TestUml: user-metrics driven Web Applications testing

Carlo Bellettini, Alessandro Marchetto, Andrea Trentini
Dipartimento di Informatica e Comunicazione,
Università degli Studi di Milano
Via Comelico 39, 20135 Milano, Italy
{Carlo.Bellettini, Alessandro.Marchetto, Andrea.Trentini}@unimi.it

ABSTRACT

Web applications have become very complex and crucial, especially when combined with areas such as CRM (Customer Relationship Management) and BPR (Business Process Reengineering), the scientific community has focused attention to Web application design, development, analysis, and testing, by studying and proposing methodologies and tools. This paper describes techniques for semi-automatic test case definition and for *user*[1]-driven testing (based on statistical testing or coverage analysis) from Web applications reverse engineered UML models. These techniques are implemented as tools in the WAAT project. WebUml is a reverse engineering tool that generates class and state diagrams through static and dynamic Web application analysis. TestUml is a testing suite that uses generated models to define test cases, coverage testing criteria and also reliability analysis.

Keywords

Reverse Engineering, UML, Application Design Model, Testing, White-Box Testing, Stop Testing, Testing Coverage, Metrics

1. INTRODUCTION

Web applications quality, reliability and functionality are important factors because software glitches could block entire businesses and determine strong embarrassments. These factors have increased the need for methodologies, tools and models to improve Web applications (design, testing, and so on). Important factors for Web applications are "speed" (in technology change, content update and fruition), complexity, large dimensions and design/use maturity. Web applications are heterogeneous, distributed, and concurrent: their testing is not an easy task. Conventional methodologies and tools may not be adequate.

This paper focuses on Web applications testing of legacy Web applications where business logic is embedded into Web pages. Analyzed applications are composed by Web documents (static, active or dynamic) and Web objects ([4]).

[1]Throughout this paper, when not specified, "user" should be considered a Web application developer.

We describe techniques for semi-automatic test case definition based on random walks analysis, and describe how to use a set of criteria (software metrics such as structural coverage criteria or reliability software analysis derived criteria) as test stop criteria. Usually these criteria are used *a priori* to define test cases (such as [26],[17]), while our system uses them also at runtime to help the *user* in drive testing operations.

TestUml architecture is composed by a test case definition module, an execution module, and metrics computation and coverage calculation modules.

TestUml uses WebUml-generated UML models and defines a set of test cases. Then it executes them and generates test results and coverage level reports. With TestUml, the *user* may verify testing coverage status step-by-step (granularity is tunable) during test case iterations and may decide (on-the-fly) when to stop testing. Alternatively, the user may *a priori* define the testing coverage level so that TestUml executes test cases until this level is reached.

Web software is often developed without a formalized proces, and Web documents and objects are directly coded in incremental way. Often, new documents and objects are obtained by duplicating (inheritance by "copy&paste") and modifying existing ones. Web software life-cycle is very compressed, in the range of tree to six months, and during this life-cycle the allotted test period is often risible. Standard Web application tests are "functional" (input data test only) and "load stress" ones. The software community needs fast, effective, useful, and complete testing methodologies.

Software testing is composed by: test cases definition (structural, functional, statistical,...), data test definition (automatic, manual,...), objects test definition(unit, integration, system...), test cases execution (stub, driver, script generator), oracles definition (test cases result validation), test stop criteria (time, resource, coverage,...), and so on.

The test phase is a very time-, cost- and computational- consuming development phase, its usefulness may be limited by the very expensive manual *user* interaction. The key for useful testing is, of course, automation.

This paper is organized as follows. Section 2 introduces a review of existing work in Web applications modeling techniques and reverse engineering tools, and in particular in testing tools and methodologies. Section 3 details WebUml Web applications modeling and the model-recovery techniques used. TestUml input is represented by WebUml model output. Section 4 defines our test model, based on automatic test cases definition, and *user*-driven test stop criteria as coverage structural criteria or reliability model metrics measurement. Section 5 describes a simple case study of Web application testing. Finally, Section 6 concludes the presented work and describes future work.

