A Practical Web Testing Model for Web Application Testing[*]

Zhongsheng Qian
School of Computer Engineering and Science, Shanghai University, Shanghai, 200072, China; School of Information Technology, Jiangxi University of Finance and Economics, Nanchang, 330013, China
changesme@163.com

Huaikou Miao
School of Computer Engineering and Science, Shanghai University, Shanghai, 200072, China
hkmiao@shu.edu.cn

Hongwei Zeng
School of Computer Engineering and Science, Shanghai University, Shanghai, 200072, China
zenghongwei@shu.edu.cn

Abstract

As an important method to ensure the quality of Web applications, Web testing attracts more and more attentions in the academic community and industrial world. Testing Web applications raises new problems and faces very high challenges. This work proposes a Web testing model for Web application testing. It starts from constructing the PFD (Page Flow Diagram) of the Web application. An algorithm is then designed to derive a PTT (Page Test Tree) from the PFD. From the PTT, a test translator is employed to extract the path expression, in order to generate all the test paths and then translates them into a test specification in XML syntax, which is the input of test engine. The test engine generates test cases and then executes them, and finally produces test report. The model presented is an important supplement to the existing Web application testing.

1. Introduction

Web applications are among the fastest growing classes of software systems today. These Web applications are being used to support a wide range of important activities: business transactions such as product sale and distribution, scientific activities such as information sharing and proposal review, and medical activities such as expert system-based diagnoses. Given the importance of such applications, bad Web applications can have far-ranging consequences on businesses, economies, scientific progress, health, and so on. Therefore, all the entities of a Web application must be tested thoroughly to ensure that the application is reliable and meets its original design specifications. Testing Web applications rises to the challenge.

Testing aims at finding errors in the tested object and giving confidence in its correct behavior by executing the tested object with selected input values. Web applications typically undergo maintenance at a faster rate than other software systems and this maintenance often consists of small incremental changes [1]. To accommodate such changes, Web testing approaches must be automatable and test suites must be adaptable. However, Web applications raise important and challenging test issues that cannot be solved directly by existing test techniques for conventional programs [2, 3, 4]. One of the key issues is the unexpected state change via the browser Back button or direct URL entry in the browser. We test the navigation between Web pages. The presentation details of the Web, such as size, position and color of page elements, although they are also of importance to the human computer interaction, will be out of scope of this paper. In testing, the set of Web pages within the Web application that are of interest is first considered, then the testing scope of the Web application is limited to any Web page that belongs to the set or is directly linked by any Web page belonging to the set. Web

[*] This work is supported by National High Technology Research and Development Program of China (863 Program, grant no. 2007AA01Z144), National Natural Science Foundation of China (grant no. 60673115) and National 973 Program of China (grant no. 2007CB310800).

pages that lead into the interest set should not be considered, for it is impossible to identify which Web pages, if any, lead into the interest set. For the sake of generality, we refer to both page hyperlinks and page buttons (such as submit, reset) as links (a hyperlink can be also seen as a button in textual style). Our testing process consists of following a path down a page test tree (edges representing links, nodes representing pages) until the end. The key difference between the presented approach and earlier approaches is that the work here proposes the construction of a tree instead of a graph thus avoiding the scalability problem.

The remainder of this paper is organized as follows. Section 2 introduces our proposed test path generation approach to Web application testing. In this section, an extended example is also demonstrated. Section 3 presents a Web testing model, which details the testing process, followed by a survey of related work in Section 4. Section 5 draws some concluding remarks and highlights the future work.

2. Generating test paths

Web testing is an effective technique to ensure the quality of Web applications. Unfortunately, it is hard to directly employ the traditional test theories and methodologies because of the particularities and complexities of Web applications. When browsing a Web application, the user traverses among various pages. The user leaves the state when some event occurs and this event is initiated by the user (clicking on the hyperlink or pressing the submit button) or by the system (being redirected to some page after some time). Due to the complexities mentioned, we must provide a practical approach to generating test paths along which to conveniently testing Web applications.

