uc3m | Universidad Carlos III de Madrid

e-Archivo

This is a postprint version of the following published document:

García, B. & Dueñas, J. C. (2015). Web browsing automation for applications quality control. *Journal of Web Engineering*, 14 (5-6), 474-502.

URL: https://dl.acm.org/doi/abs/10.5555/2871274.2871280

© 2015 Rinton Press.

Web Browsing Automation For Applications Quality Control

Boni García
Universidad Politécnica de Madrid
Avda. Complutense 30, 28040 Madrid, Spain
bgarcia@dit.upm.es

Juan Carlos Dueñas
Universidad Politécnica de Madrid
Avda. Complutense 30, 28040 Madrid, Spain
jcduenas@dit.upm.es

Context: Quality control comprises the set of activities aimed to evaluate that software meets its specification and delivers the functionality expected by the consumers. These activities are often removed in the development process and, as a result, the final soft-ware product usually lacks quality.
Objective: We propose a set of techniques to automate the quality control for web applications from the client-side, guiding the process by functional and non-functional requirements (performance, security, compatibility, usability and accessibility).
Method: The first step to achieve automation is to define the structure of the web nav-igation. Existing software artifacts in the phase of analysis and design are reused. Then, the independent paths of navigation are found, and each path is traversed automatically using real browsers while different kinds of assessments are carried out.
Results: The processes and methods proposed in this paper have been implemented by means of a reference architecture and open source tools. A laboratory experiment and an industrial case study have been performed in order to validate the proposal.
Conclusion: The definition of navigation paths is a rich approach to model web ap-plications. Grey-box (black-box and white-box) methods have been proved to be very valuable for web assessment. The Chinese Postman Problem (CPP) is an optimal way to find the independent paths in a web navigation modeled as a directed graph.

Keywords: Automated Software Testing, Web Applications, Web Navigation, Functional Assessment, Non-Functional Assessment, Graph Theory.

1 Introduction

Web applications market is shaped by a fierce global competition, with three main drivers: quality, cost, and time to market [1]. The reduction or elimination of quality control processes is a common practice in order to minimize costs and time to market, but this fact has a direct impact on the final software quality. The automation of quality control activities helps to improve the overall quality of software while reducing development time and costs [2][3].

According to the ISO-9126 standard, quality in use is the one perceived by the users who operate an application. This type of quality is determined by its external quality (properties

of the system during its execution) and internal quality (static system properties) [4]. Hence, quality in use of web applications is perceived from their client side.

In this paper, we propose a set of techniques to automate the quality control for web applications in the client-side. This goal can be divided into the following specific objectives: i) To propose a methodology for the automation of software quality control. ii) To propose specific processes to automate the testing and analysis of web applications. iii) To validate the feasibility of the approach.

The remainder of this paper is structured as follows. Section 2 presents a summary of the background for our work. Section 3 gives an overview on related research. Section 4 shows the methodology, which is further enriched by the processes to assess web applications in section 5. Section 6 presents the reference architecture and its implementation. Section 7 describes the validation of the work by means of an industrial case study. Finally, in section 8 we discuss the findings and conclusions of this piece of research.

2 Background

Software quality is a key concept in software engineering since it determines the degree in which a system meets its requirements and meets the expectations of its customers and/or users [5]. Quality control (also known as assessment or Verification and Validation, V&V) is the set of activities designed to assess a software system in order to ensure its quality [6].

Therefore, the quality control processes warranties the fulfillment of the application requirements of applications while reducing the number of defects [7]. The two core activities in quality control are testing and analysis. On one hand, testing is a dynamic method, i.e., it assesses the responses of a running system. On the other hand, analysis is static, i.e., it assesses the software artifacts (source code, models, and so on) without its execution. Static analysis and testing are often confused and both are mistakenly grouped under the term testing [8].

The evaluation of the quality of a software system is closely linked to the definition of its functional and non-functional requirements. Functional requirements are actions that the product must do to be useful to users. These requirements arise from the work that stakeholders need to do. Non-functional requirements (also known as quality attributes) are properties that the product must have. In other words, the functional requirements define what the system should do while non-functional requirements define how the system should behave.

2.1 Automated Assessment

Dustin et al. define Automated Software Testing (AST) as the "*Application and implementation of software technology throughout the entire Software Testing Lifecycle (STL) with the goal to improve efficiencies and effectiveness*" [2]. The goal of 100% testing automation described by Bertolino in [9] is still far away, but as described below, there are significant advances in several areas.

2.1.1 Test Cases Generation

Model-Based Testing (MBT) proposes the derivation of test cases from a model that describes some (at least partial) aspects of the System Under Test (SUT) [10]. MBT is a form of black-box testing because tests are generated from a model which is derived from the system

requirements. In contrast to usual black-box testing in which tests are written manually based on the specification, an MBT approach uses a model of the expected SUT behavior to capture the requirements [11].

Closely related to MBT, specification-based test cases generation techniques rely on formal specifications written in languages such as SDL [12], or Z [13]. When the specification is composed by a set of constraints, it is called a "contract", which usually includes pre-conditions, post-conditions, and invariants of system execution. The OCL (Object Constraint Language) is a language for defining constraints in UML models [14], and JML (Java Modeling Language) combines the design by contract approach [15] and the model-based specification approach of the Larch family [16].

The Record&Playback (R&P) approach is carried out firstly recording linear scripts corresponding to actions performed in the system (record stage). This script can be parameterized and automation is obtained by repeating the recorded script and exercising the system (playback stage) [17].

Golden software defines a correct version of a software artifact [18][19]. Golden software has been employed in software testing to derivate test cases by comparison of a software component with respect to its golden version. For example, in [20] those golden test cases are used to compare others test cases in order to make a test suite selection/reduction, or even to generate differential unit tests which are a hybrid of unit and system tests. Computational intelligence techniques have also been used in the area: Pedrycz and Vukovich presented in [21] a fuzzy approach to cause-effect software modeling as a basis for designing test cases in black-box testing, and Last and Friedman demonstrate the potential use of data mining algorithms for automated induction of functional requirements [22].

2.1.2 Test Data Generation

Test data is required as the input for executing a test case. The selection of test data is a very important activity because it is one of the key factors that affect the quality of testing process. The Automated Test Data Generation (ATDG) general problem is formally unsolvable [23], but research in this field is active.

Data generation for equivalence class partitioning (partition testing) was defined by Myers [24] in 1978 as "*a technique that partitions the input domain of a program into a finite number of classes (sets), it then identifies a minimal set of well selected test cases to represent these classes. There are two types of input equivalence classes, valid and invalid*". The equivalence partitioning testing theory ensures that only one test case of each partition is needed to evaluate the behavior of the program for the related partition. Close to that, boundary value analysis is a method which examines the boundaries of the input equivalence classes. Adrion et al. define it as "*a selection technique in which test data are chosen to lie along boundaries of the input domain (or output range) classes, data structures, procedure parameters*" [25].

Cause-effect graphing is another technique that can be defined as either test case generation or test case selection [15], besides test data generation. Its aim is to select the correct inputs to cover an entire effect set, and as such it deals with selection of test data. Cause-effect graphing exercises the different combinations of inputs from the equivalence classes. A different approach is found in random data generation, which consists of generating inputs at random until a useful input is found [26].

The path-oriented data generation technique first transforms source code of the program under test to a Control Flow Graph (CFG), i.e. a directed graph that represents its control structure. Then, the CFG is used to determine the paths to cover and test data for these paths are generated [24]. Constraint-based data generation is based on the path-oriented techniques but uses algebraic constraints to describe the input variables which describe the conditions necessary for the traversal of a given path. Constraint satisfaction problems are in general NP-complete [27]. Dynamic Domain Reduction (DDR) is a test data generation technique developed by DeMillo and Offutt that was originally employed as part of constraint-based testing [28].

The Goal-Oriented Approach, developed by Korel [29], explores a set of data generation techniques whose aim is to find input test data for a certain program path. Test data is selected from the available pool of candidate test data to execute the selected goal. Metaheuristic search techniques have also been used to find data at a reasonable computational cost [30].

2.1.3 Automated Test Oracles

A test oracle is a reliable source of expected outputs. The oracle problem is one of the biggest challenges in software testing: *How do we know that the software did what it was supposed to do when we ran a given test case?* [31].

Generally, expected outputs are manually generated based on specifications or developers knowledge on system behavior [32]. These manual oracles are expensive and unreliable, but complete automated test oracles can be expensive and sometimes impossible to provide [33].

The most important challenge to develop a complete automated test oracle is the expected output generation. In order to provide a reliable oracle, it is suggested that there should be a simulated model behaving like the SUT and automatically generate expected outputs for every possible inputs specified in the specification. The survey on automated test oracles carried out by Shahamiri on [33] describes the following methods:

- N-Version diverse systems and M-Model program (M-mp) testing. A gold version of the SUT is used to automate the oracle [34].