2. RELATED WORKS

One of the best known design techniques for Web applications is [6] in which Conallen extends UML with stereotypes for Web design. Reverse engineering tools are: WARE [9], ReWeb [24],and Rational Rose Web Modeler [3], WebUml [4].

Currently available Web testing tools (e.g. [2],[1]) are usually classifiable as syntax validators, HTML/XML validators, link checkers, load/stress testing tools, and regression testing tools, i.e., they are focused on low-level (implementation bound) or non-functional aspects. Some of these tools are often integrated into Web Browsers to capture user gestures and replay them in testing scripts. These tools and methodologies cannot provide structural or behavioral test artifacts. Moreover, capture and replay testing represents a good compromise when a formal model is not available and the only implicit model is the *user*.

A different approach, more strictly related to this paper, is based on functional, structural and behavioral testing, such as: G.A.Di Lucca et al. [8]; Ricca and Tonella [26],[24]; and D.C.Kung,C.H.Liu and P.Hsia [17],[16],[15]. G.A.Di Lucca et al. propose object–oriented Web testing strategy in order to build a process (unit and integration testing) essentially based on functional criteria. The testing process is based on a high level representation of Web applications derived from a reverse engineering procedure available in the WARE tool [9]. D.C.Kung et al. propose an Object–Oriented Web Test Model that captures artifacts representing three aspects: object (entities of application); behavior (navigation and state dependent behaviors); and structure (flow control and data flow information). From described model, structural and behavioral test cases can be automatically derived to support the test process. Ricca and Tonella have developed ReWeb and TestWeb tools. ReWeb, for reverse engineering Web applications into UML model, performs several traditional source code analyses, and uses UML class diagrams to represent components and navigational features. ReWeb is a semi-automatic tool because an important role is played by the *user*, mainly for form filling input values. TestWeb uses extracted model information to test with white-box criteria based essentially on Web site validation paths. Other approaches are based on statistical models, for example [25] uses reverse engineering techniques to extract UML models, then it uses log files to create a usage model to be analyzed with Markov Models. [14] defines also statistic testing: it creates an application model from log files and then analyzes it with Unified Markov Models (UMMs). Other different approaches are presented in [23] and [22] by Offutt J. et al., in [23] Offutt analyses input validation testing, while [22] describes a specific Web model for testing purposes.

3. MODEL EXTRACTION

The Web application UML model we used is based on class and state diagrams. Class diagrams are used to describe the structure and components of a Web application. E.g., forms, frames, Java applets, HTML input fields, session elements, cookies, scripts, and embedded objects. Fig. 1 shows the class diagram[2] meta model. A WebUml generated model is an instance of the meta model. State diagrams are used to represent behaviors and navigational structures according to a model derived from [20]. This navigational structure is composed by client-server pages, navigation links, frames sets, form inputs, scripting code flow control, and other static and dynamic contents. The use of statecharts let us model relevant assets, such as active documents (i.e., composed by HTML and client side scripting code). In particular, the state diagram of an active document can define function calls flow of the scripting code (how the HTML code recall the scripting function), and some relevant behaviors-navigation dynamic information (e.g., dynamic links, frames, and so on).

[2]"0..*" cardinality should be considered when not specified.