2.1. Test path generation approach

A Web application presents several different options to users, and the individual user chooses a particular sequence at the time of browsing. In addition, the users do not only browse, but also activate operations and transactions. Through hyperlinks, users can traverse among the pages of a Web application. However, a dangling hyperlink or an unreachable page can bother users. Moreover, the structure of a Web application is often complicated and difficult to maintain. To ensure whether or not the page flows are appropriate and meet the requirements, a PFD (Page Flow Diagram) is employed. The PFD intuitively reflects the relationship among pages of a Web application.

A PFD is denoted by a triple (P, L, F) where P denotes a set of pages including the pages within the Web application and the directly linked pages by them, L ($\subseteq P \times P$) denotes a set of links and F ($\subseteq P \times L \times P$) denotes a set of page flows. A PFD of an example Web application is shown in Figure 1 (the first link entering into the default page is visualized by a dashed arrow).

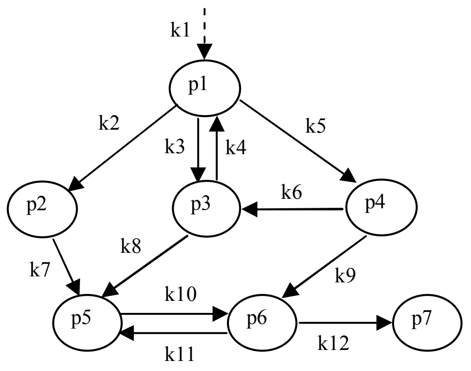

Figure 1. The PFD of an example Web application

The PFD is a graphic structure, which brings the problems that we often cannot verdict the termination of a path and the path may be cyclic, as will complicate the testing process, thus, driving the testing process using the PFD is difficult and unwieldy. Therefore, A PTT (Page Test Tree), which is based on the PFD is used. From the PTT, shorter paths can be generated than the PFD without the loss of page and link coverage. A PTT is a spanning tree constructed from a PFD, and the construction algorithm is given in Figure 2. In the algorithm, we employ two tables FIRST and SECOND. FIRST is a table that contains the page (node) identifiers, which are unmarked; SECOND is a table that contains the page (node) identifiers, which have already been marked. FIRST and SECOND are both empty initially. Note that, page identifiers and their copies are allowed to be in FIRST at the same time.

```
Algorithm 1. PFD2PTT
Input: a PFD
Output: a PTT derived from the PFD
begin
    (1) add the initial page identifier of the PFD into
        FIRST;
    (2) if FIRST is empty, then go to (6);
    (3) select the first page identifier denoted by pid from
        FIRST. If pid is within SECOND, then go to (5).
        Otherwise, add it into the end of SECOND;
    (4) if pid is linking to other pages, then
        • if some of the other page identifiers are
          within FIRST or SECOND, then generate their
          copies;
        • retain the links between pid and the other
          pages (or their copies) of the PFD, and
        • add the other page identifiers (or their copies)
          into the end of FIRST;
    (5) delete pid from FIRST and then go to (2);
    (6) output the derived PTT, which is the PFD with
        only the retained links.
end.
```

Figure 2. An algorithm deriving a PTT from a PFD

Suppose that there are n links in a PFD, we can derive a PTT in (n+1) steps using the algorithm. Figure 3 shows the PTT derived from the PFD in Figure 1, where the nodes and edges represent the pages and links in the PFD respectively (Note that the nodes underlined are the corresponding copies). The pages in FIRST and SECOND, as shown in table 1, are changed during the process of generation using algorithm PFD2PTT. In table 1, each page underlined is a copy of its corresponding page derived in one or more previous steps. Ø denotes that FIRST or SECOND is empty.

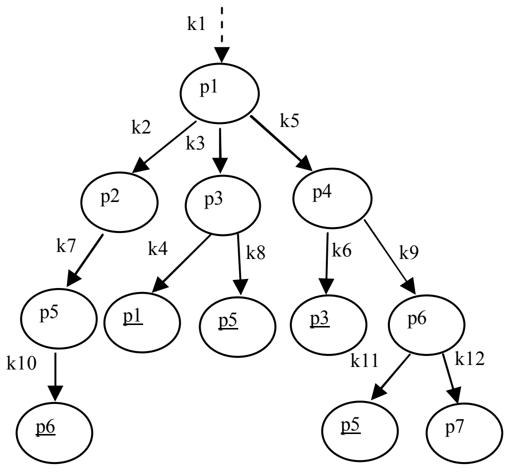

Figure 3. A PTT derived from the PFD in figure 1

In fact, the PTT is a tree expressing the successive relationship among pages.

