Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 48 (2006) 1172–1186
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

Testing Web-based applications: The state of the art and future trends

Giuseppe A. Di Lucca [a,*], Anna Rita Fasolino [b]

[a] RCOST – Research Centre on Software Technology, University of Sannio, Via Traiano, 1, 82100 Benevento, Italy
[b] Dipartimento di Informatica e Sistemistica, University of Napoli Federico II, Via Claudio 21, 80125 Napoli, Italy

Received 12 April 2006; accepted 14 June 2006
Available online 22 August 2006

Abstract

Software testing is a difficult task and testing Web-based applications may be even more difficult, due to the peculiarities of such applications. In the last years, several problems in the field of Web-based applications testing have been addressed by research work, and several methods and techniques have been defined and used to test Web-based applications effectively. This paper will present the main differences between Web-based applications and traditional ones, how these differences impact the testing of the former ones, and some relevant contributions in the field of Web application testing developed in recent years. The focus is mainly on testing the functionality of a Web-based application, even if some discussion about the testing of non-functional requirements is provided too. Some indications about future trends in Web application testing are also outlined in the paper.

Keywords: Web engineering; Web application testing; Software testing

1. Introduction

The wide diffusion of Internet has produced a significant growth of the demand of Web-based applications with more and more strict requirements of reliability, usability, inter-operability and security. Due to market pressure and very short time-to-market, the testing of Web-based applications is often neglected by developers, as it is considered too time-consuming and lacking a significant payoff [14]. This deprecable habit affects negatively the quality of the applications and, therefore triggers the need for adequate, efficient and cost effective testing approaches for verifying and validating them.

Though the testing of Web-based applications (Web applications, in the remaining of the paper) shares the same objectives of 'traditional' application testing, in most cases, traditional testing theories and methods cannot be used just as they are, because of the peculiarities and complexities of Web applications. Indeed, they have to be adapted to the specific operational environment, as well as new approaches for testing them are needed.

A Web application can be considered as a distributed system, with a client-server or multi-tier architecture, including the following main characteristics:

- a wide number of users distributed all over the world and accessing it concurrently.
- heterogeneous execution environments composed of different hardware, network connections, operating systems, Web servers and Web browsers.
- an extremely heterogeneous nature that depends on the large variety of software components that it usually includes. These components can be constructed of different technologies (i.e., different programming languages and models), and can be of different natures (i.e., new components generated from scratch, legacy ones, hypermedia components, COTS, etc.).
- the ability of generating software components at run time according to user inputs and server status.

Each aspect described in the previous list produces new testing challenges and perspectives. As an example, effective

* Corresponding author.
E-mail addresses: dilucca@unisannio.it (G.A. Di Lucca), fasolino@unina.it (A.R. Fasolino).

0950-5849/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2006.06.006

solutions for executing performance and availability testing for verifying the Web application behavior when a large number of users access it at the same time will have to be looked for. Moreover, as users may utilize browsers with different Web content rendering capabilities, the Web application should be tested in order to verify its behavior when it is accessed by clients equipped with different types of Web browsers, and running on different operating systems and middleware. Another Web application critical feature to be specifically tested is its security and ability to be protected from unauthorized accesses. Moreover, the different technologies used to implement a Web application components will affect complexity and cost of setting up a testing environment able to exercise each of them, while the different mechanisms used to integrate the distributed components will produce various levels of coupling and flow of data between them, which will impact the cost of testing them effectively. As to the existence of dynamically generated software components, a real testing issue will be to cope with the difficulty of generating and re-running the same conditions that produced each component.

Of course, the main aim of the testing of a Web application is to discover failures in the required services/functionality, in order to verify the conformance of the application behavior with the specified requirements. Web application components are usually accessed by navigation mechanisms implemented by hyper-textual links, so that a specific verification activity will have to be also devoted to link checking, for assuring that no unreachable component, nor pending/broken links are included in the application.

Problems and questions about Web application testing are, therefore, numerous and complex. In this paper we discuss these questions and present some possible solutions that researchers have proposed in the last years.

The aspects of testing both the non-functional requirements and the functionality offered by a Web application will be considered in this discussion. However, after a summary presentation of the various types of non-functional requirement testing for a Web application, that we see in Section 2, the focus of the remainder of the paper will be on functional testing of Web applications. In Section 3 different categories of models providing suitable representations of the application under test will be discussed, and in Section 4 possible definitions of testing scopes for a Web application are presented. In Section 5, several test strategies for designing test cases are discussed, while in Section 6 is reported a model and method to test interaction between a Web application and the browser. In Section 7 the main features of tools for Web application testing will be analyzed. A brief discussion about future trends in Web application testing is provided in Section 8. Finally, in Section 9 we present some conclusive remarks.

2. Web application testing challenges and perspectives

In the following, the term *Web application* (or simply application) will be used to indicate the set of software components implementing the functional requirements, while the term *running environment* will indicate the whole infrastructure (composed of hardware, software and middleware components) needed to execute a Web application.

The aim of Web application testing consists of executing the application using combinations of input and state to reveal failures. A failure is the manifested inability of a system or component to perform a required function within specified performance requirements [16]. A failure can be attributed to any fault in the application implementation. Generally, there will be failures mainly due to faults in the application itself and failures that will be mainly caused by faults in the running environment or in the interface between the application and the environment where it runs. Since a Web application is strictly interwoven to its running environment, it is not possible to test them separately and establish exactly which of them is responsible for each exhibited failure.

Since the running environment mainly affects the non-functional requirements of a Web application (such as performance, stability, or compatibility), while the application is responsible for the functional requirements, Web application testing will have to be considered from two distinct perspectives. The former perspective will comprehend the different types of testing that need to be executed for verifying the conformance of the Web application with specified *non-functional* requirements. Vice-versa, the latter one will consider the problem of testing the *functional* requirements of the Web application. Of course, it is necessary that a Web application be tested from both perspectives, since they are complementary and not mutually exclusive.

Therefore, different types of testing will have to be executed to uncover these diverse types of failures [20].

Questions and challenges that characterize both testing perspectives will be analyzed in the next sub-sections.

2.1. Testing the non-functional requirements of a Web application

There are different non-functional requirements that a Web application, either explicitly or implicitly, is usually required to satisfy. The main requirements include performance, scalability, compatibility, accessibility, usability, and security. For verifying each non functional requirement, testing activities with specific aims will have to be designed. In Table 1, a list of the verification activities that can be executed to test the main non-functional requirements of a Web application is reported.

2.2. Testing the functional requirements of a Web application

This type of testing has the responsibility of uncovering failures of the applications that are due to faults in the implementation of the specified functional requirements, rather than to the execution environment. To reach this aim, the risk that observed failures are due to the running

