Composing a Framework to Automate Testing of Operational Web-Based Software

Sreedevi Sampath
CIS
University of Delaware
Newark, DE 19716
sampath@cis.udel.edu

Valentin Mihaylov, Amie Souter
Computer Science
Drexel University
Philadelphia, PA 19104
{vlm27,souter}@cs.drexel.edu

Lori Pollock
CIS
University of Delaware
Newark, DE 19716
pollock@cis.udel.edu

Abstract

Low reliability in web-based applications can result in detrimental effects for business, government, and consumers as they become increasingly dependent on the internet for routine operations. A short time to market, large user community, demand for continuous availability, and frequent updates motivate automated, cost-effective testing strategies. To investigate the practical tradeoffs of different automated strategies for key components of the web-based software testing process, we have designed a framework for web-based software testing that focuses on scalability and evolving the test suite automatically as the application's operational profile changes. We have developed an initial prototype that not only demonstrates how existing tools can be used together but provides insight into the cost effectiveness of the overall approach. This paper describes the testing framework, discusses the issues in building and reusing tools in an integrated manner, and presents a case study that exemplifies the usability, costs, and scalability of the approach.

1. Introduction

Private consumers, businesses, and government agencies' frequent use of the internet for important routine tasks creates serious concern for the reliability, security, and usability of web-based software systems. The demand for continuously available and reliable web applications combined with their frequent updates to meet the large user community's preferences create a need for automated, cost-effective testing strategies that evolve the test suite as the operational profiles and the system change. To investigate the practical tradeoffs of different automated strategies for key components of the web-based software testing process, we have designed a framework for web-based software testing that focuses on scalability and evolving the test suite automatically as the application's operational profile changes. Our approach avoids the problem of generating artificial test cases by capturing real user interactions—rather than tester interactions—and utilizing the user sessions as representative test cases of user behavior. Thus, testers could use the collected user sessions during maintenance to enhance the original test suite. The user sessions provide the test data that represents usage not anticipated during earlier testing stages and that evolves as operational profiles change. To be cost effective as user sessions are captured and converted into test cases, we developed an approach for clustering and selecting user sessions on-the-fly to reflect use cases with a minimal test suite size.

This paper describes our overall framework for investigating automated strategies for the web-based software testing process. We summarize the key components of the general framework—a test case generator, test coverage analyzer, replay tool, test oracle, and regression tester—illustrated in Figure 1. In the figure, a control flow edge from *A* to *B* denotes that *B* is the next step of the testing process after *A*. A labeled data flow edge from *A* to *B* indicates that *A*'s output, indicated by the "label", is input to *B*. A derivation edge from *A* to *B* denotes that *B* is derived from a certain information in *A*, not necessarily the output of *A*.

- A *test case generator* is critical to an automated testing framework because it must construct test input to exercise the system under test. In the case of a web application, test input consists of URLs and name-value pairs (i.e., input data). A key issue in automatic test case generation is the efficient generation and maintenance of a practical sized set of test cases that provides desired coverage and fault detection capability.
- The *test coverage analyzer* includes a code instrumentor and the test coverage evaluator. The *code instru-*

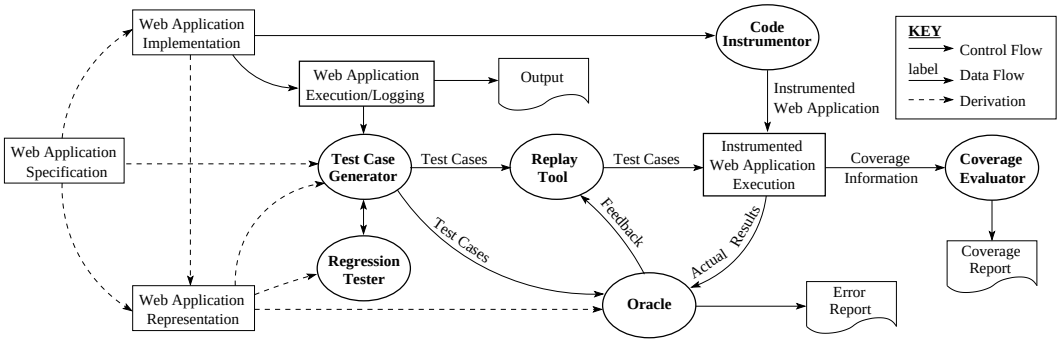

Figure 1. General Framework for Testing Web-based Applications

mentor inserts instrumentation code into the application and then the *test coverage evaluator* executes the application to measure the coverage obtained during execution. The *test coverage evaluator* determines if the application has been adequately tested according to some criterion. For traditional applications, commonly used coverage criteria include method, statement and branch. Besides standard coverage measures, we believe that additional coverage criteria need to be identified for web-based applications.

- A *test oracle* generates expected output, which can be utilized to determine if the system is executing properly during testing. The test oracle also compares the actual results of the system under test to the expected results to determine test case success. For a web application, the expected results consist of HTTP responses and data returned from the web server. Key goals in test oracle design are automation, high fidelity (ability to generate correct and complete results for any test case and distinguish between correct and incorrect responses), generality, and reasonable cost.

- A *replay tool* automates the replay of the generated test cases by sending URL requests and name-value pairs to the web server and collecting the server's responses, which the test oracle uses. A major issue for replay of web-based applications is achieving high accuracy and efficiency.

