Applying Concept Analysis to User-Session-Based Testing of Web Applications

Sreedevi Sampath, *Member*, *IEEE Computer Society*, Sara Sprenkle, Emily Gibson, Lori Pollock, *Member*, *IEEE Computer Society*, and Amie Souter Greenwald

Abstract—The continuous use of the Web for daily operations by businesses, consumers, and the government has created a great demand for reliable Web applications. One promising approach to testing the functionality of Web applications leverages the user-session data collected by Web servers. User-session-based testing 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 and a set of heuristics for test case selection. Existing incremental concept analysis algorithms are exploited to avoid collecting and maintaining large user-session data sets and to thus provide scalability. We have completely automated the process from user session collection and test suite reduction through test case replay. Our incremental test suite update algorithm, coupled with our experimental study, indicates that concept analysis provides a promising means for incrementally updating reduced test suites in response to newly captured user sessions with little loss in fault detection capability and program coverage.

Index Terms—Software testing, Web applications, user-session-based testing, test suite reduction, concept analysis, incremental test suite reduction.

1 Introduction

As the quantity and breadth of Web-based software systems continue to grow at a rapid pace, assuring the quality and reliability of this software domain is becoming critical. Low reliability can result in serious detrimental effects for businesses, consumers, and the government as they increasingly depend 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—such as complex control and value flow paths, unexpected transitions introduced by user interactions with the browser, and frequent updates—complicate testing beyond the analysis and testing considerations of the more traditional domains.

Many of the current testing tools address Web usability, performance, and portability issues [1]. 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 preset scripts into text fields, ending with pressing the submit button. Compatibility testers ensure that a Web application functions properly within different browsers.

Functional and structural testing tools have also been developed for Web applications. Tools such as Cactus [2], which utilizes JUnit [3], provide test frameworks for unit testing the functionality of Java-based Web programs. HttpUnit [4] is a Web testing tool that emulates a browser and determines the correctness of returned documents using an oracle comparator. In addition, Web-based analysis and testing tools have been developed that model the underlying structure and semantics of Web programs [5], [6], [7], [8] toward a white-box approach to testing. These white-box techniques enable the extension of path-based testing to Web applications. However, the white-box techniques often require manually identifying the input data that will exercise the paths to be tested, especially the paths that 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 to record user-induced events, gather and convert them into scripts, and replay them for testing [9], [10]. Tools such as WebKing [11] and Rational Robot [10] provide automated testing of Web applications by collecting data from users through minimal configuration changes to the Web server. The recorded events are typically base requests and name-value pairs (for example, form field data) sent as requests to the Web server. A *base request* for a Web application is the request type and resource location without the associated data (for example, *GET /apps/bookstore/Login.jsp*). The ability to record these requests is often built into the Web server, so little effort is needed to record the desired events. The testing

- *S. Sampath is with the Department of Information Systems, 1000 Hilltop Circle, University of Maryland, Baltimore County, Baltimore, MD 21250. E-mail: sampath@umbc.edu.*
- *S. Sprenkle is with the Department of Computer Science, Parmly Hall, Washington & Lee University, Lexington, VA 24450. E-mail: sprenkles@wlu.edu.*
- *E. Gibson, and L. Pollock are with the Department of Computer and Information Sciences, 103 Smith Hall, University of Delaware, Newark, DE 19716. E-mail: {gibson, pollock}@cis.udel.edu.*
- *A.S. Greenwald is with Alcatel-Lucent Bell Labs, 600 Mountain Ave., Murray Hill, NJ 07974. E-mail: agreenwald@alcatel-lucent.com.*

Manuscript received 28 Mar. 2006; revised 27 May 2007; accepted 26 June 2007; published online 10 July 2007.
Recommended for acceptance by R. Taylor.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0059-0306.
Digital Object Identifier no. 10.1109/TSE.2007.70723.

provided by WebKing [11] may not be comprehensive because WebKing requires users to record critical paths and tests for only these paths in the program. In a controlled experiment, Elbaum et al. [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 [7]. These results motivated user-session-based testing as an approach to test the functionality of Web applications while relieving the tester from generating the input data manually and also to enhance a test suite with test data that represents usage as the operational profile of an application evolves. Elbaum et al. also observed that the fault detection capability appears to increase with larger numbers of captured user sessions; unfortunately, the test preparation and execution time quickly becomes impractical. Although existing test reduction techniques [12] can be applied to reduce the number of maintained test cases, the overhead of selection and analysis of the large user-session data sets is nonscalable.

This paper presents an approach for achieving *scalable* user-session-based testing of Web applications. The key insight is formulating an approach to selecting test cases (that is, user sessions) for test suite reduction based on clustering logged user sessions by concept analysis. 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 [13]. Test suites are reduced with the criteria of covering all base requests in the original test suite and covering distinct use cases, where we specify a use case to be a set of base requests. Existing incremental concept analysis techniques can 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 reduced test suite. Using existing tools and developing some simple scripts, we automated the entire process from gathering user sessions through the identification of a reduced test suite and replay of the reduced test suite for coverage analysis and fault detection [14], [15]. In our previous experiments, the resulting program coverage of the reduced test suite is almost identical to the original test suite, with some loss in fault detection. In this paper, we extend our previous work in [16] by proposing and evaluating two new heuristics for test suite reduction and reporting significantly more experimental evaluation results with two new subject Web applications and newly collected user session data. The main contributions of this paper are the following:

1. the formulation of the test suite reduction problem for user-session-based testing of Web applications in terms of concept analysis,
2. harnessing incremental concept analysis for test suite reduction to manage large numbers of user sessions in the presence of an evolving operational profile of the application,
3. three heuristics for test suite reduction based on concept analysis, and
4. experimental evaluation of the effectiveness of the reduced suites with three subject Web applications.

In Section 2, we provide the background on Web applications, user-session-based testing, and concept analysis. We apply concept analysis to user-session-based testing and the test suite reduction problem in Section 3. In Section 4, we present three heuristics for test suite reduction. In Section 5, we present an approach to scalable test suite update with incremental concept analysis and, in Section 6, we present the space and time costs for batch and incremental concept analysis. Section 7 describes our experimental study with three subject applications. We conclude and present future work in Section 8.

2 Background and State of the Art

2.1 Web Applications

Broadly defined, a Web application consists of a set of Web pages and components that form a system that executes using Web server(s), network, HTTP, and browser(s) 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 the same for all users—or dynamic such that its content may depend on user input.

Web applications may include an integration of numerous technologies, third-party reusable modules, a well-defined layered architecture, dynamically generated pages with dynamic content, and extensions to an application framework. Large Web-based software systems can require thousands to millions of lines of code, contain many interactions between objects, and involve significant interaction with users. In addition, changing user profiles and frequent small maintenance changes complicate automated testing [17].

In this paper, we target Web applications written in Java using servlets and JSPs. The applications consist of a back-end data store, a Web server, and a client browser. Since our user-session-based testing techniques are language independent and since they require only user sessions for testing, our testing techniques can be easily extended to other Web technologies.

2.2 Testing Web Applications

2.2.1 *Program-Based Testing*

In addition to tools that test the appearance and validity of a Web application [1], there are tools that analyze and model the underlying structure and semantics of Web programs.

With the goal of providing automated data flow testing, Liu et al. [5] and Kung et al. [18] developed the object-oriented Web test model (WATM), which consists of multiple models, each targeted at capturing a different tier of the Web application. They suggest that data flow analysis can be performed at multiple levels. Though the models capture interactions between different components of a Web application, it is not clear if the models have been implemented and experimentally evaluated. With multiple models to represent control flow, we believe that the models can easily become impractical in size and complexity for a medium-sized dynamic Web application as the data flow analysis progresses from the lower (function) level to higher (application) levels. The model also is focused on HTML and XML documents and does not mention many other features inherent in Web applications.

Ricca and Tonella [7] developed a high-level Unified Modeling Language (UML)-based representation of a Web application and described how to perform page, hyperlink, def-use, all-uses, and all-paths testing based on the data dependences computed using the model. Their ReWeb tool loads and analyzes the pages of the Web application and builds the UML model and the TestWeb tool generates and executes test cases. However, the user needs to intervene to generate an input. To our knowledge, the cost effectiveness of the proposed models has not been thoroughly evaluated.

Di Lucca et al. [6] developed a Web application model and a 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 definitions 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.

Andrews et al. [8] proposed an approach to modeling Web applications with finite-state machines (FSMs) and use coverage criteria based on FSM test sequences. They represent test requirements as subsequences of states in the FSMs, generate test cases by combining the test sequences, and propose a technique to reduce the set of inputs. However, their model does not handle dynamic aspects of Web applications, such as transitions introduced by the user through the browser, and connections to remote components. To our knowledge, neither the model nor the testing strategy has been evaluated.

In summary, to enable the practical modeling and analysis of a Web application's structure, the analysis typically ignores browser interactions, does not consider dynamic user location and behaviors, and models only parts of the application. User-session-based testing addresses the challenges inherent in modeling and testing Web applications by leveraging user input data, rather than manually generating test data.

2.2.2 User-Session-Based Testing

In user-session-based testing, each *user session* is a sequence of user requests in the form of base requests and name-value pairs. When cookies are available, we use cookies to generate user sessions. Otherwise, we say a user session begins when a request from a new Internet Protocol (IP) address arrives at the server and ends when the user leaves the Web site or the session times out. We consider a 45 minute gap between two requests from a user to be equivalent to a session timing 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 for the collected user sessions [9], [10], [11], [19].

Elbaum et al. [9] provided promising results that demonstrate the fault detection capabilities and cost effectiveness of user-session-based testing. Their user-session-based techniques discovered certain types of faults; however, faults associated with rarely entered data were not detected. In addition, they observed that the effectiveness of user-session-based testing improves as the number of collected sessions increases; however, the cost of collecting, analyzing, and replaying test cases also increases.

