An Empirical Comparison of Test Suite Reduction Techniques for User-session-based Testing of Web Applications

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

Amie Souter
Computer Science
Drexel University
Philadelphia, PA 19104
souter@cs.drexel.edu

Abstract

Automated cost-effective test strategies are needed to provide reliable, secure, and usable web applications. As a software maintainer updates an application, test cases must accurately reflect usage to expose faults that users are most likely to encounter. User-session-based testing is an automated approach to enhancing an initial test suite with real user data, enabling additional testing during maintenance as well as adding test data that represents usage as operational profiles evolve. Test suite reduction techniques are critical to the cost effectiveness of user-session-based testing because a key issue is the cost of collecting, analyzing, and replaying the large number of test cases generated from user-session data. We performed an empirical study comparing the test suite size, program coverage, fault detection capability, and costs of three requirements-based reduction techniques and three variations of concept analysis reduction applied to two web applications. The statistical analysis of our results indicates that concept analysis-based reduction is a cost-effective alternative to requirements-based approaches.

1. Introduction

The demand for reliable, secure, and usable web-based applications continues to increase as private consumers, businesses, and government agencies rely on the Internet for important routine tasks. To increase the reliability of such applications, developers need automated, cost-effective test strategies that adequately test the continuously evolving web application under changing usage profiles. One effective approach to testing web applications is user-session-based testing, which relies on capturing and replaying real user sessions [7, 25, 19, 20]. As developers perform maintenance tasks, perhaps unintentionally inserting faults, they can use user-session-based testing to discover errors that users are likely to encounter. Late in the life cycle of the application, original user-session-based tests are unlikely to represent the current usage of the system. When the application is under maintenance, testers should use recent user sessions when testing. These tests provide more relevant feedback to developers who are updating the older code base, detecting potential faults exposed by current use.

A major problem with user-session-based testing is the cost of collecting, analyzing, and replaying the large number of test cases generated from user-session data. For user-session-based testing to be cost-effective, testers must be able to remove redundant test cases from the test suite without sacrificing much of the original suite's fault detection and coverage properties.

We have formulated the test suite reduction problem for user-session-based testing in terms of clustering user sessions by concept analysis. Our approach, `Concept`, can continually reflect the set of use cases representing actual executed user behavior in a minimal test suite. The approach exploits existing incremental concept analysis techniques to analyze the user sessions as they are captured and converted into test cases [24, 25]. `Concept` utilizes concept analysis to cluster together user sessions that represent similar use cases. By applying a heuristic to select user sessions from the clusters, `Concept` generates a reduced test suite that covers all the unique URLs of the original test suite, while representing the common URL subsequences of the original test suite's different use cases [26]. Our previous experiments indicated that the reduced suites have very little program coverage loss while significantly reducing the storage requirements [25].

`Concept` differs from test reduction techniques that were developed without primary focus on web applications. Other techniques require an association between test cases and the program's test requirements [11]. For example, a requirements-based approach determines an association be-

tween test cases and their statement coverage and then selects a representative set of test cases for the reduced test suite such that the test cases cover each executed statement at least once. To create this mapping, all test cases must be executed to map executed statements to corresponding test cases. Since `Concept` does not use any program coverage in its decisions, it does not need to collect program coverage information or create requirement mappings.

The primary contribution of this paper is an empirical study of the tradeoffs between reducing user-session test suites for web applications using requirements-based reduction techniques versus the `Concept` reduction approach. `Concept` does not require the overhead of program coverage collection and provides an on-the-fly approach. However, requirements-based reduction techniques may be more effective in reducing the test suite's size and maintaining fault detection capability. The goal of this paper is to gain insight into these tradeoffs. We have compared three variations of `Concept` with three requirements-based reduction techniques: `Random`, `Greedy`, and Harrold, Gupta, and Soffa's reduction [11][1]. We study each of the requirements-based techniques with respect to method, statement, conditional, and URL coverage criteria.

We compared the reduced test suite size, program coverage, fault detection, and time and space costs of each of the techniques for two web applications. The analysis of our results indicates that concept analysis-based reduction is a cost-effective alternative to requirements-based approaches.

In the next section, we provide background on web applications and user-session-based testing. We briefly describe each reduction technique and the foreseen tradeoffs in Section 3. Section 4 presents our experiment design. In Section 5, we present our results and analysis. We describe future work in Section 6.

2. User-session-based Testing

Broadly defined, a web application is an application that executes 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 [14].

In this paper, we target web applications written in Java using servlets and JSPs. The applications consist of a backend data store and a web server. However, because our technique only requires user sessions, we can extend it to other web technologies such as PHP.

[1] We do not compare incremental concept analysis as other reduction techniques do not have incremental approaches.

In user-session-based testing, each *user session* is a collection of user requests in the form of URL and name-value pairs (i.e., input field names and their values). A user session begins when a user from a new IP address makes a request from 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 can be applied to construct test cases from the collected user sessions [7, 19, 20, 28].

Elbaum et al. [7] demonstrated the fault-detection capabilities and cost-effectiveness of user-session-based testing. They found that user session techniques uncover certain types of faults but not faults associated with rarely entered data. They showed that increasing the number of collected user sessions increases the effectiveness—as well as the cost—of user-session-based testing.