Table 1
Web application non-functional testing

Testing Activity	Description
Performance testing	Performance testing objective is to verify specified system performances (e.g. response time, service availability). It is executed by simulating hundreds, or more, simultaneous users accesses over a defined time interval. Information about accesses are recorded and then analyzed to estimate the load levels exhausting system resources. For Web applications, system performances is a critical issue because Web users don't like to wait too long for a response to their requests, also they expect that services are always available. Performance testing of Web applications should be considered as an everlasting activity to be carried out by analyzing data from access log files, in order to tune the system adequately. Failures uncovered by performance testing are mainly due to running environment faults (such as scarce resources, or not well deployed resources, etc.), even if any software component of the application level may contribute to inefficiency.
Load testing	Load testing requires that system performance is evaluated with a predefined load level. It aims to measure the time needed to perform several tasks and functions under predefined conditions. The predefined conditions include the minimum configuration and the maximum activity levels of the running application. Also in this case a lot of simultaneous user accesses are simulated. Information is recorded and, when the tasks are not executed within predefined time limits, failure reports will be generated. Considerations similar to the ones made for performance testing can be done. Failures found by load testing are mainly due to faults in the running environment.
Stress testing	It is executed to evaluate a system, or component at or beyond the limits of its specified requirements. It is used to evaluate system responses at activity peaks that can exceed systems limitations, and to verify if the system crashes or it is able to recover from such conditions. Stress testing differs from performance and load testing because the system is executed on or beyond its breaking points, while performance and load testing simulate regular user activity. Failures found by stress testing are mainly due to faults in the running environment.
Compatibility testing	Compatibility testing will have to uncover failures due to the usage of different Web server platforms or client browsers, or different releases or configurations of them. The large variety of possible combinations of all the components involved in the execution of a Web application does not make it feasible to test all of them, so that usually only most common combinations are considered. As a consequence, just a subset of possible compatibility failures might be uncovered. Both the application and the running environment are responsible for compatibility failures.
Usability testing	Usability testing aims at verifying to what extend an application is easy to use. Usability testing is mainly centered on testing the user interface: issues concerning the correct rendering of the contents (e.g. graphics, text editing format, etc.) as well as the clearness of messages, prompts and commands are to be considered and verified. Usability is a critical issue for a Web application: indeed, it may determine the success of the application. As a consequence, the front end of the application and the way users interact with it often are the aspects that are devoted greater care and attention along the application development process. When Web applications usability testing is carried on, issues about the completeness, correctness and conciseness of the navigation along application are to be considered and verified too. This type of testing should be a continuing activity carried out to improve the usability of a Web application; techniques of user profiling are usually used to reach this aim. The application is mainly responsible for usability failures.
Accessibility testing	It can be considered as a particular type of usability testing whose aim is to verify that access to the content of the application is allowed even in presence of reduced hardware/ software configurations on the client side of the application (such as browser configurations disabling graphical visualization, or scripting execution), or of users with physical disabilities (such as blind people). In the case of Web applications, accessibility rules such as the one provided by the Web Content Accessibility Guidelines [27] have been established, so that accessibility testing will have to verify the compliance to such rules. The application is the main responsible for accessibility, even if some accessibility failures may be due to the configuration of the running environment (e.g., browsers where the execution of scripts is disabled).
Security testing	The objective of security testing is to verify the effectiveness of the overall Web system defenses against undesired access of unauthorized users, as well as their capability to preserve system resources from improper uses, and to grant the access to authorized users to authorized services and resources. System vulnerabilities affecting the security may be contained in the application code, or in any of the different hardware, software, middle-ware components of the systems. Both the running environment and the application can be responsible for security failures. In the case of Web applications, heterogeneous implementation and execution technologies, together with the very large number of possible users, and the possibility of accessing them from anywhere may make Web applications more vulnerable than traditional ones, and security testing more difficult to be accomplished.

environment should be avoided, or reduced at minimum. Preliminary assumptions about the running environment will have to be made before test cases design and execution.

Most of the methods and approaches used to test the functional requirements of 'traditional' software can be used for Web applications too. Likewise traditional software testing, testing the functionality of a Web application has to rely on the following basic aspects:

- *test models*, representing the relationships between elements of a representation or an implementation of a software component [3];
- *testing levels*, specifying the different scopes of the tests to be run, i.e., the collections of components to be tested;
- *test strategies*, defining heuristics or algorithms to create test cases from software representation models, implementation models or test models;
- *testing processes*, defining the flow of testing activities, and other decisions regarding when testing should be started, who should perform testing, how much effort should be used, and similar issues.

However, if we consider each aspect described in the previous list, there are specific problems and questions to be addressed in the context of Web applications. We will discuss such issues and possible solutions in the next sections.

3. Web application representation models

In software testing, models are needed to represent the essential concepts and relationships about the items to test [3]: such models can be used to support the selection of effective test cases, by expressing required behavior or focusing on aspects of software structure suspected to be buggy.

Web application models have been provided by several methodologies for Web application development; these models extend the traditional software models with new features for explicitly representing Web-related software characteristics. Examples of these models include the *Relationship Management Data Model (RMDM)* [17], the Object Oriented Hypermedia (OOH) proposed in [12], the OOHDM methodology [26] that permits the construction of customized Web applications by adopting object-oriented primitives to build a conceptual model, a navigational model and an interface model of the Web application. The Web Modeling Language (WebML) [2] is, moreover, a specification language that proposes four types of models, *Structural Model*, *Hypertext Model*, *Presentation Model* and *Personalization Model*, for specifying different characteristics of complex Web sites irrespective of their implementation details. Finally, Jim Conallen proposed to extend UML diagrams with new class stereotypes for representing specific Web application components such as HTML pages, forms, server pages, etc. [4].

Besides these models, further representation models have been proposed in the literature to be used explicitly in testing Web applications. These models belong to the category either of *Behavior models*, describing the functionality of a Web application irrespective of its implementation, or of *Structural models*, derived from the implementation of the application. While behavior models support black-box (or responsibility based) testing, structural ones can be used for white-box testing execution. As to the first category, use case models and decision tables in [6], and state machines in [1] have been utilized for designing Web application test cases in black-box testing techniques. As to the structural model category, both control-flow representation models of a Web application components [19,22,23a], and models describing the organization of the application in terms of Web pages and links between them have been proposed [6,23a]. Further details about most of these representations can be found in the next Section 5.

In [8] a meta-model of a Web application has been proposed. This meta-model is showed in Fig. 1 as a UML class diagram where various types of classes and associations represent several categories of a Web application components and relationships among them. A Web application can be modeled by a UML class diagram model instantiated from this meta-model.

The meta-model considers that a Web application is made up by *Web Pages*, that can be distinguished as *Server Pages*, i.e. pages that are deployed on the Web server, and *Client Pages*, i.e. pages that a Web server sends back in answer to a client request. As to the Client Pages, they can be classified as *Static Pages*, if their content is fixed and stored in a permanent way, or *Client Built Pages*, if their content varies over time and is generated on-the-fly by a Server Page. A *Client Page* is composed of *HTML Tags*. A Client Page may include a *Frameset*, composed of one or more *Frames*, and in each Frame a different Web Page can be loaded. Client Pages may comprise finer grained items implementing some processing action, such as *Client Scripts*. A Client Page may also include other *Web Objects* such as *Java Applets*, images and *Multimedia* Objects (like sounds or movies), *Flash Objects*, and others. A Client Script may include some *Client Modules*. Both Client Scripts and Client Modules may comprise *Client Functions*, or *Client Classes*. A Client Script may redirect the elaboration to another Web Page. In addition, a Client Page may be linked to another Web Page, through a hypertextual link to the Web Page URL: a link between a Client Page and a Web Page may be characterised by any *Parameter* that the Client Page may provide to the Web Page. A Client Page may also be associated with any *Downloadable File*, or it may include any *Form*, composed of different types of *Field* (such as select, button, text-area fields). Forms are used to collect user input and to *submit* the input to the Server Page, that is responsible for elaborating it. A Server Page may be composed of any *Server Script* (that may include any *Server Class* or *Server Function*) implementing some processing action, which may either *redirect* the request to another Web Page, or dynamically *build* a Client Built Page providing the result of an elaboration. Finally, a Server

