A Scalable Approach to User-session based Testing of Web Applications through Concept Analysis

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

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

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

Abstract

The continuous use of the web for daily operations by businesses, consumers, and government has created a great demand for reliable web applications. One promising approach to testing the functionality of web applications leverages user-session data collected by web servers. This approach automatically generates test cases based on real user profiles. The key contribution of this paper is the application of concept analysis for clustering user sessions for test suite reduction. Existing incremental concept analysis algorithms can be exploited to avoid collecting large user-session data sets and thus provide scalability. We have completely automated the process from user session collection and reduction through replay. Our incremental test suite update algorithm coupled with our experimental study indicate that concept analysis provides a promising means for incrementally updating reduced test suites in response to newly captured user sessions with some loss in fault detection capability and practically no coverage loss.

1. Introduction

As the quantity and breadth of web-based software systems continue to grow at a rapid pace, the issue of assuring the quality and reliability of this software domain is becoming critical. Low reliability can result in serious, detrimental effects for business, government, and consumers, as they have become increasingly dependent on the internet for routine daily operations. A major impediment to producing reliable software is the labor and resource-intensive nature of software testing. A short time to market dictates little motivation for time-consuming testing strategies. For web applications, additional challenges complicate testing beyond the analysis and testing considerations associated with more traditional domains.

Many of the current testing tools address web usability, performance, and portability issues [35]. For example, link testers navigate a web site and verify that all hyperlinks refer to valid documents. Form testers create scripts that initialize a form, press each button, and type pre-set scripts into text fields, ending with pressing the submit button. Compatibility testers ensure that a web application functions properly within different browsers.

Tools such as Cactus [4], which utilizes Junit [16], provide test frameworks for unit testing the functionality of Java-based web programs. In addition, web-based analysis and testing tools have been developed that model the underlying structure and semantics of web programs [22, 10, 30] towards a white-box approach to testing. While these white-box techniques enable the extension of path-based testing to web applications, they often require identifying input data that will exercise the paths to be tested, which are not covered by test cases generated from functional specifications.

One approach to testing the functionality of web applications that addresses the problems of the path-based approaches is to utilize capture and replay mechanisms which record user-induced events, gathering and converting them into scripts, which are then replayed for testing [9, 28]. Tools such as WebKing [26] and Rational Robot [28] provide automated testing of web applications by collecting data from users through minimal configuration changes to the web server. The recorded events are typically URLs and name-value pairs sent as requests to the web server. The ability to record these requests is often built into the web server, so little effort is needed to record the desired events. In a controlled experiment, Elbaum, Karre, and Rothermel [9] showed that user-session data can be used to generate test suites that are as effective overall as suites generated by two implementations of Ricca and Tonella's white-box techniques [30]. This provides a promising approach to testing the functionality of web applications which relieves the tester from generating the input data manually and provides a way to enhance an original test suite with

test data that represents usage as the operational profile of the application evolves. Unfortunately, the fault detection capability appears to increase with larger numbers of captured user sessions—but the test preparation and execution time quickly becomes impractical. While existing test reduction techniques [14] can be applied to reduce the number of maintained test cases, the initial overhead of selection and analysis of the large user-session data sets is non-scalable.

This paper presents an approach for achieving 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 [15]. The key insight is the formulation of the test case reduction problem for user-session testing in terms of concept analysis. Existing incremental concept analysis techniques can then be exploited 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 case reduction is accomplished on the fly by grouping user sessions into similar clusters. Through the creative use of a number of existing tools and the development of some simple scripts, we automated 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 [31]. In our experiments, the resulting coverage provided by the reduced test suite is almost identical to the original suite of user sessions, with some loss in fault detection.

After providing background in Section 2, we describe formulation of the test case reduction problem for user-session based testing as a concept analysis problem in Section 3. In Section 4, we present our incremental reduced test suite update algorithm, which utilizes existing incremental concept analysis techniques. Section 5 presents our complete framework for automatic test case generation, coverage analysis and fault detection for web testing. Section 6 presents the research questions, setup, methodology and results of an experimental study, which focused on the cost-effectiveness of this approach in terms of test case reduction, and maintaining coverage and fault detection capability. Related work is included in Section 7, followed by conclusions and future work in Section 8.

2. Background

2.1. Web Applications

Broadly defined, a web-based software system consists of a set of web pages and components that interact to form a system that executes using web server(s), network, HTTP, and a browser, and in which user input (navigation and data input) affects the state of the system. A web page can be either static, in which case the content is fixed, or dynamic, such that its content may depend on user input.

Web applications may have a number of characteristics, including an integration of numerous technologies; modularization into reusable components that may be constructed by third parties; a well-defined, layered architecture; dynamically generated pages with dynamic content; and typically extend an application framework. Large web-based software systems can require thousands to millions of lines of code, contain many interactions among objects, and involve significant interaction with users. In addition, changing user profiles and frequent small maintenance changes complicate automated testing [17].

