proceedings of the Second International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*, AH '02, pages 164–173, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43737-1.

[33] G.J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *Proceedings of Third International SPIN Workshop*, 1997.

[34] G.J. Holzmann. The model checker SPIN. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.

[35] W. Hua and Z. Yang. Web Data Extraction Based on Simple Tree Matching. In *Information Engineering (ICIE), 2010 WASE International Conference on*, volume 2, pages 15–18, 2010.

[36] Adobe Systems Inc. `http://www.adobe.com/en/products/flash.html`, 2013. accessed 22-July-2013.

[37] Oracle Inc. `http://www.oracle.com/technetwork/java/index-jsp-141438.html`, 2013. accessed 22-July-2013.

[38] ThoughtWorks Inc. `http://docs.seleniumhq.org`, 2013. accessed 27-July-2013.

[39] R.E. Johnson and B. Foote. Designing reusable classes. *Journal of object-oriented programming*, 1(2):22–35, 1988.

[40] J. Kent. `http://www.simplytesting.com/Downloads/Kent%20-%20From%20Rec-Playback%20To%20FrameworksV1.0.pdf`, 2013. accessed 26-July-2013.

[41] M. Khan and Khan F. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. *International Journal of Advanced Computer Science and Applications*, 3(6), 2012.

[42] E. Kit. `http://www.drdobbs.com/integrated-effective-test-design-and-aut/184415652`, 1999. accessed 26-July-2013.

[43] T. Koomen and M. Pol. *Test Process Improvement: A Practical Step-By-Step Guide to Structured Testing*. ACM Press Series. Addison Wesley Publishing Company Inc., 1999. ISBN 9780201596243.

[44] Rapid7 LLC. `http://www.metasploit.com/modules/`, 2013. accessed 24-July-2013.

[45] T. Mackinnon, S. Freeman, and P. Craig. *Extreme programming examined*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-71040-4.

[46] B. Marick. Classic testing mistakes. *STAR East*, 1997.

[47] E. Mendes and N. Mosley. *Web Engineering*. Springer-Verlag Berlin Heidelberg, 2006. ISBN 9783540282181.

[48] A. Mesbah and A. van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4.

[49] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling AJAX by Inferring User Interface State Changes. In *Web Engineering, 2008. ICWE '08. Eighth International Conference on*, pages 122–134, 2008.

[50] G. E. Moore. Cramming more components onto integrated circuits. *Electronics magazine, IEEE*, 38(8):114 ff., 1965.

[51] S.C. Ntafos. A comparison of some structural testing strategies. *Software Engineering, IEEE Transactions on*, 14(6):868–874, 1988.

[52] Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries. *IEEE Std 610*, 1991.

[53] B. Pettichord. Seven steps to test automation success. *STAR West, San Jose, NV, USA*, 1999.

[54] The Open Web Application Security Project. `https://www.owasp.org`, 2013. accessed 24-July-2013.

[55] F. Ricca and P. Tonella. Analysis and testing of Web applications. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 25–34. IEEE Computer Society, 2001. ISBN 0-7695-1050-7.

[56] Y. Rubner, C. Tomasi, and L. Guibas. The Earth Mover's Distance as a Metric for Image Retrieval. *International Journal of Computer Vision*, 40(2):99–121, 2000.

[57] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual, (Paperback)*. The Addison-Wesley object technology series. Addison Wesley Publishing Company Incorporated, 2010. ISBN 9780321718952.

[58] P. Runeson. A Survey of Unit Testing Practices. *IEEE Software*, 23(4): 22–29, 2006.

[59] A. Schatten, S. Biffl, M. Demolsky, E. Gostischa-Franta, T. Östreicher, and D. Winkler. *Best Practice Software-Engineering*. Springer London, Limited, 2010. ISBN 9783827424877.

[60] S.M. Seijcow. The Tree To Tree editing Problem. *Information Processing Letters*, 1977.

[61] K. Tai. The Tree-to-Tree Correction Problem. *J. ACM*, 26(3), July 1979.

[62] H. Tanida, M. Prasad, S. Rajan, and M. Fujita. Automated System Testing of Dynamic Web Applications. In *Software and Data Technologies*, volume 303 of *Communications in Computer and Information Science*, pages 181–196. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-36176-0.

[63] Junit Team. `http://junit.org`, 2013. accessed 23-July-2013.

[64] S. Tilley and S. Huang. Evaluating the reverse engineering capabilities of Web tools for understanding site content and structure: a case study. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 514–523, Washington, DC, USA, 2001. IEEE Computer Society.

[65] A. van Deursen and A. Mesbah. Research Issues in the Automated Testing of Ajax Applications. In *SOFSEM 2010: Theory and Practice of Computer Science*, volume 5901 of *Lecture Notes in Computer Science*, pages 16–28. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-11265-2.

[66] B. Vodde and L. Koskela. Learning Test-Driven Development by Counting Lines. *Software, IEEE*, 24(3):74–79, 2007.

[67] P. Warren, C. Boldyreff, and M. Munro. The evolution of Websites. In *Program Comprehension, 1999. Proceedings. Seventh International Workshop on*, pages 178–185, 1999.

[68] L. Wenyin, G. Huang, L. Xiaoyue, Z. Min, and X. Deng. Detection of phishing webpages based on visual similarity. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, WWW '05, pages 1060–1061, New York, NY, USA, 2005. ACM.

[69] W. Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21(7):739–755, June 1991.

[70] Y. Yang, J. Chen, and H. Zhang. Adaptive delivery of HTML contents. *WWW9 Poster Proceedings*, pages 24–25, 2000.