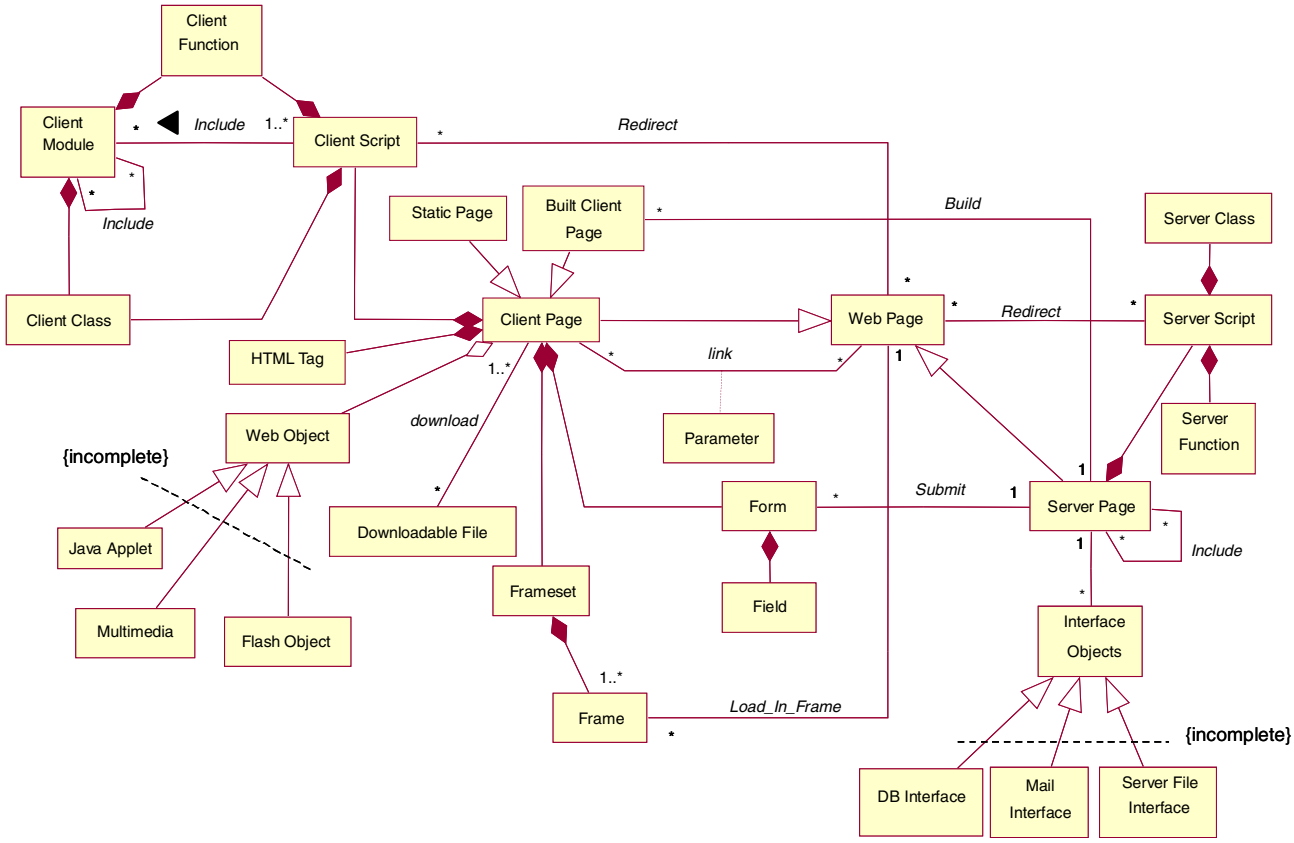

Fig. 1. The meta- model of a Web Application presented in [8].

Page may include other Server Pages, and may be associated with other *Interface Objects* allowing connection of the Web application to a DBMS, a File Server, a Mail server, or other systems.

In the following sections, we will use this model as a reference model of a Web application.

4. Unit, integration, and system testing of a Web application

A software testing process defines a set of staged testing activities, each one considering different testing levels. Usually, unit testing is the first activity of such a process followed by integration and system testing. Unit testing is carried out for verifying each individual source code component of the application, while integration testing considers combined parts of an application to verify how they work together. System testing, finally, aims at discovering defects that are properties of the entire system rather than of its individual components.

Compared with traditional software, the definition of the *testing levels* for a Web application requires a greater attention. A first question is encountered at the *unit testing* level, where the scope of a unit test cannot be defined uniquely, depending on the existence of different types of components (such as Web pages, script functions, embedded objects) residing in both the client side and server side of the application. A second problem concerns *integration testing* and regards the definition of effective strategies for integrating software units to be tested as a whole. As an example, the various mechanisms used for integrating the heterogeneous and distributed components of the Web application are able to generate various levels of coupling and flow of data between the components which will have to be considered to establish a correct integration strategy. In the following, the principal aspects regarding unit, integration, and system testing of a Web application will be discussed.

4.1. Unit testing

Different types of unit may be identified in the Web application model reported in Fig. 1, such as the Web pages, or scripting modules, forms, applets, servlets, or other Web objects normally included in these pages. Anyway, the basic unit that can be actually tested is a Web page; indeed any sub-component of a Web page should be anyway included in a Web page to be run.

As a consequence, *pages* are usually considered at the unit testing level. Of course, there are some differences between testing a client and a server page, as described in the following.

4.1.1. Client page testing

Client pages make up the end user interface of the application; they are responsible for showing textual or hyper-textual information to users, accepting user input, or

allowing user navigation throughout the application. A client page may include scripting code modules that perform simple functions, such as input validation or simple computations. Moreover, client pages may be decomposed into several frames in which other client pages can be visualized.

Testing of dynamically generated client pages (i.e. built pages) is a particular case of client page testing. The basic problem of this testing is the availability of built pages that depends on the capability of identifying and reproducing the conditions (in term of application state and user input) from which pages are built. A second problem is a state explosion problem, since the number of generable dynamic pages can be considerable, depending on the large number of possible combinations of application states and user inputs. Equivalence class partitioning criteria (such as those considering exemplar path execution of server pages) should be used to approach this question.

Of course, unit testing of client pages can be carried out by white box, black box, or gray box techniques.

Table 2 shows the typical failures that the testing of a client page will aim to identify, while Table 3 reports some criteria for client page white box test coverage.

4.1.2. Server page testing

Server pages have the main responsibility for implementing the business logic of the application, by coordinating the execution of business rules and managing the storing and retrieving of data into/from a data base.

Usually, server pages are implemented by a mixture of technologies, such as HTML, script languages (such as VBS, JSP), Java servlets, or COTS. Typical results of server page execution will be data storing into a data base, or generation of client pages showing results of the elaboration required by the user.

Unit testing of server pages can be carried out by white box, black box, or gray box techniques, likewise client page testing.

Tables 4 and 5 respectively report the typical failures that server page testing will aim to discover and some white box coverage criteria to use.

Appropriate driver and stub pages have to be generated to carry out unit page testing effectively.

4.2. Integration testing

Web application integration testing considers sets of related Web pages in order to assess how they work together, and identify failures due to their coupling. Adequate integration strategies and criteria will have to be defined; design documentation showing relationships between pages can be used to reach this aim.

For instance, the Web application model that can be obtained by instantiating the meta-model presented in Fig. 1 can be used to identify the pages to be combined: pages that are linked by direct relationships, such as *hypertextual links*, or by dependency relationships due to *redirect* or *submit* statements, or by *build* relationships may be integrated and tested. In the case of Built Client Pages, a further integration criterion may consider a Server page and each Client page it builds at run time as a whole to be

Table 5
Coverage criteria for server page white box testing

Statement/branch/path coverage in script modules;
HTML statement coverage;
Servlet, COTS, and other Web objects coverage;
Hyper-textual link coverage;
Coverage of dynamically generated pages.