2.2. User-session Testing

In user-session based testing, data is collected from users of a web application by the web server. Each *user session* is a collection of user requests in the form of URL and name-value pairs. More specifically, a user session is defined as beginning when a request from a new IP address reaches the server and ending when the user leaves the web site or the session times out.

To transform a user session into a test case, each logged request of the user session is changed into an HTTP request that can be sent to a web server. A test case consists of a set of HTTP requests that are associated with each user session. Different strategies are applied to construct test cases for the collected user sessions. A key advantage is the minimal configuration changes that need to be made to the web server to collect user requests.

Elbaum et al. [9] provide promising results that demonstrate the fault detection capabilities and cost-effectiveness of user-session testing. In particular, they show user-session techniques are able to discover certain types of faults but will not uncover faults associated with rarely entered data. In addition, they show that the effectiveness of user session techniques improves as the number of collected sessions increases. However, the cost of collecting, analyzing, and replaying test cases also increases. Elbaum et al. [9] report that they are able to achieve a high percent of test case reduction with Harrold et al's. [14] test case reduction technique. However, current test case reduction techniques are based on performing the test reduction over a complete test set. All the user-session data is collected before the reduction process occurs. An incremental update is not performed as test cases are added.

2.3. Concept Analysis

Concept analysis is a mathematical technique for clustering objects that have common discrete attributes [3]. Con-

cept analysis takes as input a set O of objects, a set A of attributes, and a binary relation $R \subseteq O \times A$, called a *context*, which relates the objects to their attributes. The relation R is implemented as a boolean-valued table in which there exists a row for each object in O and a column for each attribute in A; the entry of $table[o, a]$ is true if object o has attribute a, otherwise false.

Concept analysis identifies all of the concepts for a given tuple (O, A, R), where a *concept* is a tuple $t = (O_i, A_j)$, in which $O_i \subseteq O$ and $A_j \subseteq A$. The tuple t is defined such that all and only objects in O_i share all and only attributes in A_j. The concepts form a partial order defined as $(O_1, A_1) \leq (O_2, A_2)$, *iff* $O_1 \subseteq O_2$. Similarly, the lattice can be viewed as a superset relation on the attributes, as $(O_1, A_1) \leq (O_2, A_2)$, *iff* $A_1 \supseteq A_2$. The set of all concepts of a context and the partial ordering form a complete lattice, called the *concept lattice*, which is represented by a directed acyclic graph with a node for each concept and edges denoting the $\leq$ partial ordering. The top element $\top$ of the concept lattice is the most general concept with the set of all of the attributes that are shared by all objects in O. The bottom element $\bot$ is the most special concept with the set of all of the objects that have all of the attributes in A.

3. Applying Concept Analysis

To apply concept analysis to user-session based testing, we use objects to represent the information uniquely identifying user sessions (i.e., test cases) and attributes to represent URLs. While a user session is considered to be a set of URLs and associated name-value pairs for accurate replay, we define a user session during concept analysis to be the the set of URLs requested by the user, without the name-value pairs and without any ordering on the URLs. This considerably reduces the number of attributes to be analyzed. We present evidence of the effectiveness of choosing single URLs as attributes in [32]. A pair (user session *s*, URL *u*) is in the binary relation iff *s* requests *u*. Thus, each true entry in a row *r* of the relation table represents a URL that the single user represented by row *r* requests. For column c, the set of true entries represents the set of users who have requested the same URL.

As an example, the relation table in Figure 1(a) shows the context for a user-session based test suite for a portion of a bookstore web application [12]. User sessions are uniquely identified by the user's IP address. Consider the row for the user, us3. The (true) marks in the relation table indicate that user us3 requested the URLs GDef, GReg, GLog, PLog and GShop. We distinguish a GET (G) request from a POST (P) request when building the lattice because they are essentially different requests. The first six rows of the table are the user sessions collected initially and form the original relation table. The last two rows are the user sessions that we intend to analyze incrementally (in Section 4). Based on the original relation table, concept analysis derives the lattice in Figure 1(b)(i) as a sparse representation of the concepts. Attribute sets are drawn just above each node, while object sets are drawn just below each node.

Properties. Lattices generated from concept analysis for test suite reduction will exhibit several interesting properties and relationships. Interpreting the sparse representation, a user session *s* requests all URLs at or above the concept uniquely labeled by *s* in the lattice. For example, the user session us3 requests PLog, GShop, GDef, GReg and GLog. A node labeled with a user session *s* and no attributes indicates that *s* requests no unique URLs. For example, us6 requests no unique URL. Similarly, all user sessions at or below the concept uniquely labeling *u* access the URL *u*. In the example, user sessions us2, us3, us4, and us6 access the URL GShop.