- Decision tables. It is a requirements representation model used wherever there are many conditions affecting responses. Di Lucca et al. applied decision tables in unit and integration web testing for both client and server pages in [35].

- Info Fuzzy Network (IFN) regression tester. This approach uses artificial intelligence methods for simulating the SUT behavior using it as test oracle [21].

- AI planner test oracle. It is applied as automated GUI (Graphic User Interface) test oracle in [36], modeling the internal behavior of GUI using a representation of GUI elements and actions.

- Artificial Neural Network (ANN) based test oracle. It requires generating a neural network that simulates the software behavior by means of Input/Output (I/O) pairs as training patterns. Since ANNs can memorize or learn from I/O pairs, it is possible to apply them as test oracles [36].

- I/O analysis-based automatic expected output generator. This method changes the input values and executing the program while observing the outputs [37].

2.1.4 Automated Software Testing Frameworks

While automatic generation of tests is still a young field, the practice of automating testing activities has gained attraction in last years. AST frameworks comprise abstract concepts, processes, procedures and environments in which automated tests are designed, created and implemented. These frameworks provide tools to address test planning, test design, test construction, test execution, test results verification, and test reporting [38].

According to the Automated Testing Institute (ATI), there are three generations of AST frameworks [3]. The 1st generation primarily comprises the linear approach to automated test development. This approach typically yields a one-dimensional set of automated tests in which each automated test is treated simply as an extension of its manual counterpart. This approach is driven by the use of the R&P. The 2nd generation is comprised by two kinds of frameworks: the data-driven (where data for scripts are stored in a database or a file external to the script) and functional decomposition ones (process of producing user-defined functions in such a way that automated test scripts can be constructed to achieve a testing objective by combining these existing components. The 3rd generation includes model-based frameworks, able to create and execute tests in a semi-intelligent manner; but the automation they provide is far from complete.

2.2 Web Testing

Web-based applications testing (or simply web testing) shares the same objectives of traditional application testing, i.e. to ensure quality and finding defects in the required functionality and services [39]. However, compared with traditional software, the definition of the testing levels must be adapted for web applications [40]:

- Unit web testing. There are different types of units that can be identified in a web application: such as web pages, scripting modules, forms, applets, servlets, or other web objects. Anyway, the basic unit that can be actually perceived and then tested is a web page.

- Integration web testing. It considers a set of related web pages in order to assess how they work together, and to identify failures due to their coupling. The web application functional requirements can drive the process of page integration testing. The knowledge of both the structure (white-box testing) and the behavior of the web application (black-box testing) have to be considered.

- System web testing: Black-box (functional) testing techniques are usually employed to accomplish system testing over the externally visible behavior of the application.

According to Di Lucca and Fasolino, the non-functional requirements are as important as functional requirements for web applications, and therefore they should be taken into account for system testing. Among them, performance, scalability, compatibility, accessibility, usability, and security [41] are key for web applications. The following list presents a description of the assessment activities that can be executed for these quality attributes:

- Performance: Verification of the specified system performances, such as response time or service availability. It is evaluated by simulating many concurrent users accessing over a defined time interval.
- Security: Verification of the effectiveness of the web defenses against undesired access of unauthorized users or improper uses.
- Compatibility: Detection of defects due to the usage of different web server platforms or client browsers.
- Usability: Verification of whether or not an application is easy to use.
- Accessibility: Verification that access to the content of the application is allowed even in presence of reduced hardware/software configurations on the client side, or by users with disabilities.

3 Related Work

Since 2008, in the Real-Time Systems and Telematic Services Architecture (a research group of Universidad Politécnica de Madrid, Spain) we have been investigating, defining and applying a range of techniques for the automated assessment of web applications. Our goal is to improve the quality of the web application and expose possible failures, that is, deviations of the application from the intended behavior. This article is the culmination of our previous work in this field. Based on the test case generation proposed on [42][43] and the functional testing approach presented in [44], this piece of research presents a proposal to evaluate the overall quality of a web application based on the automatic traversal of its navigation. Static analysis and dynamic testing is carried out in order to evaluate functional and also non-functional requirements.

Previous efforts has been done by researchers and practitioners on automated assessment of web applications based in the navigation. The idea of using a graph to represent static web sites was initially proposed by Ricca and Tonella [45], obtaining navigation paths by means of the node reduction algorithm [46]. Based on this method, they created the tools ReWeb and TestWeb to support analysis and testing of web applications [47]. ReWeb downloads and analyzes web pages with the purpose of building an UML model. TestWeb generates and executes a set of test cases for a web application whose model was computed by ReWeb.

In this paper, we also use graphs to represent web navigation. Nevertheless, our proposal to model navigation as a set of states and transitions instead of web pages and links supposes a broader point of view, very convenient for asynchronous web systems. In addition, as discussed in section 4.2.4, we proved that node reduction algorithm can be enhanced using other mechanism available in the graph theory literature, i.e. the Chinese Postman Problem.

Benedikt et al. initiated the idea of "action sequences" in a patented tool called VeriWeb [48]. VeriWeb implements a web crawler that can navigate automatically through dynamic components of web sites, including form submissions and execution of client-side scripts. Starting from a pre-defined URL, this work uses a non-deterministic algorithm that explores execution paths in the state space reachable from that starting page. VeriWeb's testing is based on graphs where nodes are web pages and edges are explicit HTML links.

We share the idea of setting a starting URL to automate web navigation while error checking is performed. Nevertheless we do not use a web crawler due to the fact we think that each state should be known beforehand in order to evaluate whether or not the navigation is performed as supposed. Instead, in section 4.2 we propose several ways to define the navigation structure (UML, R&P, and XML) previous to the automated path traversal.

WebVCR [49] and WebMacros [50] were pioneer systems for web navigation sequences automation using the R&P approach. Both systems were able to record a reduced set of events (clicks and filling in form fields) on a reduced set of elements (anchors and form-related elements). In the execution phase they relied on HTTP clients that lacked the ability to execute scripting code or to support AJAX (Asynchronous JavaScript And XML) requests. In our approach R&P navigation models are also supported, but we use real browsers instead of headless HTTP clients in order to overcome the limitation of these kind of clients.

4 Methodology

The methodology presented in this paper is focused on the automated evaluation of the functional behavior of web applications. Moreover, we are going to assess the SUT response in terms of the considered most significant non-functional requirements for web applications, i.e. performance, security, compatibility, usability, and accessibility [41]. To carry out this process, the assessment is done by the automated traversal of the navigation using a real browser.

4.1 Foundations

Nowadays web applications are more and more complex, dynamic, and asynchronous. We consider necessary to redefine the traditional web navigation as a traversal of web pages through links. We propose a broader concept of navigation based on web states and transitions. A web state S_i is a functional unit that represents each of the intermediate points in the navigation of a web application W.

$$W = \{S_1, S_2, ..., S_n\} \tag{1}$$

Each web state S_i is composed by a set of elements e_{i_j} that can be accessed with the API Document Object Model (DOM) defined by the World Wide Web Consortium (W3C). These elements are the input and output units for the end users. HTML form fields are common examples of input units (text fields, radio buttons, checkboxes, and so on). Output elements can be plain text, images, multimedia and others.

$$S_i = \{e_{i_1}, e_{i_2}, ..., e_{i_m}\} \tag{2}$$

States are connected by means of transitions. A transition is composed by a sequence of atomic actions α performed on the elements e_{a_j} of a state S_a. As a result of the execution of these actions, the web state switches from S_a to S_b. Atomic actions are based on the DOM events[a]. Some examples are: *click*, *dblclick*, *mouseover*, *keypress*, etc.

$$T_{ab} = \left\{\alpha(e_{a_x}), \alpha(e_{a_y}), ..., \alpha(e_{a_y})\right\} | S_a \longrightarrow S_b \tag{3}$$

[a]http://www.w3.org/TR/DOM-Level-2-Events/events.html

The definition of a web application with these concepts enriches the classical notion of web navigation through web pages and links. A web state is a broader concept than a web page, because in its definition it fits complex forms of navigation, such as multiple *frames* or *iframes* inside a web state. Furthermore, the concept of web transition is broader than web link. For example, a transition can occur when the user performs a mouse over an item and then click on any of the displayed options.

Following these guidelines, it is possible to define the navigation of a web application using the URL which identifies the initial state, and the transitions applied, as the remaining states are determined by the execution of transitions. This is particularly valuable for the definition of asynchronous AJAX web applications, in which not every request to the server causes a change of URL, but these requests indeed change the state of the application.

Once defined the navigation of a web application, we need to establish evaluation mechanisms through dynamic testing and static analysis (quality control, i.e. V&V). Thus, it is necessary to define control and observation points. On the one hand, a control point is how the test asks the element e_{a_j} to do the action α. On the other hand an observation point is how the SUT's behavior is found out during the result verification phase. Control points are specific in the testing domain since they exercise the SUT dynamically, while observation points are applicable to testing and analysis. As a result of these assessment activities, verdicts are issued. These verdicts are aggregated as a quality control report.

