Automated Replay and Failure Detection for Web Applications

Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock
Computer and Information Sciences
University of Delaware
Newark, DE 19716
sprenkle, gibson, sampath, pollock@cis.udel.edu

ABSTRACT

User-session-based testing of web applications gathers user sessions to create and continually update test suites based on real user input in the field. To support this approach during maintenance and beta testing phases, we have built an automated framework for testing web-based software that focuses on scalability and evolving the test suite automatically as the application's operational profile changes. This paper reports on the automation of the *replay* and *oracle* components for web applications, which pose issues beyond those in the equivalent testing steps for traditional, stand-alone applications. Concurrency, nondeterminism, dependence on persistent state and previous user sessions, a complex application infrastructure, and a large number of output formats necessitate developing different replay and oracle comparator operators, which have tradeoffs in fault detection effectiveness, precision of analysis, and efficiency. We have designed, implemented, and evaluated a set of automated replay techniques and oracle comparators for user-session-based testing of web applications. This paper describes the issues, algorithms, heuristics, and an experimental case study with user sessions for two web applications. From our results, we conclude that testers performing user-session-based testing should consider their expectations for program coverage and fault detection when choosing a replay and oracle technique.

Categories and Subject Descriptors: D.2.5: [Software Engineering]: Testing and Debugging–testing tools

General Terms: Reliability, Experimentation

Keywords: software testing, replay, test oracles, web applications

1. INTRODUCTION

As the move towards web-based applications continues at a rapid rate, the need to ensure the reliability of this application domain increases accordingly. Simultaneously, companies are often logging and analyzing the usage of these applications to guide further enhancements and customization as user profiles evolve. User-session-based testing seeks to reuse this large and easily collected data bundled in the form of user sessions for testing that reflects real users' usage of the application. User sessions can be replayed as test cases, and an oracle determines whether the actual results match the expected results. A human tester could do both the replay and the oracle; however, for a human, both tasks are extremely tedious and error-prone, which leads to significant manual testing overhead.

An automated system for collecting and replaying user sessions to expose faults is desirable, but there are a number of challenges to developing an effective automated system. Web applications typically have persistent state in the form of a backend database and the server state. Thus, similar to testing databases and GUIs, the system's handling of persistent state affects replay of a sequence of user sessions. If user sessions are replayed in different orders or individual user sessions are not included in the replay, e.g., because of applying a test suite reduction technique, then the replayed sessions will most likely have different behavior than the original suite of user sessions. Furthermore, the replay might serialize the user sessions or attempt to replay sessions in parallel, as they might have occurred when collected. Thus, unlike traditional software without persistent state, reordering test cases and temporal changes to a set of test cases during replay change the behavior exhibited during testing. To address this problem, we must consider the web applications' *controllability* [6].

Oracles have traditionally been difficult and expensive to develop in software testing [3, 4, 23, 30]. An *oracle* is a mechanism to evaluate the actual results of a test case as pass or no pass. An oracle produces an expected result for an input and uses a *comparator* to check the actual results against the expected results [4]. Human oracles often evaluate the correctness of the actual output. Some approaches to automated or partially automated oracles involve developing an oracle from specifications, from simulations of the program under test, or using an implementation that is trusted [4].

The primary challenge to developing oracles for web applications is their low *observability* [6], i.e., while some application output goes to the user in the form of generated HTML pages, other application behaviors, such as generated email messages, internal server state, data state, and calls to web services, are not as easy to monitor. Existing web application testing approaches [10, 17] have employed oracles that use the HTML pages as the output of an executed test case. However, an oracle comparator that naively detects every difference in page content could mistakenly report a fault

ASE'05, November 7–11, 2005, Long Beach, California, USA.

when the difference is in real-time, dynamic information. Furthermore, an oracle comparator that focuses on specific internal details of behavior may miss reporting faults.

In a previous paper [25], we reported on our use of existing tools and the development of some simple scripts to automate the entire process from gathering user sessions through the identification of a reduced test suite and replay of that test suite for coverage analysis and fault detection. In this paper, we describe our work on replay and oracle comparators, which focuses on addressing the challenges posed by web applications' observability and controllability, derived from our experiences in experimentation with the prototype.

This paper describes the design, implementation, and evaluation of four oracle comparators, used in combination with replay with and without state information considered. We introduce alternate notions of expected and actual output, specifically for web applications. While the persistent state problem is not new to testing certain application domains, we discuss the issues of handling persistent state in web applications and consider both the structure and content of the generated HTML web pages in our oracle comparators. The main contributions of this paper are

- A set of automated strategies for handling persistent state during replay of web application user sessions
- A set of plug-n-play, automated oracle comparators for user-session-based testing of web applications
- An experimental study of two web applications and corresponding user sessions that examines the impact of different combinations of replay and oracle comparators on program coverage, composition of faults detected, and associated costs

By automating the replay and oracle comparators, the tester benefits by (1) plug-n-play oracles, tailored to the types of faults the tester wants to expose, (2) ability to replay with or without restoring state, which may expose different faults with the same test suite, (3) capture-replay that helps the developer localize faults that the user encounters, and (4) automatically removing static/invalid URLs, which reduces the cost of replaying URLs that do not exercise the application under test.

Section 2 provides background on user-session-based testing. The overall automated framework in which we integrated the replay and oracle mechanisms is described in Section 3. We describe the challenges and our approaches to automating replay of user sessions for user-session-based testing of web applications in Section 4. Section 5 presents the challenges and our set of oracle comparator strategies. The design, results, and analysis of our experimental study with two web applications are presented in Sections 6 and 7. We conclude with future work in Section 8.

2. USER-SESSION-BASED TESTING

Broadly defined, a web application is an application that runs on a web server and is available to users over a network. A web application generally encompasses a set of static and dynamic web pages. Based on user requests and server state, the web application generates dynamic responses. Large web-based software systems can require thousands to millions of lines of code, contain many interactions between objects, and involve significant user interaction. Changing user profiles and frequent small maintenance changes complicate automated testing [15].

Because of the user-oriented nature of web applications, testing focuses on code that users commonly access. User sessions are a convenient way to test executed code in the manner users access the application. In *user-session-based testing* [9], the web server logs users accessing a web application. Each *user session* is a collection of user requests in the form of URL and name-value pairs (i.e., form field names and values). More specifically, a user session begins when a request from a new IP address reaches the server and ends when the user leaves the web site or the session times out. To transform a user session into a test case, each logged request 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 [10, 20, 21, 27]. A key advantage is the minimal configuration changes to the web server to collect user requests. Capture/replay for web applications is relatively cheap, compared to other domains, as demonstrated in [19].