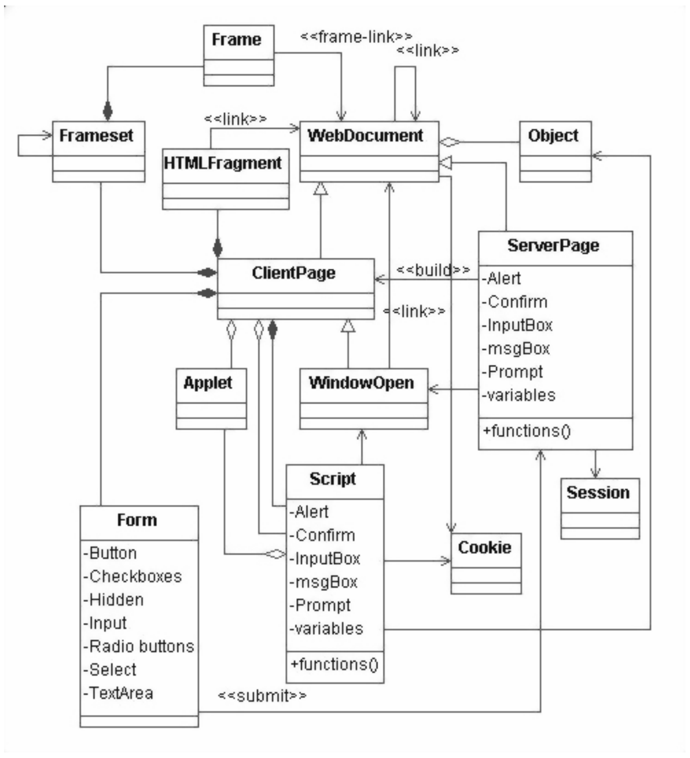

Figure 1: Class diagram meta model

The Web application reverse-engineered model[4] is based on static and dynamic analysis. The technique uses static methods derived from traditional source code analysis adapted to extract static and dynamic information from Web. Moreover a combined method based on static and dynamic analysis is used to define navigational structure and application behavior. We have paid particular attention to the server side dynamic aspects of Web applications, we analyzed it with a dynamic method based on application execution and on mutational analysis applied to source code [11]. This dynamic analysis is performed with the generation of a set of source code mutants, used into navigation simulation. Then, procedure results are analysed with traditional static source code techniques. The use of mutation lowers user interactions in the reverse engineering phase and let us define a more detailed description.

4. TESTUML ALGORITHM

Figure 2 shows the TestUml tool pseudocode. TestUml input is represented by Web applications UML model written in XMI, a Web server log file, and XML configuration files specifying Web server address, information about metrics and coverage use, and so on (Figure 2 row 3).

TestUml uses the model to define test cases based on random walks analysis (Figure 2 row 8), then analyses the log file to help the user define input data (Figure 2 row 11) and then asks the user to help compile the testing script with the remaining input data and expected output (Figure 2 row 13). Then, it executes the defined test case (Figure 2 row 14 and rows from 28 to 41), and calculates coverage measures (Figure 2 row 37). Finally, TestUml chooses test stop criterium: ask the user (based on generated reports)(Figure

```
1  Class TestUml
2
3  Function Testing(model[], logFile, cXml)
4   finalMetr[]=null
5   finalReport=''
6   infoModel[][]=Model_analysis(model[])
7   do
8    rWalk[]=Randow_Walk_paths_extraction(model[])
9    script=Write_Script_Testing(rWalk[])
10   if logFile !=null then
11   script=Log_input_add(logFile, script)
12   endIf
13   script=User_info_add(script)
14   dMetr[],report=Scp_Exe(script,infoModel[][],cXml)
15   finalMetr[]=add(Metr[],dMetr[])
16   finalReport=add(report,finalReport)
17   if cXml.whoStop=='user' then
18    output(dMetr[],finalMetr[],report,finalReport)
19    yn=ask_if_continue(finalReport)
20   else
21    if cXml.coverageTies[]==finalMetr[] then
22     yn='s';
23    endIf
24   endIf
25   endDo while(yn='n')
26 endFunction
27
28 Function Scp_Exe(script,infoModel[][],cXml)
29  do
30   rWalk[]=Path_Extraction(script)
31   if rWalk[]!=null then
32    do
33     step[]=Step_Extraction(rWalk[])
34     reportStep=Step_Exe(step[],cXml)
35     report=add(report,reportstep)
36    endDo while(step[]!=null)
37    dMtr[]=Coverage(infoModel[][],report)
38    resturn dMtr[],report
39   endIf
40  endDo while(rWalk[]!=null)
41 endFunction
42
43 endClass
```

Figure 2: TestUml pseudocode

2 rows 17, 18 and 19) or check configuration constraints (Figure 2 row 21).