In order to formalize all the concepts presented in this section, we present the metamodel illustrated in figure 1. This metamodel represents web applications as an aggregation of web states and transitions. Quality control (V&V) is also present in the metamodel, composed by (dynamic) testing and (static) analysis activities.

4.2 Navigation Modeling

We pursue the goal of automation, so the elaboration or creation of the navigation models should not represent a large effort in the development. Thus, we identify navigation information from models that have been created through the application life cycle, and in some cases we need additional models.

4.2.1 Unified Modeling Language Diagrams

If available, the UML diagrams used to describe system requirements are a good source to find navigation information. We use the following:

- **Use case** diagrams offer a perspective of the functional requirements of the application interaction with the actors.

- **Activity** diagrams show details of each use case. These diagrams describe the flow within a use case. Therefore, activity diagrams for web applications describe the navigation structure. Each activity in this diagram is mapped as a web state S_i as defined in equation 1.

- **User interface** diagrams are needed to represent the data handled by the web application. Each field defined in this diagram is mapped as a web element e_{i_j} as defined in equation 2.

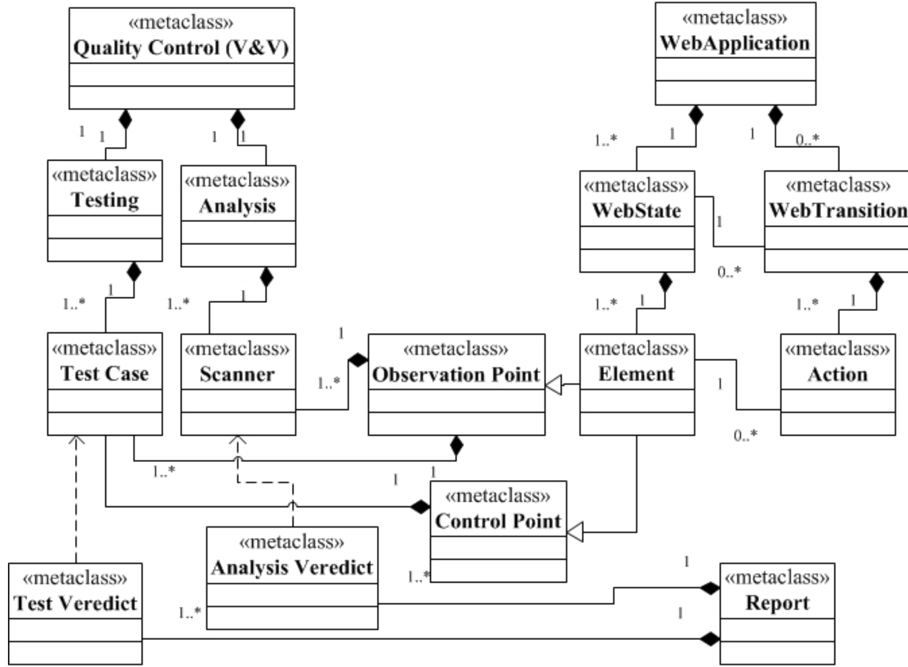

Fig. 1. Web Application and Quality Control Metamodel

Some considerations should be taken into account by testers in order to link these diagrams to the execution of the web application. First, the start node of each activity diagram is the initial URL for that flow. Second, each text description in the links between activities (label known as *guard*) describes the web transition as depicted in equation 3. We propose the following notation to define these labels:

$$[element_1, action_1, < key_1 >; ...; element_n, action_n, < key_n >] \tag{4}$$

Each group $element_i$,$action_i$,$<key_i>$ is an atomic action and it is separated from the next one using a semicolon. Each $element_i$ is a text label to locate the target HTML element in the web state. Each $action_i$ is a DOM event (*click*, *dblclick*, and so on). Finally, each $<key_i>$ is only present when a keyboard event (*keypress*, *keydown*, or *keyup*) is defined, and it represents the triggering character.

In order to locate web elements, we translate each $element_i$ label to HTML tags using XPath. XPath queries are able to locate any element in an HTML document. Nevertheless, these queries can be complex depending on the DOM structure of the page. For the sake of simplicity, other alternatives are provided to locate elements. In addition to XPath, HTML attributes are used to locate web elements. Thus, first we look for the attribute *id*. If it is not present, then we look for the attribute *name*. This process is repeated iteratively with the attributes *title*, *value*, and finally *alt*. If the element is not found neither with XPath nor with HTML attributes, the element is searched by looking for the raw text description. The pseudocode which implements this algorithm is presented in listing 1.

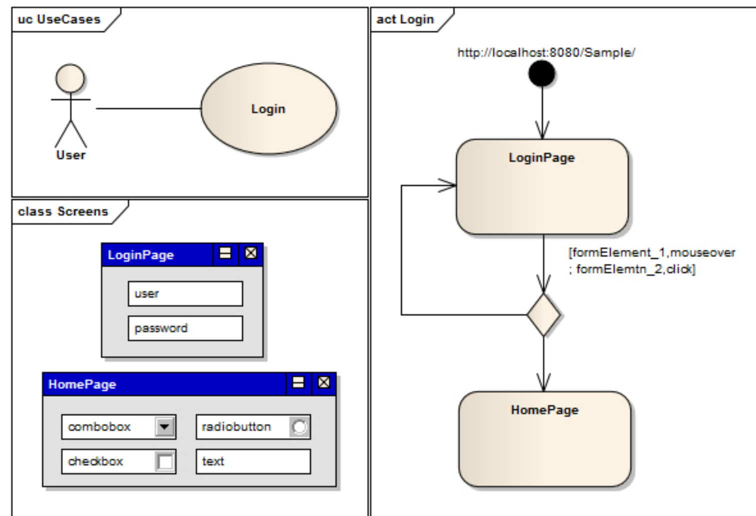

Fig. 2. Example of Navigation Modeling with UML Diagrams

Listing 1: Elements Location Algorithm

```
Function LookForElement(element_i)
  found = nothing
  For Each frame/iframe (if any)
    For Each HTML element
      If element_i is XPath
        found = Execute element_i as XPath expression
      Else
        found = Look For element_i at id/name/title/value/alt
        If Not found
          found = Look For element_i as text
        End If
      End If
    End For
  End For
  Return found
End Function
```

Figure 2 shows an example of the modeling of a navigation using UML as described in this section. First, the use case defines the high-level functional requirement, in this case, a login. Then the activity diagram defines the flow composed by two web states connected by a transition. This transition may end in another state (if the login is correct), otherwise the ending state is the same as the starting one (login not correct). The transition is composed by two atomic actions: *mouseover* on an element, and *click* on another element. Finally, the user interface diagrams describe the data of the two web states (*LoginPage* and *HomePage*).

4.2.2 Record and Playback

The second mechanism we propose to model the web navigation is the R&P approach. It is a useful way to represent the structure of a web application by recording interactions with the application through a browser once the application is running. Obviously this mechanism is used in a later phase of development lifecycle.

During the recording stage, the user interacts with the system manually via the user interface while a testing tool records the interactions. During the playback stage, that tool interacts with the system via the user interface to replay the original session. This approach is very useful to automate the navigation, since a recorded script contains itself the navigation, the data introduced, and the expected outcomes.

In this approach, a web state S_i as defined in equation 2 is composed by all the elements e_{i_j} involved in the recording until a transition is detected. A transition, as defined in equation 2 is the set of atomic actions.

As discussed in section 6, the implementation of the R&P approach we select is based on Selenium IDE[b]. This tool is a plugin that provides record, edit, debug, and playback capabilities to a Firefox browser. Selenium IDE recordings are stored as HTML pages, in which interactions with the web application are stored in a table. Each row in this table (*<tr >...</tr >*) corresponds to a Selenium command (syntax *command-target-value*).

Listing 2 shows the equivalent navigation example described in section 4.2.1 (see figure 2), this time recorded with Selenium IDE and therefore stored as a HTML page.