Elbaum et al. [10] report promising results that demonstrate the fault detection capabilities and cost-effectiveness of user-session-based testing. They show that the effectiveness of user session techniques improves as the number of collected sessions increases. Elbaum et al. [10] report that they achieve a high percent of test case reduction with Harrold et al's. [12] test case reduction technique, and they found no difference in fault detection effectiveness when they restored state as compared to when they did not.

In [26], we presented an approach to achieve scalable user-session-based testing of web applications. We view the collection of logged user sessions as a set of use cases where a use case is a behaviorally related sequence of events performed by the user through a dialogue with the system [14]. The key insight of our approach is the formulation of the test case reduction problem for user-session testing in terms of concept analysis [26]. We can then exploit existing incremental concept analysis techniques to analyze the user sessions on the fly, as sessions are captured and converted into test cases, and thus we can continually reflect the set of use cases representing actual executed user behavior by a minimal test suite. Test suite reduction is accomplished on the fly by grouping user sessions into similar clusters.

3. AUTOMATED FRAMEWORK

We implemented a prototype of an automated web application testing framework, shown in Figure 1. We designed our framework to collect user sessions, test web applications and evaluate the effectiveness of test suites with minimal human effort. Our goal was to develop an *integrated* framework for testing web-based applications, where the testing process is *automated* while maintaining the desired fault detection capability, test coverage, cost-effectiveness, and scalability. In addition to being *automated*, the main properties of the framework are

- *generality* - The framework can be used to test any web application written in Java/JSP because we do not make changes to the application code to enable the testing process. The framework can also be extended to test applications written in other web-based languages such as PHP.
- *flexibility* - Some components of the framework, e.g., the user-session logger or the replay tool, can be modified depending on the web application's requirements.
- *portability* - Test information, e.g., user sessions and coverage and failure detection reports, is platform-independent.

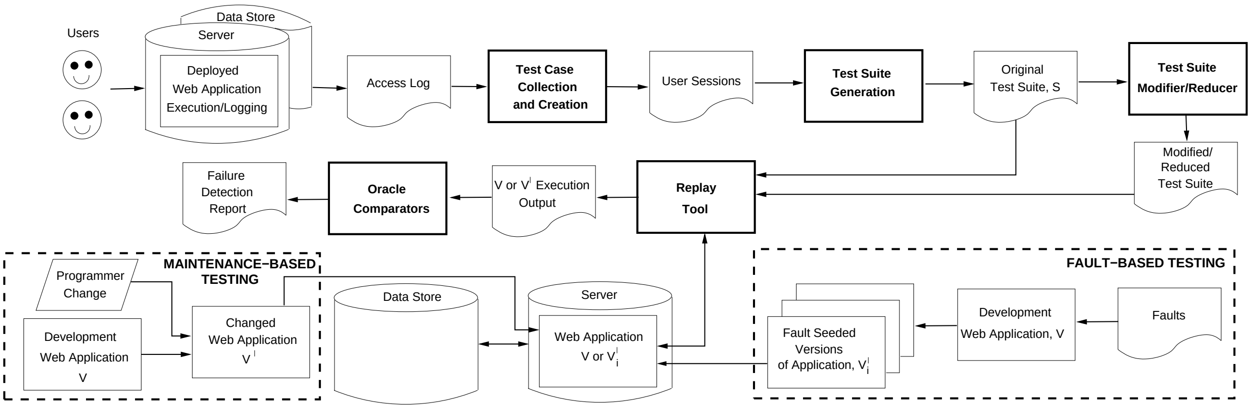
Users
Data Store
Server
Deployed Web Application Execution/Logging
Access Log
Test Case Collection and Creation
User Sessions
Test Suite Generation
Original Test Suite, S
Test Suite Modifier/Reducer
Modified/ Reduced Test Suite
Failure Detection Report
Oracle Comparators
V or V' Execution Output
Replay Tool
MAINTENANCE–BASED TESTING
Programmer Change
Changed Web Application V'
Development Web Application V
Data Store
Server
Web Application V or V'i
FAULT–BASED TESTING
Fault Seeded Versions of Application, V'i
Development Web Application, V
Faults

Figure 1: Automated Framework for Testing Web Applications

In [25], we presented an initial framework. In this section, we focus on our improvements to test case collection, test suite generation, test suite replay, and fault detection.

3.1 Test Case Collection and Creation

We create the original test suite in four steps: record web server accesses, generate user sessions, remove static sessions, and generate replayable sessions.

Record web server accesses. We augmented the Resin [22] web server's access logging class to record both GET and POST requests' encoded name-value pairs and the URL.

Generate user sessions. We parse the web server's augmented access log to create user sessions [25]. Because image requests do not affect an application's execution, we exclude them from our user sessions. We generate a list of an application's valid URL requests so that we can remove invalid URLs, such as mistyped URLs, requests for nonexistent files, or requests for other applications, from the sessions.

Remove static user sessions. Some user sessions request only static pages. Since static user sessions do not access the application, we can remove static sessions from the test suite without changing the test suite's ability to exercise the application. We maintain a list of static pages for each application so that we can identify and remove static sessions.

Generate replayable user sessions. The final step for creating test cases is a simple transformation from user sessions into files containing URLs that the system can replay.

3.2 Test Suite Generation and Reduction

User sessions may depend on previous user sessions for state. For example, if a user creates an account in one session and subsequently logs into the application in another session, the data store must contain the user account for the application to run as expected. To approximate the user session dependencies, we order the user sessions by their first recorded web server access.

Optionally, a test suite may be reduced before it is replayed. We describe our reduction technique in [26] and compare reduction techniques in our framework in [28].

3.3 Test Suite Replay

The replay tool may be run with the original web application V or a modified version of the application V'. In maintenance-based testing, V' is created from applying maintenance changes to V. Additionally, a set of versions V_i' could be created by seeding different faults into copies of V during fault-based testing.

To replay a test suite, we utilize the GNU `wget` [31] tool. For every user session in the test suite, `wget` requests each URL including name-value pairs and then downloads the pages returned in response. Currently, the framework replays user sessions consecutively in the order in which the server logged them. Before replay, the framework initializes the data store to its initial state—before we deployed the application and captured user sessions.

When replaying reduced suites, the framework does not replay all the sessions in the original suite. If a reduced test suite does not contain a user session, the framework could restore the data state as if the user session were in the original suite to account for the application's persistent state. If we do not restore the state between sessions, the application behavior for the reduced suite will not necessarily match the behavior of replaying the original suite. To save the data store between every session, a consecutive replay of the original suite is run once before replaying any reduced suites. We discuss the issues involved in maintaining state during replay in Section 4.