User-session-based testing techniques are complementary to the testing performed during the development phase of the application [5], [6], [7], [20], [21], [22], [23]. In addition, user-session-based testing is particularly useful when the program specifications and requirements are not available for test case generation.

2.3 Test Suite Reduction

A large number of user sessions can be logged for a frequently used Web application and it may not be practical to use all of the user sessions when testing the application. In addition, an evolving application can cause some test cases to become obsolete and also may require augmenting an existing test suite with new test cases that test the application's new functionality. The additional test cases can lead to redundant test cases, which waste valuable testing resources. In general, the goal of test suite reduction for a given test requirement (for example, 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. Since test suite reduction produces a smaller test suite, it has several advantages, such as reducing the cost of executing, validating, and managing test suites as the application evolves. Our goal is to select test cases for a reduced suite that covers both all base requests of the original suite and distinct use cases. In our current test suite reduction approach, we do not identify and remove test cases that become obsolete due to application version changes; instead our focus is to manage large test suites within the same application version.

Several test suite reduction techniques have been proposed [12], [24], [25], [26], [27], [28], [29], [30]. For instance, Harrold et al. [12] 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 satisfies a set of testing requirements. To our knowledge, these techniques do not include incremental approaches to test suite reduction. Harder et al. [31] proposed an operational-abstraction-based minimization technique that can be executed incrementally, but dynamically generating operational abstractions can be costly.

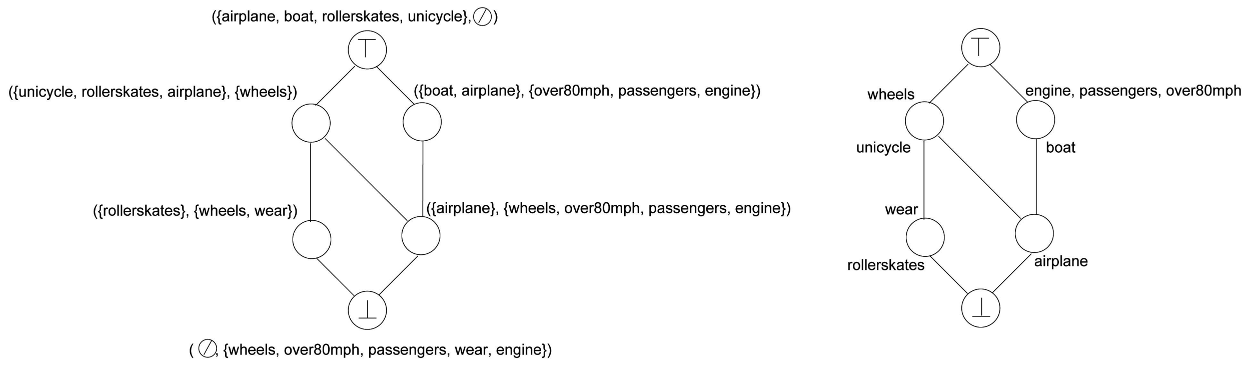
2.4 Concept Analysis

Our work focuses on applying a concept-analysis-based approach and its variations to reducing test suites for user-session-based testing of Web applications. Concept analysis is a mathematical technique for clustering objects that have common discrete attributes [32]. Concept 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, it is false. For example, consider the context depicted in Fig. 1a. The object set O is $\{airplane, boat, rollerskates, unicycle\}$; the attribute set A is

$$\{wheel(s), over80mph, passengers, wear, engine\}.$$

	wheels	**over80mph**	**passengers**	**wear**	**engine**
airplane	true	true	true	false	true
boat	false	true	true	false	true
rollerskates	true	false	false	true	false
unicycle	true	false	false	false	false

(a)


({airplane, boat, rollerskates, unicycle},∅)
⊤
({unicycle, rollerskates, airplane}, {wheels})
({boat, airplane}, {over80mph, passengers, engine})
({rollerskates}, {wheels, wear})
({airplane}, {wheels, over80mph, passengers, engine})
⊥
(∅, {wheels, over80mph, passengers, wear, engine})
⊤
wheels
engine, passengers, over80mph
unicycle
boat
wear
rollerskates
airplane
⊥

(b)

Fig. 1. Example of concept analysis for modes of transportation. (a) Relation table (that is, context). (b) Full and sparse concept lattice representations.

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 and all and only attributes in A_j share all and only the objects in O_i. The concepts form a partial order defined as $(O_1, A_1) \leq (O_2, A_2)$ *iff* $O_1 \subseteq O_2$ or, equivalently, $A_1 \supseteq A_2$. Thus, the partial order can be viewed as a subset relation on the objects or a superset relation on the attributes. 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 $\perp$ is the *most special concept* with the set of all of the objects that have all of the attributes in A. In the example, $\top$ is

$$(\{airplane,\ boat,\ rollerskates,\ unicycle\},\ \oslash),$$

whereas $\perp$ is

$$(\oslash,\ \{wheels,\ over80mph,\ passengers,\ wear,\ engine\}).$$

The full concept lattice is depicted on the left side in Fig. 1b. A sparse representation (shown on the right side) can be used to depict the concept lattice. In the sparse representation of the lattice, a node n is marked with an attribute a if the node is the most general concept that has a in its attribute set. Similarly, a node n is marked with an object o if the node is the most special concept with o in its object set. Attribute sets are shown just above each node, whereas object sets are shown just below each node. For example, consider the node labeled above by $\{wheels\}$ and below by $\{unicycle\}$. This node represents the concept $(\{unicycle,\ rollerskates,\ airplane\},\ \{wheels\})$.

Krone and Snelting first introduced the idea of concept analysis for use in software engineering tasks, specifically for configuration analysis [33]. Concept analysis has also been applied in evaluating class hierarchies [34], debugging temporal specifications [35], redocumentation [36], and recovering components [37], [38], [39], [40]. Ball introduced the use of concept analysis of test coverage data to compute dynamic analogs to static control flow relationships [41]. The binary relation consisted of tests (objects) and program entities (attributes) that a test may cover. A key benefit is an intermediate coverage criterion between statement and path-based coverage. Since our initial work [16], Tallam and Gupta [42] have presented a greedy approach to test suite minimization inspired by concept analysis.

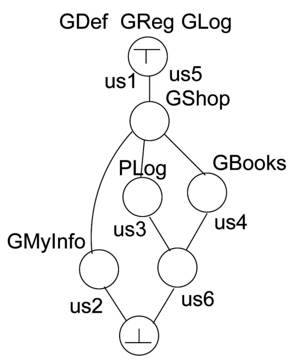
Concept analysis is one form of clustering. Researchers have applied clustering to various software engineering problems. To improve the accuracy of software reliability estimation [43], cluster analysis has also been utilized to partition a set of program executions into clusters based on the similarity of their profiles. Dickinson et al. [44] 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 [44].

3 Applying Concept Analysis to Testing

To apply concept analysis to user-session-based testing, we use objects to represent the information uniquely identifying user sessions (that is, test cases) and attributes to represent base requests. Since we use user sessions to test the Web application, we will use the terms user sessions and test cases interchangeably in this paper. Although a test case is considered to be a sequence of base requests and associated name-value pairs for accurate replay, we define a

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

(a)



(b)

Fig. 2. (a) Relation table of the original suite of user sessions. (b) Concept lattice for test suite reduction.

test case during concept analysis to be the set of base requests accessed by the user, without the name-value pairs and without any ordering on the base requests. This considerably reduces the number of attributes to be analyzed. In [14], we present the results of an experimental study with two subject Web applications—an online bookstore application and an online portal. We examined the clustering of test cases by concept analysis in terms of common subsequences of base requests. Our experiments indicate that single base requests can be effective when used as attributes for test suite reduction.

A pair (test case s, base request u) is in the binary relation iff s requests u. Thus, each *true* entry in a row r of the relation table represents a base request that the single user represented by row r accesses. For column c, the set of *true* entries represents the set of users who have requested the same base request, possibly with different name-value pairs.

As an example, the relation table in Fig. 2a shows the context for a user-session-based test suite for a portion of a bookstore Web application. Consider the row for the test case *us3*. The (true) marks in the relation table indicate that test case *us3* accessed the base requests *GDef*, *GReg*, *GLog*, *PLog*, and *GShop*. We distinguish a GET (G) request from a POST (P) request when building the lattice because they are different requests. Based on the original relation table, concept analysis derives the sparse representation of the lattice in Fig. 2b.

3.1 Properties of the Concept Lattice

Lattices generated from concept analysis for test suite reduction will exhibit several interesting properties and relationships. Interpreting the sparse representation in Fig. 2b, a test case s accesses all base requests at or above the concept uniquely labeled by s in the lattice. For example, the test case *us3* requests *PLog*, *GShop*, *GDef*, *GReg*, and *GLog*. A node labeled with a test case s and no attributes indicates that s accessed no unique base requests. For example, *us6* accesses no unique base requests. Similarly, all test cases at or below the concept uniquely labeled by base request u access the base request u. In Fig. 2b, test cases *us2*, *us3*, *us4*, and *us6* access the base request *GShop*.

The $\top$ of the lattice denotes the base requests that are accessed by all of the test cases in the lattice. In our example, *GReg*, *GDef*, and *GLog* are requested by all of the test cases in our original test suite. The $\bot$ of the lattice denotes the test cases that access all base requests in the context. Here, $\bot$ is not labeled with any test case, denoting that no test case accesses all the base requests in the context.

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

3.2 Using the Lattice for Test Suite Reduction

Our test suite reduction technique exploits the concept lattice's hierarchical use case clustering properties, where a use case is viewed as the set of base requests executed by a test case. Given a context with a set of test cases as objects O, we define the *similarity* of a set of test cases $O_i \subseteq O$ as the number of attributes shared by all of the test cases 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 base requests than nodes higher in the concept lattice along a certain path in the lattice.