Listing 2: Example of Navigation Modeling with the R&P approach

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head profile="http://selenium-ide.openqa.org/profiles/test-case">
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<link rel="selenium.base" href="http://localhost:8080/" />
<title>login</title>
</head>
<body>
<table cellpadding="1" cellspacing="1" border="1">
<thead>
<tr><td rowspan="1" colspan="3">login</td></tr>
</thead><tbody>
<tr>
        <td>open</td>
        <td>/Sample/</td>
        <td></td>
</tr>
<tr>
        <td>type</td>
        <td>id=user</td>
        <td>Administrator</td>
```

[b] http://seleniumhq.org/projects/ide/

```
</tr>
<tr>
          <td>type</td>
          <td>id=password</td>
          <td>admin</td>
</tr>
<tr>
          <td>clickAndWait</td>
          <td>id=frmDatos_0</td>
          <td></td>
</tr>
<tr>
          <td>verifyText</td>
          <td>id=input-text</td>
          <td>Welcome.</td>
</tr>
</tbody></table>
</body>
</html>
```

4.2.3 XML for Modeling the Navigation

Halfway between the UML models and R&P, we propose a syntax-neutral way of modeling the navigation using a specific XML notation. We propose this specific format with the aim of combining the benefits of previous models. On the one hand, this XML notation allows to define the complete navigation structure (like UML) and not just a single path (like R&P). On the other hand, with the XML notation we are able to define test oracles (like R&P).

An XML Schema Definition (XSD) defines this format, following the metamodel depicted in figure 1. The initial state is always unique, since it identifies the entry point of the navigation. It is a mandatory XML attribute (*base*). In addition, there is a finite number of web states connected by transitions, as depicted in figure 3, which represents the XSD *type* for a web site.

Each web state S_i (see figure 4) is appointed with a unique identifier. Each state contain a set of elements e_{i_j} that can be split into two categories: data fields and oracles. On the one hand, a data field contains:

- *Locator*: data field identifier. The function *LookForElement* described in listing 1 of section 4.2.1 is also used to find each locator defined using this XML syntax.
- *Ref*: optional reference to link transition (attribute *id* in transitions).
- *Type*: data type, corresponding to the HTML input types, i.e. *text*, *password*, *checkbox*, *radio*, *submit*, *reset*, *file*, *hidden*, *image*, *button*.
- *Required*: boolean value than indicates whether the data field is mandatory.
- *Stereotype*: category of the data field: *string*, *integer*, *email*, *date*, *name*, *surname*, or *address*.

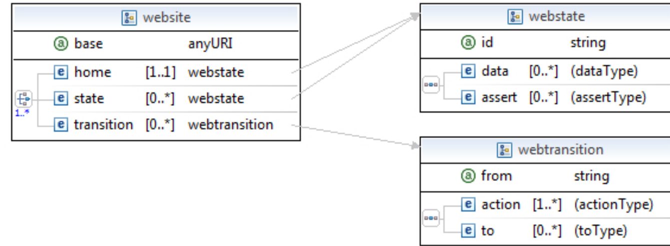

Fig. 3. XSD Graphic Representation for a Web Application

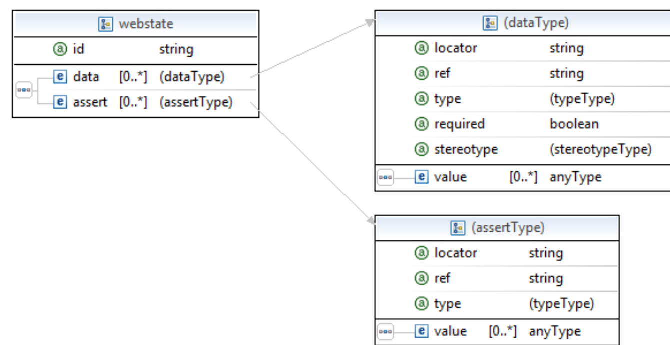

Fig. 4. XSD Graphic Representation for a Web State

- *Value*: collection of values for the data field.

On the other hand, each oracle (assertion) contains the following information:

- *Locator*: web element identifier. The function *LookForElement* is also used with this identifier.
- *Ref*: optional reference to link transition (attribute *id* in transitions).
- *Type*: comparison carried out in the oracle. It could be: *text* (assertion that compares if the text of the element identified by *locator* is exactly equal to oracle *value*), *notText* (the opposite of *text*), *textPresent* (assertion for a text contained in the in the element identified by *locator*), *textNotPresent* (the opposite of *textPresent*), *value* (assertion for the attribute *value* in the element identified by *locator*), *notValue* (the opposite of *value*).
- *Value*: Collection of values for the oracle.

Finally, each web transition (see figure 5) is composed by an attribute called *from* (which is the identifier of the source web state) and a collection of atomic actions and web targets (attribute *to*). Similarly to notation used in equation 4, the action attribute is composed by:

- *Target*: Unique identifier (*id* field) of the destination web state.

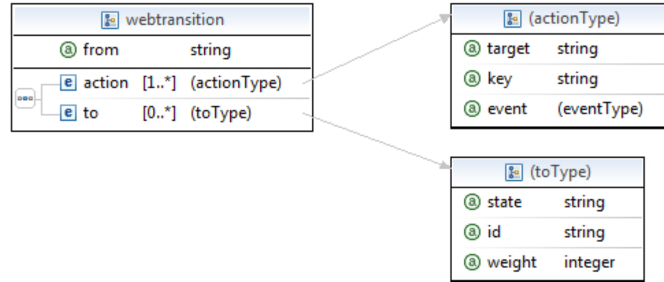

Fig. 5. XSD Graphic Representation for a Web Transition

- *Event*: Literal that describes the DOM event (*click*, *dbclick*, *mouseover*, *keydown*, and so on).
- *Key*: Optional field with the character that triggers the keyboard event (*keypress*, *keydown*, or *keyup*).

Finally, each transition T_{ab} switches the navigation from a source state S_a to a target state S_b. This destination is defined in the attribute *to*, which has the following properties:

- *State*: Target state identification (*id*).
- *Id*: Optional identification for the transition, used for conditional data fields and oracles tagged with the attribute *ref*.
- *Weight*: Numeric field that provides the capability to assign a priority to transitions. The maximum value for this field is 10, and by default each transition is assigned a weight of 5. For example, if two different transitions connect the same origin and destination state, one of them can be marked with a higher priority over the other one by establishing a higher weight. This figure will be used later in the algorithm used to split the navigation into independent paths.

Listing 3 shows an equivalent navigation example to the described in section 4.2.1 (see figure 2) and section 4.2.2 (see listing 2), but in this case using the custom XML notation proposed in this section.

Listing 3: Example of Navigation Modeling with the XML Notation

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<website xmlns="http://www.dit.upm.es/atp"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.dit.upm.es/atp
␣http://atestingp.sourceforge.net/atp.xsd"
  base="http://localhost:8080/Sample/">
   <home id="login">
      <data locator="user" ref="correct">
         <value>Administrator</value>
```

```
      </data>
      <data locator="password" ref="correct">
        <value>admin</value>
      </data>
      <data locator="username" ref="incorrect">
        <value>bad-login</value>
      </data>
      <data locator="password" ref="incorrect">
        <value>bad-password</value>
      </data>
    </home>
    <transition from="login">
      <action target="frmDatos_0" event="click" />
      <to state="init" id="correct" />
      <to state="login" id="incorrect" />
    </transition>
    <state id="init">
      <assert locator="input-text" type="text" ref="correct">
        <value>Welcome.</value>
      </assert>
    </state>
</website>
```

4.2.4 Finding the Paths

Given a navigation structure based on its states and transitions, we need to find different navigation paths. This problem leads to graph theory. In this particular case we use weighted multidigraphs, i.e. directed graphs (or digraphs, whose edges have orientation) in which a number is assigned to each edge (weight) and it is allowed having multiple edges (two or more edges that are incident to the same two vertices) and/or loops (edge that connects a vertex to itself) [51]. Web states are modeled as vertices and transitions correspond to edges in the multidigraph.

The selected coverage for the traversal of the equivalent multidigraph is all-paths, and concretely the all-transition coverage [11]. This criterion establishes that each edge (web transition) is traversed at least once (100% of transition coverage). This condition implies that each state is visited at least once too. An additional condition is to try to traverse all the paths in a single execution; while this could render useless the approach for some complex web applications in which long periods of time and external interactions must elapse to get a result (for example, granting an external permission to create a new user), this fits to the currently usual web applications where most of interactions can be performed in a single or small set of sessions.

Therefore, given a multidigraph, it is necessary to be able to select the different paths within it, so we need an algorithm to break the multidigraph into non-hamiltonian paths (a hamiltonian path visits each node exactly once) [52]. After literature review, the choice of candidates was limited to: i) Node reduction algorithm [46] and ii) Chinese Postman Problem (CPP) [53].

The node reduction algorithm finds out the path between two nodes, typically the entry

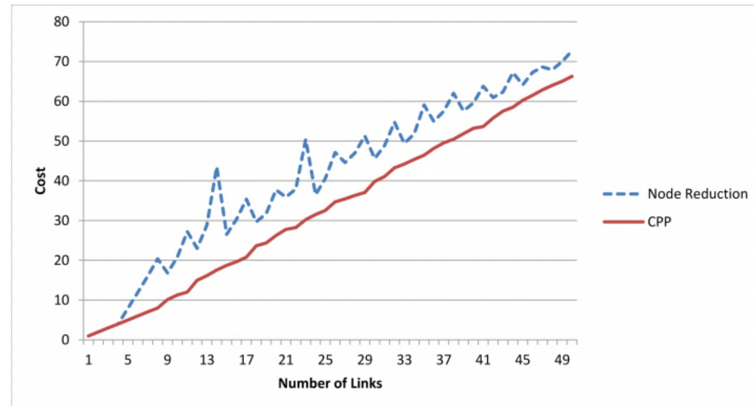

Fig. 6. Node Reduction vs. CPP Costs

and exit nodes by reducing the rest of the graph connecting these nodes. CPP is the problem of finding a shortest circuit that visits every edge of a graph at least once. Both node reduction and CPP fit with the objective of complete edge coverage, but they have a constraint that cannot be ensured for any multidigraph modeling web navigation: the input digraphs must be strongly connected. Therefore, in order to use node reduction or CPP in this approach, the input multidigraph should be converted to strongly connected digraph.