- A *regression tester* determines if modifications made to a program adversely affect the system under test. Tasks of the regression tester can include selecting test cases from the original test suite that correspond to valid test cases of the modified version of the system, determining new test cases as needed, and prioritizing regression test cases. The design of regression testing techniques for web-based applications should take into account the frequent updates to these applications. We currently do not include the regression tester in the prototype framework presented in this paper, but it appears in Figure 1 because the figure represents the complete testing framework design.

We have developed an initial prototype of this framework that not only demonstrates how existing tools can be used together but provides insight into the cost effectiveness of the overall approach to evolving the test suite.

While software testing researchers have addressed some of the challenges of testing the correctness of web-based applications (detailed in Section 2), there still remains a need for a comprehensive framework that is *robust, integrated, scalable, general,* and provides as much *automation* as possible. In this paper we describe a comprehensive web testing framework that requires little to no amount of human effort for testing processes, while attaining desirable fault detection capability, cost-effectiveness, and scalability. A unique aspect of the framework is its ability to incrementally update the suite of user sessions on-the-fly such that the updated suite reflects the changing operational profile of the web-based application. In addition, the framework provides a means for experimenting with different strategies for each testing component within an integrated and automated overall framework. Our experience in using the prototype for the testing of a bookstore application provides insight into the usability, costs, and scalability of the approach. Another paper describes the theory behind applying concept analysis and the coverage and fault detection capabilities of this approach [24].

Section 2 describes the state of the art in testing of web applications. The testing framework is presented in Section 3. Section 4 demonstrates the use of the framework with a case study as well as empirical measurements of time and space costs. The use of the framework as operational profiles change over time is discussed in Section 5, followed by conclusions and future work in Section 6.

2. Background and State of the Art

Broadly defined, a web-based software system consists of a set of web pages and components that interact to form

a system that executes using web server(s), network, HTTP, and browser, and in which user input (navigation and data input) affects the state of the system. A web page can be either static—in which case the content is fixed—or dynamic, such that its content may depend on user input. Large web-based software systems can require thousands to millions of lines of code, contain many interactions among objects, and involve significant interaction with users and reusable components from third parties. In addition to their architectural challenges, changing user profiles and frequent maintenance changes complicate automated testing [15].

Functional Testing. In addition to tools that test the appearance and validity of web applications, e.g., link, form, and compatibility testers [26], tools such as Cactus [4]—which utilizes JUnit [14]—provide test frameworks for unit testing the functionality of Java-based web programs.

In user-session based testing, data collected from users of a web application are recorded as *user sessions*. Each user's set of interactions with the web server is considered as a user session. To transform a user session into a test case, each logged request of the user session is changed into an HTTP request that can be sent to a web server. A test case consists of a set of HTTP requests that are associated with each user session. Different strategies are applied to construct test cases for the collected user sessions. Tools such as WebKing [20] and Rational Robot [21] provide automated testing for web applications by collecting data from users through minimal configuration changes to the web server. In these tools, test case generators select the most popular paths in web server logs; however, selecting only the most popular paths may lead to inadequate test suites. In addition, these tools do not provide coverage evaluators of the test suite.

Elbaum et al. [9] provided promising results that demonstrate the fault detection capabilities and cost-effectiveness of user-session based testing. They showed that user-session based testing techniques are able to discover certain types of faults but will not uncover faults associated with rarely entered data. In addition, they showed that the effectiveness of user session techniques improves as the number of collected sessions increases. However, the cost of collecting, analyzing, and storing data will also increase. With Harrold et al.'s [12] test case reduction technique, Elbaum et al. [9] report that they are able to achieve a high percent of test case reduction. However, current test case reduction techniques are based on complete data sets, i.e., all user-session data must be collected before the reduction process begins. In contrast to our approach, their technique does not incrementally update the suite as test cases are added.

Program-based Testing. Several researchers have developed analysis tools that model the underlying structure and semantics of web-based programs. With the goal of providing automated data-flow testing, Liu, Kung, Hsia, and Hsu [17] developed the object-oriented web test model (WATM), which captures the dependent relationships of an application's entities through an object model, page navigation through a page navigation diagram, and control and data flow through an interprocedural control flow graph (ICFG). They utilized this model to generate test cases, which are based on the data flow between objects in the model. However, they developed their model for HTML and XML documents and did not consider many features inherent in more recent web applications. Their technique generates def-use chains as test cases, which require additional analysis to generate test cases that can be utilized as actual input to the application. They do not indicate how this analysis would be accomplished.

Ricca and Tonella [23] developed a high level UML-based representation of a web application and described how to perform page, hyperlink, def-use, all-uses, and all-paths testing based on the data dependences computed using the model. Their ReWeb tool loads and analyzes the pages of the web application and builds a UML model. The TestWeb tool generates and executes test cases. However, the tool is not completely automated, so the user needs to intervene to generate input and act as the oracle—examining the output of each response to determine if the test case passed.

Di Lucca et al. [7] developed a web application model and set of tools for the evaluation and automation of testing web applications. They presented an object-oriented test model of a web application and proposed a definition of unit and integration levels of testing. They also developed functional testing techniques based on decision tables, which help in generating effective test cases. However, their approach to generating test input is not automated.

3. Testing Framework and Initial Prototype

Based on the framework in Figure 2, we constructed an automated system that enables the incremental generation of a reduced test suite of user sessions representing use cases for web-based applications. The reduced set of user sessions is automatically replayed and provided as input to a coverage analysis tool and an automated oracle to generate test coverage and fault detection reports, respectively. The process as illustrated in Figure 2 begins with the collection of user-session data (Step (1)). The test case generator comprises of Steps (2), (3), (8), and (9). Step (3) builds an initial reduced test suite for the initial set of user-session data, while Step (8) will incrementally update this reduced test suite as future user sessions are processed. Step (4) is the coverage analysis part of the system. Step (5), composed of the replay tool and the oracle, produces a fault detection report. Step (6) depicts the generation of the coverage report. Steps (7), (8), and (9) represent the incremental update of the reduced test suite.