3.4 Fault-Based Testing

Testers can use fault (or failure) detection to evaluate the effectiveness of a given test suite. A fault is defined as an incorrect statement, step, process, or data definition in a computer program [4]. We developed fault classifications based on the classifications identified in Elbaum et al. [10], which identified three types of faults for web applications: *scripting*, *web page*, and *database* faults. However, Elbaum et al. assume all web applications have a database backend; some web applications use files as a data store. We seed *data store faults* that exercise application code that interacts with databases and/or data stores. We renamed the scripting faults as *logic faults*, which include application code logic errors in the data or control flow. We split *web page faults* into three categories to help differentiate the severity of faults: *form faults*, which modify name/value pairs and actions, i.e., API faults because users call the web application through forms; *appearance faults*, which change the way the page appears to users, such as, layout and style; and *link faults*, which change a hyperlink's location. Thus, our fault classification is *data store, logic, form, appearance,* and *link* faults.

To detect failures, i.e., the manifestation of a fault, the fault detection framework inserts one fault into the application and replays a test suite. For our purposes of comparing test suite effectiveness, graduate and undergraduate students familiar with JSP/Java servlets/HTML manually seeded realistic faults into the applications. We are currently designing an extension to automatically insert faults through mutator operators [2]. A pluggable oracle compares the generated HTML output from the non-faulty, "clean" version of the application with the actual output from the faulty application to determine whether a test detects a failure. Section 5 describes the challenges of developing oracles web applications.

4. AUTOMATING REPLAY

Given a version of the code to be executed during replay, there are a number of different parameters for replaying user sessions: (1) the selected user sessions to replay (the original set or a selected subset based on some heuristic), (2) the timing of the user session replay (serially or with concurrency), (3) the order to replay user sessions (log-recorded order or some other established order), and (4) the restoration of persistent state during replay (no restoration or restoration at certain points during replay).

For testing purposes, it is not necessary to emulate the original, i.e., the captured, behavior of the application exactly. However, in user-session-based testing, it is desirable to exercise the application with test cases that represent real usage, e.g., as recorded in some real user session. Thus, we focus on replay in the order of the original user session's first access. When captured, the user sessions probably overlapped and interacted. In our current work, we focus on the issues in executing the test cases serially. Serialized access simplifies the testing process by removing the complexities of concurrent access and testing [8, 13, 16]; however, our current implementation does not test critical sections, i.e., blocks of code that require mutually exclusive access, of the web application. In future work, we plan to address the challenges that parallel replay pose in our framework. Since the experiments by Elbaum et al. [10] did not support combining different user sessions through random selection of requests from different sessions, we execute an entire user session from start to finish, without selecting portions of (and possibly merging) user sessions.

One of the goals of our experiments is to investigate the role of persistent state on web-application behavior during user-session replay. Thus, we examine several options for dealing with state, described in the next subsections.

The remainder of this section elaborates on the challenges and then describes explored solutions to addressing persistent state in web applications, as embodied in data and server state.

4.1 Data State

Challenges. Maintaining application state is a controllability issue for accurately replaying the reduced test suite [6]. Figure 2 illustrates the behavioral differences upon restoring or not restoring state during replay. The left subfigure shows the data store state after each user session is executed in the original suite. The middle subfigure shows how the state is updated between user sessions of a subset of the original suite to maintain the same behavior as the original suite. Note that only states just prior to an executed user session in the smaller suite need to be restored, not every state between every user session in the original suite. Note also that later user sessions may also depend on the execution of previous user sessions, i.e., user session behavior may depend on the application's persistent state. The right subfigure shows how behavior differs when the same smaller test suite is replayed without state restoration.

Approaches. Currently, we believe there are three approaches to handling persistent data state for reduced suites:

`Without state.` Ignore the persistent state problem. The results will be inaccurate as compared to the sessions when replayed consecutively and may not represent the same use cases. However, because the data state is different from when the sessions were originally recorded, e.g., the corresponding user account does not exist before a login request, the sessions will most likely execute significant amounts of error code. Therefore, replaying the same test suite without restoring the state could provide additional testing without capturing or generating any new sessions.

`With state.` The data state could be restored before the framework replays each user session. Our current implementation of this approach is to replay the original suite once and save the data state between sessions. Before running each session of a reduced suite, the framework restores the data state. This approach is expensive in terms of space and time. A more efficient implementation, which we plan to implement in the future, is to log database interactions or data store changes (deltas) while replaying the original suite. The framework could replay the database logs or apply deltas to the data store to restore the data between reduced sessions with less time and space costs.

`Augment the test suite.` It may be possible to calculate the dependencies between user sessions, created by persistent state changes. If the correct execution of session C, e.g., an internal application page, depends on sessions A, e.g., a registration request, and B, e.g., an update of account information, we could include A and B in the reduced suite along with C to make the suite *safe* [4]. However, in the worst case, the reduced suite would include all the sessions of the original suite—negating the benefits of reduction. Calculating user-session dependencies is similar to restoring the state but may require replaying a larger test suite instead of updating the state directly.

4.2 Server State

Challenges. In addition to data state, server state can also be an issue when replaying user sessions. For example, a simple page counter could be implemented by storing variables on the server and incrementing and displaying the counter associated with a web page every time a user accesses the page. The counter's behavior changes depending on the number of times the page has been accessed since the server's last restart. If a fault exists in the counter code, perhaps the first time the page is accessed after the server starts may be the only time that a failure occurs, which has implications for failure detection. Other examples of server state include cookies, current time maintained by the server, and session information.

Approach. Our current approach is to ignore server state. Another approach is to restart the server between the replay of every user session. Restarting the server is extremely impractical and proved prohibitively expensive for the machines used in our experiments. However, keeping any state on the server is not good programming practice; therefore, server state is a lesser concern than data state.