A digraph is strongly connected if every vertex is reachable from every other vertex. To increase connectivity, the graph theory offers several alternatives [51]: i) Reversing arcs; ii) Deorienting arcs; iii) Adding arcs. In this paper, we propose a simple but effective to create strongly digraphs for web navigations: adding virtual links which connects leaf nodes to the start node in the navigation. The added virtual links is later translated as a new path (the additive operator in the graph algebra) when reducing the graph into its independent paths.

In order to choose between node reduction and CPP we carried out a laboratory experiment. Random multidigraphs have been created using an incremental number of links from 1 to 50. For each multidigraph both algorithms were executed, comparing its cost (i.e. the number of visited links to traverse the graph). The experiment was carried out in a PC Intel Core2 Quad (2.66 GHz) with 4 GB of RAM memory. This experiment has been repeated 100 times. Finally, we draw the mean of the cost with respect to the number of links in figure 6.

In view of the results, we can draw a significant conclusion: CPP offered better behavior than node reduction in terms of cost. For that reason, we use the CPP algorithm created by Thimbleby [53] to find out the set of independent paths within a multidigraph representing a web application navigation.

5 Automated Assessment

So far we have described a method to model web navigation and a way to find its independent paths. While traveling along these paths, testing and static analysis should be carried out in each page in order to assess the selected quality attributes. In testing terms, the model of a web application using a multidigraph corresponds to the system testing level. The evaluation of each independent path can be considered as integration testing, and the assessment of

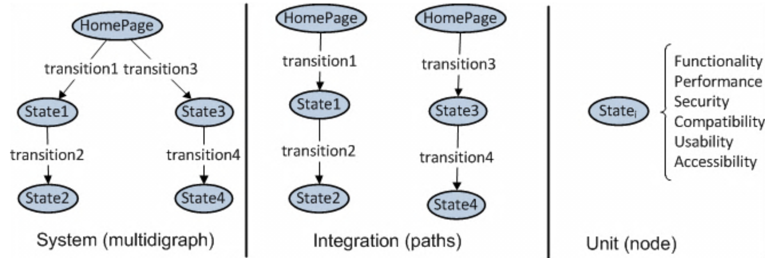

Fig. 7. Assessment Levels

each single page is the lowest level, i.e. unit testing. This approach is illustrated in figure 7. In this section we detail the processes related to automated functional and non-functional assessment.

5.1 Functional Evaluation

We propose an automated assessment process divided into four steps, as follows:

1. Pre-automation. Definition of the navigation structure using any of the proposed mechanisms (UML, R&P or XML).

2. Configuration. Test case generation settings are established. We need to set which real browser is going to be used. In addition, a test data dictionary can be also provided. This dictionary contains a collection of data that can be used as input for test cases.

3. Test case generation. In this step test cases and test data (input and output) are derived using the artifacts involved in the two previous stages. Test data are going to be stored by means of decision tables.

4. Post-automation. After the automated test case generation, additional test input data and expected outcomes can be manually added in the decision tables.

This process is illustrated in figure 8. Test case generation implies a set of activities carried out by different modules, namely:

Test logic generation. This module takes as input the model from the pre-automation stage. This model (UML, R&P or XML) is converted to a strongly connected multidigraph as depicted in section 4.2.4. Then CPP is executed to find the navigation paths. This behavior is implemented in the component called *White-Box Parser*.

Test data generation. This module extracts test data and expected outcome from the input model (*Black-Box Parser* component). XML and R&P models can include test data and expected outcome. Regarding test data (input), the parser extracts the value, data type, and stereotype. When the value is not provided (e.g. in UML models), the data type/stereotype is used to incorporate test data based on the test data dictionary provided in the configuration phase. The component called *randomizer* generates a random pointer to select a specific value of test data within the dictionary using its category (type/stereotype). Therefore, the data required for the test cases consist of the aggregation of three different sources: i) Data from

the XML and R&P models; ii) Randomly generated data from a test data dictionary; iii) Optionally, additional data can be included manually as new rows in the decision table in the post-automation phase.

Test oracle generation. This module collects data from the response of the SUT and extracts the following information: i) Navigation state; ii) Actual data returned by the application. This information is used to perform two kinds of assessment:

- White-box assessment. It is made by evaluating the expected state. As depicted in equation 2, we consider a web state as the aggregation of web elements. Therefore, the expected state is a set of web elements as defined in the navigation model (UML, XML, or R&P). In order to verify this assertion, all the expected web elements must be present in the current web state.
- Black-box assessment. This module gives verdicts by comparing expected with actual data. The expected data come from the black-box parser of the test data module. Additional expected data can be added in the post-automation stage by adding new rows in the decision table (see table 1).

Verdicts from white and black-box oracles feed the test report. The browser component is responsible for orchestrating the automation. This entity takes as input the test data and the information about the paths, i.e. the set of states to be traversed and the transitions between these states. Thus, the browser is in charge of performing the navigation using a real browser by exercising the client-side of the web application under test using the path information and test data. As a result, the web under test returns the real value of the navigation in terms of functionality (output data) and structure (navigation states).

Test data (input and output) are stored in decision tables as depicted in table 1. Each input ($in_element_i$) and output element ($out_element_j$) is located in each web state using the *LookForElement* algorithm described in listing 1 (section 4.2.1).

Table 1. Decision Table Structure

Test data (input)			Expected outcome (output)		
$in_element_1$	...	$in_element_n$	$out_element_1$	...	$out_element_m$
$data_{1_1}$	...	$data_{1_n}$	$outcoume_{1_1}$	...	$outcome_{1_m}$
$data_{2_1}$	...	$data_{2_n}$	$outcoume_{2_1}$	...	$outcome_{2_m}$
...	...	...	...	...	...

5.2 Non-Functional Evaluation

Regarding performance testing, our approach consist on a load injector that simulates concurrent users navigating the SUT. In the meantime, measures of response time, throughput and bitrate values are taken. These measures are compared against pre-defined thresholds (in milliseconds, samples/second, and KB/second) to identify potential errors. These values are established in the configuration stage defined in the previous section.

The automated assessment of the security is done by means of a black-box web scanner. This component looks for scripts and forms where it can inject data for each of the states within the navigation paths, observing its response in order to find vulnerabilities. These

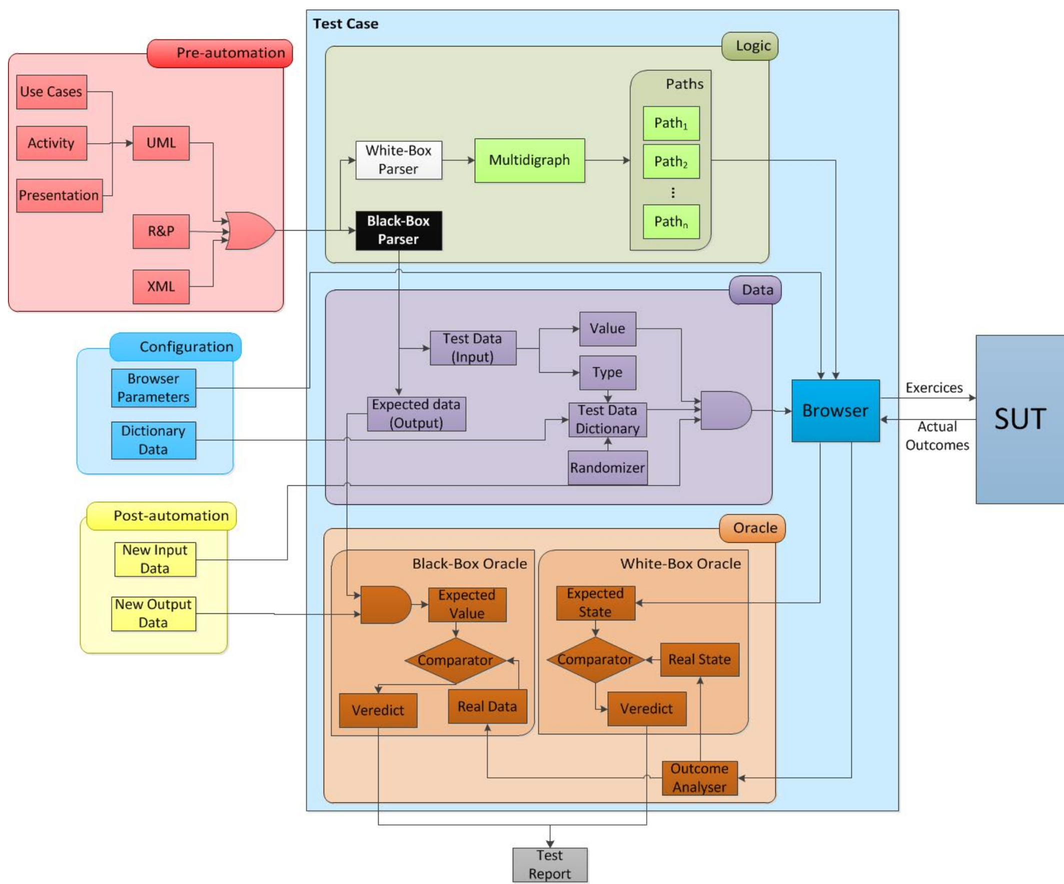

Fig. 8. Automated Functional Assessment Process

attacks correspond to known security problems. The vulnerabilities detected in our approach are:

- Cross-site scripting (XSS) vulnerabilities. This vulnerability occurs when an attacker submits malicious data to the SUT.
- Injection vulnerabilities. This includes data injection, command injection, resource injection, and SQL injection.
- Unvalidated input. It includes tainted data and forms, improper use of hidden fields, use of unvalidated data in array index, in function call, in a format string, in loop condition, in memory allocation and array allocation.