Table 2
Typical failures detectable by client page testing

Differences between the content displayed by the page and the one specified and expected by a user (e.g. the rendering in the browser of both textual content and its formatting, of forms, images and other Web objects will have to be verified);
Wrong destination of links towards other pages, (i.e., when a link is selected, the expected page is not returned);
Existence of pending links, i.e., links towards not existing pages;
Wrong actions performed when a button, or any other active object, is selected by a user (i.e. the actual behaviour is different from the specified one);
Wrong content visualized in the frames;
Failures due to scripts included in the client page.

Table 3
Coverage criteria for client page white box testing

HTML statement coverage;
Web objects coverage (i.e., each image, multimedia component, applet, etc. will have to be exercised at least once);
Script blocks coverage (i.e., each block of scripting code, such as client side functions, will have to be executed at least once);
Statement/branch/path coverage for each script module;
Hyper-textual link coverage.

Table 4
Typical failures detectable by server page testing

Failures in the executions of servlets, or COTS;
Incorrect executions of data storing into a data base;
Failures due to the existence of incorrect links between pages;
Defects in dynamically generated client pages (such as non-compliance of the client page with the output specified for the server page).

tested; in this case, due to the large number of Client pages that a Server page may generate, the problem of Built Client page explosion will have to be addressed. A possible solution entails that Built pages are partitioned into a set of equivalence classes, where each class will cluster together the built pages showing the same characteristics. The identification of clusters of equivalent Built Client pages can be based on the clone detection techniques proposed in [5b,9].

The use cases implemented by the application, or any other description of the functional requirements of the application, can drive the process of page integration. As an example, for each use case (or functional requirement), the Web pages collaborating to its implementation will have to be integrated and tested. The identification of such Web pages can be made by analyzing the development documentation or by reverse engineering the application code. For instance, clusters of interconnected pages implementing a single use case can be identified by the reverse engineering technique described in [5a].

At the integration testing level, the knowledge of both the structure and the behavior of the Web application will have to be considered: the application structure will be used to define the set of pages to be integrated, while the behavior implemented by these pages will be necessary to carry out integration testing with a black box strategy. Therefore, gray box techniques will be more suitable than pure black or white box ones to carry out integration testing.

4.3. System testing

System testing aims to discover defects that depend on the entire Web application. Black box approaches are usually exploited to accomplish system testing and identify failures in the externally visible behavior of the application. Moreover, in order to discover Web applications failures due to incorrect navigation links among pages (such as links reaching a Web page different from the specified one, pending links, or links to unreachable pages), gray box testing techniques that consider the application navigation structure besides its behavior may be more effective for designing test cases.

Typical coverage criteria for system testing will include:

- user function/use case coverage (when a black box approach is used);
- page (both client and server) coverage (when white box or gray box approaches are applied);
- link coverage (when white box or gray box approaches are applied).

5. Strategies for Web application testing

Testing strategies define the approaches for designing test cases. They can be responsibility based (also known as black box), implementation based (or white box), or hybrid (also known as gray box) [3]. Black box techniques design test cases on the basis of the specified functionality of the item to be tested. White box ones rely on source code analysis to develop test cases. Gray box testing designs test cases using both responsibility based and implementation based approaches. In [20] it is said that 'Gray-box testing is well suited for Web application testing because it factors in high-level design, environment, and interoperability conditions. It will reveal problems that are not as easily considered by a black-box or white-box analysis, especially problems of end-to-end information flow and distributed hardware/software system configuration and compatibility. Context-specific errors that are germane to Web systems are commonly uncovered in this process'.

Some representative contributions presented in the literature for white box, black box and gray box testing of Web applications will be discussed in this section.

5.1. White box strategies

White box strategies design test cases on the basis of a code representation of the component under test (i.e. the *test model*), and of a *coverage model* that specifies the parts of the representation that must be exercised by a test suite. As an example, in the case of traditional software the Control Flow Graph is a typical test model, while statement coverage, branch coverage, or basis-path coverage are possible code coverage models.

As to the code representation models adopted to test Web applications, two main families of structural models have emerged from the literature: one focuses on the level of abstraction of single statements of code components of the application, and represents the traditional information about their control-flow or data-flow; a second family considers the coarser degree of granularity of the pages of the Web application and essentially represents the navigation structure between pages of the application with some eventual additional details. As to the coverage criteria, traditional ones (such as those involving nodes, edges, or notable paths from the graphical representations of these models) have been applied to both families of models.

Two white box techniques proposed in the literature to test Web applications will be presented shortly in this section. The first technique is due to Liu et al. [19] and exploits a test model that belongs to the first family of models, while the second one has been proposed by Ricca and Tonella [22,23a] and is based on two different test models, each one belonging to a different family.

The white box technique proposed by Liu et al. [19] is an example of how data-flow testing of Web applications can be carried out. The approach is applicable to Web applications implemented in HTML and XML languages, and including interpreted scripts as well as other kinds of executable components (such as Java applets, ActiveX controls, Java beans, etc.) both at the client and server side of the application.

The approach is based on a Web application test model, WATM, that includes an Object model, and a Structure model. The *Object model* represents the heterogeneous

components of a Web application and the ways they are interconnected using an object-based approach. The model includes three types of objects (i.e. client pages, server pages, and components) and seven types of relationships among objects. Each object is associated with attributes (corresponding to program variables or other HTML specific document elements such as anchors, headers, or input buttons), and operations (corresponding to functions written in scripting or programming languages). Relationships among objects are inheritance, aggregation, association, request, response, navigation, and redirect. The former three have the classical object-oriented semantics, while the latter ones represent specific relationships among client and server pages: a *request* relationship exists between a client page and a server one, if the server page is requested by the client one; a *response* relationship exists between a client page and a server one, if the client page is generated by the server one as a response of an elaboration; for two client pages there is a *navigation* relationship if one of them includes a hyperlink towards the other page; finally, between two server pages there is a *redirect* relationship if one of them redirects an HTTP request to the other page. The *Structure model*, vice-versa, uses four types of graphs to capture various types of data-flow information of a Web application: the *Control Flow Graph* (CFG) of an individual function, the *Interprocedural Control Flow Graph* (ICFG) that involves more than one function and integrates the CFGs of functions that call each other, the *Object Control Flow Graph* (OCFG) that integrates the CFGs of an object functions that are involved in sequences of function invocations triggered by GUI events, and, finally, the *Composite Control Flow Graph* (CCFG) that captures the interaction between pages where a page passes data to the other one when the user clicks a hyperlink, or submits a form, and is constructed by connecting the CFGs of the interacting Web pages.

The data-flow testing approach derives test cases from three different perspectives: intra-object, inter-object, inter-client. For each perspective, def-use chains of variables are taken into account for defining test paths that exercise the considered def-use chains. Five testing levels specifying different scopes of the tests to be run have been defined, namely: Function, Function Cluster, Object, Object Cluster, and Application level.

For the *intra-object* perspective, test paths are selected for variables that have def-use chains within an object. The def-use chains are computed using the control flow graphs of functions included in the object, and can be defined at three different testing levels: single *function*, *cluster of functions* (i.e. set of functions that interact via function calls within an object), and *object level* (considering different sequences of function invocations within an object).

For the *inter-object* perspective, test paths are selected for variables that have def-use chains across objects. Def-use chains have to be defined at the *object cluster* level, where each cluster is composed by a set of message-passing objects.

Finally, the *inter-client* perspective derives test-paths on the basis of def-use chains of variables that span over multiple clients, since in a Web application a variable can be shared by multiple clients. This level of testing is called *application* level.

This testing technique is relevant since it represents a first tentative step to extend to the field of Web applications the data flow testing approaches applicable to traditional software. However, to make it actually usable further investigations are required. Indeed, the effectiveness of the technique has not been validated by any experiment involving more than one example Web application: to carry out these experiments, an automated environment for testing execution including code analyzers, data flow analyzers, and code instrumentation tools would have been necessary. Moreover, indication about how this data flow testing approach may be integrated in a testing process would also be needed: as an example, the various testing perspectives and levels proposed by the approach might be considered in different phases of a testing process to carry out unit test, as well as integration or system test. However, also in this case an experimental validation and tuning would be required.