Definition 1. Successor function И: $P \rightarrow 2^P$ defines the set of pages directly linked by each page. If a page p connects no other page, then И(p) = Ø. If p1 is linked by p2, then p2∈И(p1), p1 is the *previous* page of p2 and p2 is the *next* page of p1. Reflective transitive closure И* gives the successive relationship among pages: we have $И^*(p) = \cup_{i\geq 0} И^i(p)$ such that $И^0(p) = \{p\}$ and $И^1(p) = И(p)$. For $\forall$ i≥1, $И^{i+1}(p) = \cup_{q \in И(p)} И^i(q)$. If p2∈И*(p1), then p1 is the *precedent* page of p2 and p2 is the *subsequent* page of p1.

Definition 2. In a PTT, the initial page (node) is called *root page (node)*, the link entering into the *root page* is called *root link,* the leaf page (node) is called *tail page,* the link entering into the *tail page* is called *tail link,* and each of all the other pages (links) is called *branch page (link).*A *tail page* may also be a *branch page* or a *root page*.

Definition 3. A path is a possible sequence of links starting from the *root link*, and ending in any link during navigation. A path is denoted by k1→k2→…→kn where ki is a link (1≤i≤n, n>0).

Definition 4. A test path is a path from the *root link* of the tree to a *tail link*.

Table 1. Pages in FIRST and SECOND in each step using the PFD2PTT algorithm

step	FIRST	SECOND
1	p1	Ø
2	p2, p3, p4	p1
3	p3, p4, p5	p1, p2
4	p4, p5, $\underline{p1}$, $\underline{p5}$	p1, p2, p3
5	p5, $\underline{p1}$, $\underline{p5}$, $\underline{p3}$, p6	p1, p2, p3, p4
6	$\underline{p1}$, $\underline{p5}$, $\underline{p3}$, p6, $\underline{p6}$	p1, p2, p3, p4, p5
7	$\underline{p5}$, $\underline{p3}$, p6, $\underline{p6}$	p1, p2, p3, p4, p5
8	$\underline{p3}$, p6, $\underline{p6}$	p1, p2, p3, p4, p5
9	p6, $\underline{p6}$	p1, p2, p3, p4, p5
10	$\underline{p6}$, $\underline{p5}$, p7	p1, p2, p3, p4, p5, p6
11	$\underline{p5}$, p7	p1, p2, p3, p4, p5, p6
12	p7	p1, p2, p3, p4, p5, p6
13	Ø	p1, p2, p3, p4, p5, p6, p7

The PTT is used to establish path expressions [5], which are then translated into test specification by the test translator (see Section 3). A path expression is an algebraic representation of the paths in a tree. Variables in a path expression are links. They can be combined through operators + and *, associated respectively with selection and loop. Brackets can be used to group subexpressions. An example path expression in figure 3 is k1→k3→(k4+k8). In this path expression, a selection is encountered at k3, with k4 = (p3, p1) and k8 = (p3, p5). Computation of the path expression for a Web application can be performed by means of the Node-Reduction algorithm designed by B. Beizer [5]. Since the path expressions directly represent all test paths in the tree, they can be employed to generate sequences of links which satisfy any of the coverage criteria. To satisfy a given criterion, determining the minimum number of test paths from a path expression is in general a hard task. However, heuristics can be defined to compute an approximation of the minimum.