Although we view a use case as the *set* of base requests executed by a test case, traditionally, a use case is viewed as a behaviorally related *sequence* of events performed by a user when interacting with the application [13]. Our previous studies have shown that similarity in sets of base requests translates to similarity in covering similar subsequences of base requests and in covering similar program characteristics [14]. Thus, clusters of test cases based on their set of single base requests can be viewed as clusters of similar use cases of the application. We developed heuristics for selecting a subset of test cases as the current test suite, based on the clustering of the current concept lattice. In the next section, we present three heuristics for test suite reduction based on the concept lattice.

4 Test Selection Heuristics

4.1 One-Per-Node Heuristic

The *one-per-node* heuristic seeks to cover all of the base requests present in the original suite and maximize use case representation in the reduced suite by selecting one test case from every node (that is, cluster) in the lattice without duplication. The *one-per-node* heuristic can be implemented by selecting one test case from each node in the sparse representation of the concept lattice. If there are multiple test cases clustered together in a node, we select one test case at random. Selecting a session from each node in a sparse representation of a lattice is equivalent to selecting one representative for each set of duplicate rows in the relation table. Thus, *one-per-node* can also be implemented by removing duplicate rows from the relation table. In Fig. 2b, the *one-per-node* heuristic selects test cases *us2*, *us6*, *us3*, *us4*, and *us1* for the reduced suite. For the node one level below $\top$, either *us1* or *us5* could be randomly selected.

The *one-per-node* heuristic appears to naively exploit the lattice's clustering and partial ordering. Nodes that are higher in the lattice, that is, further away from $\bot$, contain base requests (attributes) that are accessed by many test cases (objects). The *one-per-node* heuristic thus selects test cases representing frequently accessed base request sets—and the program code covered on executing these base request sets—as well as test cases from each level of the lattice for the reduced suite. However, by selecting from every node in the lattice, the *one-per-node* heuristic will create a large reduced test suite.

4.2 Test-All-Exec-Requests Heuristic

In the second heuristic for user-session selection, which we call *test-all-exec-requests* (presented in [16]), we seek to identify the smallest set of test cases that will cover all of the base requests executed by the original test suite while representing different use cases.

The details of this heuristic are as follows: The reduced test suite contains a test case from $\bot$ if the set of test cases at $\bot$ is nonempty and from each node next to $\bot$ that is one level up the lattice from $\bot$, which we call *next-to-bottom* nodes. These nodes contain objects that are highly *similar* to each other. From the partial ordering in the lattice, if $\bot$'s object set is empty, then the objects in a given *next-to-bottom* node contain the exact same set of attributes. Since one of the *test-all-exec-requests* heuristic goals is to increase the use-case representation in the reduced test suite, we include test cases from both $\bot$ and *next-to-bottom* nodes. We do not select duplicate test cases when selecting test cases from $\bot$ and *next-to-bottom* nodes. When more than one test case is clustered together in a $\bot$ or *next-to-bottom* node, we randomly select one representative test case from the node. The heuristic operates on the sparse representation of the concept lattice. This heuristic can also be implemented by examining subset relations directly in the relation table.

To validate clustering test cases based on concept analysis and the *test-all-exec-requests* heuristic for test-case selection, we performed experimental studies and user session analysis [14]. These studies examined the commonality of base request subsequences of objects clustered into the same concepts and also compared the subsequence commonality of the selected test cases with those in the remainder of the suite; the study is described fully in [14]. The results support using concept analysis with a heuristic for user-session selection, where we choose representatives from different clusters of similar use cases. Since our goal is also to represent use cases in the reduced suite in addition to satisfying all of the requirements (that is, base requests), our approach differs from traditional reduction techniques that select the next test case with the most additional requirements coverage until 100 percent coverage is obtained [12].

In our example in Fig. 2a, the original test suite is all of the test cases in the original context. The reduced test suite, however, contains only test cases *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 base requests accessed by the two test cases is exactly the set of all base requests accessed by the original test suite. In addition, in previous work [14], we found that *next-to-bottom* nodes represent distinct use cases and, thus, each use case covers a different set of methods and faults in the program code. Thus, the reduced suite obtained by applying the *test-all-exec-requests* covers all base requests covered by the original suite while representing different use cases of the application. Since the *test-all-exec-requests* heuristic selects a small subset of test cases, the reduced test suite is likely to have low program-code-coverage and fault detection effectiveness compared to the original suite. This expectation motivated the development of the *k-limited* heuristic presented in the next section.

4.3 k-Limited Heuristic

The *k-limited* heuristic selects test cases such that the resulting test suite covers all of the base requests covered by the original suite while maintaining varied use cases, beyond the use case representation provided by the *test-all-exec-requests* heuristic. The *k-limited* heuristic selects a reduced suite that represents more use cases than the *test-all-exec-requests* heuristic by including test cases from nodes that are k levels above $\bot$ in the sparse representation of the lattice. For $k = 1$, the *test-all-exec-requests* heuristic and the *k-limited* heuristic will select the same reduced suite. For $k > 1$, we select one test case from each node in the lattice that is less than or equal to k levels up from $\bot$. For $k = max\_depth\_of\_lattice$, *k-limited* selects the same reduced suite as *one-per-node*. As with the other heuristics, if multiple test cases are clustered together in a node, we select one representative test case at random.

The main challenge with the *k-limited* heuristic is determining the nodes in each of the k levels. Because a concept lattice is not a tree, a node may exist at multiple levels, depending on the path traversed from $\bot$. To guarantee that each concept node in the lattice is assigned a unique level, a node is placed at the highest level possible. For example, in Fig. 3a, there are paths of lengths 2 and 3 from $\bot$ to *node 1*. Following the approach described above, *node 1* is placed at the higher of the two levels, level 3 (Fig. 3e). We assign nodes to the highest level when traversing the lattice from $\bot$ to ensure that nodes representing more specialized use cases (nodes that are lower in the lattice along a path) are included in the reduced suite before more general use cases (nodes that are higher in the lattice along a path). Also, note that assigning a node to the highest level would result in creating a smaller test suite than if the node is assigned to the lowest level. Through our experimental studies, we provide intuition on reasonable values of k and study the trade-offs between increasing the number of levels represented in the reduced test suite versus the effectiveness of the test suite.

5 Incremental Concept Analysis

The key to enabling the generation of a reduced test suite with base request and use case coverage similar to a test suite based on large user-session data sets, without the

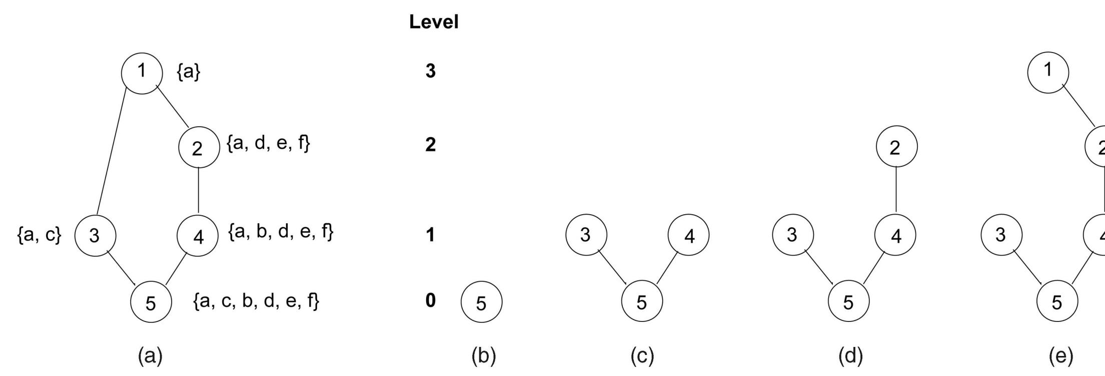


Fig. 3. Applying the k-$limited$ heuristic with different values of k. (a). Lattice. (b). $k = 0$. (c). $k = 1$. (d). $k = 2$. (e). $k = 3$.

us	GDef	GReg	GLog	PLog	GShop	GBooks	GMyInfo
1	●	●	●				
2	●	●	●		●		●
3	●	●	●	●	●		
4	●	●	●		●	●	
5	●	●	●				
6	●	●	●	●	●	●	
7	●	●	●	●	●		●
8	●		●			●	

(a)

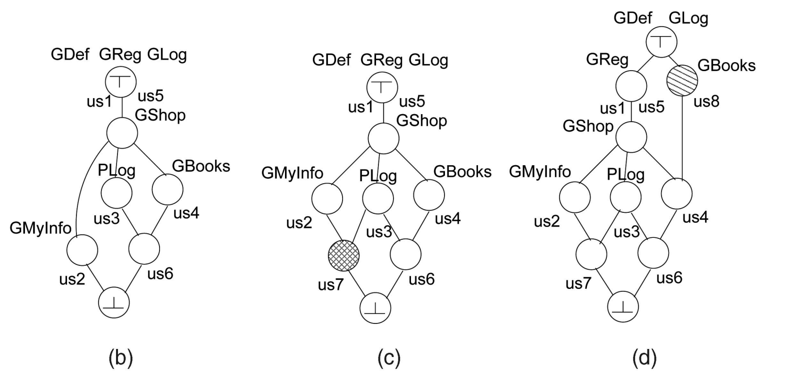


Fig. 4. (a) Relation table for the original suite of user sessions (us1 through us6) and the newly added user sessions (us7 and us8). (b) Concept Lattice for original suite of user sessions. (c) Concept lattice after adding us7. (d) Concept lattice after adding us8.

overhead for storing and processing the complete set at once, is the ability to incrementally perform concept analysis. The general approach is to start with an initial user-session data set and incrementally analyze additional test cases 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 test case s and a tuple (O, A, R, L), where O is the set of test cases (that is, objects) in the current reduced test suite, A is the set of possible base requests in the Web application (that is, attributes), R is the binary relation describing the base requests that are accessed by each test case in O, and L is the concept lattice output from the concept analysis of (O, A, R), modify the concept lattice L to incorporate the test case s and its attributes, creating an updated lattice L' without complete reanalysis to build L' and then create an updated reduced test suite T' with respect to L'.