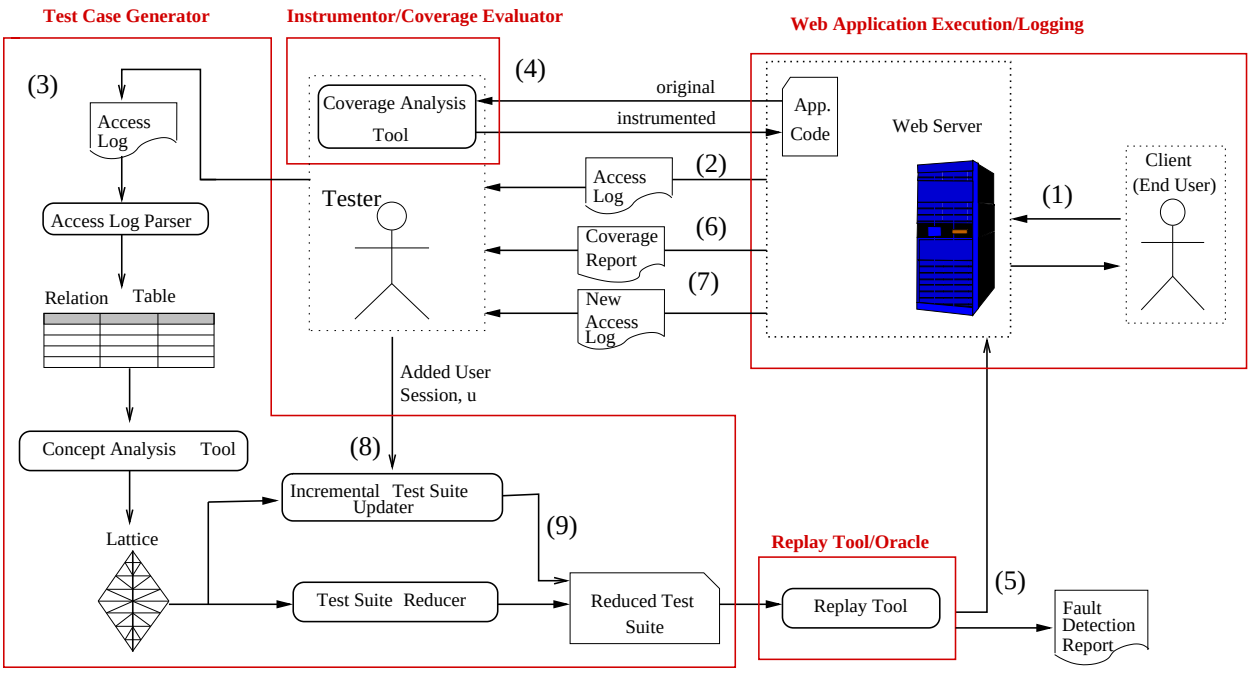

Figure 2. Current Prototype Framework

3.1. Test Case Generation

There has been a large amount of research in automatically generating test cases based on specifications, such as Z-specifications [27, 13], UML statecharts [19], ADL specifications [5], and JML specifications [3]. Researchers [18, 28] have also investigated test case generation techniques for GUI-based applications. While these techniques are useful under some situations, test case generation is not fully automated, and programmers must intervene and provide information about the system under test.

We exploit the ability of a web server to log user sessions without large configuration changes and employ these user sessions for scalable test case generation. Our key insight is to formulate user-session based test case generation in terms of concept analysis [2]. Existing incremental concept analysis techniques [10] can be used to analyze the user sessions on the fly and continually minimize the number of maintained user sessions.

We modified the *AccessLog* class of the Resin web server [22] to log user sessions in a specified format. We are specifically interested in obtaining the IP address, time stamp, the requested URL, cookies, name-value pairs traveling on the GET/POST requests, and the referer URL. To effectively use these user sessions as test cases, we require the data (name-value pairs) input to the application by the user. HTTP GET requests automatically carry the name-value pairs at the end of the request. However, we made considerable changes to the *AccessLog* class to obtain the name-value pairs traveling on HTTP POST requests.

Each user session is a collection of user requests in the form of URL and name-value pairs. We define a user session as beginning when a request arrives from a new IP address and ending when the user leaves the web site or the session expires. We also keep track of the timestamp of the request. Requests from the same IP that are more than 45 minutes apart are considered to be distinct sessions.

The **AccessLogParser** parses the collected user sessions, taking the server's **Access Log** as input and separating the URLs by the user who requested them. Because the format of the **AccessLogParser** output does not conform to the concept analysis tool's input, we further modify the output using scripts executing on a stream editor utility `sed` to create the **Relation Table** for concept analysis.

The **Concept Analysis Tool**, Lindig's `concepts` [16], takes as input a set of objects O, set of attributes A, and a binary relation $R \subseteq O \times A$ called a context (represented as a relation table), which relates the objects to the attributes. In our application of concept analysis, we define an object to be a user session and an attribute to be a URL.

A new IP address identifies each user session, which contains all the user's requests. In the relation table, we strip the name-value pairs of the request from each requested URL. Each user session is thus the IP address followed by the list of requested URLs. Concept analysis identifies all of the concepts for a given tuple (O, A, R), where a *concept* is a tuple $t = (O_i, A_j)$ for which all and only objects in O_i share all and only the attributes in A_j.