Regarding compatibility, we carry out static analysis of the source code of the different states of the web application to check that HTML and CSS code complies to standards, so they can be displayed properly in the main browsers.

The evaluation of usability and accessibility are difficult to perform automatically, since these kinds of assessment usually involve manual or user testing to identify defects of potential problems. To accomplish this automated assessment we use guideline inspectors. This kind of evaluation is based on the comparison of a set of rules (best practices, patterns, bad smells, and fault description) with the HTML/CSS source code of each web state in the navigation. We are going to perform automated analysis according to the Web Content Accessibility Guidelines (WCAG) 2.0 level AA, by checking:

- Text alternatives. Text alternatives should be provided for any non-text content (images, audio, video).
- Readability. Text content should be readable and understandable (e.g. presence of label tag to describe forms elements, content should be easy to distinguish, etc).
- Navigability: Ways to help users to navigate should be provided, find contents, and determine where they are (titles, headings, etc).

6 Architecture and Implementation

The framework which implements this proposal has been named Automatic Testing Platform (ATP)[c] and has been released as open-source under the terms of Apache license 2.0. Following the proposed processes to describe navigation, ATP accepts three kinds of inputs:

- UML diagrams. These models are created using Enterprise Architect[d]. XMI (XML Metadata Interchange) is the format in which the models are exported.
- Recordings of interactions with the SUT. These recordings are made with the tool called Selenium IDE, and stored as HTML files.
- XML navigation models, based on the proposed XSD schema.

[c] http://atestingp.sourceforge.net/
[d] http://www.sparxsystems.com.au/

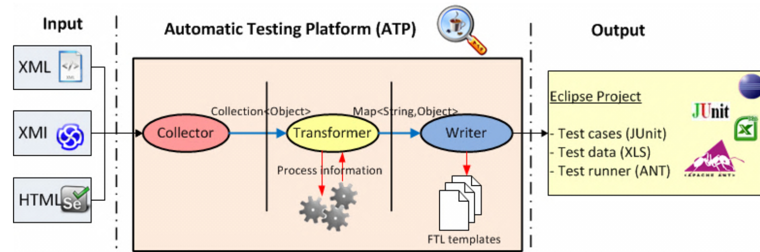

Fig. 9. ATP Architecture

	A	B	C	D	E	F
1	combobox	checkbox	radiobutton	text		anotherelment
2	yes	no	yes	This is a test		Welcome
3	no	no	yes	This is another test		Welcome
4						
5						

LoginPage | HomePage

Fig. 10. Decision Table in Excel Example

The test case generation strategy is divided into three stages (see figure 9). First, we proceed to analyze the information input by an entity called *collector*. This component extracts relevant information that is serialized to the *transformer*, which prepares the information in the form of a key-value map that is interpreted by the entity called *writer*, who is in charge of generating the test cases. The aggregation of these three entities (collector, processor and writer) is called *generator* [42][43]. In ATP, there are registered generators for each of the considered factors, i.e. functionality [44], performance, security, compatibility, usability, and accessibility.

ATP has been developed using Java. Test cases are derived using JUnit[e]. An Ant[f] script is also generated to execute these test cases. The decision table with the test data and expected outcomes are stored using Excel sheets, so an Excel file represents each path in the navigation and each sheet in the file corresponds to each web state in the path. The first row in each sheet contains labels to locate the web elements, which is translated using the *LookForElement* algorithm described in listing 1. An empty column separates the test data (input) of the expected outcome (output). In the UML example depicted in figure 2, two Excel files are generated since there are two independent paths: i) *LoginPage* → *HomePage*; ii) *LoginPage* → *LoginPage*. Regarding the first path, the generated Excel file is as illustrated in figure 10.

ATP provides a default test data dictionary. This dictionary is implemented using another Excel spreadsheet. This document contains test data categorized using the stereotypes (strings, first names, last names, addresses, dates, and email providers) as depicted in section 4.2.3. These categories are organized as independent sheets in the dictionary.

[e] http://www.junit.org/
[f] http://ant.apache.org/

All of the automatically generated artifacts by ATP are encapsulated as a Java Eclipse[g] project. As illustrated in figure 11, to implement ATP we selected several off-the-shelf open source tools:

- Selenium RC[h]. It provides a way to perform automated navigation of the web states and transitions in a real browser (Chrome, Firefox, and Explorer are supported).
- JMeter[i]. It provides a load injector to carry out performance testing.
- Wapiti[j]. Web scanner to audit the web security.
- JTidy[k]. Static analyzer of HTML code.
- CSSValidator[l]. Static analyzer of CSS code.
- WebSAT[m]. Static analyzer of usability HTML code.
- A-Checker[n]. Static analyzer of accessibility HTML code.
- JUNG[o]. Library to work with graphs, used internally to model navigation.
- FreeMarker[p]. Templates library to generate the source code for the test cases.
- JExcelAPI[q]. Library to read, write, and modify Excel spreadsheets.
- JDOM[r]. Library for manipulating XML data from Java.
- Jersey[s]. Library for accessing web services from Java.

ATP aggregates test and analysis verdicts as an HTML report (see an example in figure 12). This report contains the found defects grouped as functional, performance, security, compatibility, usability and accessibility. In addition, other useful information is compiled by ATP: error description, snapshot of each of the traversed web state, performance charts, HTTP response codes, static source code (HTML/CSS), number of received bytes, and response time.

---

[g] http://www.eclipse.org/
[h] http://seleniumhq.org/
[i] http://jakarta.apache.org/jmeter/
[j] http://wapiti.sourceforge.net/
[k] http://jtidy.sourceforge.net/
[l] http://jigsaw.w3.org/css-validator/
[m] http://zing.ncsl.nist.gov/WebTools/WebSAT/overview.html
[n] http://achecker.ca/checker/index.php
[o] http://jung.sourceforge.net/
[p] http://freemarker.sourceforge.net/
[q] http://jexcelapi.sourceforge.net/
[r] http://www.jdom.org/
[s] http://jersey.java.net/

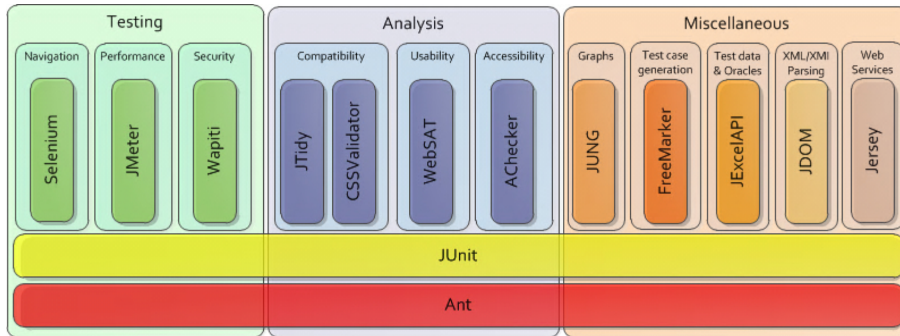

Fig. 11. ATP Implementation

Unit Test Results.

Home

Packages

es.upm.dit.atp.navigation

es.upm.dit.atp.navigation

Classes

Test_AmazonSearch

AmazonSearch - Automated Web Navigation Test/Analysis Results

Created by ATP on 08-09-2011 18:07:13.