By using incremental concept analysis, an updated reduced test suite can be created that reflects the current operational profile of the application as represented by the captured user sessions. However, our current approach to incremental concept analysis cannot be used to address test suite changes as a result of software maintenance, for example, removing obsolete test cases as a result of application code changes.

Our incremental algorithm, shown in Fig. 5, utilizes Godin et al.'s incremental lattice update algorithm (*IncrementalLatticeUpdate* in Fig. 5), which takes as input the current lattice L and the new object with its attributes [45]. Once an initial concept lattice L has been constructed from relation table 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 a specific level p (where p is defined as the length of the longest path from $\perp$ to the node) may now be raised in the lattice and new nodes may have been created at the level p. However, existing internal nodes will never "sink" in the lattice because the partial ordering between existing nodes is unchanged by the addition of new nodes [14].

The insertion of a new node and the increase in the level of some nodes upon insertion are the only changes that can immediately affect the updating of the reduced test suite for all our heuristics. Thus, all three of our heuristics, *one-per-node*, *test-all-exec-requests*, and *k-limited*, require maintaining only the test cases from the old reduced test suite and the corresponding concept lattice for incremental test suite update. Other test cases from the original test suite will not be added to form the new reduced test suite and thus need not be maintained. The old and new reduced node sets in Fig. 6 allow for identifying test cases added (due to a new node) or deleted (due to an increase in a node's level) 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.

Although the old reduced suite may be small for *test-all-exec-requests*, the other heuristics require storing a larger reduced suite. For the *one-per-node* heuristic, since the

```
Algorithm: IncrementalReducedTestSuiteUpdate

Input:  ConceptLattice, $L$
        Added user session, $s$
        Reduction heuristic, reduction-heur
        Old reduced set of nodes, old-reduced-node-set
Output: Updated Lattice, $L'$
        Updated Test Suite, $T'$
let $a$ = set of requests in $s$
/*IncrementalLatticeUpdate returns the updated lattice $L'$,
and the new nodes that were added to $L'$, add-nodes */
($L'$, add-nodes) = IncrementalLatticeUpdate(L, (s,a))
new-reduced-node-set =
        ComputeReducedNodeSet (reduction-heur, $L'$, add-nodes,
            old-reduced-node-set )
$T' = \oslash$
foreach node $n$ in new-reduced-node-set
        let $(o, a)$ be the label on $n$ in the sparse lattice $L'$
        if ($o$ != null)
            randomly select user session $s'$ in $o$
            add $s'$ to $T'$
return $L'$, $T'$
```

Fig. 5. Incremental reduced test suite update.

```
Algorithm: ComputeReducedNodeSet

Input:  Reduction heuristic, reduction-heur
        Updated Lattice, $L'$
        Nodes added to the updated lattice, add-nodes
        Old reduced set of nodes, old-reduced-node-set
Output: Updated reduced nodes set, reduced-node-set
let $k$ = stopping level of the reduction-heur
let reduced-node-set = old-reduced-node-set
AssignLevelsToNodes($L'$, k) /* See Section IV */
foreach node $n$ in reduced-node-set
        if level of $n$ in $L' > k$
            remove $n$ from reduced-node-set
foreach node $n$ in add-nodes
        if (level of $n$ in $L' \leq k$)
            add $n$ to reduced-node-set
return reduced-node-set
```

Fig. 6. Compute the reduced node set.

heuristic selects from every node in the lattice, maintaining the reduced suite entails storing one test case per node in the lattice (that is, the reduced suite before update). As a result, the saved space may not be large. For the *k-limited* heuristic, depending on the value of k, a differently sized subset of test cases from the original suite is maintained. Also, a batch analysis of all the sessions yields the same reduced set of nodes as the batch analysis of an initial subset of the sessions followed by an incremental analysis of the remaining sessions [14].

To demonstrate the incremental test suite update algorithm, we begin with the initial concept relation table in Fig. 4a (excluding the *us7* and *us8* rows) and its corresponding concept lattice in Fig. 4b. Consider the addition of the test case *us7*, which contains all base requests except *GBooks*. Fig. 4c 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 nodes at the *next-to-bottom* level (level 1) up in the concept lattice.

For the *one-per-node* heuristic, the new reduced test suite would be updated to include the new user session *us7* in addition to the old reduced test suite. Upon applying the *test-all-exec-requests* heuristic, the incremental update to the reduced test suite deletes test case *us2* and adds test case *us7*. The updated reduced suite selected by the *k-limited* heuristic depends on the value of k. For $k = 3$, the incremental update to the reduced suite adds the new test case *us7* and deletes test case *us1*.

Now, consider the addition of the test case *us8*, which contains only three base requests, as indicated by the last row of the augmented relation table. Fig. 4d shows the incrementally updated concept lattice after the addition of both test cases *us7* and *us8*. In this case, *us8* resulted in a new node and edges higher in the lattice. Because nodes at the *next-to-bottom* level remain unchanged, the reduced test suite selected by the *test-all-exec-requests* heuristic remains unchanged and the new test case is not stored. The *one-per-node* heuristic, however, would include *us8* in the reduced test suite.

6 Theoretical Space and Time Costs

In this analysis, $|O|$ is defined as the cardinality of the set of objects, that is, user sessions, and $|A|$ is defined as the cardinality of the set of attributes, that is, the base requests of the Web application, in the initial user-session set. The relation table is $\mathbf{O}(|O| \times |A|)$.

The initial batch concept analysis requires space for the initial user-session test set, relation table, and concept lattice. Incrementally updating the reduced test suite does not need the relation table; we need to maintain space only for the concept lattice (with objects and attributes from the reduced test suite), the current reduced user-session test set, and the new user sessions. During concept analysis, the lattice L can have at most 2^m concepts, where $m = min(|O|, |A|)$ in the worst case, but the worst case is not expected in practice [45]. Assuming, for each test case in O, there is a reasonable fixed upper bound q_{max} on the number of requests from A that it requests, the lattice grows linearly with respect to $|O|$ such that $|L| = \mathbf{O}(2^{q_{max}}|O|)$. In our experience, the size of the lattice is even much smaller than $\mathbf{O}(2^{q_{max}}|O|)$; we contrast the theoretical size bounds with our empirical results in Section 7.2.

Time complexity for Lindig's *batch* concept analysis algorithm is quadratic in the size of the input relation [46]. The time complexity of Godin et al.'s *incremental* algorithm for our problem (where the number of base requests any given test case will request is limited) is $\mathbf{O}(|O|)$, linearly bounded by the number of user sessions in the current reduced test suite [45].

7 Experiments

In our experimental studies, we focused on providing evidence of the potential effectiveness of applying our proposed methodology for automatic but scalable test suite reduction. The objective of the experimental study was to address the following research questions:

TABLE 1
Objects of Analysis

Metrics	Book	CPM	MASPLAS
Classes	11	75	9
Methods	319	173	22
Conditions	1720	1260	108
NCLOC	7615	9401	999
Seeded faults	40	135	29
Total number of user sessions ($\|O\|$)	125	890	169
Total number of requests accessed	3640	12352	1107
Number of unique requests ($\|A\|$)	10	69	24
Largest user session in number of requests	160	585	69
Average user session in number of requests	29	14	7
Maximum number of unique requests requested by a test case, q_{max}	10	40	14
Mean number of unique requests requested by a test case, q_{mean}	6.9	6.7	4.8
Theoretical maximum number of concepts, $2^m, m = min(O, A)$	$2^{10} = 1024$	$2^{69} = 5.9 \times 10^{20}$	$2^{24} = 1.6 \times 10^7$
Theoretical max when limited number of unique requests per test case, $2^{q_{max}}\|O\|$	since $q_{max} == \|A\|$, $2^{\|A\|} = 2^{10} = 1024$	$2^{40} \times 890 =$ 9.8×10^{14}	$2^{14} \times 169 =$ 2.7×10^6
From Godin's observations, $q_{mean}\|O\|$	$6.9 \times 125 = 862$	$6.7 \times 890 = 5963$	$4.8 \times 169 = 811$
Our measured number of concepts	72	2695	682

1. How much test-case reduction can be achieved by applying the different reduction heuristics?
2. How effective are the reduced test suites in terms of program coverage and fault detection?
3. What is the cost effectiveness of incremental and batch concept analysis for reduced test suite update?

7.1 Independent and Dependent Variables

The *independent variables* of our study are the objects (test cases) and attributes (base requests) input to concept analysis, as well as the test suite reduction heuristic applied. The objects and attributes are based on the original suite. To answer our research questions, we used four *dependent variables* as our measures: reduced suite size, program coverage, fault detection, and space required for the reduced suite.

7.2 Objects of Analysis

The subject applications are an open source e-commerce bookstore (Book) [47], a course project manager (CPM) developed and first deployed at Duke University in 2001, and a conference registration and paper submissions system (MASPLAS). Table 1 shows the characteristics of our subject programs and collected user sessions. The noncommented lines of code (NCLOC) shown in Table 1 are the number of noncommented lines of Java code; since the Web application server automatically compiles JSPs into Java servlets, we also measure Java code for JSPs.