A second interesting proposal in the field of structural testing of Web applications has been due to Ricca and Tonella, who proposed in [22] a first approach for white box testing of primarily static Web applications. This approach was based on a test model named the *Navigational model* that focuses on HTML pages and navigational links of the application. Later, the same authors presented an additional lower layer model, the *Control flow model* representing the internal structure of Web pages in terms of the execution flow followed [23a]. This latter model has been used to carry out structural testing too.

In the Navigational model two types of HTML pages are represented: static page, whose content is immutable, and dynamic pages, whose content is established at run time by server computation, on the basis of user input and server status. Server programs (such as scripts or other executable objects) running on the server side of the application, and other page components that are relevant for navigational purposes, such as forms and frames, are also part of the model. Hyperlinks between HTML pages, and various types of link between pages and other model components are included in this code representation.

As to the Control Flow model, it takes into account the heterogeneous nature of statements written in different coding languages, and the different mechanisms used to transfer the control between statements in a Web application. It is represented by a directed graph whose nodes correspond to statements that are executed either by the Web server or by the Internet browser on the client side, and whose edges represent control transfer. Different types of nodes are shown in this model, according to the programming language of the respective statements.

A *test case* for a Web application is defined as a sequence of pages to be visited, plus the input values to be provided to pages containing forms. Various coverage

criteria applicable on both models have been proposed to design test cases: they include *Path coverage* (requiring that all paths in the Web application model are traversed in some test case), *Branch coverage* (requiring that all branches in the model are traversed in some test case), and *Node coverage* (requiring that all nodes in the model are traversed in some test case).

Assuming that nodes of the representation models can be annotated by definitions or uses of data variables, further data flow coverage criteria have been described too: *All def-use* (all definition-clear path from every definition to every use of all Web application variables are traversed in some test case), *All uses* (at least one def-clear path – if any exists – from every definition to every use of all Web application variables are traversed in some test case), *All defs* (at least one def-clear path- if any exists- from every definition to at least one use of all Web application variables is traversed in some test case).

This testing approach is partially supported by a first tool, ReWeb that analyzes the pages of the Web application and builds the corresponding Navigational model, and a second tool, TestWeb, that generates and executes test cases. However the latter tool is not completely automated, since user intervention is required to generate input and act as the oracle. The main limitation of this testing approach also consists of its scalability (consider the problem of path explosion in presence of cycles on the graphs, or the unfeasibility of all def-use coverage criterion).

Some conclusive considerations about the testing levels supported by white box techniques can be made. Some approaches will be applicable at the unit level, while other ones can be considered at the integration and system levels. For instance, the first approach proposed by Liu et al. [19] is applicable at various testing levels, ranging from unit level to integration level. As an example, the intra-object perspective can be used to obtain various types of units to be tested, while inter-object and inter-application perspectives can be considered for establishing the items to be tested at the integration level. Vice-versa, the approaches of Ricca and Tonella are applicable exclusively at the system level. As a consequence, the choice of a testing technique to be applied in a testing process will also depend on the scope of the test to run.

5.2. Black box strategies

Black box techniques do not require the knowledge of the implementation of the software items under test but design test cases on the basis of the specified or expected functionality of the item.

One main issue with black box testing of Web applications is the choice of a suitable model for specifying the behavior of the application to be tested and deriving test cases. Indeed, this behavior may significantly depend on the state of data managed by the application and on user input, with the consequence of a state explosion problem even in presence of applications implementing a few simple requirements.

To solve this problem, some solutions have been investigated and presented in the literature. Two solutions will be discussed in this sub-section. The first one is offered by the black box testing approach proposed by Di Lucca et al. [6] that exploits decision tables as a combinatorial model for representing the behavior of the Web application and producing test cases. The second one is provided by Andrews et al. [1] that propose state machines to model state dependent behavior of Web applications and designing test cases.

Di Lucca et al. [6] propose a two-stage black box testing approach that in the first stage addresses *unit test* of a Web application, while in the second one *integration test* is considered. The scope of a unit test is a single page of the application, either client or server page, while the scope of an integration test is a set of Web pages that collaborate to implement a use case of the Web application.

Unit test is carried out with a responsibility based approach that uses *decision tables* for representing page features and therefore for deriving test cases. A decision table can be used to represent the behavior of software components whose responses are each one associated with a specific condition. Usually a decision table has two parts: the *condition section* (listing conditions and combinations of conditions) and the *action section* (listing responses to be produced when corresponding combinations of conditions are true). Each unique combination of conditions and actions is a *variant*, represented as a single row in the table.

As to the *unit testing* of client and server pages, the approach requires that each page under test is preliminarily associated with a decision table describing a set of variants of the page. Each variant represents an alternative behavior offered by the page and is defined in terms of an Input section and an Output section. In the case of client pages, the *Input Section* describes a condition in terms of *Input variables* to the page, *Input actions*, and *State before test* where the state is defined by the values assumed, before test execution, by page variables, tag attributes, cookies, and by the state of other Web objects used by page scripts. In the *Output Section*, the action associated with each condition is described by the *Expected Results*, *Expected Output* actions, and *Expected State after test* (defined as for the State before Test). Table 6 reports the template of the decision table for client page testing.

Of course, this specification technique may be affected by the problem of variant explosion. However, criteria for partitioning input section data into equivalence classes may be defined and used to reduce the set of variants to be taken into account.

In the case of server pages, the Decision Table template is slightly different (see Table 7) : for each page variant the *Input Section* includes the *Input variables* field that comprises the variables provided to the server page when it is executed, and the *State before test* field that is defined by the values assumed, before test execution, by page session variables and cookies, as well as by the state of the session

Table 6
A decision table template for client page testing

Variant	Input section			Output section		
	Input variables	Input actions	State before test	Expected results	Expected output actions	Expected state after test
…	…			…		

Table 7
A decision table template for server page testing

Variant	Input section		Output section		
	Input variables	State before test	Expected results	Expected output actions	Expected state after test
	…		…		

objects used by the page scripts. In the *Output Section*, the *Expected Results* field represents the values of the output variables computed by the server page scripts, the *Expected Output* field includes the actions performed by the server side scripts (such as composing and sending an e-mail message), and the *Expected State after test* field includes the values of variables and cookies, as well as the state of session objects, after execution.

As to the definition of the Decision Tables, the authors propose to compile them by analyzing the development documentation (if available) or by reverse engineering the Web application code, and focusing on the page inner components that contribute to define conditions and actions of each variant. An object model of a Web application representing each component of the application relevant for testing purposes is specifically presented by the authors to support this type of analysis. This model is actually an extended version of the one we reported in Fig. 1, including some additional relevant details for the aims of testing (such as session variables).

The *test case selection strategy* is based on the decision tables and requires that test cases are defined in order to cover each table variant for both true and false values. Other criteria based on partitioning of the input sets into equivalence classes are also suggested for defining test cases.

In this testing approach, decision tables are also used to develop *driver* and *stub* modules which will be needed to execute the client page testing. A driver module will be a Web page that interacts with the client page by populating its input forms and generating the events specified for the test case. The driver page will include some script functions, and the DOM will allow its interaction with the tested page. Stub modules can be developed either as client pages, server pages or Web objects. The complexity of the stub will depend both on the type of interaction between the tested page and the component to be substituted, and on the complexity of the function globally implemented by the pair of components.