Summary

Iteration	Functionality		Performance		Security			Compatibility		Usability	Accessibility		
#1	0 Warnings	1 Errors	4 Warnings	0 Errors	0 Low Errors	0 Medium Errors	0 High Errors	227 Warnings	45 Errors	39 Warning	879 Likely Errors	1.137 Potential Errors	13 Errors

Iteration #1

Performance Test Plan - Performance Samples

State: Home
Screenshoot - Traffic

Components

URL	Source	Status	Method	Bytes	Time (ms)
http://g-ecx.images-amazon.com/images/G/01/x-locale/common/transparent-pixel._V192234675_.gif		200	GET	43	324
http://z-ecx.images-amazon.com/images/G/01/nav2/gamma/websiteGridCSS/websiteGridCSS-websiteGridCSS-48346.css._V176526456_.css	Source	200	GET	22017	43
http://z-ecx.images-amazon.com/images/G/01/browser-scripts/us-site-wide-css-quirks/site-wide-4009516206.css._V153887713_.css	Source	200	GET	30272	45
http://g-ecx.images-amazon.com/images/G/01/Books/grutty/fall11/300x75_textbooks-only._V157725335_.gif		200	GET	7608	297
http://g-ecx.images-amazon.com/images/G/01/gno/images/orangeBlue/navPackedSprites-US-22._V183711641_.png		200	GET	7279	388
http://g-ecx.images-amazon.com/images/G/01/kindle/merch/gw/shasta/kindle-international-redirect-475x398._V153145029_.png		200	GET	32962	354
http://z-ecx.images-amazon.com/images/G/01/browser-scripts/us-site-wide-js-1.2.6/site-wide-10462726834.js._V152235461_.js		200	GET	39391	12
http://g-ecx.images-amazon.com/images/G/01/gno/popover/sprites-h2._V173407700_.gif		200	GET	134	21
http://g-ecx.images-amazon.com/images/G/01/gno/popover/sprites-v2._V173407313_.gif		200	GET	136	39
http://z-ecx.images-amazon.com/images/G/01/s9-campaigns/s9-widget._V152375922_.css	Source	200	GET	11663	8
http://ecx.images-amazon.com/images/I/51OQD0PF%2BcL._SL135_.jpg		200	GET	5383	24
http://ecx.images-amazon.com/images/I/51XqBeKo6UL._SL135_.jpg		200	GET	4525	25
http://z-ecx.images-amazon.com/images/G/01/s9-campaigns/s9-multipack-min._V154344867_.js		200	GET	1142	5
http://g-ecx.images-amazon.com/images/G/01/watches/B001A62M04._V192212155_.jpg		200	GET	3654	30
http://g-ecx.images-amazon.com/images/G/01/watches/B0019FP47E._V192598089_.jpg		200	GET	3399	34
http://g-ecx.images-amazon.com/images/G/01/watches/B000EQR6H0._V192598437_.jpg		200	GET	3167	40
http://g-ecx.images-amazon.com/images/G/01/kitchen/a-plus/sleep-innovations/41m9WSTdPRLSS3502res._SL75_.jpg		200	GET	1428	25
http://g-ecx.images-amazon.com/images/G/01/electronics/detail-page/asus-2._V189604747_.gif		200	GET	7922	69
http://g-ecx.images-amazon.com/images/G/01/digital/video/trident/merch/launch/0812_r1-gateway_nonprime._V154659899_.png		200	GET	7535	39
http://g-ecx.images-amazon.com/images/G/01/gno/images/general/navAmazonLogoFooter._V169459313_.gif		200	GET	1216	20
http://g-ecx.images-amazon.com/images/G/01/x-locale/personalization/yourstore/bluebox/bluebox-corners._V192187813_.gif		200	GET	636	21

Fig. 12. ATP Report

7 Validation

In order to carry out the validation of our approach, an industrial case study has been performed. The Research Questions (RQ) that drove this case study were the following:

- RQ1: Does the SUT accomplish its functional requirements?
- RQ2: Does the SUT have an acceptable behavior in terms of non-functional requirements?
- RQ3: Is ATP able to reveal defects?
- RQ4: What are the advantages and disadvantages of different types of input (UML, XML, and R&P) to ATP?
- RQ5: Does ATP provide any advantages in testing and analysis (defects revealed, reduction of efforts, and so on)?

The selected SUT is an electronic invoice web management system created by Universidad Politécnica de Madrid in the context of an innovation project. This web has been developed using Java technology with Spring Framework in the server-side and XHTML, CSS, and JavaScript in the client-side. The presentation layer is based on Apache Struts and JSPs. Web services have been performed using Apache CXF. Security management is used by Spring Security. The databases used are MySQL and Oracle, accessed by means of Hibernate with C3P0. The application server is JBoss. As a result, the SUT has more than 27.000 lines of code, and during its development lifecycle manual testing was performed during 1 month by 1 tester. In this process 7 defects were found.

In order to perform the case study, first we need the input models. ATP must use one of the three types of entries (XML, XMI, or R&P). In order to perform a complete case study, the three kinds of input were used. ATP completed the assessment of the SUT in 1 day. Moreover, ATP found 2184 defects in the SUT. Table 2 summarizes these results. In this table, the defects found are structured as follows:

- Functional. In this category ATP founds two types of defects: i) Warnings. Broken links detected in each of the web states during the path traversal. ii) Errors: Navigation errors, i.e. unexpected states in the navigation.
- Performance. Concurrent exercise of the SUT can also detect navigation errors. Performance warnings are reported when the measured values for response time, throughput and bitrate are out of range.
- Security: Following the OWASP (Open Web Application Security Project) rating methodology, ATP classifies security risks into three categories (low, medium, and high), depending on the severity of the detected vulnerability.
- Compatibility. ATP classifies compatibility issues in two groups: i) Warnings: potential compatibility problems. For example empty tags, deprecated attributes, etc. ii) Errors: parse errors in HTML or CSS. For example not allowed tags, bad values in attributes, etc.

- Usability. Each defect for this quality attribute is categorized as a warning. Examples of this kind of evaluation are blinking texts, absence of head tag information, etc.

- Accessibility. Divided into three categories: i) Likely: Problems identified as probable barriers, but require a human to make a decision. For example, readability issues. ii) Potential: These problems also require a human that can make a judgment. For example, format or color issues. iii) Errors: Known problems as accessibility barrier, for example, not providing text alternatives for any non-text content.

Table 2. Case Study Results

	Functionality		Performance		Security			Compatibility		Usability	Accessibility		
Test Case	Warning	Error	Warning	Error	Low	Medium	High	Warning	Error	Warning	Likely	Potential	Error
login	0	0	3	2	43	0	0	65	40	150	42	87	20
new_company	0	0	2	1	29	0	0	74	40	130	29	60	14
edit_company	2	0	1	1	35	0	0	112	60	184	27	54	13
new_admin	1	0	1	1	29	0	0	69	40	130	29	60	14
edit_admin	1	0	1	1	35	0	0	117	60	181	27	54	13
TOTAL	4	0	8	6	171	0	0	437	240	775	154	315	74

In light of these results we can draw certain conclusions in order to answer the research questions. Regarding RQ1 (*Does the SUT accomplish its functional requirements?*) we conclude that the system has been well implemented since no navigation problem has been detected. The only functionality defects found were warnings caused by broken links.

Let's move now to RQ2 (*Does the SUT have an acceptable behavior in terms of non-functional requirements?*). Regarding performance, warnings and errors were detected. This means that some expected figures on response time, throughput and bitrate are out of range. In addition, navigation errors are detected when performing concurrent requests to the server. Regarding security, only low security errors were detected. Regarding compatibility, usability, and accessibility, many defects were detected. All in all, we conclude that the system has an acceptable behavior in terms of security, but it can be improved in terms of performance, compatibility, usability and accessibility.

The answer to RQ3 (*Is ATP able to reveal defects in web applications?*) is straightforward: it does. ATP has proved to be an effective tool to find warning, errors, and potential problems. In addition, it is fully automated and data-driven, so it can be a useful framework for web developers and testers.

As for RQ4 (*What are the advantages and disadvantages of different types of input (UML, XML, and R&P) to ATP?*) we found these conclusions:

- UML: Pros: i) Analysis/design models are reused for assessment. ii) Every possible path is depicted in the models. Cons: i) It is not possible to attach test data nor oracle in the models. ii) Post-automation step is mandatory to establish expected outcomes.

- XML: Pros: i) Every possible path can be depicted using XML files. ii) Data and oracles can be attached to XML files. Cons: i) The XML files must be coded and maintained by hand.

- R&P: Pros: i) The creation of the scripts is done using Selenium IDE against the real application. ii) Data and oracles can be attached to HTML scripts. Cons: i) Each

recording is linear, therefore there is always a single path in each HTML script. ii) Error paths should be defined in different scripts. iii) ATP uses a subset of Selenium commands, so the recorded script should be done using only that subset.

Finally, for RQ5 (*Does ATP provide any advantages in testing and analysis (defects revealed, reduction of efforts, and so on)?*), we conclude that ATP does help to perform quality control and find defects in a short amount of time, reducing efforts. The main advantage of the presented approach is that it is based on the navigation of web applications in which each state is a point of observation reached via the previous states and transitions. In the traditional testing/analysis approach, a tester would need to do this browsing manually. This fact involves performing the same actions in a mechanical way again and again. Furthermore, in the case study we are able to compare the figures of the effort in testing with the results of assessing this application with ATP. The effort is much lower by using ATP (1 day vs. 1 month). In addition, the number of defects found by ATP is significantly higher, and the kind of assessment is much more complete (functionality, performance, security, compatibility, usability and accessibility vs. only functionality).