Book allows users to register, log in, browse for books, search for books by keyword, rate books, add books to a shopping cart, modify personal information, and log out. Book uses JSPs for its front end and a MySQL database for the back end. To collect the 125 user sessions for Book, we sent e-mail to local newsgroups and posted advertisements in the university's classifieds Web page asking for volunteer users. Since we did not include administrative functionality in our study, we removed requests to administration-related pages from the user sessions.

In **CPM**, course instructors log in 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 e-mail 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 data store. We collected 890 user sessions from instructors, teaching assistants, and students using CPM during the 2004-2005 academic year and the 2005 Fall semester at the University of Delaware.

MASPLAS is a Web application developed by one of the paper's authors for a regional workshop where we collected 169 user sessions. Users can register for the workshop, upload abstracts and papers, and view the schedule, proceedings, and other related information. MASPLAS displays front-end pages in HTML, the back-end code is implemented in both Java and JSP, and a MySQL database is at the back end.

Table 1 also shows the theoretical and measured number of concepts on applying concept analysis to our subject applications. The lattice's size for each application is much smaller than the theoretical bounds discussed in Section 6. For example, CPM's resulting lattice has 2,695 concepts, which is far less than $2^{q_{max}}|O| = 9.8 \times 10^{14}$. Similarly to Godin et al.'s conclusions from experimental results that the growth factor in practice is far less than $2^{q_{max}}$ [45], our experiments showed that the size of the lattice is less than $q_{mean}|O|$ ($6.7 \times 890 = 5,963$ concepts for CPM), where q_{mean} is the mean number of the test cases' unique requests. We believe that, in general, the size of the lattice will be much smaller than the theoretical bounds because users (and, therefore, the test cases) do not cover all possible subsets of some attributes: Some attributes will always appear together and others will never appear together. Also, because most users are likely to access the application similarly, the set of distinct requests is similar, that is, only a few test cases give rise to a unique set of requests.

7.3 Experimental Framework

We constructed an automated system, illustrated in Fig. 7, that enables the incremental generation of a reduced test suite of user sessions, replaying the original and reduced test suite, and generation of program coverage and fault

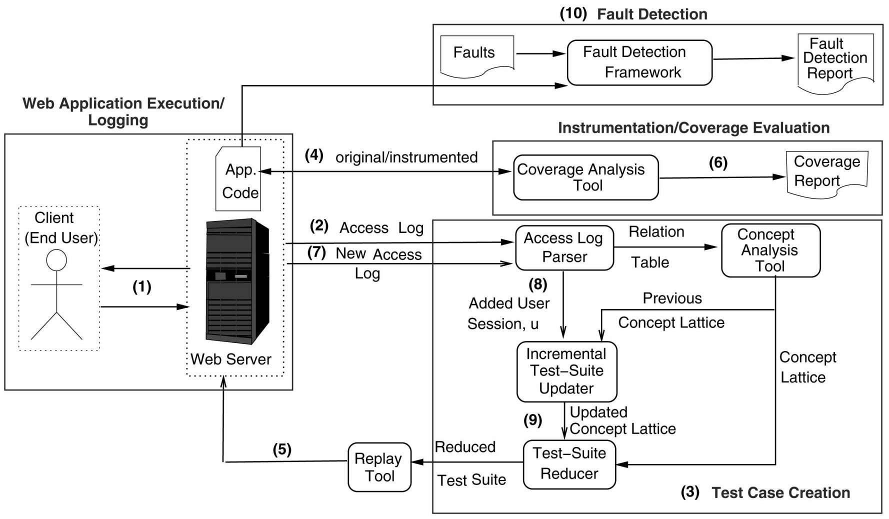


Fig. 7. Experimental framework.

detection reports of the replayed suites. The process begins with the collection of user-session data, as shown in Step 1 in Fig. 7. For each user request, we record the originating IP address, timestamp, base request, cookie information, and GET or POST data. The access log is sent to the test suite creation engine in Step 2. In Step 3, we reduce the initial set of user sessions, whereas, in Step 8, we incrementally update the reduced test suite with new user sessions from the new access log in Step 7. In Step 3, we parse the access log to generate a relation table, which is input into the *Concept Analysis Tool*, Lindig's `concepts` tool [46], to construct a concept lattice. The heuristic for creating the reduced test suite from the lattice is embedded in the *Test Suite Reducer*.

In Step 4, the Clover [48] *Coverage Evaluator* instruments the application's source files for program coverage analysis and generates the coverage report in Step 6. For JSPs, the server compiles the JSP into a Java servlet, which Clover then instruments for coverage. Since the compiled Java servlet contains server-dependent code, similar to libraries in traditional programs, we want to ignore the server-dependent code in our coverage reports. Thus, during the instrumentation phase, we instruct Clover to ignore server-dependent statements and instrument only the Java code that maps directly back to the JSPs. Steps 4 and 6 are the only steps that assume a Java-based Web application; to handle other languages, we need only plug in a coverage tool that handles that language.

Step 5 replays the reduced suite. We implemented a customized replay tool using HTTPClient [49], which handles GET and POST requests, file upload, and maintaining the client's session state. Our replay tool takes as input the user sessions in the form of sequences of base requests and name-value pairs and replays the sessions on the application while maintaining the state of the user session and the application.

There are a number of ways to maintain the Web application state during replay [15]. Our current implementation, `with_state` replay, replays the original suite and records the application state before each user session. The framework then restores the application state before replaying each user session in the reduced suite.

Steps 7, 8, and 9 incrementally update the reduced test suite. The *Incremental Test-Suite Updater* implements the incremental concept analysis algorithm [45], which takes as input the lattice and a new user session. The *Test-Suite Reducer* then uses the updated lattice to create the reduced test suite. Step 10 is the *Fault Detection Phase*.

7.4 Methodology

To answer our research questions, we generated reduced suites for each subject application using each of the proposed heuristics. Because of the nondeterminism in selecting test cases when multiple test cases are clustered together in a node, we generated 30 reduced test suites for each heuristic. We replayed each reduced suite and measured the program code covered and the number of faults detected.

For the fault detection experiments, graduate and undergraduate students familiar with JSP, Java servlets, and HTML manually seeded realistic faults in Book, CPM, and MASPLAS. Faults were seeded in the application—one fault per version. In general, five types of faults were seeded in the applications—data store faults (faults that exercise the application code interacting with the data store), logic faults (application code logic errors in the data and control flow), form faults (modifications to name-value pairs and form actions), appearance faults (faults that change the way in which the user views the page), and link faults (faults that change the hyperlink's location). The total number of faults seeded in each application is presented in Table 1. The original and reduced suites were replayed on the nonfaulty or "clean" application to generate the expected output, as well as on each faulty version to generate the actual output. Although we developed several automated, modular oracle comparators for Web applications in previous work [15], we manually created the fault detection report for each reduced test suite in this paper to eliminate the threat to validity from an oracle comparator that could have false positives and false negatives.

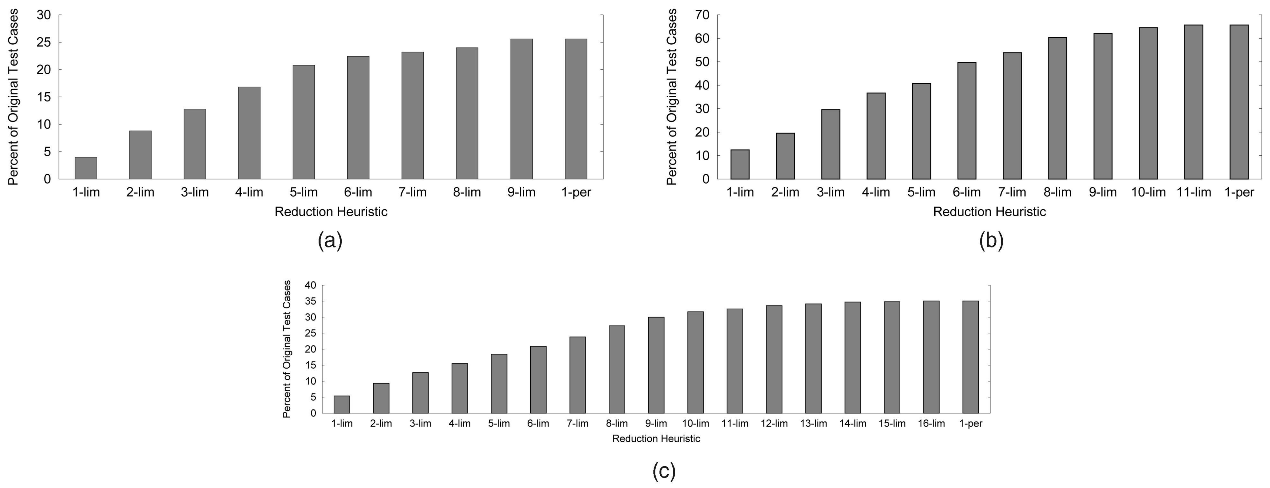


Fig. 8. Reduced test suite size. (a) Book. (b) MASPLAS. (c) CPM.

7.5 Threats to Validity

Because of the nondeterminism in selecting test cases when multiple test cases are in a concept node, we may select a test case that does not have the same program coverage and fault detection effectiveness as every other test case in the node. We attempt to minimize this threat by executing the reduction algorithm 30 times and reporting the mean, the standard deviation, and the overall distribution of each metric. The size of our subject Web applications may not show large differences in program code coverage and fault detection when comparing the reduction heuristics. Because we conducted our experiments on three applications, generalizing our results to other Web applications may not be fair. The types of users and usage of each application could affect the amount of test suite reduction because our results depend on the collected user sessions.

Since we only consider the number of faults detected by the reduced suites and not the severity of the faults missed, the conclusions of our experiment could be different if the results were weighted by fault severity. Manually seeded faults may be more difficult to expose than naturally occurring faults [50]. Though we tried to model the seeded faults as closely as possible to naturally occurring faults —even including naturally occurring faults from previous deployments of CPM and MASPLAS—some of the seeded faults may not be accurate representations of natural faults.

