Applying Concept Analysis to User-session-based Testing of Web Applications

Technical Report No. 2006-329

Sreedevi Sampath, Sara Sprenkle, Emily Gibson and Lori Pollock

Department of Computer & Information Sciences

University of Delaware

Newark, DE 19716

March 29, 2006

Applying Concept Analysis to User-session-based Testing of Web Applications

Sreedevi Sampath, Sara Sprenkle, Emily Gibson,

Lori Pollock, *Member, IEEE Computer Society*,

Amie Souter, *Member, IEEE Computer Society*

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 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 thus to 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 indicate 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

I. 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 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 associated with 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 pre-set 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]–[8] towards 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 input data that will exercise the paths to be tested, especially 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 URLs and name-value pairs (e.g., form field data) 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. Testing 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, 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 [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 provide a way to enhance an original 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. While 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 non-scalable.

This paper presents an approach for achieving scalable user-session-based testing of web applications. The key insight is formulating an approach to the test case selection problem for user-session-based testing based on clustering 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]. 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 minimal test suite. Through the 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 the reduced test suite for coverage analysis and fault detection [14], [15]. In our experiments, the resulting program coverage provided by the reduced test suite is almost identical to the original suite of user sessions, 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

1) 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
4) Experimental evaluation of the effectiveness of the reduced suites with three subject web applications

In Section II, we provide the background on web applications, user-session-based testing, and concept analysis. We apply concept analysis to user-session-based testing in Section III and formulate the test suite reduction problem using concept analysis. In Section IV, we present three heuristics for test suite reduction. In Section V, we present an approach to scalable test suite update with incremental concept analysis and in Section VI we present the space and time costs for incremental concept analysis. Section VII describes our experimental evaluation study with three subject applications. We conclude and present future work in Section VIII.

II. Background and State of the Art

A. Web Applications

Broadly defined, a web-based software system consists of a set of web pages and components that form a system that executes using web server(s), network, HTTP, and browser 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.

B. Testing Web Applications

1) Program-based Testing: In addition to tools that test the appearance and validity of a web application [1], tools that analyze and model the underlying structure and semantics of web programs exist.

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 easily can 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 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 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 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 and use coverage criteria based on Finite State Machine (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. It is also not clear if the model or testing strategy have been evaluated.

In summary, to enable 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.

2) User-session-based Testing: In user-session-based testing, each *user session* is a collection of user requests in the form of URL and name-value pairs. A user session begins when a new IP address makes a request from the server and ends when the user leaves the web site or the session times out. User sessions are thus uniquely identified by the user's IP address and requests that arrive from the same IP but after an interval of 45 minutes are considered as a new session. 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]–[11], [19].

Elbaum et al. [20] 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 testing performed during the development phase of the application [5]–[7], [21]–[24]. In addition, user-session-based testing is particularly useful in the absence of program specifications and requirements.