The $\top$ of the lattice denotes the URLs that are requested by all the user sessions in the lattice. In our example, GReg, GDef and GLog are requested by all the user sessions in our original test suite. The $\bot$ of the lattice denotes the user sessions that access all URLs in the context. Here, $\bot$ is not labeled with any user session, denoting that no user session accesses all the URLs in the context.

To determine the common URLs requested by two separate user sessions *s1* and *s2*, the closest common node c towards $\top$, starting at the nodes labeled with *s1* and *s2* is identified. User sessions *s1* and *s2* jointly access all the URLs at or above c. For example, user sessions us3 and us4 jointly access the URLs GShop, GDef, GReg, and GLog. Similarly, to identify the user sessions that jointly request two URLs *u1* and *u2*, the closest common node d towards $\bot$ starting at the nodes uniquely labeled by *u1* and *u2* is determined. All user sessions at or below d jointly request *u1* and *u2*. For example, user sessions us3 and us6 jointly request URLs PLog and GShop.

Use for Test Suite Reduction. Test suite reduction exploits the concept lattice's hierarchical clustering properties. In particular, we developed a heuristic for selecting a subset of user sessions to be maintained as the current test suite, based on the current concept lattice. Given a context with a set of user sessions as objects O, we define the *similarity* of a set of user sessions $O_i \subseteq O$ as the number of attributes shared by all of the user sessions in O_i. Based on the partial ordering reflected in the concept lattice, if $(O_1, A_1) \leq (O_2, A_2)$, then the set of objects O_1 are more similar than O_2. User sessions labeling nodes closer to $\bot$ are more similar in their set of URL requests than nodes higher in the concept lattice.

Our current heuristic for user-session selection, which we call *test-all-exec-URLs*, seeks to identify the smallest set of user sessions that will cover all of the URLs executed by

	GDef	GReg	GLog	PLog	GShop	GBooks	GMyInfo
us1	X	X	X				
us2	X	X	X		X		X
us3	X	X	X	X	X		
us4	X	X	X		X	X	
us5	X	X	X				
us6	X	X	X	X	X	X	
us7	X	X	X	X	X		X
us8	X		X			X	

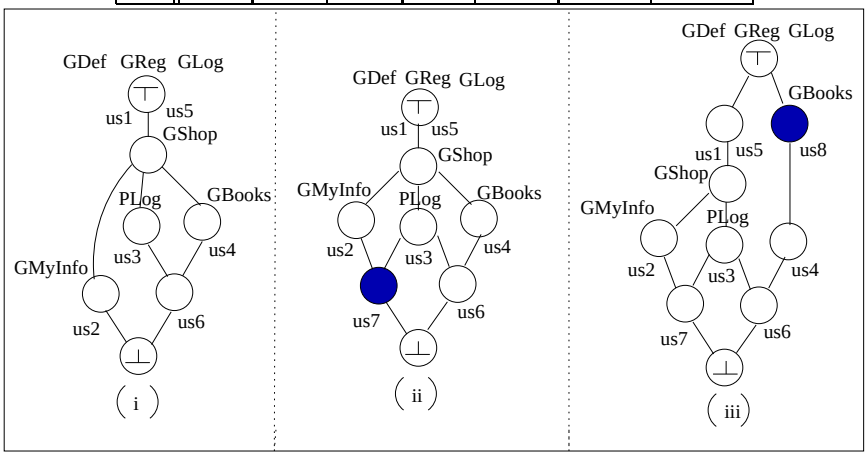

Figure 1. (a) Relation table and (b) concept lattices for test suite reduction

the original test suite while representing the common URL subsequences of the different use cases represented by the original test suite. This heuristic is implemented as follows: The reduced test suite is set to contain a user session from each node next to $\perp$, that is one level up the lattice from $\perp$. These nodes contain objects that are highly *similar* to each other. If the set of user sessions at $\perp$ is nonempty, those user sessions are also included in the reduced test suite.

We have performed experimental studies and user-session analysis to investigate the intuitive reasoning behind clustering user sessions based on concept analysis and our heuristic for user-session selection. These studies examined the commonality of URL subsequences of objects clustered into the same concepts and also compared the subsequence commonality of the selected user sessions with those in the remainder of the suite. The study is described fully in another paper [32]; the results provide support for using concept analysis with a heuristic for user-session selection where we choose representatives from different clusters of similar use cases. Thus, our approach differs from traditional reduction techniques such as selecting the next user session with most additional coverage until 100% coverage is obtained.

In our example in Figure 1, the original test suite is all the user sessions in the original context. The reduced test suite however contains only user sessions us2 and us6, which label the *next-to-bottom* nodes. By traversing the concept lattice to $\top$ along all paths from these nodes, we will find that the set of URLs accessed by these two user sessions is exactly the set of all URLs requested by the original test suite.