As to the *Integration testing*, a fundamental question is the one of determining which Web pages have to be integrated and tested. The authors of this approach propose to integrate Web pages that collaborate to the implementation of each use case (or functional requirement) of the application. They propose to analyze the object model of the Web application in order to find client and server pages to be gathered together. A valuable support for the identification of clusters of interconnected pages may be provided by *clustering techniques*, such as the one proposed in [5a]. This technique produces clusters of pages on the basis of a measure of coupling of interconnected pages that associates different weights to different types of relationship (*Link, Submit, Redirect, Build, Load_in_Frame*, *Include*) between pages. Once clusters have been defined and use cases have been associated to each of them, the set of pages included in each cluster will make up the item to be tested. For each use case a decision table can be defined to drive integration testing. Such a decision table can be derived from the ones defined for the unit testing of the single pages included in the cluster.

The second black box approach for Web application testing considered in this section exploits *Finite State Machines* (FSM) for modeling software behavior and deriving test cases from them [1]. This approach explicitly takes into account the state dependent behavior of Web applications, and proposes specific solutions for addressing the problem of state explosion.

The process for test generation comprises two phases: in the first phase, the Web application is modeled by a hierarchical collection of FSMs, where the bottom level FSMs are formed by Web pages and parts of Web pages, while a top level FSM represents the whole application. In the second phase, test cases are generated from this representation.

The model of the Web application is obtained in more steps: first, the application is partitioned into clusters, that are collections of Web pages and software modules that implement some logical function. This clustering task is made manually and is, of course, subjective. In a second step, Web pages that include more than one HTML form, each of which is connected to a different back-end software module, will be modeled as multiple Logical Web Pages (LWP), in order to facilitate testing of these modules. In the third step, a Finite State Machine will be derived for each cluster, starting from bottom-level clusters containing only modules and Web pages (no clusters), and therefore aggregating lower-level FSMs into a higher-level FSM. Ultimately, an Application FSM (AFSM) will define a finite state model of the entire Web application. In each FSM,

nodes will represent clusters and edges will represent valid navigation among clusters. Moreover, edges of the FSMs will be annotated with inputs and constraints that may be associated with the transitions. Constraints on inputs, for instance, will indicate if input data are optional and their eventual input order. Information will be also propagated between lower-level FSMs.

Annotated FSMs and aggregate FSMs are thus used to generate tests. Tests are considered as sequences of transitions in a FSM and the associated constraints. Test sequences for lower level FSMs are combined to form the test sequences for the aggregate FSMs. Standard graph coverage criteria, such as *all nodes* and *all edges* are used to generate sequences of transitions for cluster and aggregate FSMs.

While the approach of Di Lucca et al., provides a method for both unit and integration testing, the one of Andrews et al., mainly addresses integration and system testing. Both approaches use clustering to identify groups of related pages to be integrated, even if in the second one the clustering is made manually, and this may limit the applicability of the approach when large size applications are tested.

The second method can be classified as gray box rather than pure black box technique. Indeed, test cases are generated to cover all the transitions among the clusters of LWP, and therefore the knowledge of the internal structure of the application is needed.

5.3. Gray box testing strategies

Gray box testing strategies combine black box and white box testing approaches to design test cases: they aim at testing a piece of software against its specification but using some knowledge of its internal workings.

Among the gray box strategies we will consider the ones based on the collection of user session data. These methods can be classified as gray box since they use collected data to test the behavior of the application in a black box fashion, but they also aim at verifying the coverage of any internal component of the application, such as page or link coverage.

Two approaches based on user session data will be described in the following.

5.3.1. User session based testing

Approaches based on data captured in user sessions collect user interactions with the Web server and transform them into test cases using some strategy.

Data to be captured include clients' requests expressed in form of URLs and name-value pairs. These data can be obtained from the log files stored by the Web servers, or by adding some script module on the requested server pages that captures the name-value pairs of exchanged parameters. Captured data about user sessions can be transformed into a set of HTTP requests, each one providing a separate test case.

The main advantage of this approach is the possibility of generating test cases without analyzing the internal structure of the Web application, thus reducing the costs of finding inputs. In addition, generating test cases utilizing user session data is less dependent on the heterogeneous and fast changing technologies used by Web applications, which is one of the major limitations of white box testing techniques. However, it can be argued that the effectiveness of user session techniques depends on the set of user session data collected: the wider this set, the greater the effectiveness of the approach to detect faults, but the wider the user session data set, the greater the cost of collecting, analyzing and storing data. Therefore there is a tradeoff between test suite size and fault detection capability.

Elbaum et al. [10] propose a user session approach to test a Web application and present the results of an empirical study where effectiveness of white box techniques and user session ones were compared.

In the study, user session collected data consist of sequences of HTTP requests made by users. Each sequence reports the pages (both client and server ones) the user visited together with the data he/she inputted, as well as the data resulting from the elaboration of requests the user made.

The study considered two implementations of the white box testing approach proposed by Ricca and Tonella [22], and three different implementations of the user session approach: a first implementation that transforms each individual user session into a test case, a second one that combines interactions from different user sessions, and a third one that inserts user session data into a white box testing technique. The study explored: the effectiveness of the techniques in terms of the fault-detection they provide, the cost-effectiveness of user-session based techniques, and the relationship between the number of user sessions and the effectiveness of the test suites generated based on those sessions' interactions. As a general result, the effectiveness of white box and user session techniques were comparable in term of fault detection capability, even if the techniques showed to be able to find different types of faults: in particular, user-session techniques were not able to discover faults associated with rarely entered data. The experiment also showed that effectiveness of user session techniques improves as the number of collected user sessions increased. However, as the authors recognized, the growth of this number puts additional challenges to the cost of collecting and managing sessions, such as the problem of finding an oracle to establish the expected output of each user request. The possibility of using reduction techniques, such as the one described in [13], is suggested by the authors as a feasible approach for reducing test suite size, but the applicability of such an approach still needs to be explored. A second empirical study carried out by the same authors and described in [11] essentially confirmed the results of the first experiment.

Sampath et al. [24] have explored the possibility of using *concept analysis* for achieving scalability in user-session

based testing of Web applications. Concept analysis is a technique for clustering objects that have common discrete attributes: it is used in [24] to reduce a set of user-sessions to a minimal test suite, that still represents actual executed user behavior. In particular, a user session is considered as a sequence of URLs requested by the user, and represents a separate use case offered by the application. Starting from an original test suite including a number of user sessions, this test suite is reduced finding the smallest set of user sessions that cover all of the URLs of the original test suite, while representing the common URL subsequences of the different use cases represented by the original test suite. This technique enables an incremental approach that updates the test suite on-the-fly, by incrementally analyzing additional user sessions. The experiments carried out showed the actual test suite reduction achievable by the approach, while preserving the coverage obtained by the original user sessions' suite, and with a minimal loss in fault detection. The authors have developed a framework that automates the entire testing process, from gathering user sessions through the identification of a reduced test suite to the replay of that test suite for coverage analysis and fault detection. A detailed description of this framework can be found in [25].

6. Testing the interaction between a Web application and the browser

A user, while navigating through a Web application, in addition to following the Web application links to reach a target page, uses the browser buttons, i.e., the back(ward), forward or reload buttons, to re-display some of the pages already visited along the navigation. The usage of these browser elements may negatively affect the navigation, because they might introduce some inconsistencies (or violate any functional/not-functional requirements of the application). For example:

- The user fills up a form, clicks the submit button and reaches the next page; if he/she goes back using the *back(ward)* button, then the inputted data might not be displayed again in the form;
- When using a Web-mail system, the user selects a message in the incoming list, then the text of that message is displayed; if the user returns to the incoming list page using the back button, the previously displayed message is not checked as "*read*";
- If a user reaches a page containing an access counter, and then he/she uses just the back and forward buttons to re-display the page, then the value of the counter might not be incremented.[1]

[1] This example does not consider page counters counting once more requests coming from the same IP address in the same time interval.