From the path expressions, test paths can be extracted automatically and then test cases (see section 3). The path expression for the entire PTT in figure 3 is k1→(k2→k7→k10+k3→(k4+k8)+k5→(k6+k9→(k11 +k12))). There are six test paths in this path expression because there exist six tail links in the PTT. These test paths are k1→k2→k7→k10, k1→k3→k4, k1→k3→k8, k1→k5→k6, k1→k5→k9→k11 and k1→k5→k9→k12. Among these test paths, it is possible to generate more than one test path from the root page to some tail page. The paths k1→k3→k8 and k1→k5→k9→k11 are, for example, two test paths for ensuring normal navigational behavior from p1 to p5. Note that the path k1→k2→k7 is not a test path from p1 to p5, since k7 is not a tail link in this path.

In the tree, all the paths from the *root link* to each *tail link* cover all the links, as is termed *all links coverage*. If all the nodes underlined and the corresponding edges pointing to them are removed from the tree, then we get the minimum number of test paths that cover all the pages under consideration, as is termed *all pages coverage*. The *all pages coverage* tests whether or not all the pages can be accessed. Obviously, the test paths of *all links coverage* pass the test paths of *all pages coverage*.

A Web application often contains numerous pages and links. Pages can be connected by various links in an interesting way, which complicates the Web application. And the PFD and PTT of the Web application will be too large to manage. So, exhaustive exploration could hardly be achieved for the non-trivial Web application. We divide the Web application into a set of modules according to its functional requirements, each of which can be considered as a sub Web application (subWA). The subWAs can be conquered respectively. Suppose that a WA is divided into m subWAs. For subWA WA_k, the PFD_k and PTT_k can be constructed. Thus, we obtain $\{PFD_1, PFD_2, \ldots, PFD_m\}$ and $\{PTT_1, PTT_2, \ldots, PTT_m\}$ respectively. Let P_i be the set of pages in subWA WA_i, P_j be the set of pages in subWA WA_j, $Q_i \subset P_i$ and $Q_j \subset P_j$, if each page in Q_i points to (or is pointed by) some page in Q_j, then a $PFD_{i.j}$ based on $Q_i \cup Q_j$ will be constructed, thus $PTT_{i.j}$ is easily built. So, we get a *tree-like page flow diagram* TPFD, in which the leaves are $PFD_1, PFD_2, \ldots, PFD_m$ for the subWAs, the branch nodes are formed by $PFD_{i.j}$ if there is a relationship between Q_i and Q_j ($1 \le i, j \le m$, $i \ne j$) (they may form a higher level if necessary), and the root is the PFD for the entire WA. Note that, the root PFD is a virtual node which is “realized” by all its sub-nodes. The *tree-like page test tree* TPTT is constructed in a similar way. In this way, our method treats the WA in a “divide and conquer” manner. Therefore, the testing of the entire WA can be divided into the testing of its subWAs, which is less complex and more controllable. The subWAs can be tested simultaneously on different machines by different testers. So, the testing is very flexible. This also restricts the state space explosion in a sense. In addition, TPFD and TPTT is a higher level of abstraction of the WA.

2.2. An extended example

Consider a typical miniature Web application that we have developed to demonstrate our approach, the SWLS (Simple Web Login System) (see its page flow diagram in figure 4). Starting at the first page (indicated by a dashed arrow, reasonably, a *blank* page can be used to request for the first page of a Web application), i.e., a *home page (p1)*, the user can enter into the *news page (p2)* to list the news by clicking on the *view* link, or enter into the *login page (p3) by pressing the login button*. In page *p3*, the user enters the *userid* and *password, and presses the submit button.* Upon this pressing, the *userid* and *password* are sent to the Web server for authentication. A *logged page (p4)* will be loaded if both *userid* and *password* are correct. On the contrary, an *error page (p7)* containing an error message is displayed if at least one of the submitted values for *userid* and *password* is wrong. From the *logged page*, it is possible to go to *info page (p5)* for secure information viewing by just clicking on the *browse* link. The user can click on the intra-page link *continue* to view the different parts of the same page *p5* if it is too long. A *logout page (p6)* will be displayed when the user presses the *exit* or *logout* button. Then, the user may come back to the *home page* for login again. Note that each time the *login page* is displayed, both the *userid* and *password* fields must be empty. According to the PFD2PTT algorithm, figure 5 is the corresponding PTT derived from the PFD of SWLS.