4. Incremental Reduced Test Suite Update

The key to enabling the generation of a reduced test suite with URL coverage similar to a test suite based on large user-session data sets, without the overhead for storing and processing the complete set at once, is the ability to incrementally perform concept analysis. The set of attributes A can be fixed to be the set of all possible URLs related to the web application being tested. The general approach is to start with an initial user-session data set and incrementally analyze additional user sessions with respect to the current reduced test suite. The incremental analysis results in an updated concept lattice, which is then used to incrementally update the reduced test suite.

More specifically, the incremental update problem can be formulated as follows:

> Given an additional user session s and a tuple (O, A, R, L, T), where O is the current set of user sessions (i.e., objects), A is the set of possible URLs in the web application (i.e., attributes), R is the binary relation describing the URLs that are requested by each user session in O, L is the concept lattice output from an initial concept analysis of (O, A, R), and T is the reduced test suite with respect to L, modify the concept lattice L to incorporate the user session s and its attributes, creating an updated lattice L' without building L'

Algorithm: Incremental Reduced Test Suite Update.
Input: Concept Lattice L
(Reduced) Test Suite T
Added user session s
Output: Updated Lattice L'
Updated Test Suite T'
begin
old-next-to-bottom-set = $\perp$
new-next-to-bottom-set = $\perp$
If object set labeling $\perp$ = $\oslash$
old-next-to-bottom-set = $\oslash$
Foreach node *n* with edge $(n, \perp)$ in L
Add n to *old-next-to-bottom-set*

Identify a = set of URLs in s
$L' = IncrementalLatticeUpdate(L, (s, \{a\}))$
If object set labelling $\perp$ = $\oslash$
new-next-to-bottom-set = $\oslash$
Foreach node n with edge $(n, \perp)$
Add n to *new-next-to-bottom-set*
addtests = *new-next-to-bottom-set – old-next-to-bottom-set*
deletetests = *old-next-to-bottom-set – new-next-to-bottom-set*
Foreach node n in $deletetests$
Let(o, a) be the label on n in the sparse lattice L'
Foreach user session s' in o
If $s' \in T'$ Delete s' from T'
Foreach node n in $addtests$
Let (o, a) be the label on n in the sparse lattice L'
Foreach user session s' in o
If $s' \notin T'$ Add s' to T'
end.

Figure 2. Reduced test suite update.

from scratch, and then incrementally update the reduced test suite T to T' with respect to L'.

Our incremental algorithm, shown in Figure 2, utilizes the incremental concept formation algorithm developed by Godin, Missaoui, and Alaoui [11]. Godin et al's. incremental lattice update algorithm takes as input the current lattice L and the new object with its attributes. Once an initial concept lattice L has been constructed from R, there is no need to maintain R. The incremental lattice update may create new nodes, modify existing nodes, add new edges, and change existing edges that represent the partial ordering. As a result, nodes that were at the *next-to-bottom* location may now be raised in the lattice, new nodes may have been created at the *next-to-bottom* location, but existing internal nodes will never "sink" to the *next-to-bottom* location because the partial ordering of existing internal nodes with respect to the existing *next-to-bottom* nodes is unchanged by addition of new nodes. These changes are the only ones that immediately affect the updating of the reduced test suite. Thus, test cases are not maintained for internal nodes. The design of the algorithm in Figure 2 allows for identifying test cases added or deleted from the current reduced test suite. If the software engineer is not interested in this change information, the update can simply replace the old reduced test suite by the new test suite by identifying the *new-next-to-bottom-set* after incremental lattice update.

Example. To demonstrate the incremental test suite update algorithm, we begin with the initial concept relation table (excluding us7 and us8 rows) and its corresponding concept lattice in Figure 1(b)(i). Consider the addition of the user session us7, which contains all URLs except the GBooks. Figure 1(b)(ii) shows the incrementally updated concept lattice as output by the incremental concept analysis. The changes include a new (shaded) node and the edge updates and additions, which move one of the old *next-to-bottom* nodes up in the concept lattice. Thus, the incremental update to the reduced test suite is a deletion of the user session us2 and addition of user session us7.

Now, consider the addition of the user session us8, which contains only three URLs as indicated by the last row of the augmented relation table. Figure 1(b)(iii) shows the incrementally updated concept lattice after the addition of both the user sessions us7 and us8. In this case, the new user session resulted in a new node and edges higher in the lattice. The *next-to-bottom* node set remained unchanged. Thus, the reduced test suite remains unchanged, and the new user session is not stored. Note that our previous studies [32] provide evidence that the use case represented by us8 is similar in short URL subsequences to use cases represented by objects in *next-to-bottom* nodes below the new node with us8.

Space and Time Costs. The initial batch concept analysis requires space for the initial user-session data set, relation table, and concept lattice. The relation table is **O(**$|O| \times |A|$**)** for an initial user-session data set of $|O|$ and $|A|$ URLs relating to the web application. The concept lattice can grow exponentially as $2^k|O|$, where k is the upper bound on the number of attributes of any single object in O; however, this behavior is not expected in practice [11]. In our application of concept analysis, k is limited by the number of URLs that any given user session would request relating to the web application; since k is fixed and a given user in a single session is expected to traverse a small percentage of a web application's URL set, the space requirements can be viewed as **O(**$|O|$**)** with a small constant k. There is no need for the relation table during the incremental reduced test suite update. We need to maintain space only for the current concept lattice (with objects and attributes), current reduced user-session data set, and new user sessions being analyzed.