When TestUml executes a script, it submits a set of requests to the Web server and saves every response. For every executed path, TestUml computes coverage level based on user-selected criteria and the UML application model.

Random walks testing ([19, 18, 27]) is used to define a set of paths from application graph where every path-step is randomly defined, so that every step and every path have the same probability to be chosen. TestUml performs random walks integration testing for Web application and computes testing coverage based on metrics measurements. Coverage measurements let the user define when to stop testing. User may interact with TestUml to decide when to stop testing, or may define coverage constraint levels in the configuration file to automatize coverage analysis. The user may operate TestUml in interactive mode. In the tool computes current coverage testing level and asks the user if he wants to stop testing after every test script execution. If the user has set coverage constraints, TestUml defines a test case, creates a testing script (with user help), executes the script, and then computes the current coverage level, it stops testing if coverage is greater than defined constraints.

TestUml uses random walks analysis to define test cases from the application UML model. Test case granularity (i.e., number of test cases executed before coverage computation) is tunable: single test or set of tests. *Single* is when TestUml helps the user in defining a single test case, executes it, computes the coverage defined with this test case and then asks the user whether to stop testing. *Set* granularity is when TestUml defines a set of test cases, executed in batch before querying the user. These granularities characterize different types of testing, *single* involves heavy manual user interaction, so it is used to define more specific, accurate, limited (as test cases number), target-based testing. *Set* granularity let the user define more invasive tests, often based on randomly generated input data, so it is used with higher numbers of test cases.

To define coverage measurements, typical white-box coverage criteria are used, and, in particular, size and complexity measurements such as: number of selected documents or objects, number of internal links or element relations, paths (with or without cycles), number of states or substates, numbers of functions or variables, connectivity, dominant or post-dominant analysis, distance measures, elements classification based on functionality (for pages such as home page, index, reference, contents, and so on), tour definition, and node clustering (see [5],[7],[12]). Small set of used metrics are shown in Table 1. Other metrics are based on reliability analysis ([21], [25]), such as defects density, probability of failure, rate of fault occurrence, mean time to failure, availability, and reliability.

Test cases defined by TestUml are Web pages (or objects) sequences with user defined input values. A typical test case is a tuple composed by:

(Web page, input variables & values, parameters, [actions])

Test case specification is an extended version of the one described in [13] (with particular attention to the dynamically generated pages), where the specification is based on *request* and *response*, and *predicate* definition. The testing specification are described using XML, with the format specified by a DTD file. *Request* specification defines a pattern of HTTP requests, while *response* specifies assertions on the HTTP response, and, finally, the *predicate* may be HTTP protocol specific assertion (such as match, contain, comparison) or logical connective predicate (such as not, and, or, implies). Figure 4 shows an example of test specification.

Web server log file are used to automatize testing phase thus reducing the need for user interactions. TestUml analyzes log files and defines input as described in [10]. In particular, used techniques are: transformation of each individual user session into a test case, generation of new user sessions based on pools of collected data belonging to different users. TestUml integrates log file analysis with user test data (not all inputs are present in log files) and random input values. The user must also specify expected outputs, i.e., the test oracles used to verify test-case success.

5. APPLICATION EXAMPLE

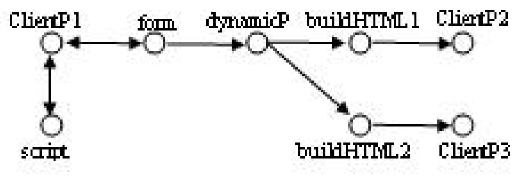

Figure 3: Application graph

To show TestUml usage the same example web application modeled in [4] will be used here. This application builds two type of XML nodes using user values (insert by HTML form). The two different type of XML nodes are built via local javascript function or server side elaboration, based on user choice.