The concepts form a partial order defined as $(O_1, A_1) \leq (O_2, A_2)$, *iff* $O_1 \subseteq O_2$. The set of all concepts of a context and the partial ordering form a complete lattice, called the *concept lattice*, which is represented by a directed acyclic graph with a node for each concept and edges denoting a $\leq$ partial ordering. The **Concept Analysis Tool** can output the objects and attributes of each concept in the *concept lattice* in textual or graphical format.

The test suite reduction phase uses the textual output of the concept analysis tool. We apply a heuristic for selecting a subset of user sessions to be included in the reduced test suite based on the concept lattice. Our current heuristic, which we call *test-all-exec-URLs*, seeks to identify the smallest set of user sessions that will cover all of the URLs executed by the original test suite while representing the common URL subsequences of the different use cases portrayed in the original test suite. We define this set to be the **Reduced Test Suite**. We have performed studies that support our heuristic and evaluated its effectiveness [25] when applied to test case reduction. Based on our heuristic, we identify the corresponding concepts in the concept lattice using a Java class we call *IdentifyNextToBottomNodes*. Another custom Java class, *ExtractURLs*, obtains the objects associated with these concepts from the server's access log. These user sessions contain the URL and the associated name-value pairs. Thus, we obtain the reduced test suite for further testing of the application. The two classes—*IdentifyNextToBottomNodes* and *ExtractURLs*—essentially form the **Test Suite Reducer**.

To avoid full reanalysis and space for all user sessions as the test suite evolves, (i.e., address the problem of scalability) we apply incremental concept formation followed by our heuristic for user session selection. We perform incremental concept formation by applying the algorithm developed by Godin, Missaoui, and Alaoui [10]. Godin et al.'s incremental lattice update algorithm takes as input the current lattice L and the new user session with its attributes (i.e., the information in its row of the relation table if it were added to the table). Our incremental update algorithm processes each new user session in its relation-table form. The execution of our **Incremental Test Suite Updater** coupled with the **Test Suite Reducer** results in the new reduced test suite.

3.2. Replay Tool

To replay user sessions, i.e., the test suite, we currently use `wget`, a GNU utility for non-interactive download of files from the Web, as our **Replay Tool**. To replicate the replay of user sessions, we invoke `wget` for each request contained in the logged user sessions. Additional input parameters to `wget` include cookie information and POST or GET data that is associated with the original request. We wrote scripts executing on `sed` to convert the logged user sessions into the appropriate input format for `wget`.

As described in Section 3.1, we record cookies and name-value pairs traveling on the URL. The cookie information is essential to maintain the state of the application. Servers recognize the identity of the user requesting a certain page through cookies, URL rewriting, and hidden fields. When replaying the reduced suite to test the application, we require that the user session is emulated with high accuracy as if a user were initiating requests. We log the cookies when collecting the user sessions initially and use them during replay. Thus, the server recognizes the user and interacts with the replay tool as if the user were logged in. The name-value pairs similarly help in enacting the user session with high accuracy and are representative of data flow through the application.

In addition to preserving the state of the application through cookies to guarantee consistency in the replay of user sessions, we restore the state of the database to its original state, i.e., the state before logging the first user session. By resetting the data in the database tables to their original state, we can replay subsequent runs of user sessions on the same database state as used when the server collected the user sessions initially. To reset the state of the database, we create copies of database tables for which data has been updated as a result of the user's requests. After we use MySql commands to copy the tables, we restore the state of the database from the duplicate tables as needed during the replay of subsequent user sessions.

3.3. Coverage Analysis

To evaluate the effectiveness of a generated test suite, we compute statement and method coverage of the web-based application with the **Coverage Analysis Tool**, Clover [6]. On execution of an `ant` [1] task, Clover instruments and compiles the application. We place the instrumented class files in the directory that the web server can access. Clover also creates an empty fresh database of the instrumented class files to assist in computing the coverage. After replaying the test suite, we stop the server and Clover writes coverage results to its database. By running another `ant` task on the database, we obtain HTML output of the coverage results. Clover produces the number and percent of statements and methods covered individually for each file of the application and also computes overall coverage values for the whole application.

We measured the adequacy of the test suites using statement and method coverage as our preliminary criteria. We believe that the unique characteristics of web applications require other coverage criteria to be identified. These unique characteristics include intrapage control flow through GUI components, interpage con-

trol flow via links, indirect method calls through config files, user input (name-value pairs) through forms, data flow within applets on the client side, the state of the underlying database, and session information maintained by the application through cookies, URL rewriting, and hidden fields.

3.4. Oracle and Fault Detection Tools

Similar to other software testing case studies, our current oracle generates expected results using a "correct" (i.e., non-fault seeded) version of the application. We execute the correct version of the application, save the results (HTML pages), and then compare them to the results obtained when executing a faulty (fault-seeded) version of the application. We use `diff` to compare the downloaded pages of the two versions of the application.

Utilizing an oracle based on the non-fault seeded version of an application is useful for fault detection experiments. We exercised this component of the framework using the following process, which relies on using the **Replay Tool** and **Oracle**. First, we execute a set-up script that separates the logged user requests into distinct sessions. Next, we create a directory with JSP files—each JSP file contains one fault to be inserted, which allows us to create different faulty versions of the application easily. Finally, we create copies of the database, which affects the state of the application.