Time for the batch algorithm is exponential in the worst case, with time **O(**$2^k|O|$**)**, while the incremental algorithm for our problem where the number of attributes is fixed is **O(**$|O|$**)**, linearly bounded by the number of user sessions in the current reduced test suite [11].

5. Framework for Web Testing

To apply concept analysis to user-session data, we constructed an automated prototype framework that enables the

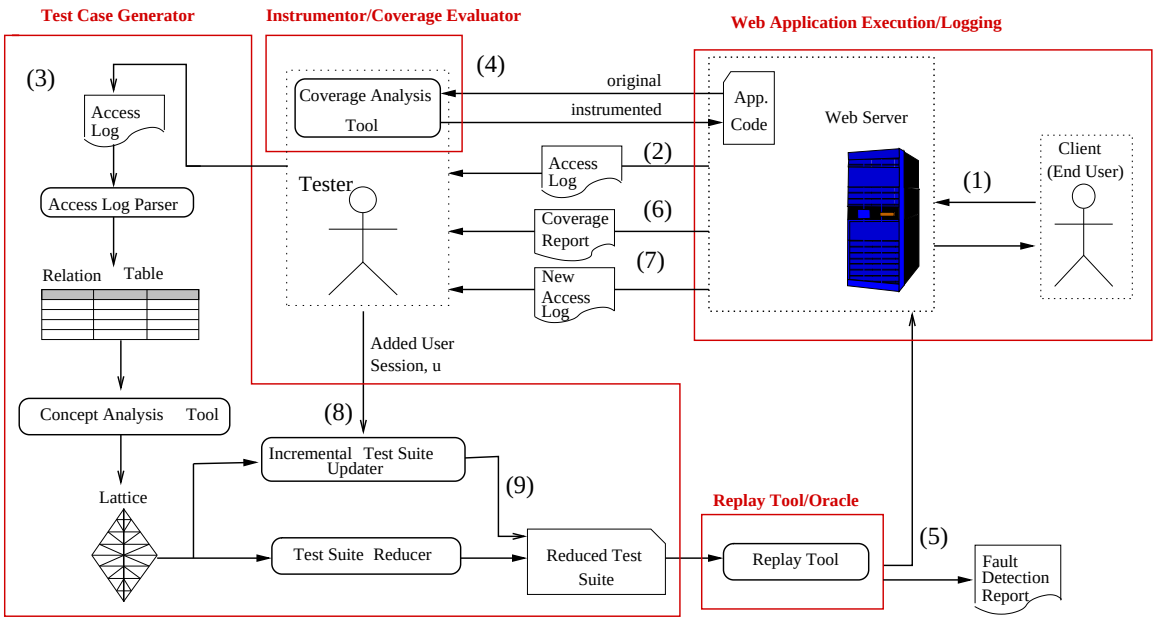

Figure 3. Current Prototype Framework

generation of a reduced test suite, which can be replayed to generate test coverage reports and perform fault detection studies. The process as illustrated in Figure 3 begins with the collection of user-session data (Step (1)). This is the **web application execution/logging** phase of our prototype framework. To obtain user-session data, we rewrote the Resin [29] web server's *AccessLog* class to collect the specific data of interest. In particular, we collected information pertaining to each user request, such as IP address, time stamp, URL requested, GET/POST data, and cookies. Once the user-session data is logged, the process of generating test cases can begin. Note that we define user sessions as beginning when a request arrives from a new IP address and ending when the user leaves the website or the session times out. In addition, we keep track of the timestamp of the request. Requests from the same IP that are more than 45 minutes apart are considered to be distinct sessions. In the future we plan to extend the definition of user sessions to the more general case where multiple users' requests can originate from the same IP address.

We assume the access log is provided to a tester (Step (2)). Step 3 is the **test case generation** phase of our framework. It begins with parsing the access log to generate a relation table. The concept lattice is constructed utilizing Lindig's `concepts` tool [20]. This tool takes a relation table as input and outputs a lattice. The test suite reducer implements the heuristic for identifying the reduced test suite.