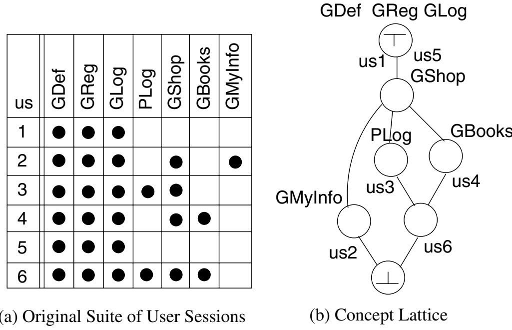
Techniques and tools have also been developed to aid in the initial stages of testing web applications [1, 6, 8, 16, 17, 18, 22]. User-session-based testing approaches are complementary to these techniques.

3. Test Suite Reduction Techniques

An evolving program can require augmenting an existing test suite with new test cases that test the program's new functionality. The additional test cases can lead to redundant test cases, which waste valuable testing resources, in the testing process. The goal of test suite reduction for a given test requirement (e.g., statement or all-uses coverage) is to produce a test suite that is smaller than the original suite's size yet still satisfies the original suite's test requirement. Advantages of test-suite reduction techniques include reducing the cost of executing, validating, and managing test suites as the application evolves.

Our study focuses on requirements-based reduction techniques and a concept analysis-based approach. Reduction techniques have also been developed based on bi-criteria models, which derive a reduced test suite that maximizes coverage and improves the error-detection rate of the suite [3]. Harder et al. [10] applied an operational difference technique to generate and minimize the number of test cases. Their technique dynamically generates operational abstractions from test suite executions by adding test cases until the operational abstractions do not change. However, the technique requires executing the test cases to determine their impact on the operational abstraction.

We use the example test suite in Figure 1(a) to illustrate our study's reduction techniques. The rows show a subset of user sessions from a bookstore web application. The corresponding URLs of the application label the columns. Each table entry represents a user session's URL request.



us	GDef	GReg	GLog	PLog	GShop	GBooks	GMyInfo
1	●	●	●				
2	●	●	●		●		●
3	●	●	●	●	●		
4	●	●	●		●	●	
5	●	●	●				
6	●	●	●	●	●	●	

(a) Original Suite of User Sessions (b) Concept Lattice

Figure 1. Example for Reduction Techniques

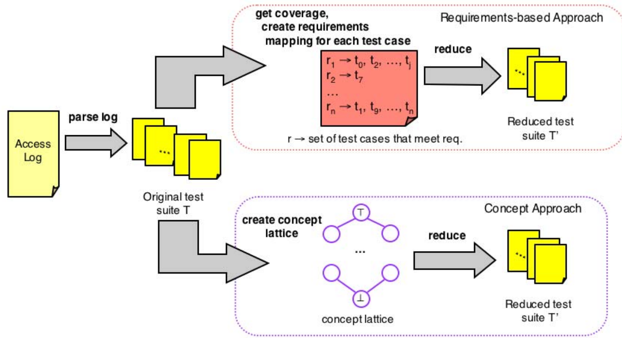


Figure 2. Test Suite Reduction Process

3.1. Requirements-based Approaches

We show the reduction process for requirements-based reduction techniques in Figure 2 and describe the **reduce** algorithm more formally: Let *P* be a program under test, *R* be a test requirement, *T* be a test suite developed for *P*, and *T'* be the reduced test suite obtained by analyzing *P* and *T*.

1. Begin with an empty reduced test suite, *T'*.
2. Select a candidate test case *t* from *T*.
3. Add *t* to *T'*.
4. Repeat 2 and 3 until the suite satisfies *R*.

The distinguishing property of requirements-based reduction approaches is test case selection in step 2 above [11, 13]. A naive approach, which we call `Random`, is to select test cases *randomly* until the suite satisfies the criterion. Consider reducing the test suite in Figure 1(a) using the `Random` approach with the *all-URL coverage* criterion. *all-URL coverage* covers each URL of the application at least once. The algorithm might first select us1 for the reduced suite and then select us6. Then, the algorithm might select us3 as a candidate and include it, even though us3 does not provide any more coverage than the current reduced test suite. The random selection process continues until the reduced test suite satisfies the *all-URL coverage* criterion. The final reduced test suite could thus be {*us1, us2, us3, us6*}. The disadvantage of randomly selecting test cases is that the reduced test suite may not be minimal for a given original test suite and criterion.

Alternatively, we can reduce the suite by choosing the test case that provides the most *marginal* improvement of the test suite in terms of the criterion. This reduction approach, which we call `Greedy`, selects test cases such that each subsequent test case provides maximum coverage of the targeted criterion.

In Figure 1(a), us6 is the candidate test case with maximum URL coverage and is the first test case selected for the reduced suite using `Greedy`. Then, since us2 provides the most marginal improvement for the *all-URL coverage* criterion, us2 is added to the reduced suite. The combination of us6 and us2 satisfies the *all-URL coverage* criterion; therefore, `Greedy` reduces the test suite to {*us6, us2*}. In the presence of ties, `Greedy` randomly selects a session.

In this example, `Greedy` reduces the test suite more than `Random`. However, `Greedy`'s disadvantage is the time to determine the test case that provides maximum coverage improvement. `Random` and `Greedy` are approximations to the solution of the minimum set covering problem. Wong et al. [30] developed a tool, ATACMIN, which uses heuristics to find exact solutions to the minimum set covering problem applied to test suite reduction.