Next, we run all user session requests on the non-fault-seeded version of the application to generate the expected output of the application. Each response from the server is saved to be utilized by the oracle for comparison purposes. Then a new version of the application is deployed that contains one fault. The database is restored to reflect the original state of the database. All user sessions are replayed, and all responses are logged. The logged responses are stored in a directory structure. To ensure that the requests of a user session are never treated as a part of a previous session, we generate random cookie information for requests from the replay tool, i.e., `wget`. After replaying all the user sessions, we remove the current fault and re-deploy the application with a new inserted fault. The process of replaying the user sessions repeats until all the user sessions have been executed on all the faulty versions of the application.

After all the faulty versions of the application have been executed, we invoke the oracle. The oracle compares the expected web pages with the set of stored actual web page results and saves the results, which are then automatically analyzed to determine the detected faults. The resulting fault detection reports are listings of the name and number of faults detected by each user session.

We automate the fault detection process using several scripts, which can be reused for other web applications. Currently, the process of copying database tables requires user intervention to provide information about which database tables are affected by user requests. Once the tables are identified, the process of copying and restoring the contents of the tables can be automated.

4. A Case Study Using the Framework

The goal of performing this case study was to demonstrate the framework's usability and different test components with example snapshots of input and output at various steps of the process and to examine the space and time costs for using this framework to test web applications.

4.1. Study Setup

We used a medium-sized web application from an open source e-commerce site [11] as our system under test. The application is a bookstore, where users can register, login, browse for books, search for specific books given a keyword, rate the books, buy books by adding them to the shopping cart, modify personal information, and logout. The bookstore application has 9,748 lines of code, 385 methods and 11 classes. Since our interest was in user sessions, we considered only the code available to the user when computing these metrics, not the code for bookstore administration. The application uses JSP for its front-end and MySql database for the backend. The application was hosted on the open source Resin web server [22].

Emails were sent to various local newsgroups and advertisements were posted in the university's classifieds webpage, asking for volunteers to browse the bookstore. We collected 123 user sessions, all of which were used in these experiments. Some of the URLs of bookstore mapped directly to the 11 classes/JSP files and the rest were requests for gif and jpeg images of the application. The size of the largest user session in bookstore was 863 URLs, and on average a user session was 166 URLs.

4.2. Using each Component

In this section, we demonstrate the operation and interaction of the different components of the current prototype framework (Figure 2). Execution of the web-based application and logging the users' sessions (Step (1) of Figure 2) results in creating the **Access Log**. Figure 3 shows a snapshot of the data we recorded in the access log of the web server. An IP address identifies each user session and is the first entry of the recorded request. Each request has the cookie information and the referer URL as well as the IP address, timestamp and the requested URL. The third request depicts how the server records name-value pairs (bold-faced

```
10.197.37.159 [03/Feb/2004:16:14:05 −0500] GET /apps/bookstore/Default.jsp ]
        −−cookies=off] −

10.197.37.159 [03/Feb/2004:16:16:27 −0500] GET /apps/bookstore/Registration.jsp ]
        −−cookies=off −−header "Cookie:JSESSIONID=a7mpavbwGTf6"]
       http://dwalin.cis.udel.edu:8080/apps/bookstore/Default.jsp

10.197.37.159 [03/Feb/2004:16:17:22 −0500] GET /apps/bookstore/Registration.jsp
        ?member_login=bobmason&
        member_password=14921492&member_password2=14921492&
        first_name=bob&last_name=mason&
        email=bobmason%40udel.edu&address=&phone=&
        card_type_id=&card_number=&
        FormName=Reg&FormAction=insert&
        member_id=&PK_member_id= ]
       −−cookies=off −−header "Cookie:JSESSIONID=a7mpavbwGTf6"]
       http://dwalin.cis.udel.edu:8080/apps/bookstore/Registration.jsp

10.197.37.159 [03/Feb/2004:16:14:45 −0500] GET /apps/bookstore/Login.jsp ]
        −−cookies=off −−header "Cookie:JSESSIONID=a7mpavbwGTf6"/
        http://dwalin.cis.udel.edu:8080/apps/bookstore/Default.jsp

10.197.37.159 [03/Feb/2004:16:17:23 −0500] GET /apps/bookstore/Default.jsp ]
        −−cookies=off −−header "Cookie:JSESSIONID=a7mpavbwGTf6"]
       http://dwalin.cis.udel.edu:8080/apps/bookstore/Registration.jsp

10.197.37.159 [03/Feb/2004:16:17:41 −0500] POST /apps/bookstore/Login.jsp
        −−post−data="&querystring=&Password=14921492&
        FormName=Login&FormAction=login
        ret_page=&Login=bobmason"]
       −−cookies=off −−header "Cookie:JSESSIONID=a7mpavbwGTf6"]
       http://dwalin.cis.udel.edu:8080/apps/bookstore/Login.jsp
```

Figure 3. User Session Log

in figure) traveling on the GET request for the Registration.jsp page. The last request in the figure demonstrates the name-value pairs that are part of the POST requests. Since web server loggers usually do not record data traveling on a POST request, we made changes to the web server's *AccessLog* to record this information. The **AccessLogParser** parses the web server's access logs. The output of the parser is further edited by a script file to create the **Relation Table**. Figure 4(a) shows the relation table for a small example containing two user sessions. Each IP address in the table is followed by a list of its corresponding URLs. In the relation table a colon terminates the IP address and a semi-colon terminates the user session.