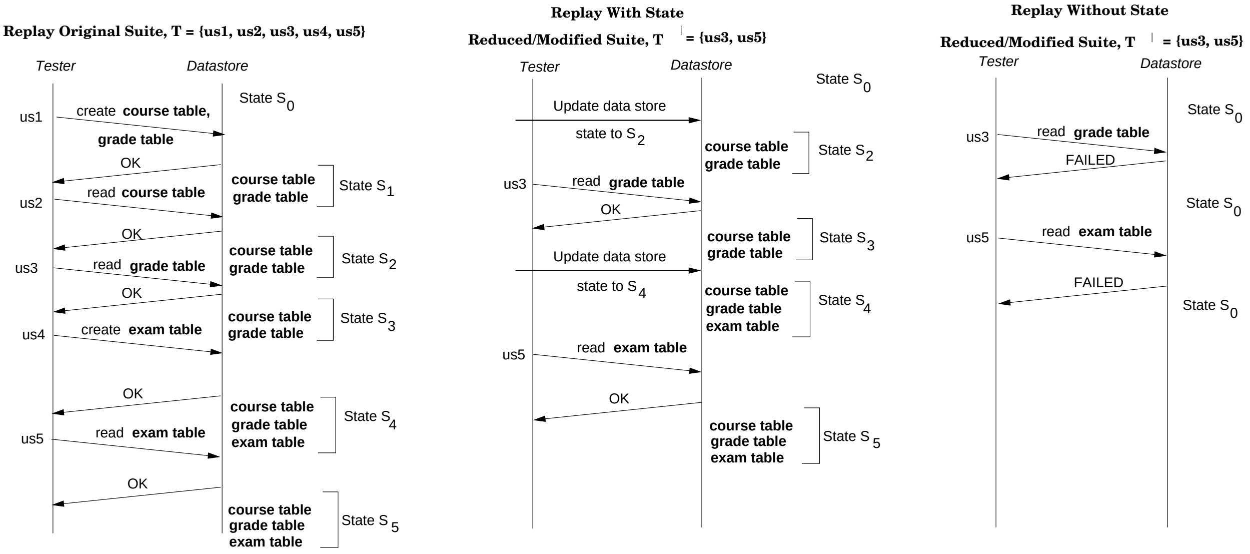

Figure 2: Illustrating the Role of State in Web Applications

4.3 Failure Detection Replay

State is an issue for replaying fault-seeded versions of an application for both the original and reduced suites. During a faulty run, sessions may affect the application or data state in such a way that subsequent sessions may execute erroneously, even without executing the fault. However, if the framework restores the state after each session, then it will only report a fault in a session that executes the faulty code if the fault manifests itself in the output of the session. If the framework does not restore state, replaying the reduced suites will suffer from the state problems discussed earlier.

5. AUTOMATING ORACLES

5.1 Challenges

Software testing researchers often assume that oracles exist, but developing them is difficult [24, 30]. Some researchers propose using model-based oracles [5, 18, 29]. However, there are major challenges in accurately modeling web applications, such as addressing the ability to navigate web pages through a browser (e.g., a user accesses pages through back and forward buttons or by typing the URL directly); the inability to statically determine control flow due to user input and location; and the existence of numerous technologies and languages. Partial oracles based on models for web applications are presented in [1]. Since many web applications have a database backend, we could use a state validator, such as the one in AGENDA [6], to validate the application's database state. However, web applications may use other state, such as files or internal server state.

The ideal oracle accurately reports erroneous application behavior. When developing an oracle, we must analyze its accuracy in terms of its false positives—reporting a failure when the application behaved appropriately—and its false negatives—not reporting a failure. False negatives allow faulty code to be passed into production use, while false positives may cause a developer to waste time tracking a non-existent bug. Note that, in this paper, our goal is not precise fault localization; we may extend our framework to automate more precise fault localization in future work. Instead, we propose several automated oracle comparators, which plug easily into our framework, and discuss their relative precision.

5.2 Comparison Algorithms

Since many web applications generate HTML output, we focus our efforts on validating HTML output—a pseudo oracle for a subset of the application's output [30]. By analyzing the HTML output, we can evaluate what the user sees. We start with a “gold standard”—a previous, working version of the application [4]—and compare the gold standard's expected output with the actual output.

One approach to validating HTML output is using a simple `diff` of the expected and actual HTML output, as in [10]. While an HTML diff, which we call **Raw**, is inexpensive, the oracle will report a failure for any change in the HTML code, including changes in dynamic content, such as changes in the date or time; the ideal oracle does not report changes in dynamic content as failures. Additionally, the **Raw** oracle will miss faults that do not reveal themselves in the HTML output. An example is a database transaction that does not commit and the HTML output matches the output that would be generated if the transaction did commit. Any oracle that looks only at the HTML output is susceptible to these kinds of false negatives. However, subsequent sessions that depend on the session that exercised the faulty code typically expose these faults.

To reduce the number of false positives, we developed two additional output-based oracles: one that ignores the HTML tags and one that only considers the HTML tags. By ignoring the HTML tags, the HTML **Content** oracle will not falsely fail for web pages with simple changes to the UI, e.g., changes in the page layout (tables versus CSS) or display (different font color), which do not affect the overall execution of the application. However, the HTML **Content** oracle will falsely fail applications for changes in the generated dynamic output. Besides missing faults not manifested in the HTML content, the **Content** oracle will miss faults in the HTML tags, such as omitted form input tags.

The HTML **Structure**-based oracle compares only the

HTML tags, including the attributes and values, of the actual and expected output. The oracle will incorrectly fail pages that have slight changes in the UI or display that do not affect the behavior or appearance of the application. For example, the oracle will flag changes in layout of hidden fields and breaks, e.g., combinations of `<BR>` with `<INPUT type=hidden ...>`. The oracle will miss data or content errors.

The cost of comparing each request's output can be reduced by first performing a simple diff on the file names that `wget` has downloaded. This comparison, which we call **Flist**, catches errors that cause differences in page-level control flow, e.g., URL redirection, or errors that caused the server not to generate HTML, e.g., a fatal exception. **Flist** has no false positives because if a session took a different path through the application and therefore downloaded different files, then a failure occurred. The oracle misses errors in page content. Since the other proposed output-based oracles subsume **Flist**, **Flist** should be executed before the other oracles.

6. EXPERIMENTAL STUDY

In previous sections, we presented several approaches for replaying test suites and for comparing expected with actual output. In this section, we describe our experiments for evaluating the approaches' effectiveness and practicality. We then use our experimental results to guide testers in choosing an appropriate replay technique and oracle comparator.

6.1 Research Questions and Hypotheses

We designed our experiments to answer the following questions about our approaches:

Question 1. How does the choice of replay technique affect the test suite's program coverage?

Hypothesis 1. We believe that we will see a difference in the program coverage between `with_state` and `without_state`. We expect `with_state` to have higher coverage, but `without_state` will cover error code that `with_state` does not. We also believe that replaying smaller reduced suites will cause a bigger difference in code coverage because `without_state` will miss the state from more user sessions and thus cover more error code than `with_state` replay of the reduced suite.

Question 2. How do the replay technique and oracle comparator affect the set of reported faults?