In Step (4), our **test coverage evaluator**, consisting of the Java code coverage tool Clover [6], instruments the Java source files that the server generates. The Java source files correspond to the JSP code of the application. Step (5) is the replay of the reduced test suite. We utilized the GNU Unix utility `wget`, which supports the non-interactive download of files over the HTTP protocol, as our replay tool. The replay tool is the core of our **replay tool/oracle** phase of the framework. Additional input parameters to `wget` include cookie information and POST or GET data associated with the original request in order to maintain application state information. To guarantee consistency in replay, we also restore the state of the database to its original state (i.e., the state before logging) before replay. Step (6) is the generation of the coverage report. From a software tester's point of view, the above process provides a tester with the means to automatically generate test cases with real data as input and then replay the test cases, and gather coverage information. Steps (7), (8), and (9) depict the incremental update of the reduced test suite, with a user session from a new access log and the current concept lattice input to the incremental test suite updater, which outputs the updated reduced test suite and updated lattice. Step (5) also plays a role in determining the fault detection capability of the reduced suite against the original suite. Comparison of the JSP pages retrieved on replaying the test suites (original or reduced) is performed to determine the number of detected faults.

6. Experimental Study

Our initial experimental studies focus on providing evidence of the potential effectiveness of applying our proposed methodology for automatic but scalable test case gen-

eration. The objective of the experimental study was to address the following research questions.

1. What test case reduction can be achieved through concept analysis for user-session testing?
2. How does reducing the test suite affect replay time and oracle time?
3. What is the cost-effectiveness of incremental versus batch concept analysis for reduced test suite update?
4. Does concept analysis reduce the test suite while maintaining good coverage?
5. How does the reduced test suite fare against the original test suite in terms of fault detection capabilities?

6.1. Experimental Setup

We used an application from an open source e-commerce site [12] to experiment with applying concept analysis to user sessions to generate a reduced test suite. The application is a bookstore, where users can register, login, browse for books, search for specific books giving a keyword, rate the books, buy books by adding them to the shopping cart, modify personal information, and logout. The bookstore application has 9,748 lines of code, 385 methods and 11 classes. Since our interest was in user sessions, we considered only the code available to the user when computing these metrics, not the administration code. The application uses JSP for its front-end and MySql database for the back-end.

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

The application was hosted on the Resin web server. User requests can be HTTP GET requests or HTTP POST requests, depending on the page being accessed in the bookstore. The process of collecting user-session data differs for each of the HTTP methods. Modifications were made to Resin to allow for logging data from either request.

The setup for the fault detection study consisted of manually seeding faults in the application such that each version of the application contained one seeded fault. Details of the fault detection study are presented in Section 6.5.

6.2. Test Suite Reduction

The first experiment involved evaluating the resulting test size reduction and its impact on the cost of executing and validating test cases. We measured reduction in test suite size by applying concept analysis and also computed the time saved during oracle comparisons and replay.

Methodology. We applied concept analysis on the 123 user sessions collected for the bookstore application to obtain the lattice. The original suite consists of all the 123 user sessions. In accordance with our heuristic, the reduced suite consists of the user sessions corresponding to the *next-to-bottom* nodes.

Metrics. We evaluated the effectiveness of applying concept analysis to the problem of test suite reduction by comparing the number of test cases (user sessions) in the original and the reduced suite. We assessed the benefits of test suite reduction in terms of the time to replay and time to perform oracle comparison. Replay time is the time required to replay the original or reduced suite of user sessions. Oracle time is defined as the sum of times to reinitialize the database, replay, and compare actual versus expected results for one run of the (original or reduced) suite.

Results. In our study, we observed that concept analysis achieved considerable reduction in test suite size. In particular, a reduction of 87.8% in test suite size occurred as shown in the first row of Table 1 (Question 1). As observed in rows 2 and 3 of Table 1, the large percent reduction in test suite size also significantly reduces the time to replay the user sessions and perform oracle comparisons, respectively (Question 2). When performing fault detection studies for a large number of faults and multiple runs of the suite, this time reduction could prove to be considerable.

6.3. Incremental vs. Batch

This experiment involved assessing the scalability of incremental concept analysis.

Methodology. First we employed the publicly available tool `concepts` to reduce the test suite in batch. This involved inputting the original set of user sessions to the concept analysis tool and then selecting the *next-to-bottom* nodes to be in the reduced suite. To perform incremental concept analysis, we first processed 100 of the bookstore's user sessions in batch to obtain a reduced test suite, and then the 23 remaining sessions were incrementally added to the lattice to obtain an updated test suite.

Metrics. The relative sizes of the files required by the incremental and batch techniques to eventually produce the reduced suite are used to evaluate the effectiveness of incremental concept analysis. Time costs are not reported

Metrics	Original suite		Reduced suite		Reduction/Loss(%)
	abs	%	abs	%	
Test Suite Size	123	-	15	-	87.8% red
Replay Time	16m56s	-	4m22s	-	74.2% red
Oracle Time	25m30s	-	5m17s	-	79.3% red
Statement Coverage	5878	60.3	5654	58	3.8% loss
Function Coverage	205	53.2	205	53.2	0% loss
Faults Detected	35	87.5	28	70	20% loss

Table 1. Test suite reduction, coverage and faults detected

for incremental versus batch because we believe it is unfair to compare our implementation of the incremental algorithm, which is an unoptimized and slow academic prototype, against the publicly available tool `concepts`.