Harrold et al. [11] developed a test suite reduction technique, which we call `HGS`, with a heuristic that selects a representative test set from the original test suite by approximating the optimal reduced set. Determining the optimal reduced set is an NP-complete problem [11]. Before the algorithm can reduce the test set, test cases must be associated with the requirements they meet. The number of test cases that cover a requirement is the requirement's cardinality. After a test case is added to the reduced set, the algorithm marks the test case's covered requirements. The algorithm then selects the most frequently occurring test case among the unmarked test cases with lowest requirement cardinality, i.e., the test case that covers the most unmarked requirements. In the case of ties, the algorithm chooses the test case that occurs most frequently at the next higher requirement cardinality. Breaking ties repeats until cardinality equals the maximum cardinality, at which point the algorithm randomly selects from the tied test cases. `HGS` stops selecting test cases when at least one test case in the reduced set covers each test requirement.

In Figure 1(a), the columns in the matrix represent the association between *all-URL coverage* requirement and the test cases. `HGS` will choose us2 first because it is the only user session that covers GMyInfo and marks the requirement GMyInfo as met. The algorithm then considers requirements with cardinality two—PLog and GBooks—and then selects the test case that occurs most frequently in the union of these two columns. Since us6 occurs two times while us3 and us4 each occur only once, us6 is added to the

reduced set, and the algorithm marks the requirements that us6 meets. Since the set {*us2,us6*} meets all requirements, `HGS` selection halts. `HGS` can also reduce the test suite with multiple test selection criteria. For example, the algorithm can be used to reduce the test suite such that it satisfies both structural and functional criteria.

Elbaum et al. [7] applied `HGS` to a suite of user sessions—consisting of URL and name–value pairs for each user session—with the selection criterion of program coverage. The authors reported that they achieve 98% reduction with a loss of three faults when applied at function-level and 93% reduction at block-level with one less fault detected. Their application of `HGS` requires the entire original test suite of user sessions. We are unaware of any algorithms that incrementally update the reduced test suite using `HGS`.

3.2. Concept Analysis Approach

Concept analysis is a mathematical technique for clustering *objects* that have common discrete *attributes* [2]. To apply concept analysis to the test suite reduction in user-session-based testing, the objects, O, represent the information uniquely identifying user sessions (i.e., test cases) and attributes, A, represent URLs.

As shown in Figure 2, concept analysis-based test suite reduction first performs concept analysis to build a concept lattice where each node of the lattice is a tuple (O_i, A_i) such that all the objects in $O_i \subseteq O$ share all and only the attributes in $A_i \subseteq A$ and vice versa. The edges of the lattice denote the partial ordering between the concept nodes. Figure 1 (b) shows the sparse representation of the concept lattice for the example user sessions. In user-session-based-testing, the test cases (i.e., user sessions) are representative of the applications' use cases [12]. We developed a heuristic based on the concept lattice for selecting a subset of user sessions to be maintained as the current test suite. Our heuristic for user-session selection, called *test-all-exec-URLs*, seeks to identify the smallest set of user sessions that covers the original test suite's executed URLs and use cases. The reduced test suite contains a user session from the bottom node, $\perp$[2] of the concept lattice, and a user session from each concept node that is one level up the lattice from $\perp$ (also called *next-to-bottom* nodes). In Figure 1(b) the *next-to-bottom* nodes are labeled by us2 and us6; therefore by applying the `Concept` heuristic, the reduced test suite satisfying the *all-URL coverage* criterion would be {*us2, us6*}.

Though the `Concept` approach may not result in the minimum test suite for a given criterion, we believe this approach maintains the original suite's use case representation in the reduced test suite [27]. In addition, we have shown that the concept analysis approach is able to incrementally update the test suite of user sessions so the original suite need not be maintained for test suite reduction. Details on applying concept analysis and the heuristic for test suite reduction can be found in our previous papers [26, 25].

[2]The $\perp$ contains the URLs that all the user sessions request.

Metrics	Bookstore	CPM
Classes	11	75
Methods	385	172
Conditions	1808	1274
Non-comment LOC	7791	9300
Seeded Faults	40	86
Number of User Sessions	125	261
Total URLs Requested	3640	3881
Largest User Session	160 URLs	152 URLs
Average User Session	29 URLs	15 URLs

Table 1. Objects of Analysis

3.3. Expected Cost-Benefit Tradeoffs

We expect the different test suite reduction techniques to vary in the size, contents, program coverage, and fault detection capability of the reduced test suite, as well as reduction time and space costs. We expect `Concept`'s test suite size to be larger than that generated by `HGS`. However, we believe the tradeoff in size will be offset by the fault detection effectiveness of `Concept` because of the heuristic's use-case clustering. In our earlier small examples of each algorithm, the reduced test sets were similar or identical. In practice, we believe the `Concept` test suite will be more representative of the application's use cases than the test suites generated by the other techniques. We expect the overhead for building the requirement mappings to dominate the time for the requirement-based reduction techniques. We also expect the concept lattice and the requirements mapping to have similar space needs.

4. Empirical Study

We compare three requirements-based approaches (with different coverage criteria) and three variations of the concept analysis approach by analyzing their reduced test suite size, program coverage, fault detection effectiveness, and time and space requirements. Contradictory results [23, 30] indicate that a reduction technique's fault-detection capability can vary with the program domain. Our goal is to evaluate the tradeoffs of the concept analysis-based approach (in addition to the benefits of incremental test suite update) with respect to other techniques for web applications.

In this section, we present our goals and design.

4.1. Research Questions and Hypotheses

We designed our study to investigate the following research questions and hypotheses:

Question 1. By what percentages do the various reduction techniques reduce the test suite size?

Question 2. How effective is the program coverage of the reduced suites as compared to the original suite?

Question 3. Compared to the original suite, how effective are the reduced suites with respect to fault detection?

Question 4. What are the relative time and space costs of the reduction techniques?