The **Concept Analysis Tool** takes this relation table as input and produces the concept **Lattice** in textual format. The output from the concept analysis tool, a sparse representation of the lattice for the two user sessions in Figure 4(a), is shown in Figure 4(b). In the lattice representation, a unique number identifies each concept. The objects and attributes that belong to the concept are listed beside the concept number. For example, concept *001* includes user session *10.82.161.133* with attributes *GET.apps.bookstore.images.icon_home.gif* and *GET.apps.bookstore.images.icon_reg.gif*. The edges of the lattice are implicit in the partial ordering between the objects and attributes of different concepts. Based on our heuristic we pass the lattice through the **Test Suite Reducer**, which selects the objects that constitute the **Reduced Test Suite**, i.e., reduced set of user sessions.

The suite is then input to our **Replay Tool**, `wget`. A

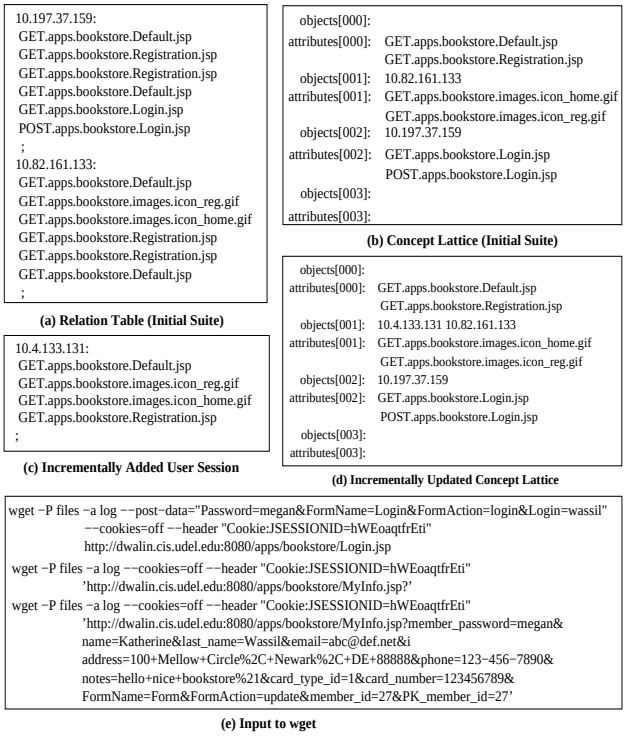

Figure 4. Test Case Generation and Execution

small portion of the input to the replay tool is shown in Figure 4(e). The input consists of the cookie information, the requested URL, name-value pairs of GET requests appended to the request, and name-value pairs of POST requests as a flag (*–post-data*) parameter. We store the files retrieved by `wget` in a directory specified by the *-P* flag and also preserve a log of the replay activity. We only require the retrieved pages for fault detection evaluation—not for computing coverage information with our current criteria. The log provides insight into any errors encountered by the replay tool and could be used as one of the measures of fault detection for future evaluation.

After the framework replays the suite, Clover [6], the **Coverage Analysis Tool**, generates the coverage reports. In Figure 5, we show an example snapshot of Clover's coverage report, which includes the percent of statements, conditionals, and methods covered for the whole application. The lefthand frame shows the percent coverage achieved for each class in the application.

A **New Access Log** from the web server results in adding each new user session, **Added User Session**, to the **Incremental Test Suite Updater**. An example of adding a new user session is shown in Figure 4(c). The original concept lattice is incrementally updated to a new lattice as shown in Figure 4(d). The **Reduced Test Suite** adapts to the change

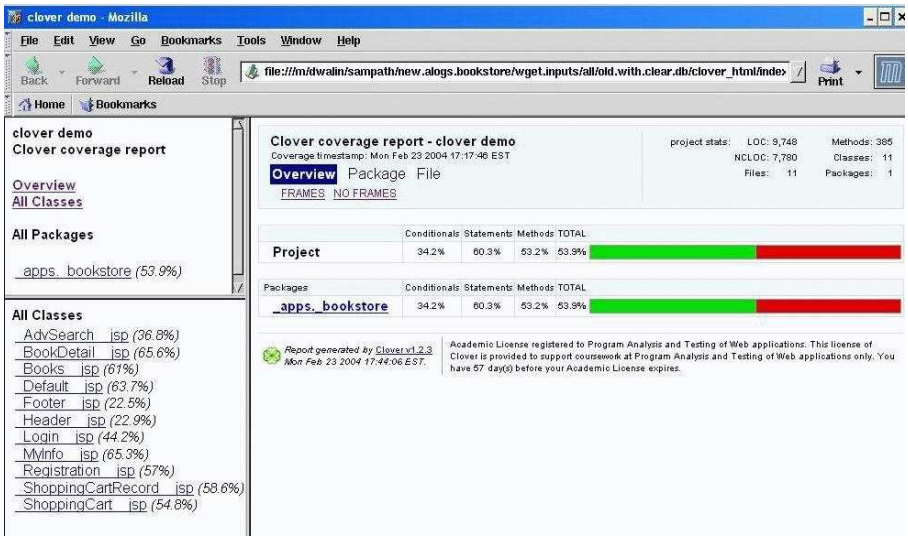

Figure 5. Clover Coverage Report

in the concept lattice according to the *test-all-exec-URLs* heuristic.

For fault detection, we inserted a total of 40 faults into separate copies of the application (one fault per copy). The faults inserted into the code can be broadly classified as data-flow and control-flow based, changes to name-value pairs of the web page, and faults in database queries. Figure 6 illustrates a control-flow-based fault, in which the target page is changed (as shown with the first bold text section in Figure 6). To detect faults, our prototype logs the web server's response pages during replay so that the **Oracle** can determine if a test case passes or fails. We currently utilize a fine-grain text-based comparator (`diff`), which compares the expected output with the actual output. The oracle indicates that a test case fails if it detects any differences between the two pages. The second bold text section of Figure 6 illustrates how the oracle detects a seeded fault.