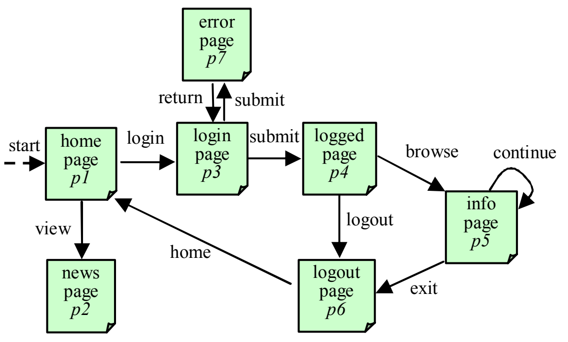

Figure 4. Page flow diagram of SWLS

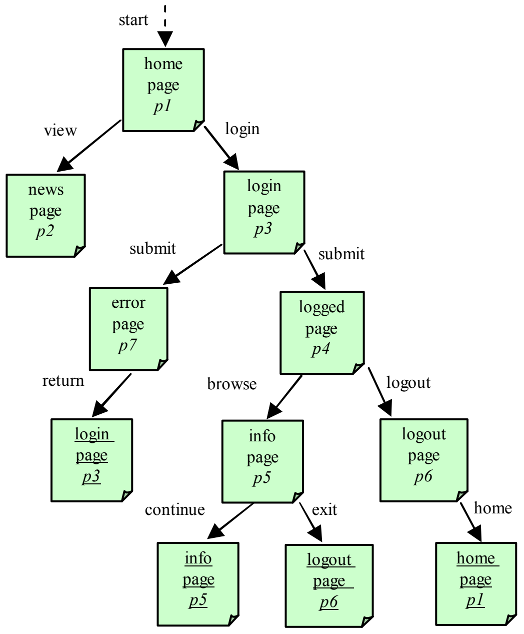

Figure 5. Page test tree of SWLS

The path expression based on the PTT of the SWLS is start→(view+login→(submit→return +submit→(browse→(continue+exit)+logout→home))). So, there are five test paths, which start from the home page. These test paths are:

- start→view
- start→login→submit→return
- start→login→submit→browse→continue
- start→login→submit→browse→exit
- start→login→submit→logout→home

3. A Web testing model

To test a Web application, test cases must be generated. Our test paths produced can be easily employed to construct test cases. A test case defined by our Web testing model (see figure 6) is one test path with user input values. So, a test path may be used to construct multiple test cases if only the tester provides different user input values for the test path. The user input values can be obtained by various existing approaches such as the user session data proposed by S. Elbaum et al. [6] or the input unit by J. Offutt et al. [7]. A test specification is a hierarchy of test suite, test case, and test step [8]. In our test specification, a test case is defined as a tree of test step. The links on a path represent test steps to be carried out successively. In other words, a *next* link represents a test step to be taken following the test step represented by its *previous* one. *Sibling* links represent alternative test steps in test paths. Our test specification is an extended version of the one described in [8], where the specification is depicted using XML. The test specification is based on request specification, response specification, and predicate definition. Request specification specifies a pattern of HTTP requests, while response specification specifies the assertions on the HTTP response generated from the HTTP request in the same test step, and, finally, the predicate definition specifies the assertions on the results of testing.

The following is a similar test specification for the last test path start→login→submit→logout→home of the Web application SWLS:

```
<testsuite name = “the SWLS test suite”>
    <testcase name = “the last test case”>
        <teststep name = “start”>
            the request and response specification of test
            step start
            <teststep name = “login”>
                the request and response specification of
                test step login
                <teststep name = “submit”>
                    the request and response specification
                    of test step submit
                    <teststep name = “logout”>
                        the request and response
                        specification of test step logout
                        <teststep name = “home”>
                            the request and response
                            specification of test step home
                        <teststep>
                    <teststep>
                <teststep>
            <teststep>
        </teststep>
    </testcase>
    ……
</testsuite>
```