Our hypotheses with regard to the questions are

Suite Size Hypothesis: Concept will generate larger reduced suites than HGS and Greedy but smaller than those generated by Random. Because of concept analysis's hierarchical clustering, we expect Concept's reduced suite to be more diverse in terms of use case representation,while avoiding redundancy.

Coverage Hypothesis: The program coverage of the reduced suite generated by Concept will be similar to the original suite's coverage and less than the suites generated to satisfy the program-based requirements. Because of its use-case representation, Concept's reduced suite will have higher URL coverage than Greedy or HGS with the URL criterion. The relations may not hold between Random and Concept for both the URL criterion and program coverage criteria because we believe a tradeoff between reduced test suite size and the suite's coverage exists.

Fault Detection Hypothesis: Concept's reduced suite will have similar fault detection to the original suite and greater fault detection than the requirements-based approaches with the URL criterion because of the use-case representation. Concept and the requirements-based techniques with program coverage criteria will detect similar numbers of faults. The relations may not hold between Random and Concept because the larger test suite size will increase Random's ability to detect faults.

Costs Hypothesis: To perform the reduction, Concept only requires the concept lattice, while HGS, Greedy, and Random need to create and store the requirement mappings. As a result, Concept will require far less space and time overhead than HGS, Greedy, and Random.

4.2. Subject Programs and Data Collection

Table 1 shows the characteristics of our two subject programs: an open-source, e-commerce Bookstore [9] and a course project manager (CPM) 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 MySQL database for the backend.

To collect our 125 user sessions for Bookstore, we sent email to local newsgroups and posted padvertisements, asking for volunteer Bookstore users to mimic their typical online bookstore usage. Since we did not include administrative functionality in our study, we removed requests to administration-related pages from the user sessions. 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 and students using CPM during the 2004-05 summer, fall, and spring sessions at the University of Delaware. The URLs in the user sessions mapped to the application's 60 servlet classes and to its HTML and JSP pages.

For the fault detection experiments, graduate and undergraduate students familiar with JSP/Java servlets/HTML manually seeded realistic faults in Bookstore and CPM. In general, the seeded faults inserted errors into the application's control flow, the generated UI web pages, and/or the datastore interactions.

4.3. Variables and Measures

Our study's *independent variable* is the applied reduction technique with its reduction requirement. As mentioned earlier, we do not compare incremental concept analysis as we are not aware of any existing incremental approaches for the other reduction techniques. The *dependent variables* are the size, program coverage, and fault detection effectiveness of the reduced test suite and the reduction's time/space costs. A tester wants small reduced suites, which require little replay time. Thus, we measure size both as a percent of the test cases in the original test suite and as the total number of URLs requested. We include requested URLs in our results as a better replay-time and space approximation than the number of test cases because the user sessions vary in number of requested URLs.

4.4. General Methodology

For each subject application, our experiments consisted of three phases: generating user sessions, creating reduced suites for each technique, and replaying each reduced suite to collect coverage and fault detection data.

Generating User Sessions. We augmented the Resin [21] web server's access logging class to record the GET and POST name-value pairs. We then parsed the web server's augmented access log to create the user sessions. A user session begins when a user from a new IP address accesses the application and ends when the user leaves the application or when the session expires after 45 minutes of inactivity. Because image requests do not affect the application's execution, we do not include them in our user sessions.

Creating Reduced Suites. We generated reduced suites for the requirements-based and concept analysis techniques, as shown in Figure 2.

For the requirements-based approaches, we map user sessions to requirements (program or URL coverage). The three techniques (Random, Greedy, HGS) use the mappings to generate reduced test suites, labeled in our results as Ran-S, Ran-M, Ran-C, Ran-U, Grd-S, Grd-M, Grd-C, Grd-U, HGS-S, HGS-M, HGS-C, HGS-U, where