The framework requires the tester to provide the initial access logs and any newly-produced access log to the **Incremental Test Suite Updater**. The tester must also identify the web application's database tables that need to be copied and restored in the fault detection phase of the framework. No other component in the testing framework requires tester intervention. At the end of analysis, the tester would obviously view the coverage and fault detection reports to evaluate the correctness of the web-based application.

```
<%!
 void Footer_Show (javax.servlet.http.HttpServletRequest request,
                   javax.servlet.http.HttpServletResponse response,
                   javax.servlet.http.HttpSession session,
                   javax.servlet.jsp.JspWriter out,
                   String sFooterErr, String sForm, String sAction,
                   java.sql.Connection conn, java.sql.Statement stat) throws java.io.IOException {
      try {
           out.println("    <table style=\"\">");
           out.print("    <tr>");
           // Set URLs
           String fldField1 = "Default.jsp";
           String fldField3 = "Registration.jsp";
           //faulty code: here is the fault
           String fldField5  = "Registration.jsp";//= "ShoppingCart.jsp";
           String fldField2 = "Login.jsp";
           // Show fields
           out.print("\n      <td style=\"background-color: #FFFFFF; border-width: 1\">
              <a href=\""+fldField1+"\"><font style=\"font-size: 10pt; color: #000000\">Home</font>
              </a></td>");
           out.print("\n      <td style=\"background-color: #FFFFFF; border-width: 1\">
              <a href=\""+fldField3+"\"><font style=\"font-size: 10pt; color: #000000\">Registration</font>
              </a></td>");
           out.print("\n      <td style=\"background-color: #FFFFFF; border-width: 1\">
              <a href=\""+fldField5+"\"><font style=\"font-size: 10pt; color: #000000\">
              Shopping Cart</font>
              </a></td>");
           out.print("\n      <td style=\"background-color: #FFFFFF; border-width: 1\">
              <a href=\""+fldField2+"\"><font style=\"font-size: 10pt; color: #000000\">Sign In</font>
              </a></td>");
           out.println("\n    </tr>\n    </table>");
         }
       catch (Exception e) { out.println(e.toString()); }
     }
%>
```

Figure 6. Example of Seeded Fault

4.3. Costs and Scalability

To investigate the costs and scalability potential of the framework and individual components, we evaluated the execution time of each testing component and the space occupied by the output of that component. In each case, we used the bookstore application and 123 collected, real user sessions.

Table 1 summarizes the results. The first column represents the component of the framework. Each row of the table shows the time taken to execute the component, output obtained by executing the component, and the space required for maintaining the output. We evaluated the time and space computations for all of the original 123 user sessions because our goal in this paper is to gain insight into the effectiveness of utilizing this framework in processing

Component	Execution Time	Component Output	Output Space
Test Case Generation	19s	Reduced Test Suite	1MB
Replay Tool Execution	16m56s	Pages Retrieved	152 MB
Coverage Analysis	1m52s	Coverage Reports/Clover Database	3.5MB
Test Oracle	14s	Diff Output	0.5MB

Table 1. Time to Execute Component and Space for Outputs (with 123 User Sessions)

large sets of user sessions as they evolve, as is typical for any frequently accessed web-based application.

The first component, *Test Case Generation*, comprised of the Steps (2) and (3) in Figure 2. The number of user sessions in the original access log (original suite, Step (2)) was 123 and used 4.3MB of disk space. The framework's test case generation component took 19 seconds to execute completely—from starting the access log through generating the reduced suite. The test case generator's output is the reduced test suite, containing 15 user sessions (87.8% reduction) and occupying 1MB of space (76.7% reduction). The time to generate test cases and the space for the reduced suite are positive indications of effectiveness of the test case generation approach. Evaluation of the incremental update (Step (8) and (9)) produced the same reduced suite and thus provides scalability. As mentioned in Section 3.1, we do not always maintain the original suite and thus the space requirement is for the reduced suite and the new set of user sessions that are added incrementally.

The second row of the table gives the time and space for *Replay Tool Execution*, Step (5) in the framework (Figure 2). The replay tool component involves only the replay of the test suite with the instrumented web application code. We noted that 123 user sessions replayed in 16 minutes and 56 seconds. The output of the test execution, which is the set of pages that the web server returns in response to replayed user sessions, required 152MB of space. The pages retrieved by the replay tool are not required for coverage information but for fault detection studies.

Coverage Analysis (third row of Table 1) is Step (4) in the framework (Figure 2). The time for this component to execute includes the time to instrument the application code, the time for Clover to create and write to its database on completion of replay, and the time taken to generate coverage reports. We considered the time for coverage analysis—1 minute and 52 seconds for the entire 123 user sessions—to be negligible. The output of coverage analysis, which consists of the coverage reports and Clover's updated database, occupied 3.5MB.

The last row of the table reports the time and space to execute the *Oracle*, that is, the time to compare the actual output of 123 user sessions with the expected output of the web application when run with no seeded faults and the space necessary to store the output of `diff`. The oracle took 14 seconds to execute, and the output occupied 0.5MB. Storing all the `diff` output is not necessary to determine the number of faults; however, it is useful to log the `diff` output to better understand how the faults manifest as failures in web pages.