Hypothesis 2. Because we expect the replay technique to affect the covered code, the replay technique will also affect the reported failures. As described earlier, the oracles are not always precise and will report failures that are not caused by code faults (false positives) and miss other failures (false negatives). Since each oracle is susceptible to its own false positives and false negatives, the set of reported faults for each oracle comparator will be different.

Question 3. What are the costs of each replay technique and oracle comparator in terms of their requirements for time, space, and human intervention?

Hypothesis 3. We believe that `with_state` will have slightly higher space and time costs to store and update the data store between missing sessions when replaying reduced/ modified test suites. Since **Structure** and **Content** process the HTML file, they will have higher time costs than **Raw** and **Flist**.

The remainder of this section describes our methodology for answering our research questions.

Test Suites	# US	Total URLs		Largest US	Avg US
		mode	set avg	(# URLs)	(# URLs)
Book_Orig	125	3640	N/A	160	29
Book_Mod2	63	1644	N/A	81	26
Book_HGS-C	12	526	568	74	44
Book_HGS-M	4	161	166	66	40
Book_HGS-S	11	387	393	74	35
Book_HGS-U	1	66	67	66	66
Book_Con-U	5	264	N/A	81	53
CPM_Orig	261	3881	N/A	152	15
CPM_Mod2	131	1989	N/A	152	15
CPM_HGS-C	30	1240	1246	152	41
CPM_HGS-M	12	486	500	132	40
CPM_HGS-S	36	1295	1222	152	36
CPM_HGS-U	12	384	422	132	32
CPM_Con-U	49	1709	N/A	152	35

Table 1: Test Suite Characteristics

6.2 Independent and Dependent Variables

The *independent variables* in our study are the test suites, subject applications, replay techniques, and the oracle comparators. The *dependent variables* in our study are program coverage (measured in statement coverage), reported failures, and the time and space costs.

6.3 Subject Applications and User Sessions

In our experiments we used two subject programs: an open-source, e-commerce **Bookstore** [11] *(Classes:11, Methods:385, Conditionals: 1808, NCLOC:7791, Seeded Faults: 39)* and a course project manager (**CPM**) *(Classes: 75, Methods: 172, Conditionals:1274, NCLOC:9300, Seeded Faults: 86)* developed and first deployed at Duke University in 2001.

Bookstore allows users to register, login, browse for books, search for books by keyword, rate books, add books to a shopping cart, modify personal information, and logout. Bookstore uses JSP for its front-end and a MySQL database for the backend. To collect 125 Bookstore user sessions, we sent email to local newsgroups and posted advertisements in the University's classifieds web page asking for volunteer Bookstore users. Since we did not include administrative functionality in our study, we removed requests to administration-related pages from the user sessions. The first row of Table 1 presents the characteristics of the collected user sessions.

In CPM, course instructors login and create *grader* accounts for teaching assistants. Instructors and teaching assistants set up *group* accounts for students, assign grades, and create schedules for demonstration time slots for students. CPM also sends emails to notify users about account creation, grade postings, and changes to reserved time slots. Users interact with an HTML application interface generated by Java servlets and JSPs. CPM manages its state in a file-based datastore.

We collected 261 user sessions from instructors, teaching assistants, and students using CPM during the 2004-05 academic year at the University of Delaware. The URLs in the user sessions mapped to the application's 60 servlet classes and its HTML and JSP pages. The row labeled **CPM_Orig** in Table 1 shows the user sessions' characteristics.

6.4 Methodology

In addition to the original suites, Table 1 lists the characteristics of our study's test suites. To create a medium-sized, non-contiguous test suite, we selected every other user session from the original suite of collected user sessions to create

Mod2. We applied Harrold et al.'s reduction technique [12] with conditional, method, statement and URL requirements to create the **HGS-C, HGS-M, HGS-S** and **HGS-U** test suites, respectively. We applied our concept analysis-based reduction technique [26] to create the **Con-U** test suite. Because the Harrold et al. reduction technique is nondeterministic, we executed the reduction technique 100 times to create 100 reduced test suites for each requirement (C, M, S, U). The second column of Table 1 shows the total URLs in (a) the mode suite of the 100 reduced suites and (b) the average of the set of 100 reduced suites. Since the **Mod2** and the concept analysis-based reduction are deterministic, we use only one test suite for each technique.

To collect program coverage, we used the Java coverage tool Clover [7]. In addition to the test suites from **Mod2** and **Con-U** for program coverage, we executed each of the 100 generated HGS reduced suites for each requirement and computed the mean program coverage.

As mentioned in Section 3, for the fault detection experiments, graduate and undergraduate students familiar with JSP/Java servlets/ HTML manually seeded faults in Bookstore and CPM. In general, we seeded faults in the application code that affect the program's control flow and the generated web pages as described in Section 3.4. For CPM, we used some faults from previous versions of the code. In this paper we show the results for faults reported by the mode reduced suite obtained by Harrold et al.'s reduction as well as the reduced suites from **Mod2** and **Con-U**.

We implemented the **Content** oracle in Perl and **Structure** oracle in Java—first filtering the HTML pages and then using `diff` to compare the filtered output. **Raw** and **Flist** use `diff` alone.

6.5 Threats to Validity

In our experiments, *internal threats to validity* arise from the relatively small number of user sessions. Since the limited nature of the original suite of user sessions could be considered an internal threat to validity, we created multiple test suites to use in our experiments to alleviate this threat. In addition, our applications may not be complex enough to show large differences in program coverage and fault detection when comparing the replay and oracle techniques. Our first subject application, Bookstore was an open-source code, and the second application, CPM, was developed and modified consistently by a single developer—thus threats that arise from different implementation styles are not a threat to internal validity of our study. In addition, the machines we used to run experiments were not dedicated to our experiments; other users, other experiments, backups, and network activity may affect the timing results. Since we do not consider the severity of the faults or their potential impact on the replay techniques and oracle comparators, our experiments are liable to a *construct threat to validity.*

In [2], Andrews et al. question the use of seeded faults because, for all but one of their subject programs, the seeded faults were more difficult to expose than naturally-occurring faults. Though we tried to model the seeded faults as closely as possible to naturally occurring faults—and even included faults from previous versions of CPM, some of the seeded faults may not be accurate representations of natural faults, resulting in an *external threat to validity.* Finally, since we conducted our experiments on two applications, generalizing our results to all web applications may not be fair.