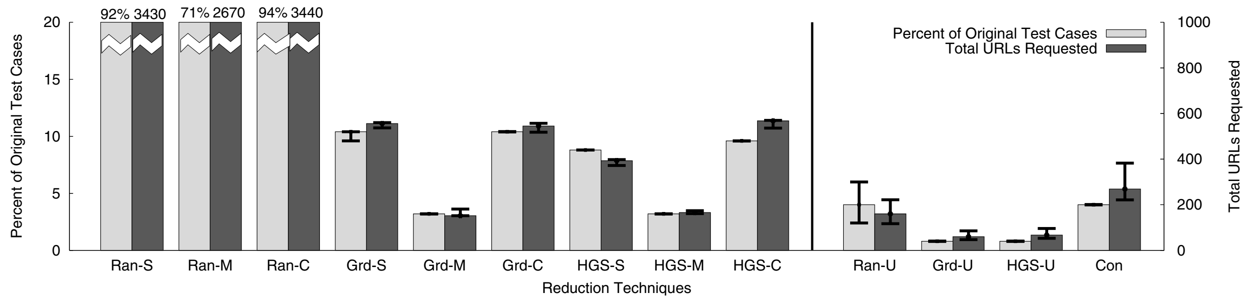


Figure 3. Bookstore: Reduced Suite Size for Reduction Techniques

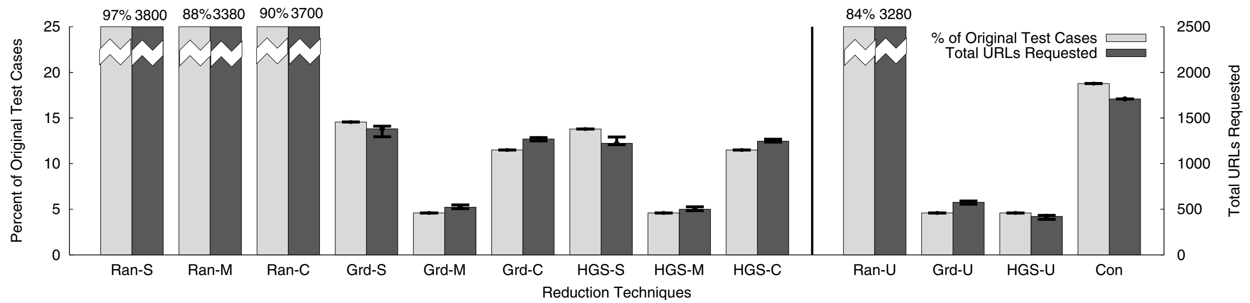


Figure 4. CPM: Reduced Suite Size for Reduction Techniques

`S`, `M`, `C`, and `U` represent statement, method, conditional, and URL, respectively. We implemented the mapping programs in Perl and the reduction techniques in Java. Since the requirements-based reduction techniques are non-deterministic, we executed each reduction algorithm 100 times for each reduction criterion. We chose to generate 100 reduced suites to ensure that the sample means of our results are normally distributed.

We apply concept analysis to the URL coverage requirement only. Given the original suite of user sessions, Lindig's `concepts` tool [15], written in C, creates a lattice that clusters sessions by their common requested URLs. We apply our heuristic *test-all-exec-URLs*, implemented in Java, on the generated lattice to create the reduced suite.

We generated two additional reduced suites from the lattice by changing how the heuristic selects user sessions from the selected concept nodes containing multiple user sessions. Our hypothesis is that sessions clustered together should have the same use cases and therefore detect the same faults [27]. Thus, the session chosen from each node should not affect the generated reduced suite's effectiveness. To speed up replaying `Concept`'s reduced suite, we created `Con-Min`, which selects from each *next to bottom* node the session that requests the least URLs. As another reference point, we include `Con-Max`, which selects the session that requests the most URLs.

Replaying Reduced Suites. We used the framework from [29] to measure program coverage and detected faults of each user session and the generated reduced suites.

Maintaining application state is a controllability issue for accurately replaying the reduced test suite [4]. The data store must be populated with appropriate data for the application to execute as it did originally. To maintain accurate state for our applications, we replay the original suite of user sessions and save the state after executing each session. We then restore the saved state as the current state when replaying the corresponding user session in the reduced suite.

We used Cenqua's Clover coverage tool [5] to collect our program coverage results. We used application-specific oracles to detect faults. For bookstore, we could use a simple `diff` to detect differences between the faulty and original web page results. Because CPM includes dynamic behavior, such as time-dependent functionality, we used an oracle that detects faults based only on differences in the HTML tag structure.

4.5. Threats to Validity

Internal Validity. While the Bookstore application is similar to a real e-commerce application, users could not complete a monetary transaction. The reduced functionality may have simplified the pool of sessions collected, thus presenting an internal threat to validity. To reduce another threat to internal validity, one developer implemented the requirements-based reduction techniques using a common reduction superclass.

Conclusion Validity. Accurate web application testing requires maintaining the state of the application. Our fault de-

Technique		Random				Greedy				HGS			
		S	M	C	U	S	M	C	U	S	M	C	U
H_0 (Null)		$\leq$ C	$\leq$ C	$\leq$ C	$\leq$ C	$\geq$ C	$\geq$ C	$\geq$ C	$\geq$ C	$\geq$ C	$\geq$ C	$\geq$ C	$\geq$ C
H_a (Orig)		$>$ C	$>$ C	$>$ C	$>$ C	$<$ C	$<$ C	$<$ C	$<$ C	$<$ C	$<$ C	$<$ C	$<$ C
Accept H_a?	**Book (% conf)**	81,79	62,61	88,87	50,54	95,99	100,63	99,86	99,70	99,78	99,81	99,95	99,71
	CPM (% conf)	99,98	67,61	98,95	67,61	99,73	100,97	99,97	100,99	99,86	100,99	99,99	100,99

Table 2. Significance Levels (percent of test cases, number of URLs) for Suite Size Hypothesis

tection and coverage results may be skewed because of how we maintain the application state. This could be viewed as a potential threat to the conclusion validity of the fault detection and coverage experiments.

External Validity. This empirical comparison needs to be evaluated further by experimenting with larger real-world web applications. The in-house nature of the subject applications and the data collection are threats to the external validity of the experiment.

5. Results and Analysis

5.1. Reduced Test Suite Size

We compared the size of the reduced test suites generated by each reduction technique. Figures 3 and 4 show the results for Bookstore and CPM, respectively. The left y-axis represents the number of test cases in the reduced suite as a percent of the original test suite size. The right y-axis represents the total URLs requested (including repeating URLs) by the user sessions in the reduced suite. The x-axis lists the reduction techniques. The solid bars represent the results from the **median** reduced suite for each technique. In addition to the median, for each of the nondeterministic techniques, we plot the **25th** and **75th** percentiles of test cases (for the lighter bars) and total URLs requested (for the darker bars). For `Concept`, the range lines represent the results for `Con-Min` and `Con-Max`. On the left side of the vertical line are the results from the nine program criterion requirements-based (`PRG_REQ`) techniques. To the right of the vertical line are the results from the URL criterion requirements-based (`URL_REQ`) and concept analysis (`URL_CON`) techniques. We consistently use this separated layout for all our graphs.