As the results in this section demonstrate, the times to execute the test case generation component and the test oracle comparison were not longer than a few seconds, and the time to perform coverage analysis was also quite short. The test execution component is the most time consuming because the replays are instrumented executions, and this time is proportional to the number of user sessions being replayed (i.e., if only the reduced suite is replayed, then the time to replay would be much smaller). The heuristic applied to test case reduction and the incremental update of the test suite address the issue of scalability because we never maintain the original large set of user sessions but only the reduced suite of user sessions. The framework does not maintain the new user sessions for the next incremental update after it processes them and creates the new reduced suite. The output of the test execution component need not be maintained for coverage analysis purposes but may be required for fault detection analysis. The space occupied by the output of the test oracle is also not that high and thus does not negatively affect the scalability of the system.

5. Using the Framework as Operational Profiles Change

The many users, and even the same user, of a web application may navigate and access the capabilities of an application in different ways at different times. Furthermore, the patterns of usage evolve as the content and application evolve to fit the users' interests.

By basing automatic test case generation on user sessions gathered during the use of the web application in the field, a test suite can reflect the application's various use cases. Web usage pattern mining is a promising approach to drive the adaptation of a web site to improve the accessibility of its offered content [8]. Web usage data is collected and then mined to extract frequent usage patterns, where an access pattern is a recurring sequential pattern among the entries in the web server log. While web usage data is also the target of our testing analysis, the problem of evolving

a test suite as the operational profiles change is slightly different from adapting a website based on web usage patterns. Particularly, the main concern is evolving the test suite in a scalable manner.

By clustering user sessions on-the-fly through concept analysis, a single day's user sessions can be gathered in a production environment, and then overnight, the incremental concept analysis algorithm run to incrementally update the concept lattice, followed by application of the test selection heuristic on the newly updated concept lattice. This process results in a continually updated reduced suite of user sessions that reflects the evolving operational profile. Space and testing efforts are saved by not maintaining the original set of user sessions or the entire set of all user sessions gathered to date.

6. Conclusions and Future Work

The main contribution of this paper is the description of a comprehensive framework for automating the testing process for web-based software that focuses on scalability and evolving the test suite automatically as the application's operational profile changes. The framework provides a means for experimenting with different strategies for each testing component, within an integrated and automated framework. We have gained valuable insight into the usability, costs, and scalability of our approach to evolving the test suite with change in operational profiles through a case study.

We plan to use this framework to investigate additional heuristics for evolving the test suite, performing experimental studies of a large collection of web applications, and developing more sophisticated test components. We are also working towards augmenting the current prototype with a regression testing component.

Acknowledgments. We sincerely thank Sara Sprenkle and Emily Gibson at U. of Delaware for their editorial suggestions which greatly improved the readability of the paper.

References

[1] Apache Software Foundation. <http://ant.apache.org/>, 2000.

[2] G. Birkhoff. *Lattice Theory*, volume 5. American Mathematical Soc. Colloquium Publications, 1940.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2002.

[4] Cactus. <http://jakarta.apache.org/cactus/>, 2002.

[5] J. Chang and D. J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Proceedings of the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 1999.

[6] Clover: Code coverage tool for Java. <http://www.thecortex.net/clover/>, 2003.

[7] G. A. Di Lucca, A. Fasolino, F. Faralli, and U. D. Carlini. Testing web applications. In *International Conference on Software Maintenance*, 2002.

[8] M. El-Ramly and E. Stroulia. Analysis of web-usage behavior for focused web sites: A case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 2004.

[9] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *International Conference on Software Engineering*, 2003.

[10] R. Godin, R. Missaoui, and H. Alaoui. Incremental concept formation algorithms based on Galois (concept) lattices. *Computational Intelligence*, 11(2):246–267, 1995.

[11] Open source web applications with source code. <http://www.gotocode.com>, 2003.

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

[13] H.-M. Horcher. Improving software tests using Z specifications. In *Proceedings of the 9th International Conference of Z Users, The Z Formal Specification Notation*, 1995.

[14] Junit. <http://www.junit.org>, 2002.

[15] E. Kirda, M. Jazayeri, C. Kerer, and M. Schranz. Experiences in engineering flexible web service. *IEEE MultiMedia*, 8(1):58–65, 2001.

[16] C. Lindig. ftp.ips.cs.tu-bs.de:pub/local/softech/misc, 2002.

[17] C.-H. Liu, D. C. Kung, and P. Hsia. Object-based data flow testing of web applications. In *Proceedings of the First Asia-Pacific Conference on Quality Software*, 2000.

[18] A. M. Memon. *A Comprehensive Framework for Testing Graphical-user Interfaces*. PhD thesis, University of Pittsburgh, 2001.

[19] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proceedings of the Second International Conference on the Unified Modeling Language*, 1999.

[20] Parasoft WebKing. <http://www.parsoft.com>, 2004.

[21] Rational Robot. <http://www-306.ibm.com/software/awdtools/tester/robot/>, 2003.

[22] Caucho resin. <http://www.caucho.com/resin/>, 2002.

[23] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the International Conference on Software Engineering*, May 2001.

[24] S. Sampath, V. Mihaylov, A. Souter, and L. Pollock. A scalable approach to user-session based testing of web applications through concept analysis. In *Proceedings of the Automated Software Engineering Conference*, September 2004.

[25] S. Sampath, A. Souter, and L. Pollock. Towards defining and exploiting similarities in web application use cases through user session analysis. In *Proceedings of the Second International Workshop on Dynamic Analysis*, May 2004.

[26] Web site test tools and site management tools. <http://www.softwareqatest.com/qatweb1.html>, 2003.

[27] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.

[28] L. White and H. Almezen. Generating test cases for GUI representation using complete interaction sequences. In *Proceedings of the International Symposium on Software Reliability Engineering*, 2000.