Metrics	Book			CPM		
	State	No State	Dif	State	No State	Dif
Orig	59.2	N/A		69.3	N/A	
Mod2	58.9	58.9	0.0	64.4	58	6.4
HGS-C	59.1 (.07)	58.6 (.17)	0.5	67.5 (.05)	64.4 (1.1)	3.1
HGS-M	57.5 (.35)	54.1 (3)	3.4	64.6 (.08)	51.5 (.99)	13.1
HGS-S	59.2 (.05)	55.2 (.05)	3.9	69.3 (.0)	64.8 (.84)	4.5
HGS-U	41.9 (10.7)	32.7 (.63)	9.3	62.5 (.26)	47.2 (1.1)	15.4
Con-U	58.1	58.0	0.1	67.6	66.0	1.6

Table 2: Percent Statement Coverage, with and without restoring state

7. RESULTS AND ANALYSIS

7.1 Effect of Replay on Program Coverage

Table 2 shows the results of executing test suites on Bookstore and CPM. We used two replay techniques, `with_state` and `without_state`, for replaying the test suites and measured the corresponding statement coverage. The rows in Table 2 for the nondeterministic reduced suites (**HGS-C, HGS-M, HGS-S** and **HGS-U**) show the average statement coverage of the 100 generated reduced suites for each replay technique and the standard deviation in paranthesis. For the other reduced suites, since only a single suite is present, we only report its statement coverage.

We observe that `with_state` always covers more statements than `without_state`, and the standard deviation is less than the difference except for Bookstore's HGS-U. The difference is more pronounced in CPM than in Bookstore because CPM has more dynamic and real-time user session operations than Bookstore.

To verify that the two techniques cover different statements and not simply different amounts of code, we present CPM's statement coverage—the most precise coverage information we collected—for representative reduced suites using each replay technique in Figure 3. We do not show the results for Bookstore because the difference is not as pronounced as in CPM. The y-axis shows the test suites for each replay technique, and the x-axis denotes the randomly ordered individual statements that each test suite covers. Besides the difference in total statements covered, we see that test suites cover different statements (denoted by the x-values in the figure). On manual inspection of the code covered by the test suites with the different replay techniques, we observed that `without_state` was more likely to cover error code, such as code in catch blocks, than the corresponding `with_state` replay technique.

Contrary to our hypothesis, we did not observe a correlation between test suite size and the percent difference in covered statements. We note that the difference between the replay techniques is not in the amount of code covered but in which statements each technique covers.

7.2 Effect of Replay and Oracle on Failure Detection

Figures 4, 5, 6, 7 show the faults reported for `without_state` and `with_state` replay techniques for Bookstore and CPM, respectively. Since the standard deviation in program coverage for the nondeterministic reduced suites is small (from Table 2), we use the mode reduced suite in the failure detection experiments.

The x-axis represents the oracle comparators and the y-

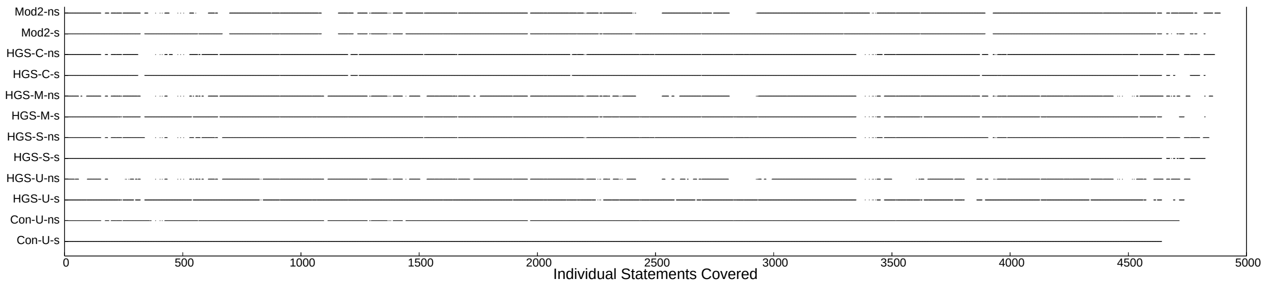

Figure 3: CPM statement coverage, with and without restoring state

axis denotes the faults seeded in the application. The total number of faults reported by the oracle and corresponding test suite is the numerical label above the individual faults. In all the figures, we observe that different oracles detect different numbers of faults. **Raw** reports the most faults, followed by **Content** and **Structure**, and **Flist** reports the least faults. Since **Raw** compares the raw HTML file, it is likely to report differences in real-time data as a fault, whereas **Content** and **Structure** oracles are more conservative. We also note that **Structure**, which compares differences in HTML tags, reports more *form faults* than the **Content** oracle. **Flist** reports mostly *logic* and *data store* faults because failures caused by *logic* faults may affect URL redirection or cause fatal exception and, therefore, affect the filenames that **Flist** uses for comparison.

We note differences between faults reported by with_state (Figures 5 and 7) and without_state (Figures 4 and 6). For the Original suites, without_state has more opportunities to detect failures because the framework does not correct faulty state; i.e., without_state exposed some faults that subsequent sessions could detect. In general, the reduced suites replayed with_ state reported more faults than without_state, which matches our coverage results from Table 2. Also, without_state exposed some faults that with_state did not detect, such as faults in error code.

To accurately evaluate our oracle comparators, we wanted to determine how many of the faults reported are *false positives*. A *false positive* is when the oracle reports a fault based on output differences, but, on manual inspection, the seeded fault did not cause the difference in the output. An example of a false positive is when the comparator detects a difference in the current time displayed because the expected result was generated earlier; however, displaying the current time is the correct application behavior. Due to the dynamic and real-time nature of CPM, we believe that false positives are more likely in CPM than in Bookstore. Since determining *false positives* and *false negatives* requires a huge amount of manual labor, we performed a small qualitative study on *false positives* for CPM. In the future we plan to investigate *false negatives* reported by our oracles.

In our inspection of the faults the original suite reported, we found that **Raw** and **Content** reported false positives for replay with_state. **Raw** reported errors when the selected option in a form changes, e.g., the selected start time changes from a.m. to p.m. **Content** reports an error when the order of form options changes, e.g., the order of available demos changed. However, in the without_state version **Content** did not report a fault because the demos printed in the same order as in the clean version. **Raw** reported false positives in without_state as well, but **Content** reported the same errors as **Structure**, for without_state.

Metrics	Bookstore		CPM	
	State	No State	State	No State
Replay Time Per URL	0.33 s	0.27 s	.04 s	.04 s
Replay Time Per Session	10.59 s	8.43 s	.55 s	.57 s
Space for Replay	4.2 MB	N/A	18 MB	N/A

Table 3: Replay: Time and Space Costs

While our study did not expose any false positives for **Structure**, we can construct situations where they would occur. However, we believe those situations are less realistic and will be easy to identify as false positives. We conclude that **Raw** is most likely to report false positives, followed by **Content** and **Structure**.