Figure 3 shows the schematized application graph, where every node is an application document or object and every edge is a link (link, redirection, submission link, and so on) or relation between application elements.

```
<testsuite>
 <testcase name="4-7">
 <teststep name="dynamicP" >
  <request url="http://website/dynamicP.asp" />
  <response statuscode="200" />
 </teststep>
 <teststep name="dynamicP" >
  <request url="http://website/dynamicP.asp">
   <parameter name="code" value="056978" method="get"/>
   <parameter name="name" value="Alex" method="get"/>
  </request>
  <response statuscode="200" >
   <and>
    <match op="structureElements" equalTo="buildHTML1"/>
   </and>
  </response>
 </teststep>
 <teststep name="buildHTML1" >
  <request url="http://website/dynamicP.asp">
   <parameter name="code" value="056978" method="get"/>
   <parameter name="name" value="Alex" method="get"/>
  </request>
  <response statuscode="200" >
   <and>
    <match op="contains" equalTo="buildHTML1"/>
   </and>
  </response>
  <teststep name="buildHTML1_2" >
   <request onresponse="true" select="descendent:
   :a/@href" value="http://website/clientP2.html" />
   <response statuscode="200" />
  </teststep>
 </teststep>
 <teststep name="clientP2" >
  <request url="http://website/clientP2.html" />
  <response statuscode="200" />
 </teststep>
</testcase>
...
</testsuite>
```

Figure 4: Test-Case specification

TestUml extracts application element relations from the UML model. Then it defines random walk based test cases (a test case example is the sequence: "dynamicP, buildHTML1, clientP2") and defines testing script skeletons (e.g., the no-bold lines in Figure 4).

This skeleton defines the test suite (<*testsuite*> XML tag), containing test case sequences (<*testcase*> XML tag). Every test case is a sequence of test case steps (<*teststep*> XML tag). One step for every document (or object) to verify. Every step defines HTTP requests (<*request*> XML tag), expected response (<*response*> XML tag) and eventually predicates with condition definitions (<*and*>, <*match*>, <*or*> XML tags). Other useful and definable elements such as parameters (<*parameter*> XML tag), may be defined through UML model analysis, so that the tool knows inputs needed by a particular Web document through information extracted from the model. The test script does not contain values for inputs variables, since they will be added through log analysis or manually by the user. Moreover, the skeleton does not contain hardcoded expected output, TestUml defines skeletons with empty <*response*> XML tags, since they will be filled manually by the user.

If the application has a log file, TestUml analyses it in search of clickstreams related to the defined test case and then analyses it to extract input data. TestUml may use two different techniques ([10]): 1) individual user session transformation; 2) generation of new *user* sessions based on pools of collected data belonging to particular users. For example, for the test case defined in Figure 4 the selected log file user session may be:

```
127.0.0.1 – [26/May/2004:17:55:52 +0200]
"GET /website/dynamicP.asp?code=056978&name=Alex HTTP/1.1" 200
1802
127.0.0.1 – [26/May/2004:17:55:52 +0200]
"GET /website/clientP2.html HTTP/1.1" 200 1727
```

Metrics	Case-Test	Application
#pages	2	4
#objects	1	4
#class	4	11
#class type	3	7
#state	4	17
#state level	2	3
connectivity	2	9
#paths ind.	0	3
#paths cycl.	1	9
#functions	0	4
#var.	8	22
#central nodes	1	2
#central nodes edge	1	4
total page complexity	2.75	8.75
vtotal page complexity	5.75	6.5
#cycles	0	2
#strongly connected components	0	3
#periphery nodes	1	3
....	..	..

Table 1: Fragments of Sample Metrics

If the user wants to change automatic-defined input values he may edit the testing script. The script must then be completed with expected output. Black bold lines in Figure 4 show user-added information. TestUml executes the script with the test cases by sending requests to the Web server and verifying the response.