7.6 Data and Analysis

7.6.1 Reduced Test Suite Size

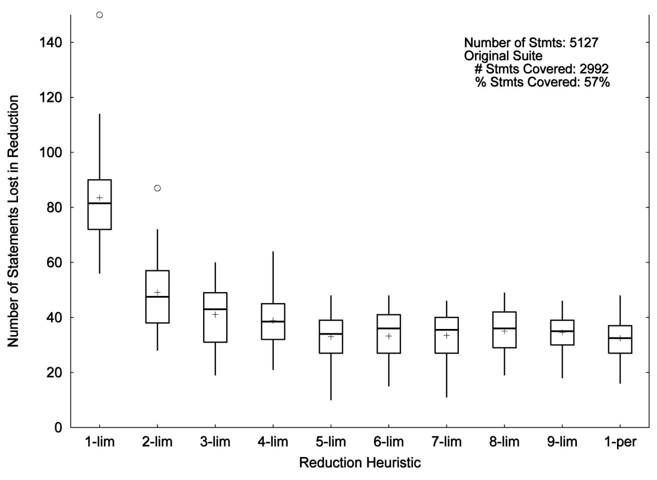
Fig. 8 shows the size of the reduced test suites generated for our three subject applications using the various heuristics. The x-axis represents the heuristics. We abbreviate the heuristics as follows: 1-lim refers to the *test-all-exec-requests* heuristic, k-lim refers to the k*-limited* heuristic for $k > 2$, and 1-per refers to the *one-per-node* heuristic. We use the same x-axis labels for all the graphs in this paper. The y-axis represents the reduction in test suite size as a percentage of the original test suite. Unlike coverage and fault detection, whose results vary with each reduced suite selected, reduced suite size is constant for a given subject application and reduction heuristic, regardless of the number of times the reduction heuristic is applied. Therefore, we use a bar graph, rather than a box plot, to represent the reduction in test suite size.

For all of the subject applications, we observe that *test-all-exec-requests* (1-lim in figures) selects the smallest reduced suite, the *one-per-node* heuristic selects the largest test suite, and, as the k value increases, the k*-limited* heuristic selects larger reduced suites. As expected, *one-per-node* provides the least reduction in test suite size since the heuristic selects one test case from each concept node, effectively removing duplicate test cases in terms of sets of base requests. However, the "duplicate" test cases contain data along with the base requests and, therefore, are not necessarily duplicates in terms of program coverage and fault detection.

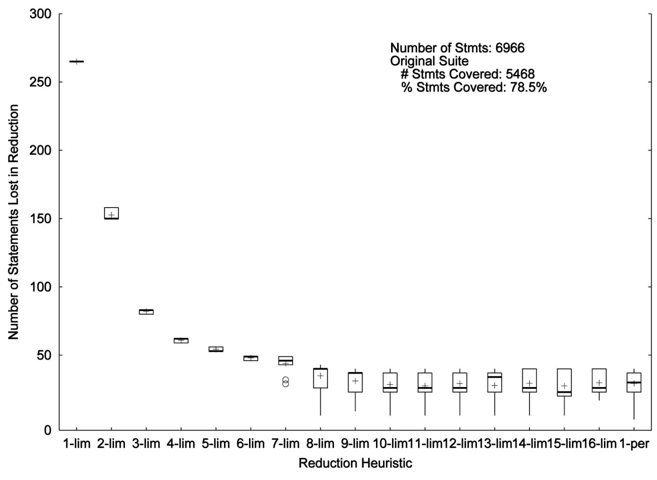
7.6.2 Program Coverage Effectiveness

Fig. 9 shows the program coverage loss of the Book and CPM reduced suites. The y-axis is the number of statements covered by the original suite that the reduced suite does not cover. In each figure, the box represents 50 percent of the data and spans the width of the inner quartile range (IQR), with each whisker extending $1.5 * IQR$ beyond the top and bottom of the box. The center horizontal line within each box denotes the median coverage loss, $+$ represents the mean, and $\circ$ represents an outlier. A similar graph is used to show the fault detection effectiveness results.

As expected, in Fig. 9, we observe that, as the reduced test suite size increases, the loss in program coverage decreases. The heuristic *test-all-exec-requests* loses much more program code coverage than the reduced suites selected by the *2-limited* heuristics for the small increase in reduced test suite size. In addition, the statements missed between the k*-limited* reduced suites becomes almost constant as k increases, especially for $k > 3$ (Book) and $k > 8$ (CPM). To support our observation that there is little difference in coverage between reduced suites for higher values of k, we examined the individual statements covered by each test case. We found that there exist statements covered only by a single test case and that these test cases are clustered lower in the lattice—$k \leq 5$ for Book



(a)



(b)

Fig. 9. Program coverage effectiveness. (a) Book. (b) CPM.

Suite	μ	σ
1-lim	83.53	18.90
2-lim	49.10	13.69
3-lim	41.03	11.35
4-lim	38.87	10.15
5-lim	33.00	8.99
6-lim	33.23	9.38
7-lim	33.47	8.97
8-lim	35.03	8.06
9-lim	34.57	6.12
1-per	32.43	7.96

(a)

Suite	μ	σ
1-lim	2.53	0.51
2-lim	1.93	0.69
3-lim	1.93	0.64
4-lim	1.73	0.74
5-lim	0.90	0.61
6-lim	0.67	0.55
7-lim	0.93	0.64
8-lim	0.73	0.64
9-lim	0.97	0.56
1-per	0.83	0.65

(b)

Suite	μ	σ
1-lim	265.00	0.00
2-lim	152.67	3.84
3-lim	82.00	1.44
4-lim	60.90	1.47
5-lim	54.40	1.52
6-lim	48.10	1.40
7-lim	44.00	7.12
8-lim	34.97	10.04
9-lim	31.13	9.52
10-lim	28.57	8.64
11-lim	27.50	9.24
12-lim	29.30	9.41
13-lim	27.89	11.36
14-lim	29.40	9.64
15-lim	27.50	9.94
16-lim	29.83	8.24
1-per	29.37	9.84

(c)

Suite	μ	σ
1-lim	10.00	0.00
2-lim	9.00	0.00
3-lim	5.67	0.52
4-lim	5.59	0.51
5-lim	5.47	0.51
6-lim	5.70	0.47
7-lim	5.70	0.47
8-lim	3.17	.95
9-lim	1.93	1.05
10-lim	1.77	1.19
11-lim	1.93	1.14
12-lim	1.83	1.15
13-lim	1.80	1.24
14-lim	1.80	1.10
15-lim	1.93	1.17
16-lim	1.83	1.05
1-per	1.60	1.16

(d)

Fig. 10. Mean (μ) and standard deviation (σ) for program coverage and fault detection effectiveness. (a) Book coverage loss. (b) Book fault loss. (c) CPM coverage loss. (d) CPM fault loss.

and $k \leq 9$ in CPM. Therefore, selecting k values above these levels does not significantly improve the amount of program code covered by the reduced suites. These results also support our design of the reduction heuristics to traverse the lattice from the bottom up, rather than top down, because traversing bottom up ensures selecting test cases that cover unique statements. Also, none of the reduced suites cover all of the statements covered by the original suite. It may not be possible to get a reduced suite with 100 percent original suite statement coverage with our heuristics because concept analysis clusters test cases based on base request coverage and not statement coverage.

Figs. 10a and 10c show the mean and standard deviation of the program coverage lost on executing 30 reduced suites generated using each heuristic. The variation in code loss between the reduced suites selected by a given heuristic is larger for Book than CPM, as seen by the standard deviation in Figs. 10a and 10c, respectively. On examining the lattice, we found that, at lower levels, Book has a larger number of test cases clustered together in a node. The clustered test cases do not all cover the same code. Thus, selecting test cases at random causes the variation in code covered. For CPM (Fig. 10c), the variation in code loss is smaller because, often, a node contains only one test case at lower levels.

We do not present graphs of the MASPLAS coverage and fault detection results because the *test-all-exec-requests* reduced test suite for MASPLAS covers only one less statement than the original suite. Since the other heuristics are at least as good as *test-all-exec-requests*, that is, the reduced suites generated by the other heuristics cover all of the statements covered by the original suite, we did not observe significant improvement by using the other heuristics.

7.6.3 Fault Detection Effectiveness

Fig. 11 presents the fault detection results of the reduced suites for Book and CPM. The y-axis presents the number of faults detected by the original suite that the reduced suite misses. Figs. 10b and 10d show the mean and standard deviation of the faults lost on executing 30 reduced suites for each heuristic.