The Web testing model extends the testing approach proposed by X. Jia, et al. [8]. At the beginning, the PFD is constructed through the commonly-used requiring and analyzing methods of Web applications. The test translator of the testing model extracts page relations from the PTT, which is derived from the PFD according the PFD2PTT algorithm, to build path expressions [5]. It then translates the path expressions into test script framework (of test specification). This framework defines the test suite, containing test cases. Every test case is a sequence of test steps. Each test step is for one page to be verified. Every test step defines HTTP requests, expected response and predicates with condition definitions (such as <not>, <and>, <or>, <match>, and <forall>). The test script framework does not contain values for input variables, since they will be added later. Some inputs have to be created by hand, including user ids and passwords. Other inputs are either automatically extracted from the HTML files or randomly generated. Moreover, the test script framework does not contain hard-coded expected output, that is to say, the Web testing model defines frameworks with empty response, since they will be filled manually by the tester.

The following is an example of the request and response specification of a possible test step for the login page p3 of the SWLS:

```
<request url = “http://mytestwebsite/login.asp”>
    <parameter name = “name” value = “computer”/>
    <parameter name = “password” value = “hello”/>
</request>
<response>
    <match op = “contains” regexp=false select =
     “/html/body” value = “Login Error!”/>
</response>
```

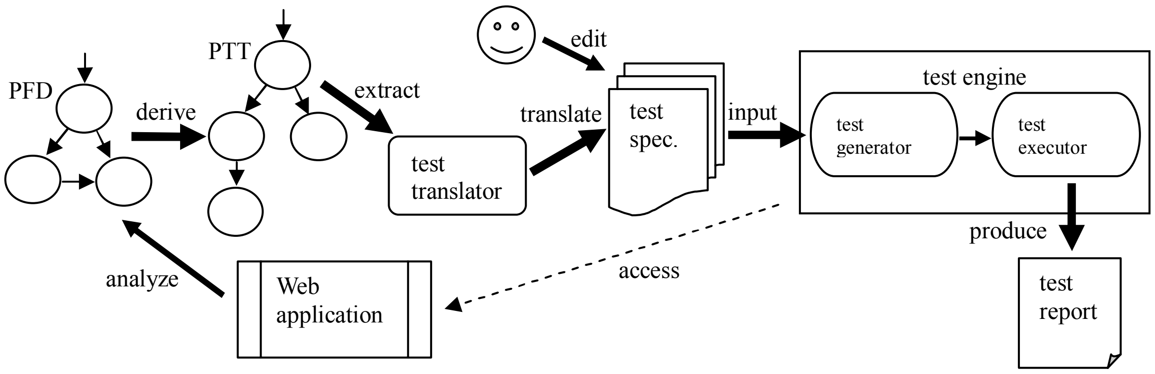

Figure 6. **A Web testing model**

After editing the necessary information of the test script framework, the tester gets a formal specification in XML syntax, which is the input to test engine. The specification contains templates of test cases. Test engine includes a test case generation engine (test generator) that is able to determine the test paths from the test specification, and to generate test cases from it, provided that a test criterion is specified. Generated test cases are sequences of links which, once executed, grant the coverage of the selected criterion. Then the test engine executes the test cases and validates its result against the test oracles specified as expected result. That is to say, test engine’s test executor can now provide the link sequences of each test case to the Web server, attaching proper inputs to each form. After the execution, test engine produces a test report summarizing the results of all test cases. For such evaluation, the tester opens the output pages on a Web browser and checks whether the output is correct for each given input. By the way, the test engine behaves as a Web client accessing the Web application.

4. Related work

A number of Web testing techniques for Web applications have been already proposed, each of which has different origins and pursuing different test goals for dealing with the unique characteristics of Web applications. Link testers traverse a Web application and verify that all hyperlinks refer to valid documents. Form testers create scripts that initialize a form, press each button, and type preset scripts into text fields, ending with pressing a submit button. Compatibility testers ensure that a Web application behaves properly within different Web browsers.

Object driven performance testing is proposed by B.M. Subraya and S.V. Subrahmanya [9]. They illustrated a new testing process that employs the concept of decomposing the behavior of a Web application into testable components. Different from theirs, our approach decomposes a Web application according to its functional requirements, not its behavior.