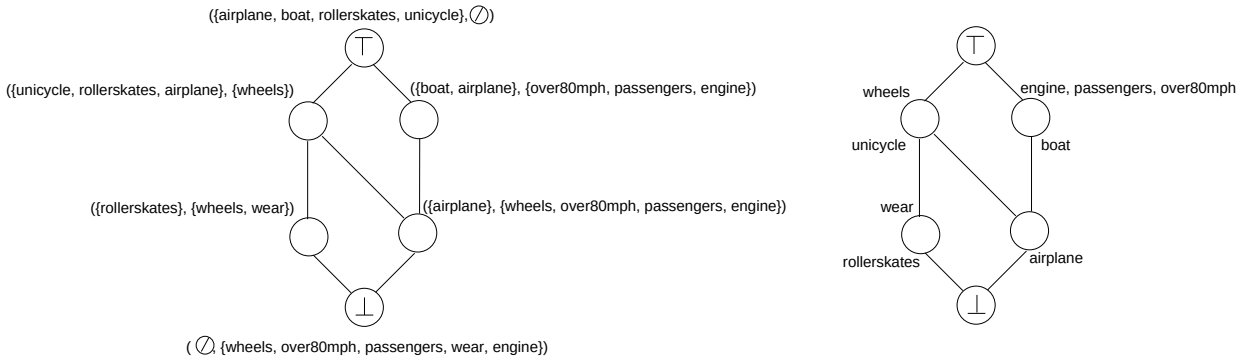
C. Test Suite Reduction

A large number of user sessions can be logged for a frequently used web application, and it may not be practical to use all the user sessions when testing the application. In addition, an evolving program can cause some test cases to become obsolete and also may require augmenting an existing test suite with new test cases that test the program's new functionality. The additional test cases can lead to redundant test cases, which waste valuable testing resources. The goal of test suite reduction for a given test requirement (e.g., statement or all-uses coverage) is to produce a test suite that is smaller than the original suite's size yet still satisfies the original suite's test requirement. Advantages to test-suite reduction techniques include reducing the cost of executing, validating, and managing test suites as the application evolves.

Several test suite reduction techniques have been proposed [12], [25]–[29]. 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. Such techniques assume that the test suite to be reduced is complete and associations between test cases and test requirements are known before reduction heuristics are applied. Harder et al. [30] proposed

	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) Relation table (i.e., context)



(b) Full and sparse concept lattice representations

Fig. 1. Example of concept analysis for modes of transportation

a technique to generate, augment, and minimize test suites, but their technique involves dynamically generating operational abstractions, which can be costly.

D. 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 [31]. 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 false. For example, consider the context depicted in Figure 1(a). The object set O is $\{$*airplane, boat, rollerskates, unicycle*$\}$; the attribute set A is $\{$*wheel(s), over80mph, passengers, wear, engine* $\}$.

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* $O_1 \subseteq O_2$. Similarly, the partial order 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 $\perp$ 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 $(\{airplane, boat, rollerskates, unicycle\}, \oslash)$, while $\perp$ is $(\oslash, \{wheels, over80mph, passengers, wear, engine\})$.

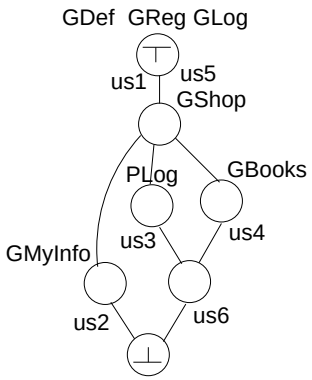
The full concept lattice is depicted on the left side of Figure 1(b). A sparse representation (shown on the right side) can be used to depict the concept lattice. In the sparse representation, a particular node n is labeled only with each attribute $a \in A$ and each object $o \in O$ that is introduced by node n. Attribute sets are shown just above each node, while 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 $(\{unicycle, rollerskates, airplane\}, \{wheels\})$.

Snelting first introduced the idea of concept analysis for use in software engineering tasks, specifically for configuration analysis [32]. Concept analysis has also been applied to evaluating class hierarchies [33], debugging temporal specifications [34], redocumentation [35], and recovering components [36]–[39]. Ball introduced the use of concept analysis of test coverage data to compute dynamic analogs to static control flow relationships [40]. 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. Since our initial work [16], Tallam et al. [41] 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 [42], cluster analysis

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

(a) Original Suite of User Sessions



(b) Concept Lattice

Fig. 2. (a) Relation table and (b) Concept lattice for test suite reduction

has also been utilized to partition a set of program executions into clusters based on the similarity of their profiles. Dickinson et al. [43] 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 [43]. Clustering has also been used to reverse engineer systems [44]–[46].

III. 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 (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 [47]. 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, possibly with different name-value pairs.

As an example, the relation table in Figure 2(a) shows the context for a user-session based test suite

for a portion of a bookstore web application [48]. 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. Based on the original relation table, concept analysis derives the sparse representation of the lattice lattice in Figure 2 (b).