The fault detection effectiveness of a reduced suite depends on 1) the number of test cases in the reduced

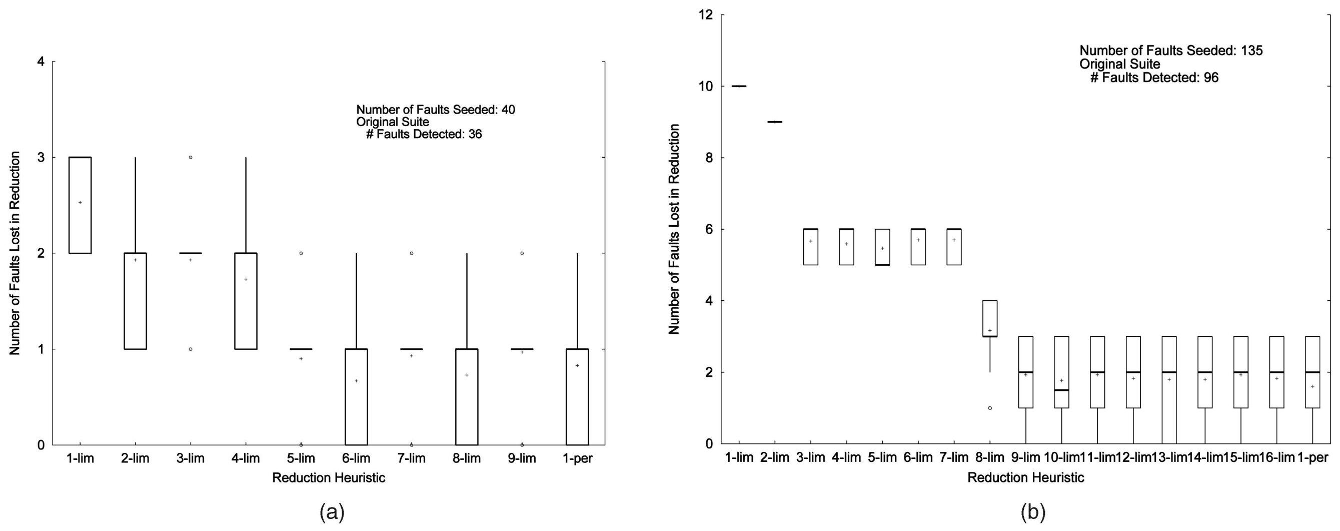


Fig. 11. Fault detection effectiveness. (a) Book. (b) CPM.

suite and 2) *the specific* test cases that are selected by the reduction heuristic from each concept in the lattice. In Fig. 11, the number of test cases in the reduced suite affects the number of faults lost—as k and the size of the reduced suites increase, the number of faults lost decreases. Similarly to program coverage, the faults missed between the *k-limited* reduced suites becomes almost constant as k increases, especially for $k \geq 5$ (Book) and $k \geq 9$ (CPM). The specific test cases that are selected appear to have little effect on the number of faults detected, from the lack of variation in Fig. 11 (1-2 faults for Book and 0-3 for CPM).

We note that one fault missed by all reduced suites in Book interacts with the server state of the system. During replay, we reset the database state to the time before the test case execution, but we ignored the server state, so the fault is never caught. In previous work [15], we presented the challenges and reasoning for ignoring the server state.

The *test-all-exec-requests* reduced test suite for MASPLAS detects all of the faults detected by the original suite. Since the other heuristics generate test suites that are at least as good (that is, they also detect all of the faults detected by the original suite), graphs of these MASPLAS results are not included.

7.6.4 Incremental versus Batch

We evaluated the effectiveness of incremental concept analysis by comparing the size of the files required by the incremental and batch techniques to eventually produce the reduced suite. Time costs are not reported for incremental and 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 academic tool `concepts` [46].

Table 2 presents the suite sizes and space savings for incremental and batch concept analysis for the *test-all-exec-requests* heuristic. The batch algorithm requires the complete original set of test cases to generate the reduced test suite, whereas the incremental concept analysis algorithm only requires the reduced test suite and the lattice. We do not show the space requirements for the lattice in Table 2 because the space required depends on the lattice implementation. For example, the sparse representation of the lattice contains all of the information in the full lattice representation with smaller space requirements. In Table 2, we note that, for all of the subject applications, space savings greater than 82 percent are obtained by using the incremental concept analysis. A batch analysis of all of the test cases yields the same reduced suite as the batch analysis of an initial subset of the test cases followed by an incremental analysis of the remaining test cases [14].

Thus, the incremental reduction process saves considerable space costs by not maintaining the original suite of test cases. In a production environment, the incremental reduction process could be performed overnight with the collection of that day's test cases to produce a fresh updated suite of test cases for testing while still saving space by keeping a continually reduced suite. In its current form, the incremental algorithm can be used only in situations where the user sessions (test cases) are changing and not where application versions change.

7.6.5 Analysis Summary

From our results, concept analysis-based reduction with the *test-all-exec-requests* heuristic produces a reduced test suite much smaller in size, but loses some of the original suite's effectiveness in terms of program coverage and fault detection. By reducing the test suite size, the tester saves time because the tester need not execute the large original suite. We also note that, as various heuristics are applied, the resulting reduced suites have varied fault detection and

TABLE 2
Space Requirements for Incremental and Batch for *test-all-exec-requests*

Applications	Original Suite	Reduced Suite	Space Savings
MASPLAS	336KB	20KB	94%
Book	1.1MB	65KB	94%
CPM	1.5MB	268KB	82%

program coverage effectiveness, as expected. Hence, a tester can choose an appropriate heuristic to obtain the desired suite characteristics.

Our results suggest that a trade-off exists between test suite size and effectiveness. Although *one-per-node* performs as effectively as the original suite in terms of program code coverage and fault detection for all of the applications, the trade-off between the reduced test suite size and the effectiveness of the suite needs to be considered. However, we observed that the *2-limited* heuristic selects a test suite that is only slightly larger (3-5 percent) in size than the *test-all-exec-requests* heuristic, but is more effective in terms of program coverage and fault detection effectiveness. For larger values of k, the tester must determine if the small increase in effectiveness is worth the large increase in test suite size.

We investigated two other approaches to converting the lattice into a tree prior to applying the *k-limited* heuristic. In cases of conflict regarding a node's position, we followed two other approaches: 1) Place the node at the lower of the two levels and 2) compare the node's attribute set to its predecessors' attribute sets (when traversing the node from $\perp$) and assign the node to the level where the difference is the least [14]. In our experimental investigation, we found that the reduced suites generated by these two techniques are similar in program coverage and fault detection effectiveness to the suites generated by the *k-limited* technique presented in this paper.

We observed that, when a large number of test cases are clustered together, the variation in suite effectiveness is large for a given selection heuristic. If a large number of test cases are clustered together, our results suggest that using different attributes for clustering test cases, such as sequences of requests or data associated with the request, could reduce the variation caused by using the current attribute, base request only [51]. The associated trade-off, however, is the increase in the reduced test suite size.

We also evaluated the effectiveness of incremental versus batch concept analysis in terms of space savings and found that considerable storage can be saved by using incremental concept analysis for test suite update.

8 Conclusions and Future Work

By applying concept analysis to cluster test cases and then carefully selecting test cases 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. We presented three heuristics for test suite selection. Our experiments show that statement coverage similar to that of the original suite can be sustained while reducing the storage requirements. Similarly to other experiments [9], our experiments show that there is a trade-off between the amount of test suite reduction and fault detection capability. However, the incremental update algorithm enables a continuous examination of new test cases that could increase fault detection capability without storing the larger set of session data to determine the reduced test suite.

From our experimental evaluations, the *2-limited* heuristic appears to be a good compromise between maintaining the test suite size that covers all base requests and maintaining distinct use case representation while still being effective in terms of program coverage and fault detection. Our experimental results suggest that applying concept analysis with base requests as an attribute clusters sessions that uniquely detect a fault together with sessions that do not detect the fault but have similar attribute coverage. The results motivated clustering by concept analysis with other attributes, such as with the data associated with the base request and base request sequences [51].

To our knowledge, incremental approaches do not exist for the existing requirements-based reduction techniques [12], [25], [26]. Thus, the incremental approach to concept analysis described in this paper provides a space-saving alternative to the requirements-based reduction techniques. In our previous studies [52], we found that, for the domain of Web applications, reduced suites based on reduction by base request coverage that maintains use case representation is as effective in terms of program code coverage and fault detection as reduced suites from reduction techniques that use program-based requirements [12]. In addition, clustering test cases by concept analysis and applying the reduction heuristics is cheaper than the reduction techniques based on program-based requirements since our technique saves time by not computing the requirements' mappings prior to reduction.

In the future, we plan to investigate issues in regression user-session-based testing of Web applications, such as identifying and removing obsolete test cases and test case prioritization.

Acknowledgments

Valentin Mihaylov from Drexel University and Stacey Ecott from Tufts University assisted in the fault seeding process for Bookstore and MASPLAS, respectively. The authors thank the anonymous reviewers of the paper for their helpful comments. Portions of this paper previously appeared in [16].

References

[1] R. Hower, "Web Site Test Tools and Site Management Tools," http://www.softwareqatest.com/qatweb1.html, 2007.
[2] "Jakarta Cactus," http://jakarta.apache.org/cactus/, 2007.
[3] "JUnit," http://www.junit.org, 2007.
[4] "HttpUnit," http://httpunit.sourceforge.net, 2007.
[5] C.-H. Liu, D.C. Kung, and P. Hsia, "Object-Based Data Flow Testing of Web Applications," *Proc. First Asia-Pacific Conf. Quality Software*, pp. 7-16, 2000.
[6] G. Di Lucca, A. Fasolino, F. Faralli, and U.D. Carlini, "Testing Web Applications," *Proc. 18th IEEE Int'l Conf. Software Maintenance*, pp. 310-319, 2002.
[7] F. Ricca and P. Tonella, "Analysis and Testing of Web Applications," *Proc. 23rd Int'l Conf. Software Eng.*, pp. 25-34, 2001.
[8] A. Andrews, J. Offutt, and R. Alexander, "Testing Web Applications by Modeling with FSMs," *Software and Systems Modeling*, vol. 4, no. 3, pp. 326-345, July 2005.
[9] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II, "Leveraging User Session Data to Support Web Application Testing," *IEEE Trans. Software Eng.*, vol. 31, no. 3, pp. 187-202, Mar. 2005.
[10] "Rational Robot," http://www.ibm.com/software/awdtools/tester/robot/, 2007.
[11] "Parasoft WebKing," http://www.parasoft.com, 2007
[12] M.J. Harrold, R. Gupta, and M.L. Soffa, "A Methodology for Controlling the Size of a Test Suite," *ACM Trans. Software Eng. and Methodology*, vol. 2, no. 3, pp. 270-285, July 1993.