Results. The batch algorithm required the complete original set of user sessions to derive the reduced test suite. The original set of user sessions occupied 4.3MB. After processing the initial 100 user sessions in batch, the incremental algorithm maintained only the reduced set when generating new test cases (i.e., the updated test suite). The space for the reduced set was 1MB, a savings of 76.7% in space. Thus, the incremental reduction process saves space costs considerably by not maintaining the original suite of user sessions (Question 3). In a production environment, the incremental reduction process could be performed overnight with the collection of that day's user sessions to produce a fresh updated suite of test cases for testing, while still saving space by keeping a continually reduced suite. It can be shown to hold in general that the batch analysis of all 123 sessions would result in the same reduced test suite as batch analysis of 100 followed by incremental update for the last 23 sessions.

6.4. Program Coverage

The next experiment investigated the loss in program coverage due to test suite reduction.

Methodology. We used the commercially available tool Clover to evaluate program coverage obtained on replaying the reduced suite and the original suite. On deploying the bookstore application, the Resin web server generates corresponding Java files for each JSP file in the application. We instrumented these Java classes using Clover and replayed using `wget` both the original and reduced suite of user sessions. During replay for coverage, we restored the database to the state before logging and resent cookies and name-value pairs recorded during logging the user sessions to make certain that the replay is accurate and the session is maintained by the server.

Metrics. We measured the effectiveness of the reduced suite in terms of statement and method coverage.

Results. Table 1 presents the statement and method coverage obtained after replaying both the original and reduced test suite. The percentage of coverage loss with the reduced suite is negligible (Question 4). We attribute the low loss in coverage to the following properties of the reduced test suite: the commonality of URL subsequences in objects of the same concept node, the coverage of all URLs that appear in any user session, and the low percent of URL subsequences lost through test selection for short common subsequences (see paper [32]). We believe that common URL subsequences replayed with name-value pairs follow similar paths in the program; in the future we plan to examine this relationship more closely.

6.5. Fault Detection Capability

The last aspect of our study involved evaluating the fault detection capability of the reduced suite compared to the original suite.

Methodology. First we replayed both the original and reduced suites through the fault-free application and recorded the expected results (i.e., JSP pages). The "correct" version of the program is considered to be the expected result (i.e., the set of JSP pages returned as a result of executing the correct version of the program). Then we inserted a total of 40 faults into separate copies of the application (one fault per copy). The faults inserted into the code can be broadly classified as: data and control flow based, changes to name-value pairs of the web page and faults in database queries.

The original and reduced suites were re-run through the faulty versions and the actual results were computed. To detect faults, the pages sent in response by the web server were logged during replay in order for the oracle to determine if a test case passes or fails. We utilized a fine grain text based comparator (`diff`), which compares the expected output with the actual output. Any differences detected between two pages results in the oracle indicat-

ing that a test case failed. For every user session in each run of the suites, a count of detected faults and list of specific faults discovered by the suites was recorded.

Metrics. Comparisons were made based on the number of faults detected by each suite to evaluate how well the reduced suite performed in detecting faults versus the original suite.

Results. The last row of Table 1 shows the number of faults detected by the original and reduced suites. The original suite found seven more faults than the reduced suite. The faults not detected by the reduced suite were mainly related to name-value pairs. This indicates that the fault detection capabilities were not hampered largely by reducing the test suite size. We believe that the reduced suite performs reasonably well in detecting faults, though with some loss (Question 5). We are examining changes to the heuristic to increase the fault detection capability.

7. Related Work

7.1. Concept Analysis in Software Engineering

Snelting first introduced the idea of concept analysis for use in software engineering tasks, specifically for configuration analysis [18]. Concept analysis has also been applied to evaluating class hierarchies [34], debugging temporal specifications [1], redocumentation [19], and recovering components [21, 36, 33, 8].

Ball introduced the use of concept analysis of test coverage data to compute dynamic analogs to static control flow relationships [2]. The binary relation consisted of tests (objects) and program entities (attributes) that a test may cover. A key benefit is an intermediate coverage criteria between statement and path-based coverage.

Concept analysis is one form of clustering. To improve the accuracy of software reliability estimation [27], cluster analysis has also been utilized to partition a set of program executions into clusters based on the similarity of their profiles. Dickinson et al. have utilized different cluster analysis techniques along with a failure pursuit sampling technique to select profiles to reveal failures. They have experimentally shown that such techniques are effective [7]. Clustering has also been used to reverse engineer systems [23, 24, 37].

7.2. Reducing Test Suites

Several test suite reduction techniques have been proposed [14, 5, 25]. The goal is to attempt to reduce the cost of maintaining and executing a large test suite during software maintenance by eliminating test cases that result in covering or executing the same statements in a program. For instance, Harrold et al. [14] developed a test suite reduction technique that employs a heuristic based on the minimum cardinality hitting set to select a representative set of test cases that satisfy a set of testing requirements. Such techniques assume that the entire test suite is complete and associations between test cases and test requirements known before reduction heuristics are applied. Harder et al. [13] proposed a technique to generate, augment, and minimize the test suites, but their technique involves dynamically generating operational abstractions, which can be costly.