Then the tool uses UML models to calculate coverage level with user selected metrics[3] and creates reports. The available reports are: testing results (pass/fail) and metrics output (small set in Table 1). In this case we have defined a single test-case with single log-derived input data, but the user may also exploit the random input data definition procedure in TestUml to repeat more than once the same test-case, and/or may define a set of different test cases.

Finally, the tool shows reports to the user and asks him if new test cases extraction and execution has to be performed. The user may choose to execute only one test case at a time (e.g., for totally manual defined input values in test case) or may choose to execute a set of test cases (e.g., to use more random input data test cases) before computing current coverage level and decide wether to stop testing. Alternatively, the user may automatize test stop verification defining the desired coverage level (e.g., 70% pages, 75% of index pages, 85% of Web object with high coupling, or so on). The user may also define a single constraint level for all metrics, or he may define coverage constraints for groups of metrics or for every metric. In this case TestUml loops continuously: define test case, describe testing script (with user help), execute, compute metrics to verify current coverage testing level. If the user defines more than one coverage constraint TestUml stops testing only if all constraints are satisfied. For example, for the Web application shown in Figure 3, the user may want to stop test when 50% or 70% of the application pages are executed at least once, so he defines only one constraint for "# pages" metric with 50% or 70% limit. In this cases the test defined in the specification file in Figure 4 may determine test stop after execution of the only defined test case if the user sets 50%, while with 70% TestUml must execute other test cases definition and execution until the coverage constraint is satisfied (see Table 1 to view the current metrics measurements after the defined Figure 4 test case execution).

6. CONCLUSIONS

[3]In this example reliability metrics are not used because the application is very simple.

This paper describes TestUml (part of the WAAT -*Web Application Analysis and Testing*- project). TestUml implements a new semi-automatic Web testing approach based on a mix of techniques: model based testing, random walks test case definition, log files analysis for input data definition, metrics computation to calculate current coverage and to stop testing. This mix is semi-automatic because the user may tune it with a set of parameters, in particular with desirable coverage level and with test oracles.

TestUml takes as input a set of UML models, usually reverse engineered by WebUml. UML models are used by TestUml to define semi-automatic random walk integration test cases. After test cases definition, TestUml generates testing scripts to be executed when the user has filled in input values (also using log file analysis) and expected outputs. TestUml also supports user-driven stop testing, based on run-time metrics measurements to continuously compute testing coverage level and determine when to stop testing. The user may interact with the tool to select coverage criteria, to select granularity test cases definition, to refine the testing script, and to decide when to stop testing based on coverage level. Since our testing is derived from a model which is reverse engineered from the implementation, it seems that test cases can only be derived from the implementation. But in our system the *user* role is to augment knowledge level towards "meant" implementation.

We are currently studying the implications of testing specification granularity. We are also interested in developing test case composition/aggregation to evaluate their effectiveness. Moreover, we want to study the correlation between chosen metric and coverage testing level trend. Also in connection with the combination of user defined, randomly defined, and log-file generated input data. This study may help analyse more accurate and useful input data generation testing methods.

Finally, through a statistic study based on TestUml usage, we want to investigate how accurate is the use of metrics coverage to define stop criteria, because good coverage level does not always lead to a good test suite.

7. REFERENCES

[1] Bitmechanic. *http://www.bitmechanic.com*.

[2] Mercury interactive. *http://www.merc–int.com*.

[3] Rational Rose Web Modeler. *http://www.rational.com*.

[4] C. Bellettini, A. Marchetto, and A. Trentini. WebUml: Reverse Engineering of Web Applications. *19th Annual ACM Symposium on Applied Computing. Web Technologies and Applications track(SAC 2004)*, Nicosia, Cyprus. March 2004.