From Figures 3 and 4 all techniques except `Random` reduced the original test suite to at most a quarter of its original number of test cases. Among the `URL_REQ` and `URL_CON` techniques `HGS-U` and `Grd-U` provide the smallest reduced suite both in terms of number of user sessions and total URLs requested. `Concept` creates the largest reduced suites with the goal of maintaining use case representation. `URL_REQ` and `URL_CON` techniques produced the smallest reduced suites because there are many fewer URLs in the applications than methods, conditions, or statements (Table 1). Also, multiple user sessions requested each URL; thus, even `Ran-U` could produce small test suites. For example, twenty-seven user sessions requested `AdvSearch.jsp`, the least frequently accessed URL in Bookstore.

Table 2 shows the results of evaluating our null hypotheses using the t-test for **Suite Size Hypothesis**. For Bookstore (Figure 3), we can reject the null hypotheses for `HGS-M`, `HGS-U`, `Grd-M`, and `Grd-U` with statistical significance. Contrary to what we expected, `HGS-S`, `HGS-C`, `Grd-S`, and `Grd-C` create larger reduced suites than `Concept` both in the number of user sessions and total URLs requested. We believe Bookstore's simple functionality and few use cases explain this anomaly. We are investigating if the larger reduced suites contain redundant use cases that `Concept` eliminates.

The CPM reduced suites (Figure 4) for all techniques (with a more pronounced difference for `Concept`) contain more test cases than Bookstore because of CPM's complex functionality and larger number of classes, URLs, and use cases. CPM's user sessions were more diverse than Bookstore's; some sessions uniquely accessed certain URLs, and therefore, certain code. The diversity in user sessions resulted in no variations of `Concept` because each next to $\perp$ node contained only one session. Therefore, `Concept` results do not have error bars for CPM.

From Table 2, CPM's results support rejecting our null hypotheses for number of test cases `Greedy` and `HGS` at the .05 significance (95% confidence) level. Therefore we conclude the `Greedy` and `HGS` reduction techniques produce smaller suites than `Concept`. The `Random` techniques produced larger suites than `Concept`, and we can reject the `Ran-S` and `Ran-C` null hypotheses with 99% confidence. Because of the high variability in size for `Ran-M` and `Ran-U`, we cannot reject these null hypotheses with statistical significance. However, the test suites generated by `Ran-M` and `Ran-U` are not desirable because their size characteristics are inconsistent across executions.

5.2. Program Coverage

Figures 5 and 6 show the results from program coverage for each of our subject applications. We represent the reduction techniques along the x-axis and percent program coverage along the y-axis. The horizontal lines represent the original test suite's method/statement/condition coverage. The lines on each bar show the range of coverage for the 25th and 75th percentiles of the nondeterministic approaches and the minimum and maximum program coverage for `Concept`.

We can make similar, expected observations about Bookstore's and CPM's results: larger reduced suites cover more

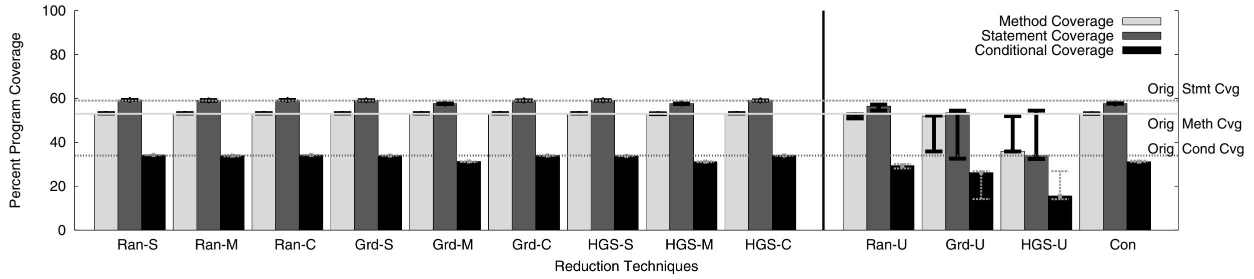


Figure 5. Bookstore: Program Coverage Effectiveness of Reduced Test Suites

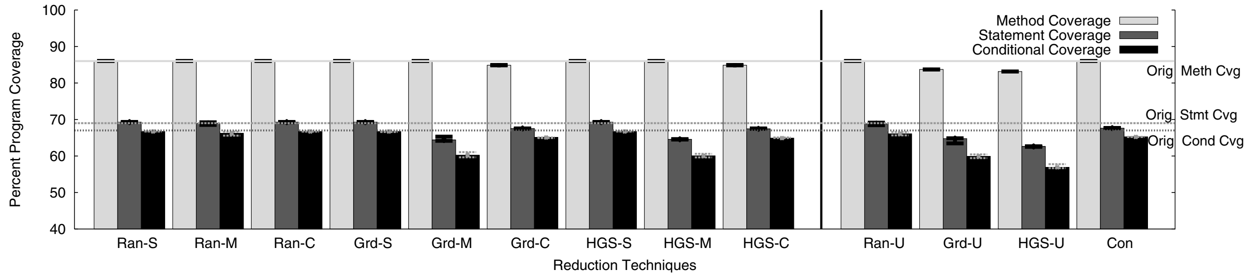


Figure 6. CPM: Program Coverage Effectiveness of Reduced Test Suites

code. For the PRG_REQ techniques, the reduced suites produced the same coverage for the requirement as the original suite because the reduced suite was generated to satisfy the program coverage requirement, e.g., reduced suites generated to satisfy the statement-coverage requirement covered as many statements as the original suite. Overall, the techniques generated for statement coverage covered the most code in terms of methods, statements, and conditionals.