Some inconsistencies, as well as the un-satisfying of any Web application requirement may be encountered in all of the above cases. Of course, all possible inconsistencies are to be found with respect to the Web application specifications. For the first example above mentioned (filling a form), such a behavior can be just the one specified for that particular functionality, i.e., the specifications said that forms must be cleared in case of backward navigation performed using the browser button.

To avoid that the usage of browser features may negatively affect the navigation of a Web application, such features should be worth of consideration when testing the behavior of the Web application. In particular, each specified functional behavior should be tested by considering it together with each possible browser feature.

In [7] a model and an approach has been proposed to test the interaction between a Web application and browser buttons: the browser is modeled by a state chart diagram where each state is defined by the page displayed and by the state of the Back/Forward buttons, while user actions on page links or browser buttons determine the state transitions. Test cases are derived to satisfy a defined coverage criterion (e.g. All states, All transitions, All transition k-tuples, All round-trip paths) on the flattened state chart describing the possible interaction with the browser buttons that a user may trigger by that test case.

7. Tools for Web application testing

The tools used to support the testing activities may significantly affect the effectiveness of the testing process itself. In general, testing tools are able to automate some of the tasks required by the process, such as test case generation, test case execution, and evaluation of test case results. Moreover, testing tools may support the production of useful testing documentation and provide a configuration management of it.

In the last years, a large variety of tools for supporting Web application testing has been proposed. A list of more than 200 both commercial, and free-on-the Web testing tools for Web applications is presented in [15]. Most of them, however, have been designed to carry out either performance and load testing, or security testing, or they implement link and accessibility checking and HTML validation. As to the functional testing, existing tools main contribution is limited to manage test case suites manually created, and to match the test case results with respect to a manually created oracle.

Therefore, greater support to automatic test case generation would be needed to enhance the practice of testing Web applications.

Existing testing tools can be classified according to the type of testing they are able to support. Table 8 lists six main categories of testing tools and their descriptions.

Tools belonging to categories (a), (b) and (e) can be used to support non-functional requirement testing, while tools from (c) and (d) categories are more oriented to verify the

Table 8
Categories of Web application testing tools

Category	Description
a	Load, performance and stress test tools
b	Web site security test tools
c	HTML/XML Validators
d	Link checkers
e	Usability and accessibility test tools
f	Web functional/regression test tools

conformance of a Web application code to syntactical rules, or the navigability of its structure. This functionality is often offered by Web site management tools, used to develop Web sites and applications. Tools from the last category support the functionality testing of Web applications and they include, besides capture and replay tools, other tools supporting different testing strategies such as the one we analyzed in Section 5.

Focusing on this last category, the main characteristics that such a testing tool should met are discussed in the following, where main differences from tools usable for traditional applications testing will be also highlighted.

The services that a generic tool designed to support Web application functionality testing should include:

- *Test Model Generation*: to produce an instance of the desired/specified test model of the subject application. This model may be either one of the models already produced along the development process, and the tool would have just to import it, or it may be produced by reverse engineering the application code;
- *Code Instrumentation*: to instrument automatically the code of the Web pages to be tested, by inserting probes that automatically will collect data about test case execution;
- *Driver and Stub Generation*: to produce automatically the code of the Web pages implementing driver and stub modules, needed for test case execution;
- *Test Case Management*: to support test case design and testing documentation management; utilities for automatic generation of the test cases would be desirable;
- *Test Result Analysis*: this service will analyze and automatically evaluate test case results;
- *Report Generation*: this service will produce adequate reports about analysis result, such as coverage reports about the components exercised during the test.

Of course, some services offered by such a tool, such as Driver and Stub Generation, as well as Code Instrumentation and Test Model Generation, will be more depending on the specific technologies used to implement the Web application, while other ones will be largely independent on the technologies. As an example, different kind of drivers and stubs will have to be generated for testing client and server Web pages as the technology (e.g. the scripting languages used to code the Web pages) affects the way drivers and stubs will be developed. In general, the driver of a client page will have the responsibility of loading the client page into a browser, where it will be executed, while the driver of a server page will require the execution of the page on the Web server. Stubs of a client page will have to simulate the behavior of pages that are reachable from the page under test by hyperlinks, or whose execution on the Web server is required by the page. Stubs of a server page will have to simulate the behavior of other software components whose execution is required by the server page under test. Of course, specific approaches will have to be designed to implement drivers and stubs for Web pages created dynamically at run time.

Depending on the specific technology used to code the Web pages of the application, different Code Instrumentation components will have to be implemented too. Code analyzers, including different language parsers, will have to be used to identify automatically the points where probes are to be inserted in the original page code.

Analogously, the Test Model generator component that will have to reverse engineering the application code for generating the test model will be of course largely dependent on the technologies used to implement the application. Code analyzers will be required in this case too.

The other modules of such a generic tool will be less affected by the Web application technologies, and can be developed as in the case of traditional applications.

8. Future trends in testing Web-based applications

With the wide diffusion of the service-oriented software paradigm, the more recent Web based applications are being developed as Web services, as well as many 'legacy' Web applications and legacy systems are being migrated towards Web services. Web services introduce new peculiarities in the context of Web applications, raising new interesting testing issues and questions, such as those regarding testing models and strategies.

For example, testing models will have to take into consideration new types of software unit (such as SOAP and WSDL modules) and represent them and their relationships with other units, in a suitable and effective way.

Of course, for each possible testing level, testing strategies, methods and approaches will need to be adapted for effectively testing Web services: for example, new methods have to be defined to test SOAP and WSDL modules and new integration strategies are needed to aggregate together sets of Web service components.

In the case of Web services, a test suite can be used as a contract between the service provider and the service consumer, as well as the various stakeholders (i.e. providers, users, third-parties) could execute the test suite to check whether the service evolution meets the contract. Therefore the availability of test suites provided together with the service as a test facet, such as a XML-encoded test suite, could be valuable for the potential clients of that service.

Besides issues regarding Web services testing, more other issues will have to be considered in the next future. The

following list is intended to provide just some suggestions about the main topics that should be addressed in the field of Web based application testing:

- To provide greater support (i.e. methods and tools) for dynamic analysis of Web applications. Indeed, the most of current Web applications are dynamic ones (i.e. many of their components, in particular the client side components, are generated at run time) and then only a dynamic analysis of the application can provide a deep and correct knowledge of it.
- To provide greater support to the automatic generation of test cases. Testing techniques that exploit user sessions data and genetic algorithms for analysing these data should be investigated for obtaining effective and efficient solutions to this issue.
- To exploit mutation testing to validate test case suites; in the case of Web based applications, new and appropriate mutation operators should be defined.
- To develop and validate testing approaches based on new technologies, such as software agent based ones [21], that may better fit to the characteristics of dynamic behaviors, novel control and data flow mechanisms of Web applications.
- To exploit statistical testing to reduce the testing effort by focusing on those parts of the Web application (user functionalities and the components implementing them) that are most frequently requested/executed. Web based applications are usually used by a very large number of users, then collecting and analysing data from user sessions could drive the identification of the most critical components of the application in order to focus testing efforts on these components [18].
- To carry out empirical studies to verify and validate the effectiveness and efficiency of the existing testing approaches as well as to understand which kind of failures are better uncovered by each approach (i.e. which approach is better suitable to identify each type of failure) and how to integrate different approaches to improve their effectiveness and efficiency. In [23b] there is a first contribution about this issue.