A. 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 Figure 2 (b), 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 labeled by URL *u* access the URL *u*. In Figure 2 (b), 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 $\perp$ of the lattice denotes the user sessions that access all URLs in the context. Here, $\perp$ 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* both 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 $\perp$ 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*.

B. 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 URLs executed by a user session. 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 $\perp$ are more similar in their set of URL requests than nodes higher in the concept lattice along a certain path in the lattice. Our previous studies have shown that similarity in URLs translates to similarity in covering similar subsequences of URLs [47] and covering similar program characteristics [49]. We interpret the similarities as representing similar use cases. We developed heuristics for selecting a subset of user sessions to be maintained as the current test suite, based on the current concept lattice. In the next section we present three heuristics for test suite reduction based on the concept lattice.

IV. Test Selection Heuristics

A. One-per-node Heuristic

The *one-per-node* heuristic seeks to cover all the URLs present in the original suite and maximize use case representation in the reduced suite by selecting one session from every node (i.e., cluster) in the lattice without duplication. To implement the *one-per-node* heuristic, one session is selected from each node in the concept lattice starting at $\perp$. In the presence of multiple sessions in a cluster, a single session is selected, since each cluster is viewed as a collection of similar use cases [49]. In Figure 2 (b) the *one-per-node* heuristic selects the sessions *us2, us6, us3, us4, us1* for the reduced suite on traversing the lattice from $\perp$ to $\top$.

The *one-per-node* heuristic appears to naively exploit the lattice's clustering and partial ordering. However, nodes that are higher in the lattice, i.e., further away from $\perp$ contain URLs (attributes) that are

accessed by many user sessions (objects). The *one-per-node* heuristic thus targets popular URL sets—and program code covered on executing these URL sets—by including sessions from each level of the lattice in the reduced suite. However, by selecting from every node in the lattice, the *one-per-node* heuristic will create a large reduced test suite.

B. Test-all-exec-URLs Heuristic

In our heuristic for user-session selection, which we call *test-all-exec-URLs* (presented in [16]), we seek to identify the smallest set of user sessions that will cover all of the URLs executed by the original test suite while representing different use cases. This heuristic is implemented as follows: the reduced test suite is set to contain a user session from $\perp$, if the set of user sessions at $\perp$ is nonempty, and from each node next to $\perp$, that is one level up the lattice from $\perp$, 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 $\perp$'s object set is empty then the objects in a given *next-to-bottom* node share the exact same set of attributes. Since one of the *test-all-exec-URLs* heuristic goals is to increase the use case representation in the reduced test suite, we include user sessions from both $\perp$ and *next-to-bottom*. We do not select duplicate sessions when selecting user sessions from $\perp$ and *next-to-bottom* nodes.

We have performed experimental studies and user-session analysis to investigate the intuitive reasoning behind clustering user sessions based on concept analysis and the *test-all-exec-URLs* heuristic for test case 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 [47]. 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 also is to represent use cases in the reduced suite in addition to satisfying all the requirements (i.e., URLs), our approach differs from traditional reduction techniques which select the next user session with most additional coverage until 100% coverage is obtained [12].

In our example in Figure 2 (a), the original test suite is all the user sessions in the original context.