[5] R. Botafogo, E. Rivlin, and B. Shneiderman. Structural analysis of hypertexts: Identifying hierarchies and useful metrics. *ACM Transaction Information System*, 1992.

[6] J. Conallen. *Building Web Applications with UML*. Addison-Wesley, 2000.

[7] J. Dhyani, W. Keong, and S. Bhowmick. A Survey of Web Metrics. *ACM Computing Surveys*, 2002.

[8] G. A. Di Lucca, A. Fasolino, F. Faralli, and U. De Carlini. Testing web applications. *International Conference on Software Maintenance (ICSM'02)*, Montreal, Canada. October 2002.

[9] G. A. Di Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. De Carlini. WARE: A Tool for the Reverse Engineering of Web Applications. *6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, Budapest, Hungary. March 2002.

[10] S. Elbaum, S. Karre, and G. Rothermel. Improving Web Application Testing with User Session Data. *25th International Conference on Software Engineering*, Portland, USA. May 2003.

[11] M. A. Friedman and J. M. Voas. *Software Assessment: Reliability, Safety, Testability*. John Wiley & Sons, 1995.

[12] E. Herder. Metrics for the Adaptation of Site Structure. *German Workshop on Adaptivity and User Modeling in Interactive Systems (ABIS02)*, 2002.

[13] X. Jia and H. Liu. Formal Structured Specification for Web Applications Testing. *2003 Midwest Software Engineering Conference (MSEC 2003)*, Chicago, USA. June 2003.

[14] C. Kallepalli and J. Tian. Measuring and Modeling Usage and Reliability for Statistical Web Testing. *Ieee Transactions on Software Engineering*, November 2001.

[15] D. C. Kung, P. Hsia, and J. Gao. *Testing Object-Oriented Software*. Wiley-IEEE Press, 2002.

[16] D. C. Kung, C. H. Liu, and P. Hsia. Object Based Data Flow Testing of Web Applications. *The First Asia-Pacific Conference on Quality Software (APAQS'00)*, Hong Kong, China. October 2000.

[17] D. C. Kung, C. H. Liu, and P. Hsia. An Object Oriented Web Test Model for Testing Web Applications. *24th International Computer Software and Applications Conference (COMPSAC 2000)*, Taipei, Taiwan. October 2000.

[18] D. Lee, K. Sabnani, D. M. Kristol, and S. S. Paul. Conformance Testing of Protocols Specified as Communicating Finite State Machines - a Guided Random Walk Based Approach. *IEEE Trans. on Communications*, 1993.

[19] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey. *IEEE Transaction*, August 1996.

[20] K. R. P. H. Leung, L. C. K. Hui, S. Yiu, and R. W. M. Tang. Modelling Web Navigation by Statechart. *24th International Computer Software and Applications Conference (COMPSAC 2000)*, Taipei, Taiwan. October 2000.

[21] J. Musa. *Software Reliability Engineering*. McGraw-Hill, NY. 1998.

[22] J. Offutt, Y. Wu, and X. Du. Modeling and Testing of Dynamic Aspects of Web Applications. *Submitted*, January 2004.

[23] J. Offutt, Y. Wu, X. Du, and H. Huang. Bypass Testing of Web Applications. *under Submission*, April 2004.

[24] F. Ricca and P. Tonella. Building a Tool for the Analysis and Testing of Web Applications: Problems and Solutions. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'200)*, Genova, Italy. April 2001.

[25] F. Ricca and P. Tonella. Dynamic Model Extraction and Statistical Analysis of Web Applications. *4th International Workshop on Web Site Evolution (WSE 2002)*, Montreal, Canada. October 2002.

[26] F. Ricca and P. Tonella. Analysis and Testing of Web Applications. *23th International Conference on Software Engineering (ICSE'2001)*, Toronto, Canada. May 2001.

[27] C. West. Protocol Validation by Random State Exploration. *6th Intl. Symp. on Protocol Specification, Testing, and Verification*, North-Holland. 1986.