7.3 Relative Costs of Techniques

We present the cost of replaying test suites of user sessions in Table 3. When replaying a test suite, the cost is the time to send a URL request and receive and store the downloaded page and the time to restore the data state, if applicable. Since the framework updates data state per session, we report the replay time at the granularity of a single URL (first row in Table 3) and of a session (second row in Table 3). Replaying with_state requires additional space to store the data state updates.

The cost to update the data state between each session is relatively small: about 2 seconds for Bookstore and negligible for CPM. Bookstore stores 4.2 MB while CPM requires 18 MB in updates for their original suites. Our applications' space requirements are small, which is not realistic for many web applications. We did not implement any space optimizations, such as compressing the data or logging the differences between states, which will allow our approach to scale to larger state requirements. Replaying CPM is much faster than replaying Bookstore because of the cost of accessing and updating the database rather than a filestore.

We measured the time to perform an oracle comparison at the granularity of a URL. The comparison time varies because each comparator measures different properties in expected and actual output. **Raw**, **Content**, and **Structure** all require the same expected/actual output to perform the comparison, while **Flist** only requires the file names. The space occupied by the downloaded HTML pages is approximately 8 GB (149 MB compressed) for Bookstore and 4 GB (107 MB compressed) for CPM.

As expected, **Flist** (Book: 0.01s, CPM: 0.01s) and **Raw** (Book: 0.01s, CPM: 0.01s) are the cheapest operators be-

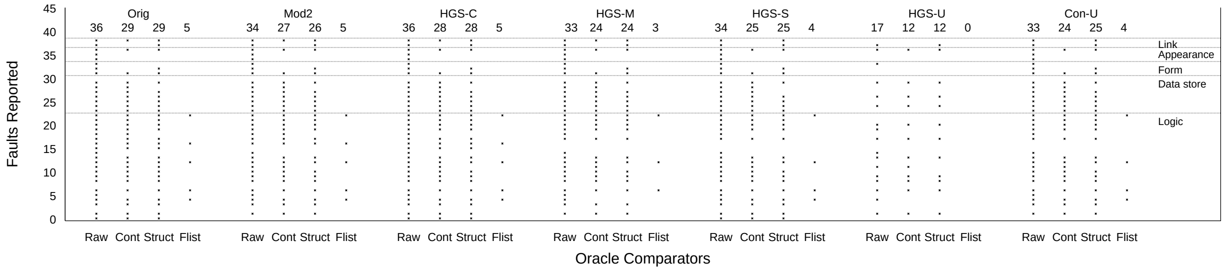

Figure 4: Faults Reported in Bookstore without State and Different Oracle Comparators

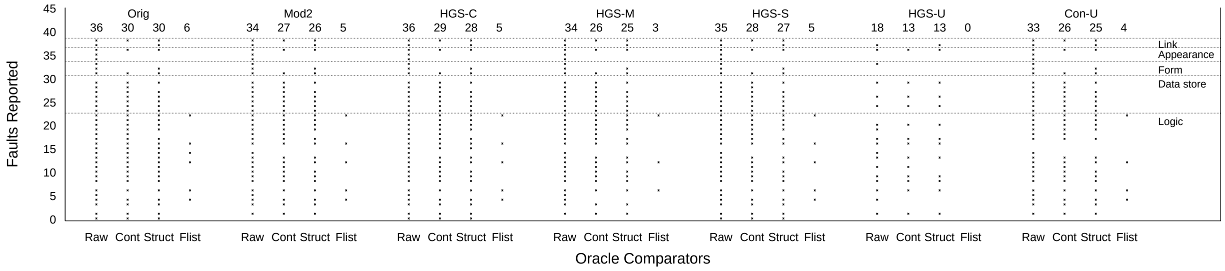

Figure 5: Faults Reported in Bookstore with State and Different Oracle Comparators

cause they require no additional filtering of the HTML output. **Content** (Book: 0.06s, CPM: 0.06s) and **Structure** (Book: 0.14s, CPM: 0.04s) required the most computing resources. We can reduce these costs by optimizing the implementations. Bookstore's HTML pages are larger than CPM's, which explains the higher cost for **Structure**.

7.4 Analysis Summary and Guidance to Testers

Based on our results, we suggest the following to testers:

Choice of Replay Technique for Program Coverage. If the web application has real-time dynamic data, the tester may want to execute test suites by both `with_state` and `without_state` replay techniques, since our results (Table 2) confirm the differences in code covered by the two techniques and Figures 3, 4, 5, 6, and 7 suggest that the covered code and reported faults by replaying with the two techniques do not exhibit a subset relation. However, if the application maintains little persistent state, the tester can execute the test suites using the `without_state` replay technique.

Choice of Replay Technique for Fault-based Testing. When conducting fault-based testing, the tester/maintainer may want to execute test suites `without_state` if the intent is to locate faults in error code of the application because our statement coverage and fault detection results suggest that `without_state` is likely to cover error code.

Choice of Oracle Comparator. From our results it is evident that the relation **Raw** > **Content**, **Structure** > **Flist** exists between the faults reported by the respective oracle comparators. Depending on the time available to the tester, the tester can choose to execute different oracles. For example, if the tester is not inclined to spend time tracking false positives, the tester should use a combination of **Content** and **Structure** oracles. However, the **Raw** oracle is an option if the tester is interested in comprehensive testing of the application—and in the process investigates some false positives. In addition, the **Flist** oracle can be used as a quick filter to detect faults prior to executing the other, more expensive oracles. Finally, depending on the kinds of faults the tester wants to find, different oracles may be used. It is intuitive to execute **Raw** and **Structure** oracle if the tester intends to isolate *form* faults. Similarly, all oracles except **Flist** can be used to isolate *appearance* faults.

An Automated Plug-and-play Framework. A tester/maintainer may find it advantageous to use a combination of plug-and-play oracles and replay techniques and have an automated framework such as ours that enables flexible combinations.

We expect that as the dynamic real-time content of the application increases, the faults reported by the **Structure** and **Content** oracles and between `with_state` and `without_state` will differ more. However, we need more subject applications with more real-time dynamic content to verify our intuition.