7.3. Web Testing

In addition to tools that test the appearance and validity of a web application [35], there are web-based analysis tools that model the underlying structure and semantics of web programs.

With the goal of providing automated data flow testing, Liu, Kung, Hsia, and Hsu [22] developed the object-oriented web test model (WATM). However, they developed their model for HTML and XML documents and did not consider many features inherent in other web applications.

Ricca and Tonella [30] developed a high-level UML-based representation of a web application and described how to perform page, hyperlink, def-use, all-uses, and all-paths testing based on the data dependences computed using the model. However, the user needs to intervene to generate input.

Di Lucca et al. [10] developed a web application model and set of tools for the evaluation and automation of testing web applications. They presented an object-oriented test model of a web application and proposed a definition of unit and integration levels of testing. They developed functional testing techniques based on decision tables, which help in generating effective test cases. However, their approach to generating test input is not automated.

8. Conclusions and Future Work

By applying concept analysis to cluster user sessions and carefully selecting user sessions from the resulting concept lattice, we are able to maintain and incrementally update a reduced test suite for user-session based testing of web applications. Our experiments show that quite similar statement and method coverage can be sustained, while reducing the storage requirements. Similar to other experiments [9], our experiments show that there is a tradeoff between test suite size and fault detection capability; however, the incremental update algorithm enables a continuous examination of possibly new test cases that could increase fault detection capability without the need to ever store the larger set of session data to determine the reduced test suite.

In the short term, our future work includes a more significant empirical investigation of this approach with larger web applications to which we have been kindly given access from the University of Delaware's IT team and more extensive user-session data. We plan to extend the incremental concept analysis algorithm to handle new URLs introduced by program evolution. Furthermore, we plan to experimentally investigate the use of alternate heuristics for test case reduction that maintain high fault detection capability.

References

[1] G. Ammons, D. Mandelin, and R. Bodik. Debugging temporal specifications with concept analysis. In *ACM SIGPLAN Conf on Prog Lang Design and Implem*, 2003.

[2] T. Ball. The concept of dynamic analysis. In *ESEC / SIGSOFT FSE*, pages 216–234, 1999.

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

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

[5] T. Y. Chen and M. F. Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135–141, Mar 1996.

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

[7] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *Proceedings of the 8th European Software Engineering Conference*, pages 246–255. ACM Press, 2001.

[8] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans on Soft Eng*, 29(3):210–224, Mar 2003.

[9] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *Int Conf on Soft Eng*, 2003.

[10] G. A. Di Lucca, A. Fasolino, F. Faralli, and U. D. Carlini. Testing web applications. In *Int Conf on Soft Maint*, 2002.

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

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

[13] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *Proceedings of the 25th International Conference on Software Engineering*, pages 60–71. IEEE Computer Society, 2003.

[14] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans on Soft Eng and Meth*, 2(3):270–285, July 1993.

[15] 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.

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

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

[18] M. Krone and G. Snelting. On the inference of configuration structures from source code. In *Int Conf on Soft Eng*, 1994.

[19] T. Kuipers and L. Moonen. Types and concept analysis for legacy systems. In *Int Workshop on Prog Compr*, 2000.

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

[21] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Int Conf on Soft Eng*, 1997.

[22] 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.

[23] C.-H. Lung. Software architecture recovery and restructuring through clustering techniques. In *Proceedings of the Third International Workshop on Software Architecture*, pages 101–104, 1998.

[24] B. S. Mitchell, S. Mancoridis, and M. Traverso. Search based reverse engineering. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 431–438, 2002.

[25] J. Offutt, J. Pan, and J. Voas. Procedures for reducing the size of coverage-based test sets. In *Twelfth International Conference on Testing Computer Software*, 1995.

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

[27] A. Podgurski, W. Masri, Y. McCleese, F. G. Wolff, and C. Yang. Estimation of software reliability by stratified sampling. *ACM Trans. Softw. Eng. Methodol.*, 8(3):263–283, 1999.

[28] Rational corporation. rational testing robot. <http://www.rational.com/products/robot/>.

[29] Caucho resin. http://www.caucho.com/resin/, 2002.

[30] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Int Conf on Soft Eng*, 2001.

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

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

[33] M. Siff and T. Reps. Identifying modules via concept analysis. In *International Conf on Software Maintenance*, 1997.

[34] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *SIGSOFT Foundations of Software Engineering*, 1998.

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

[36] P. Tonella. Concept analysis for module restructuring. *IEEE Trans on Soft Eng*, 27(4):351–363, Apr 2001.

[37] T. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Fourth Working Conference on Reverse Engineering (WCRE '97)*, 1997.