8 Conclusions

This paper proposes a set of mechanisms to automate quality control of web applications from the client side. The first conclusion has to do with the possibility of assessment automation. To automate the evaluation of quality attributes at least the expected results of execution are required. In this paper, the automation of tests and analysis have been made on the basis of a pre-defined navigation structure. To define the navigation structure we used three types of methods: UML (use case diagrams, activity diagrams, and user interface diagrams); recordings of interaction with the web application (R&P), and XML files created with a specific notation.

A valuable contribution of this work is the definition of a web navigation using **states** and **transitions** instead of the traditional way of web pages and links. Using this approach, modern asynchronous web applications can be modeled easily since it is not necessary to rely on HTTP request and responses to change the functional state of a web application. Another important conclusion of this paper is the value of **grey-box** (mixture of black-box and white-box) methods to assess web applications. On the one hand, the structure of the SUT should be known beforehand the V&V activities, i.e. the navigation structure (white-box). On the other hand, it is also important to know the behavior of the expected outcome, i.e. the output data (black-box). In this paper we have implemented this mechanism using simple scalable **decision tables** implemented by common spreadsheets.

Regarding the work on digraphs, we conclude that the most suitable algorithm to find the independent paths in a multidigraph is **CPP**. To fit the algorithm to the specific features of the web navigation, the input digraph must be converted into a strongly connected digraph by adding arcs from the leaves nodes to the root. Another important consideration is that it is difficult to get assessment completely automated. In this work we had the aim of assessing functional requirements and the most significant non-functional requirements for web applications. We conclude that following the methodology and processes depicted in this paper a reasonable reduction of effort in the development can be achieved.

Finally, we conclude that the developed framework in this research (ATP) can be a valuable tool for web developers and testers. It can be seen as the aggregation of the three kinds of AST frameworks defined by ATI: 1st generation (linear scripts based on R&P); 2nd generation (data-driven approach that stores test data and expected outcomes in spreadsheets); 3rd generation (model-based approaches that use the navigation structure of the SUT to drive the assessment).

References

1. S.A. Becker and A. Berkemeyer. Rapid application design and testing of web usability. *MultiMedia, IEEE*, 9(4):38 – 46, oct-dec 2002.
2. E. Dustin, T. Garrett, and B. Gauf. *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality.* Prentice Hall, 2009.
3. Automated Testing Institute. *Test Automation Body of Knowledge (TABOK).* Automated Testing Institute, 2011.
4. ISO. Software engineering – Product quality – Part 1: Quality model. Technical Report ISO/IEC 9126-1, International Organization for Standardization, 2001.
5. The Institute of Electrical and Eletronics Engineers. Ieee standard glossary of software engineering terminology. IEEE Standard, September 1990.
6. J. Tian. *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement.* Wiley-IEEE Computer Society Press, 1 edition, 2005.
7. I. Sommerville. *Software Engineering 9.* Pearson Education, 2011.
8. A. Abran, P. Bourque, R. Dupuis, J.W. Moore, and L.L. Tripp. *Guide to the Software Engineering Body of Knowledge - SWEBOK.* IEEE Press, Piscataway, NJ, USA, 2004 version edition, 2004.
9. A. Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.
10. P. Baker, Z.R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams. *Model-Driven Testing: Using the UML Testing Profile.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
11. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
12. ITU-T. *ITU-T Rec. Z.100 – Formal description techniques (FDT) – Specification and Description Language (SDL)*, 2002.
13. J.M. Spivey. Specifying a real-time kernel. *IEEE Softw.*, 7(5):21–28, September 1990.
14. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1999.
15. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.
16. J.V. Guttag, J.J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and tools for formal specification.* Springer-Verlag, 1993.
17. A. Sirotkin. Web application testing with selenium. *Linux J.*, 2010(192), April 2010.
18. J. Su and P.R. Ritter. Experience in testing the motif interface. *IEEE Softw.*, 8(2):26–33, March 1991.
19. P.A. Vogel. An integrated general purpose automated test environment. In *Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '93, pages 61–69, New York, NY, USA, 1993. ACM.
20. S. Elbaum, C.H. Nee, M.B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE Transactions on Software Engineering*, 35(1):29–45, 2009.
21. M. Last. Multi-objective classification with info-fuzzy networks. In *ECML*, pages 239–249, 2004.
22. N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and*

analysis, ISSTA '98, pages 73–81, New York, NY, USA, 1998. ACM.
23. J. Offutt. *Automatically Generating Test Data for Web Applications*. 5th Annual Google Test Automation Conference (GTAC), 2010.
24. G.J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.
25. W.R. Adrion, M.A. Branstad, and J.C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Comput. Surv.*, 14(2):159–192, June 1982.
26. W. Duran, J and S.C. Ntafos. An evaluation of random testing. *IEEE Trans. Softw. Eng.*, 10(4):438–444, July 1984.
27. P. McMinn. Search-based software test data generation: a survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.
28. R.A. DeMillo and A.J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, September 1991.
29. B. Korel. Dynamic method of software test data generation. *Softw. Test., Verif. Reliab.*, 2(4):203–213, 1992.
30. M. Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 342–357, Washington, DC, USA, 2007. IEEE Computer Society.
31. J.A. Whittaker. *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*. Addison-Wesley Professional, 1st edition, 2009.
32. P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.
33. S.R. Shahamiri, W.M. Kadir, and S.Z. Mohd-Hashim. A comparative study on automated software test oracle methods. In *Proceedings of the 2009 Fourth International Conference on Software Engineering Advances*, ICSEA '09, pages 140–145, Washington, DC, USA, 2009. IEEE Computer Society.
34. L.I. Manolache and D.G. Kourie. Software testing using model programs. *Softw. Pract. Exper.*, 31(13):1211–1236, October 2001.
35. G.A. Di Lucca, A.R. Fasolino, F. Faralli, and U. Carlini. Testing web applications. In *ICSM*, pages 310–319, 2002.
36. M. Vanmali, M. Last, and A. Kandel. Using a neural network in the software testing process. *Int. J. Intell. Syst.*, 17(1):45–62, 2002.
37. M.S. Phadke. Planning efficient software tests. *CrossTalk - Journal of Defense Software Engineering*, 10(10):11–15, October 1997.
38. D.J. Mosley and B. Posey. *Just Enough Software Test Automation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.
39. E. Mendes and N. Mosley. *Web Engineering*. Springer, 2010.
40. G. Rossi, O. Pastor, D. Schwabe, and L. Olsina, editors. *Web Engineering: Modelling and Implementing Web Applications*. Springer, 2008.
41. G.A. Di Lucca and A.R. Fasolino. Testing web-based applications: The state of the art and future trends. *Inf. Softw. Technol.*, 48(12):1172–1186, December 2006.
42. B. García, J.C. Dueñas, and H.A. Parada. Automatic functional and structural test case generation for web applications based on agile frameworks. In *5th International Workshop on Automated Specification and Verification of Web Systems (WWV 2009)*, pages 99–114, Hagenberg, Austria, July 2009.
43. B. García, J.C. Dueñas, and H.A. Parada. Functional testing based on web navigation with contracts. In *IADIS International Conference (WWW/INTERNET)*, pages 168–173, Rome, Italy, November 2009.
44. B. García and J.C. Dueñas. Automated functional testing based on the navigation of web applications. In *7th International Workshop on Automated Specification and Verification of Web Systems (WWV 2011)*, pages 49–65, Reykjavik, Iceland, June 2011.
45. F. Ricca and P. Tonella. Web site analysis: structure and evolution. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 76–86, 2000.

46. 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, Washington, DC, USA, 2001. IEEE Computer Society.
47. F. Ricca and P. Tonella. Detecting anomaly and failure in web applications. *MultiMedia, IEEE*, 13(2):44–51, April 2006.
48. M. Benedikt, J. Freire, and P. Godefroid. Veriweb: Automatically testing dynamic web sites. In *In Proceedings of 11th International World Wide Web Conference, WWW 2002*, 2002.
49. V. Anupam, J. Freire, B. Kumar, and D. Lieuwen. Automating web navigation with the webvcr. *Comput. Netw.*, 33(1-6):503–517, June 2000.
50. A. Safonov, J. Konstan, and J. Carlis. Beyond hard-to-reach pages: Interactive, parametric web macros. In *In Proceedings of of the 7th Conference on Human Factors & the Web*, 2001.
51. J. Bang-Jensen and G.Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer Publishing Company, Incorporated, 2nd edition, 2008.
52. B. Hasling, H. Goetz, and K. Beetz. Model based testing of system requirements using uml use case models. *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 367–376, April 2008.
53. H. Thimbleby. The directed chinese postman problem. *In journal of Software - Practice and Experience*, 33:2003, 2003.