[13] I. Jacobson, "The Use-Case Construct in Object-Oriented Software Engineering," *Scenario-Based Design: Envisioning Work and Technology in System Development,* J.M. Carroll, ed., 1995.

[14] S. Sampath, "Cost-Effective Techniques for User-Session-Based Testing of Web Applications," PhD dissertation, Univ. of Delaware, 2006.

[15] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock, "Automated Replay and Failure Detection for Web Applications," *Proc. 20th Int'l Conf. Automated Software Eng.,* pp. 253-262, 2005.

[16] S. Sampath, V. Mihaylov, A. Souter, and L. Pollock, "A Scalable Approach to User-Session Based Testing of Web Applications through Concept Analysis," *Proc. 19th Int'l Conf. Automated Software Eng.,* pp. 132-141, 2004.

[17] E. Kirda, M. Jazayeri, C. Kerer, and M. Schranz, "Experiences in Engineering Flexible Web Service," *IEEE MultiMedia,* vol. 8, no. 1, pp. 58-65, Jan.-Mar. 2001.

[18] D.C. Kung, C.-H. Liu, and P. Hsia, "An Object-Oriented Web Test Model for Testing Web Applications," *Proc. First Asia-Pacific Conf. Quality Software,* pp. 111-120, 2000.

[19] J. Sant, A. Souter, and L. Greenwald, "An Exploration of Statistical Models of Automated Test Case Generation," *Proc. Third Int'l Workshop Dynamic Analysis,* May 2005.

[20] J. Offutt and W. Xu, "Generating Test Cases for Web Services Using Data Perturbation," *Proc. Workshop Testing, Analysis, and Verification of Web Services,* 2004.

[21] C. Fu, B. Ryder, A. Milanova, and D. Wonnacott, "Testing of Java Web Services for Robustness," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis,* pp. 23-34, 2004.

[22] Y. Deng, P. Frankl, and J. Wang, "Testing Web Database Applications," *SIGSOFT Software Eng. Notes,* vol. 29, no. 5, pp. 1-10, 2004.

[23] M. Benedikt, J. Freire, and P. Godefroid, "VeriWeb: Automatically Testing Dynamic Web Sites," *Proc. 11th Int'l Conf. World Wide Web,* May 2002.

[24] T.Y. Chen and M.F. Lau, "Dividing Strategies for the Optimization of a Test Suite," *Information Processing Letters,* vol. 60, no. 3, pp. 135-141, Mar. 1996.

[25] J. Offutt, J. Pan, and J. Voas, "Procedures for Reducing the Size of Coverage-Based Test Sets," *Proc. 12th Int'l Conf. Testing Computer Software,* pp. 111-123, 1995.

[26] J.A. Jones and M.J. Harrold, "Test Suite Reduction and Prioritization for Modified Condition/Decision Coverage," *IEEE Trans. Software Eng.,* vol. 29, no. 3, pp. 195-209, Mar. 2003.

[27] D. Jeffrey and N. Gupta, "Test Suite Reduction with Selective Redundancy," *Proc. 21st IEEE Int'l Conf. Software Maintenance,* pp. 549-558, 2005.

[28] S.M. Master and A. Memon, "Call Stack Coverage for Test Suite Reduction," *Proc. 21st IEEE Int'l Conf. Software Maintenance,* pp. 539-548, 2005.

[29] D. Leon, W. Masri, and A. Podgurski, "An Empirical Evaluation of Test Case Filtering Techniques Based on Exercising Complex Information Flows," *Proc. 27th Int'l Conf. Software Eng.,* pp. 412-421, 2005.

[30] D. Leon and A. Podgurski, "A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases," *Proc. 14th Int'l Symp. Software Reliability Eng.,* pp. 442-453, 2003.

[31] M. Harder, J. Mellen, and M.D. Ernst, "Improving Test Suites via Operational Abstraction," *Proc. 25th Int'l Conf. Software Eng.,* pp. 60-71, 2003.

[32] G. Birkhoff, *Lattice Theory,* vol. 5. American Math. Soc. Colloquium Publications, 1940.

[33] M. Krone and G. Snelting, "On the Inference of Configuration Structures from Source Code," *Proc. 16th Int'l Conf. Software Eng.,* pp. 49-57, 1994.

[34] G. Snelting and F. Tip, "Reengineering Class Hierarchies Using Concept Analysis," *Proc. Sixth ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 99-110, 1998.

[35] G. Ammons, D. Mandelin, and R. Bodik, "Debugging Temporal Specifications with Concept Analysis," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 182-195, 2003.

[36] T. Kuipers and L. Moonen, "Types and Concept Analysis for Legacy Systems," *Proc. Eighth Int'l Workshop Program Comprehension,* pp. 221-230, 2000.

[37] C. Lindig and G. Snelting, "Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis," *Proc. 19th Int'l Conf. Software Eng.,* pp. 349-359, 1997.

[38] P. Tonella, "Concept Analysis for Module Restructuring," *IEEE Trans. Software Eng.,* vol. 27, no. 4, pp. 351-363, Apr. 2001.

[39] M. Siff and T. Reps, "Identifying Modules via Concept Analysis," *Proc. 13th Int'l Conf. Software Maintenance,* pp. 170-179, 1997.

[40] T. Eisenbarth, R. Koschke, and D. Simon, "Locating Features in Source Code," *IEEE Trans. Software Eng.,* vol. 29, no. 3, pp. 210-224, Mar. 2003.

[41] T. Ball, "The Concept of Dynamic Analysis," *Proc. Seventh ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 216-234, 1999.

[42] S. Tallam and N. Gupta, "A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization," *Proc. Sixth ACM SIGPLAN-SIGSOFT Workshop Program Analysis for Software Tools and Eng.,* pp. 35-42, 2005.

[43] A. Podgurski, W. Masri, Y. McCleese, F.G. Wolff, and C. Yang, "Estimation of Software Reliability by Stratified Sampling," *ACM Trans. Software Eng. and Methodology,* vol. 8, no. 3, pp. 263-283, 1999.

[44] W. Dickinson, D. Leon, and A. Podgurski, "Pursuing Failure: The Distribution of Program Failures in a Profile Space," *Proc. Ninth ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 246-255, 2001.

[45] R. Godin, R. Missaoui, and H. Alaoui, "Incremental Concept Formation Algorithms Based on Galois (Concept) Lattices," *Computational Intelligence,* vol. 11, no. 2, pp. 246-267, 1995.

[46] C. Lindig, "Christian Lindig > Software > Concepts," http://www.st.cs.uni-sb.de/~lindig/src/concepts.html, 2007.

[47] "Open Source Web Applications with Source Code," http://www.gotocode.com, 2007.

[48] "Cenqua: Clover," http://www.cenqua.com/clover/, 2007.

[49] "HTTPClient v0.3-3," http://www.innovation.ch/java/HTTPClient/, 2007.

[50] J.H. Andrews, L.C. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments," *Proc. 27th Int'l Conf. Software Eng.,* pp. 402-411, 2005.

[51] S. Sampath, S. Sprenkle, E. Gibson, and L. Pollock, "Web Application Testing with Customized Test Requirements—An Experimental Comparison Study," *Proc. 17th Int'l Symp. Software Reliability Eng.,* pp. 266-278, Nov. 2006.

[52] 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," *Proc. 21st Int'l Conf. Software Maintenance,* pp. 587-596, 2005.

Sreedevi Sampath received the BE degree in computer science and engineering from Osmania University in 2000 and the MS and PhD degrees in computer and information sciences from the University of Delaware in 2002 and 2006, respectively. She is an assistant professor in the Department of Information Systems at the University of Maryland, Baltimore County. Her research interests include software testing, Web applications, and software maintenance. She is interested in exploring the uses of Web application usage data, regression testing of Web applications, test-case generation for Web applications, and testing for security in applications. She is a member of the IEEE Computer Society.

Sara Sprenkle received the BS degree in computer science and the BS degree in mathematics from Gettysburg College, the MS degree in computer science from Duke University, and the PhD degree in computer and information sciences from the University of Delaware. She is currently an assistant professor at Washington and Lee University. Her research interests include software testing and distributed systems. She is a member of the ACM and ACM SIGSOFT.

Emily Gibson received the BS degree in computer science from the College of New Jersey in 2003 and the MS degree in computer and information sciences from the University of Delaware in 2005. She is currently working toward the PhD degree in computer and information sciences at the University of Delaware. Her research interests include natural language program analysis, software testing, and debugging. She is a member of the ACM.

Lori Pollock received the BS degree in computer science and economics from Allegheny College in 1981 and the MS and PhD degrees in computer science from the University of Pittsburgh in 1983 and 1986, respectively. She is a professor in the Department of Computer and Information Sciences at the University of Delaware. Her research focuses on program analysis for optimizing compilers, software testing, software tool development, parallel and distributed systems, and natural language analysis of software. She is currently a cochair of the Computing Research Association's Committee on the Status of Women in Computing (CRA-W). She received the University of Delaware's Excellence in Teaching Award in 2001 and the University of Delaware's E.A. Trabant Award for Women's Equity in 2004. She served on the executive committee and as an officer of ACM SIGPLAN for several terms. She is a member of the IEEE Computer Society.

Amie Souter Greenwald received the PhD degree in computer and information sciences from the University of Delaware. She is the software test lead and a developer focusing both on software quality and integrating cutting-edge technology into key components of location-based services at Alcatel-Lucent Bell Laboratories. Prior to joining Bell Laboratories, she was a member of the technical staff at Sarnoff Corp. Her testing expertise was used to develop a company-wide test process for software projects, in addition to being an integral developer on several projects. Before working in industry, she was an assistant professor at Drexel University with a research focus in the area of software testing.