Though `Concept` is URL-based reduction, it provides program coverage comparable to (within 2% of) PRG_REQ techniques. `Concept` has slightly less coverage than the original suite and `Random`'s reduced suites, which contain many more test cases than `Concept`. `Concept` provides more program coverage than the URL_REQ techniques, `Greedy` and `HGS`. We note that the samples for `Grd-U` and `HGS-U` have standard deviations of 10. Thus, a tester will not know if the generated test suite from `Grd-U` and `HGS-U` will obtain high coverage.

5.3. Fault Detection Effectiveness

Figures 7 and 8 show each technique's fault detection. The x-axis represents the reduction techniques. The left y-axis represents the percent of the original suite's detected faults that the reduced suite also detects, and the right y-axis represents the average number of faults detected per test case. Bookstore's original suite detected 36 of the 40 faults seeded, while CPM's original suite detected 40 of the 86 seeded faults (using HTML-tag oracle). The lines on each bar represent the range for the 25th and 75th percentiles of fault detection (for the lighter bars) and the number of faults per test case (for the darker bars) for the nondeterministic approaches and the minimum and maximum for `Concept`.

Similar to the coverage experiments, the number of faults detected by the reduction techniques is directly related to the size of the reduced suite. `Random` PRG_REQ techniques achieve the best fault detection at the cost of larger reduced suites. The reduced test suites generated by `Random` PRG_REQ techniques have low numbers of faults per test case as compared to the other suites.

For both Bookstore and CPM, the `Concept` variations show similar fault detection to the best PRG_REQ techniques, detecting all but one of CPM's faults and all but two of Bookstore's faults. `Concept` also detects more faults than `HGS-U`.

5.4. Analysis of Time and Space Costs

Figure 9 depicts the major stages of the requirements-based and concept analysis approaches as nodes with edges indicating the flow between phases. The `Concept` phases for test suite reduction are Parse Log, Create Lattice and Reduce, while Parse Log and the remaining phases are part of the requirements-based techniques, illustrated in Figure 2. Bookstore and CPM's measurements are prefaced by **B:** and **C:**, respectively. We present approximate measurements for the times for each phase as well as the space for the concept lattice and requirement mappings. The times in each reduce phase is the range of time required to generate one reduced suite for each technique.

The size of the requirement mappings is the product of the number of requirements, e.g., statements, methods,

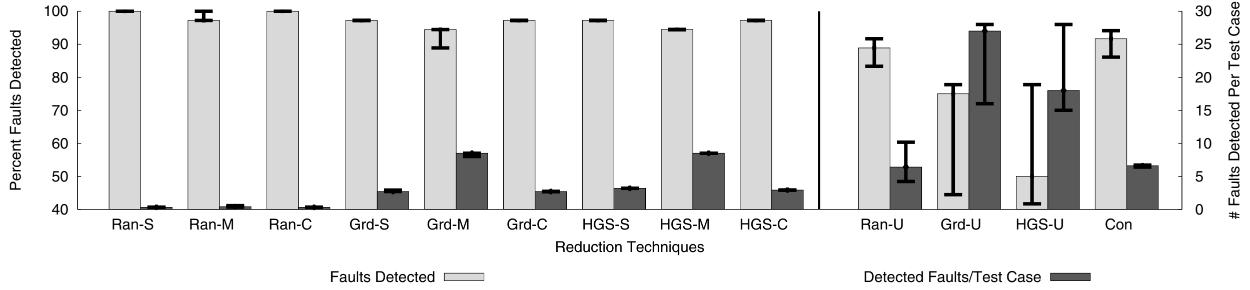


Figure 7. Bookstore: Fault Detection Effectiveness of Reduced Test Suites

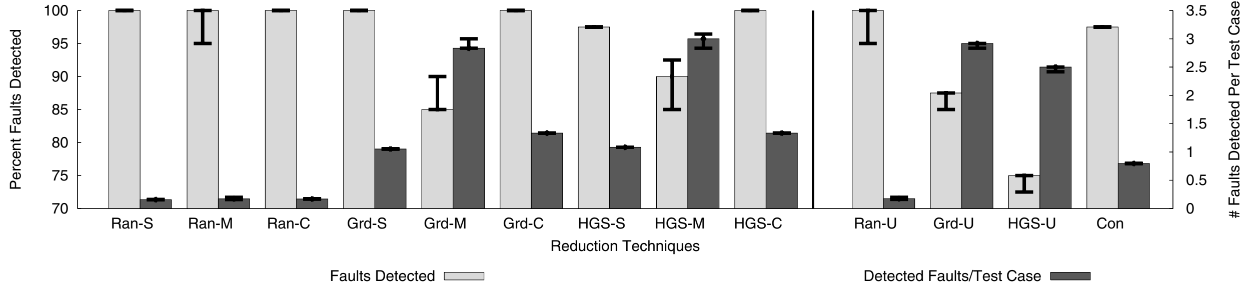


Figure 8. CPM: Fault Detection Effectiveness of Reduced Test Suites

conditionals, or URLs, and the number of sessions that cover each requirement. For Bookstore, the concept lattice's space requirements are smaller than all of the program coverage requirement maps and considerably smaller than statement coverage. In contrast, CPM's space needs for the requirement mappings in all cases but statement coverage are less than the concept lattice—the statement map is four times larger than the concept lattice.