8. CONCLUSIONS AND FUTURE WORK

In this paper we presented two replay techniques and various oracle comparators in the context of an automated framework for testing web applications. We presented an experimental study evaluating the tradeoffs associated with different combinations and provide guidance to testers based on our results. Our results suggest that a combination of `with_state` and `without_state` replay covers more and different code than either technique alone. From our results we conclude that different oracle comparators expose different kinds of code faults. A combination of the oracle comparator and the replay technique can be used to detect faults in different areas of code. The primary limitation in our oracles is that we cannot quantify "equivalence" of actual and expected results, which cause false positives. As future work, we need to identify which differences "matter" and which ones are equivalent. In the future, we plan to calcu-

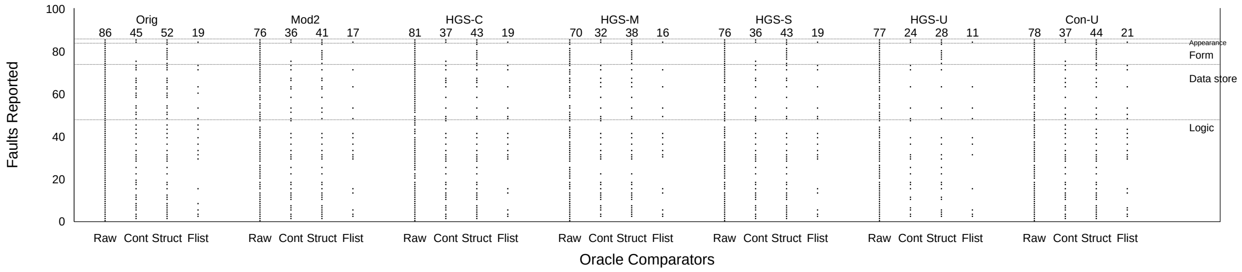

Figure 6: Faults Reported in CPM without State and Different Oracle Comparators

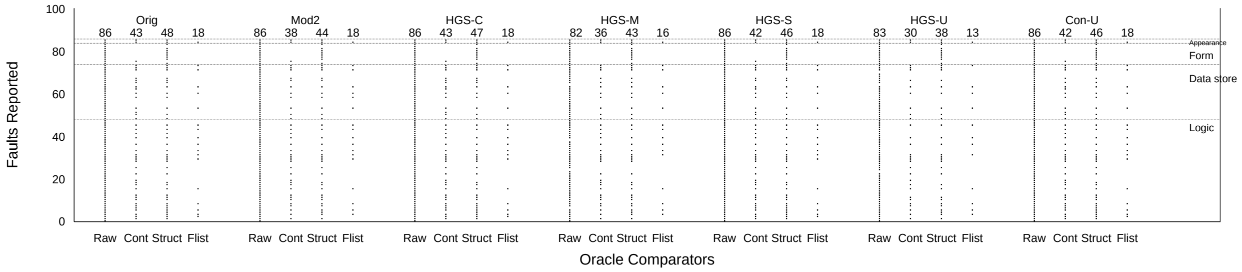

Figure 7: Faults Reported in CPM with State and Different Oracle Comparators

late user session dependencies as an alternate approach to managing data state and incorporate approaches to handle server state during replay. We also plan to explore concurrent user session replay instead of serialized replay.

Acknowledgments. We thank Jeffrey Chase and Richard Kisley for developing parts of the Course Project Manager. We also thank Stacey Ecott for developing one of the fault detection framework tools. Finally, we thank the anonymous reviewers for providing suggestions to improve our paper.

9. REFERENCES

[1] A. Andrews, J. Offutt, and R. Alexander. Testing web applications by modeling with FSMs. *Software Systems and Modeling*, 4(2), April 2005.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE*, 2005.

[3] L. Baresi and M. Young. Test oracles. Technical Report CIS-TR01-02, University of Oregon, 2001.

[4] R. Binder. *Testing Object-Oriented Systems*. Addison Wesley, 2000.

[5] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. In *SPIN Wksp*, 1996.

[6] D. Chays, Y. Deng, P. Frankl, S. Dan, F. Vokolos, and E. Weyuker. An agenda for testing relational database applications. *Soft Testing, Ver and Rel*, 14:17–44, Mar. 2004.

[7] Clover: Code coverage tool for Java. <http://www.cenqua.com/clover/>, 2005.

[8] L. K. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. In *SIGSOFT FSE*, 1994.

[9] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *ICSE*, 2003.

[10] S. Elbaum, G. Rothermel, S. Karre, and M. F. II. Leveraging user session data to support web application testing. *IEEE Trans on Soft Eng*, 2005.

[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 Trans on Soft Eng Meth*, 2(3):270–285, 1993.

[13] C. Hunter and P. Strooper. Systematically deriving partial oracles for testing concurrent programs. In *Proceedings of the 24th Australasian conference on Computer science*, 2001.

[14] I. Jacobson. The use-case construct in object-oriented software engineering. In J. M. Carroll, editor, *Scenario-based Design: Envisioning Work and Technology in System Development*, 1995.

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

[16] B. Long, D. Hoffman, and P. Strooper. Tool support for testing concurrent Java components. *Trans on Software Engineering*, 29(6):555–566, June 2003.

[17] G. D. Lucca, A. Fasolino, F. Faralli, and U. D. Carlini. Testing web applications. In *Int Conf on Software Maintenance*, 2002.

[18] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for guis. In *SIGSOFT/FSE-8*, 2000.

[19] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *Int Work on Dyn Anal (WODA)*, 2005.

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

[21] Rational Robot. <http://www-306.ibm.com/software/awdtools/tester/robot/>, 2005.

[22] Caucho resin. <http://www.caucho.com/resin/>, 2005.

[23] D. Richardson. TAOS: testing with analysis and oracle support. In *Int Sym on Software Testing and Analysis*, 1994.

[24] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *ICSE*, 1992.

[25] S. Sampath, V. Mihaylov, A. Souter, and L. Pollock. Composing a framework to automate testing of operational web-based software. In *Int Conf on Soft Main*, 2004.

[26] S. Sampath, V. Mihaylov, A. Souter, and L. Pollock. A scalable approach to user-session based testing of web applications through concept analysis. In *Automated Soft Eng Conf*, 2004.

[27] J. Sant, A. Souter, and L. Greenwald. An exploration of statistical models of automated test case generation. In *Int Work on Dyn Anal (WODA)*, 2005.

[28] S. Sprenkle, S. Sampath, E. Gibson, L. Pollock, and A. Souter. An empirical comparison of test suite reduction techniques for user-session-based testing of web applications. In *Int Conf on Software Maintenance*, 2005.

[29] S. D. Stoller. Model-checking multi-threaded distributed Java programs. *Int Journal on Soft Tools for Technology Transfer*, 4(1):71–91, Oct. 2002.

[30] E. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–70, 1982.

[31] GNU wget. <http://www.gnu.org/software/wget/>, 2005.