A final consideration is about the testing process. Testing processes typically used in the development of traditional applications are not very suitable for Web applications that usually are developed without a well defined process or by a very flexible and agile one. All that would suggest that 'agile' testing process should be defined and experimented for Web applications, or that appropriate testing driven development methods could be defined for such applications.

9. Conclusions

The very large number of users characterizing Web applications and the strategic value of the services they offer, make the verification of both non-functional and functional requirements of a Web application a critical issue.

While new and specific approaches must be necessarily used for the verification of non-functional requirements (see the problems of security or accessibility testing that are specific for Web applications), most of the knowledge and expertise in the field of traditional application testing may be reused for testing the functional requirements of a Web application.

In this paper, we have reported main differences and points of similarities between testing a Web application and testing a traditional software application. We considered testing of the functional requirements with respect to four main aspects, i.e., testing scopes, test models, test strategies, and testing tools. The main contributions on these topics presented in the literature have been taken into account to carry out this analysis.

The main conclusion is that all the testing aspects that are directly dependent on the implementation technologies have to be deeply adapted to the heterogeneous and 'dynamic' nature of the Web applications, while other aspects may be reused with a reduced adaptation effort. This finding also remarks that further research efforts should be spent to define and assess the effectiveness of testing models, methods, techniques and tools that combine traditional testing approaches with new and specific ones. Moreover, as additional future trends, we expect that new relevant issues will arise in the field of Web services testing, as well as a new challenge will consist in the introduction of 'agile' methods in Web application testing processes for improving their effectiveness and efficiency.

References

[1] A.A. Andrews, J. Offutt, R.T. Alexander, Testing Web applications by modeling with FSMs. Software Systems and Modeling, Springer-Verlag Ed., 2005, 4(2).
[2] A. Bangio, S. Ceri, P. Fraternali, Web modeling language (WebML): a modeling language for designing Web sites, in: Proceedings of the Ninth International Conference on the WWW (WWW9). Elsevier: Amsterdam, Holland, 2000, pp. 137–157.
[3] R.V. Binder, Testing Object-Oriented Systems-Models, Patterns, and Tools, Addison-Wesley, Boston, MA, USA, 1999.
[4] J. Conallen, Building Web Applications with UML, Addison-Wesley Publishing Company, Reading, MA, 2000.
[5] (a) G.A. Di Lucca, A.R. Fasolino, U. De Carlini, F. Pace, P. Tramontana, Comprehending Web applications by a clustering based approach, in: Proceedings of 10th Workshop on Program Comprehension, IEEE Computer Society Press, 2002, pp. 261–270.
(b) G.A. Di Lucca, M. Di Penta, A.R. Fasolino, An approach to identify duplicated Web pages, in: Proceedings of the 26th Annual International Computer Software and Applications Conference, IEEE Computer Society Press, Los Alamitos (CA), 2002, pp. 481–486.
[6] G.A. Di Lucca, A.R. Fasolino, F. Faralli, U. De Carlini, Testing Web applications, in: Proceedings of International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos (CA), 2002, pp. 310–319.
[7] G.A. Di Lucca, M. Di Penta, Considering browser interaction in Web application testing, in: Proceedings of the Fifth IEEE International Workshop on Web Site Evolution, IEEE Computer Society Press, Los Alamitos, CA, 2003, pp. 74–83.

[8] G.A. Di Lucca, A.R. Fasolino, P. Tramontana, Reverse Engineering Web Applications: the WARE Approach; Journal of Software Maintenance and Evolution: Research and Practice. John Wiley and Sons Ltd., 2004, 16: 71–101.

[9] G.A. Di Lucca, M. Di Penta, Integrating static and dynamic analysis to improve the comprehension of existing Web applications, in: Proceedings of Seventh IEEE International Symposium on Web Site Evolution, IEEE Computer Society Press, Los Alamitos, CA, 2005, pp. 87–94.

[10] S. Elbaum, S. Karre, G. Rothermel, Improving Web application testing with user session data, in: Proceedings of International Conference on Software Engineering, IEEE Computer Society Press, 2003, pp. 49–59.

[11] S. Elbaum, G. Rothermel, S. Karre, M. Fisher, Leveraging user-session data to support Web application testing, IEEE Transactions on Software Engineering 31 (3) (2005) 187–202.

[12] J. Gomez, C. Canchero, O. Pastor, Conceptual modeling of device-independent Web applications, IEEE Multimedia 8 (2) (2001) 26–39.

[13] M.J. Harrold, R. Gupta, M.L. Soffa, A methodology for controlling the size of a test suite, ACM Transactions on Software Engineering and Methodology 2 (3) (1993) 270–285.

[14] E. Hieatt, R. Mee, Going faster: Testing the Web application, IEEE Software 19 (2) (2002) 60–65.

[15] R. Hower, Web site test tools and site management tools. Software QA and Testing Resource Center. <http://www.softwareqatest.com/qatWeb1.html>, 2005 (accessed June 5, 2005).

[16] IEEE Std. 610.12-1990, Glossary of Software Engineering Terminology, in Software Engineering Standard Collection,1990, IEEE CS Press, Los Alamitos, California.

[17] T. Isakowitz, A. Kamis, M. Koufaris, Extending the capabilities of RMM: Russian dolls and hypertext, in: Proceedings of 30th Hawaii International Conference on System Science, Maui, HI, 1997, (6), pp. 177–186.

[18] C. Kallepalli, J. Tian, Measuring and modeling usage and reliability for statistical Web testing, IEEE Transaction on Software Engineering 27 (11) (2001) 1023–1036.

[19] C. Liu, D.C. Kung, P. Hsia, C. Hsu, Object-based data flow testing of Web applications, in: Proceedinds of the First Asia-Pacific Conference on Quality Software, IEEE Computer Society Press, Los Alamitos (CA), 2000, pp. 7–16.

[20] H.Q. Nguyen, Testing Applications on the Web: Test Planning for Internet-Based Systems, John Wiley & Sons, Inc., 2000.

[21] Y. Qi, D. Kung, E. Wong, An agent-based testing approach for Web applications, in: Proceedings of the 29th Computer Software and Applications Conference – Workshop Papers and Fast Abstract – Workshop on Quality Assurance and Testing of Web-based Applications, IEEE Computer Society Press, Los Alamitos (CA), 2005, pp. 45–50.

[22] F. Ricca, P. Tonella, Analysis and testing of Web applications, in: Proceedings of the International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos (CA), 2001, pp. 25–34.

[23] (a) F. Ricca, P. Tonella, A 2-layer model for the white-box testing of Web applications, in: Proceedings of the Sixth IEEE Workshop on Web Site Evolution, IEEE Computer Society Press, Los Alamitos, CA, 2004, pp. 11–19.
(b) F. Ricca, P. Tonella, Web testing: a roadmap for the empirical research, in: Proceedings of the Seventh IEEE International Symposium on Web Site Evolution, IEEE Computer Society Press, Los Alamitos, CA, 2005, pp. 63–70.

[24] S. Sampath, V. Mihaylov, A. Souter, L. Pollock, A Scalable approach to user-session based testing of Web applications through concept analysis, in: Proceedings of the 19th International Conference on Automated Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, 2004, pp. 132–141.

[25] S. Sampath, V. Mihaylov, A. Souter, L. Pollock, Composing a framework to automate testing of operational Web-based software, in: Proceedings of the 20th International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, CA, 2004, pp. 104–113.

[26] D. Schwabe, R.M. Guimaraes, G. Rossi, Cohesive design of personalized Web applications, IEEE Internet Computing 6 (2) (2002) 34–43.

[27] Web content accessibility guidelines 2.0 (2005), <http://www.w3.org/TR/WCAG20> (accessed June 5, 2005).