M. Benedikt et al. [10] built VeriWeb, a dynamic navigation test tool for Web applications. VeriWeb explores sequences of links in Web applications by nondeterministically searching action sequences, starting from a given URL. VeriWeb’s test is based on graphs where nodes are Web pages and edges are explicit HTML links, and the size of the graphs is controlled by a pruning process. Our approach is based on a PTT and each test case is defined as a tree of test step.

In [6], S. Elbaum et al. presented a method to use what they called *user session data* to generate test cases for Web applications. Instead of looking at the data kept in J2EE servlet session, their *user session data* is the input data collected and remembered from previous user sessions. The *user session data* is captured from HTML forms and includes name-value pairs. Experimental results from comparing their method with existing methods show that user session data can help produce effective test suites with very little expense. Our approach is flexible, and the user input data can be produced by various methods presented by existing research work.

A. Andrews et al. [11] proposed a method for deriving tests from FSMs. The method tries to use input constraints to restrict the state space explosion. Theoretically, Web applications can be completely modeled with FSMs, however, even simple Web applications can suffer from the problem of state space explosion. While our approach treats Web applications in a "divide and conquer" manner, which divides the Web application under test (or WAUT for short) into subWAs and we contend that each subWA will not raise the state space explosion problem if the WAUT is divided reasonably.

Existing tools can conduce to Web applications testing. An extensive listing of existing Web testing support tools is on a Web site maintained by Hower [12]. The list includes syntax validators, HTML/XML validators, link checkers, load/stress test tools, and regression test tools. These are all static validation and measurement tools, none of which supports functional testing or black-box testing. Recently, several automated Web test tools have been developed to support Web application testing [13, 14]. These tools are integrated with specific Web browsers to capture tests of user interactions with Web applications and create test scripts that can be replayed automatically to check the GUI components and verify the functionality of Web applications. However, these automated tools cannot provide structural and behavioral information of Web applications for testers. Thus, it is still ad hoc for testers to design test cases effectively, especially if the design documents are incomplete or missing.

Web-based analysis and test tools have been also developed that model the underlying structure and semantics of Web applications [15, 16, 17, 18] towards a white-box testing approach. While these white-box techniques enable the extension of path-based testing to Web applications, they often require identifying input data that will exercise the paths to be tested, which are not covered by test cases generated from functional specifications. In essence, these white-box approaches build system models from inspection of code, identify test requirements from those models, and require extensive human participation in the generation of test cases to fulfill those requirements. These approaches focus on the internal structural aspect and involve in the details of a Web application. While our approach concerns mainly the functional aspect towards a black-box testing (a functional test of some sort) at the page level of abstraction.

Although all these Web testing methods and tools exist, the Web testing model proposed in this work is still an important supplement to them.

5. Conclusions and future work

Many Web testing challenges were discussed in [2, 3]. Web applications testing is a specialization of software testing. Testing a Web application is challenging, not only for the mission critical tasks it carries out, but also for factors such as the security, the scalability, and the high dependency of the outputs on the Web browser and on the way a user interacts with the Web browser.

One of the main advantages of our Web testing approach is that it does not require accessing the source of the back-end software. To derive a PTT from a PFD, an algorithm is presented. Test paths can be generated from the PTT by constructing the path expression. A possible way of describing the possible test paths using XML is sketched. Test engine takes the test specification as input and then outputs test report. Two important concepts: *all links coverage* and *all pages coverage* are given. The *all pages coverage* expresses the minimum number of test paths that cover all the pages under consideration. An extended example from a Simple Web Login System (SWLS) is given to illustrate the proposed test path generation approach, which lays a promising foundation for the page flow testing methodology.

The testing phase is a very time-, cost- and computation-consuming development process; its usefulness may be limited by the very expensive manual user interaction. The key for useful testing is, of course, automation. A PTT can be derived from a PFD automatically according to the algorithm described above. The test translator then extracts path expressions from the PTT and translates the path expressions into test script framework (of test specification). Test engine reads and executes the test specification to produce a test report. On the whole, our Web testing approach is automatic except the manual construction of PFD and a little editing of the test script framework.