We report all time measurements in seconds except for the requirement mapping construction for program coverage, which is reported in hours as appropriate. None of our developed tools were optimized for performance and none are meant for production-level work. However, we can conclude that the reduction techniques have comparable execution times. We did not measure the time to generate each requirements-based reduced suite because collecting coverage information for each session (the shaded phase) is clearly the bottleneck of the requirements-based approaches. Considering all steps required in the reduction process, `Concept` costs considerably less than the `PRG_REQ` techniques.

5.5. Analysis Summary and Discussion

The experiments provide evidence that URL-based reduced test suites are competitive with program coverage requirements-based techniques with respect to reduced test suite size, program coverage, and fault detection. As expected, there is a tradeoff between reduced suite size and fault detection, but other factors—such as choosing

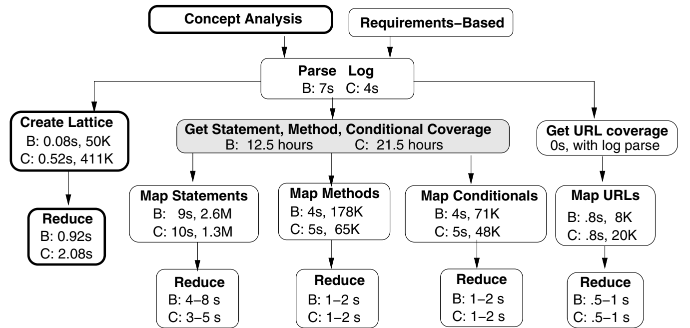


Figure 9. Approximate Time and Space Costs

test cases based on their use case representation—can result in better fault detection. A problem with the `HGS`, `Greedy`, and `Random` reduction techniques is their nondeterminism. Even in our small case studies, we noted some techniques generated reduced suites with a wide range in size, coverage, and fault detection. With `Concept`, the reduced suites have consistently small size, high coverage, and high fault detection. From our experiments, the **Suite Size Hypothesis** is satisfied when `Concept` is compared to `URL_REQ` techniques; however, the hypothesis does not hold for some of the `PRG_REQ` techniques. Our results support the **Coverage Hypothesis**, **Fault Detection Hypothesis** and the **Costs Hypothesis** as discussed in Sections 5.2,

5.3 and 5.4.

We argue that incremental test suite reduction based on concept-analysis clustering of user sessions is necessary for a web-based system that undergoes maintenance, evolution and usage changes. In this paper, we empirically evaluate several reduction algorithms to identify the tradeoffs in practice. `Concept`'s significantly lower reduction cost coupled with its high program coverage and fault detection effectiveness promote `Concept` as an appealing alternative to test suite reduction for user-session-based testing of web applications.

6. Future Work

Future work includes performing the experiments on different sets of user sessions as well as with more applications and larger user session sets. We are also investigating a solution to the data state problem for more accurate replay of user sessions.

Acknowledgments

We thank Dr. Allen Gibson, a statistics instructor with the Stillman School of Business at Seton Hall University, for his assistance in the statistical analysis of our results. We also thank Jeffrey Chase, Richard Kisley, and Sara Sprenkle for their development efforts of CPM. Finally, we thank the anonymous reviewers for their constructive feedback.

References

[1] M. Benedikt, J. Freire, and P. Godefroid. VeriWeb: Automatically testing dynamic web sites. In *International Conference on World Wide Web*, May 2002.

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

[3] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *International Conference on Software Engineering*, 2004.

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

[5] Cenqua clover code coverage for Java. <http://www.cenqua.com/clover/>, 2005.

[6] Y. Deng, P. Frankl, and J. Wang. Testing of web database applications. In *Workshop on Testing, Analysis and Verification of Web Services*, July 2004.

[7] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher. Leveraging user session data to support web application testing. *IEEE Transactions on Software Engineering*, May 2005.

[8] C. Fu, B. Ryder, A. Milanova, and D. Wonnacott. Testing of Java Web Services for Robustness. In *International Symposium on Software Testing and Analysis*, July 2004.

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

[10] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *International Conference on Software Engineering*, 2003.

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

[12] 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.

[13] J. A. Jones and M. J. Harrold. Test suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3), March 2003.

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

[15] C. Lindig. Concepts tool. <http://www.st.cs.uni-sb.de/ lindig/src/concepts.html>, 2005.

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

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

[18] J. Offutt and W. Xu. Generating test cases for web services using data perturbation. In *Workshop on Testing, Analysis and Verification of Web Services*, July 2004.

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

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

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

[22] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Int'l Conf. on Software Engineering*, 2001.

[23] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong. Empirical studies of test suite reduction. *Journal of Software Testing, Verification, and Reliability*, 4(2), Dec. 2002.

[24] S. Sampath, V. Mihaylov, A. Souter, and L. Pollock. Composing a framework to automate testing of operational web-based software. In *International Conference on Software Maintenance*, September 2004.

[25] 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 Software Engineering Conference*, September 2004.

[26] S. Sampath, A. Souter, and L. Pollock. Towards defining and exploiting similarities in web application use cases through user session analysis. *Workshop on Dyn Analysis*, May 2004.

[27] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. Souter. Analyzing clusters of web application user sessions. *Workshop on Dyn Analysis*, May 2005.

[28] J. Sant, A. Souter, and L. Greenwald. An exploration of statistical models of automated test case generation. *Workshop on Dyn Analysis*, May 2005.

[29] 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. Technical Report 2005-09, Univ. of Delaware, April 2005.

[30] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. *International Conference on Software Engineering*, 1995.