To generate a PFD, we integrate the traditional requiring and analyzing methods with the characteristics of Web applications. To establish a test case, an alternative is applying random inputs to the PFD until every link is traversed at least once. However, the path may contain loops and can be very long. Moreover, in order to achieve a test path that traverses each link once and only once, the PFD has to be an Euler graph. If it is not an Euler graph, the problem of constructing a test path with optimal length is NP-complete.

Some points that are not addressed, but are certainly beyond the scope of this paper, are the new breed of Web applications that utilize Ajax (Asynchronous JavaScript plus XML) and client-side html generation, thus doing away with the page-centric view, and pages that change their structure depending on the input. It may well be that a certain link only occurs on a page if a certain input was supplied. Next step is investigating more changeable issues in the new breed of Web applications to improve the proposed test path generation approach and developing a prototype tool to execute the Web testing model.

References

[1] E. Kirda, M. Jazayeri, and C. Kerer, et al., “Experiences in Engineering Flexible Web Services”, IEEE MultiMedia, 2001, 8 (1), pp. 58-65.

[2] G. A. Stout, “Testing a Website: Best Practices”, A Whitepaper, http://www.reveregroup.com.

[3] E. Hieatt, R. Mee, “Going Faster: Testing the Web Application”, IEEE Software, Mar. 2002, pp. 60-65.

[4] G. A. D. Lucca and A. R. Fasolino, “Testing Web-based Applications: The State of the Art and Future Trends”, *Information and Software Technology*, vol. 48, 2006, pp. 1172-1186.

[5] B. Beizer, “Software Testing Techniques”, 2nd edition, International Thomson Computer Press, 1990.

[6] S. Elbaum, S. Karre, and G. Rothermel, “Improving Web Application Testing with User Session Data”, In Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, May 2003, pp. 49-59.

[7] J. Offutt, Y. Wu, and X. Du, et al., “Bypass Testing of Web Applications”, In Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering, Saint-Malo, Bretagne, France, Nov. 2004.

[8] X. Jia, H. Liu and L. Qin. “Formal Structured Specification for Web Applications Testing”, 2003 Midwest Software Engineering Conference, Chicago, USA. June 2003.

[9] B.M. Subraya, S.V. Subrahmanya, “Object Driven Performance Testing of Web Applications”, The First Asia-Pacific Conference on Quality Software, HongKong, China, Oct. 2000.

[10] M. Benedikt, J. Freire, and P. Godefroid, “VeriWeb: Automatically Testing Dynamic Web Sites”, In Proceedings of 11th International World Wide Web Conference, Honolulu, HI, May 2002.

[11] A. Andrews, J. Offutt, and R. Alexander, “Testing Web Applications by Modeling with FSMs”, *Software and Systems Modeling*. 2004.

[12] Rick Hower, “Web Site Test Tools and Site Management Tools”, Software QA and Testing Resource Center, 2002, http://www.softwareqatest.com/qatweb1.html.

[13] Mercury Interactive, http://www.merc-int.com.

[14] E. Miller, “WebSite Testing”, http://www.soft.com/products/web/technology/websitetesting.html.

[15] C.-H. Liu, D. C. Kung, and P. Hsia, “Object-based Data Flow Testing of Web Applications”, The First Asia-Pacific Conference on Quality Software, HongKong, China, Oct. 2000.

[16] G. A. Di Lucca, A. R. Fasolino, and F. Faralli, et al., “Testing Web Applications”, In Proceedings of the IEEE International Conference on Software Maintenance, Montréal, QC, Canada, Oct. 2002, pp. 310-319.

[17] F. Ricca, P. Tonella, “Analysis and Testing of Web Applications”, In Proceedings of the 23rd International Conference on Software Engineering, Toronto, Ontario, Canada, 2001, pp.25-34.

[18] C.-H. Liu, D. C. Kung, and P. Hsia, et al., “Structural Testing of Web Applications”, In Proceedings of the 11th IEEE International Symposium on Software Reliability Engineering, San Jose, CA, Oct. 2000, pp. 84-96.