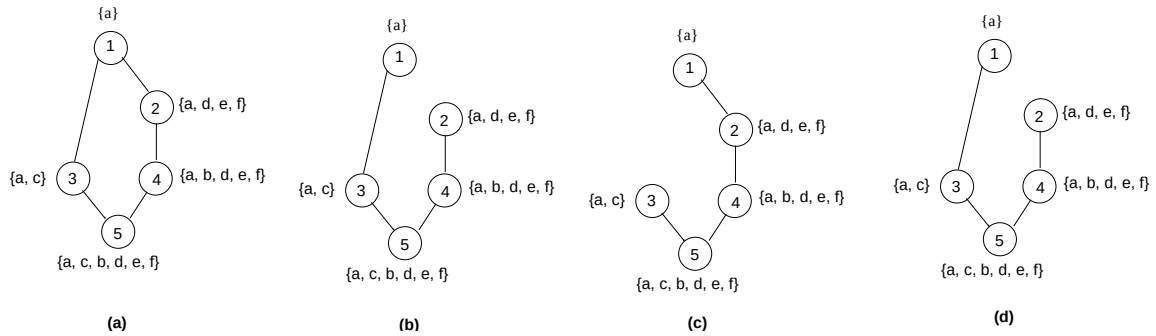


Fig. 3. (a) Initial Lattice (b) Approach *LLF* (c) Approach *HLF* (d) Approach *CLF*

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 the two user sessions is exactly the set of all URLs requested by the original test suite. In addition, in previous work [49] we found that *next-to-bottom* nodes represent distinct use cases and thus cover a different set of methods and faults in the program code. Thus, the reduced suite obtained by applying the *test-all-exec-URLs* cover all URLs covered by the original suite, while representing different use cases of the application. Since *test-all-exec-URLs* heuristic selects a small subset of sessions, the reduced test suite selected is likely to have low program code coverage and fault detection effectiveness compared to the original suite. This expectation motivated the development of the the *k-limited* heuristics presented in the next section.

C. k-limited Heuristic

The *k-limited* heuristic selects a suite of user sessions to cover all the URLs covered by the original suite while maintaining varied use cases in the reduced suite, beyond the use case representation provided by the *test-all-exec-URLs* heuristic. We hypothesize that a tradeoff exists between the use cases represented in the reduced suite and the program coverage and fault detection capabilities of the suite. The first step in the *k-limited* heuristic is to convert the concept lattice into a tree. For a specified *k*, we then select one session from each node in the tree that is *k levels* up from $\perp$.

In the initial concept lattice, depending on the path traversed from $\perp$, a node can be considered at

multiple levels. Figure 3 (a) shows an example lattice with only the attributes labeling each node. For presentation only, each node is also numbered. In Figure 3 (a), assuming $\perp$ is at *level 0*, *node 1* can be considered to be at *level 3* along the path $\langle$*node 5, node 4, node 2, node 1*$\rangle$, and at *level 2* along the path $\langle$*node 5, node 3, node 1*$\rangle$. To resolve the conflict of a node appearing at more than one level, we followed three different approaches to convert the lattice into a tree.

In the first approach, called *lowest level first* (*LLF*), if a node appears at two levels, we assign the node to the lower of the two levels. *LLF* creates a breadth first tree representation of the lattice. In Figure 3 (a), *node 1* would be placed at *level 2*. The resulting tree representation of the example lattice is shown in Figure 3 (b). The advantage of adopting the *LLF* strategy and applying the *k-limited* heuristic is that when we traverse the lattice bottom-up to a certain *k* level, we include as many nodes as possible in the reduced suite, thus increasing the use cases represented by the test suite. However, the disadvantage of the *LLF* approach is that we may select nodes that have a small set of attributes in common among the objects of the node. In earlier work [49] we have shown that as the number of attributes in a node decreases, the user session similarity in terms of program characteristics, such as program code covered and faults detected by the sessions decreases. Thus, choosing sessions from nodes with small attribute sets may result in choosing sessions that are not very representative of the other sessions in the node in terms of use case representation.

In the second approach, which we call *highest level first* (*HLF*), if a node is found to exist at two levels in the concept lattice, the node is considered to be at the higher of the two levels. In Figure 3 (a), *node 1* would be assigned to *level 3*, creating the tree in Figure 3 (c). The *HLF* approach of converting the lattice into a tree coupled with a *k-limited* heuristic tries to avoid selecting nodes with small attribute sets for a given *k*.

In the third approach, *closest level first* (*CLF*), when traversing the lattice bottom-up, if a node *n* exists at two levels, we compare the difference in the number of attributes between *n* and the two parents of the node *n*, *n1* and *n2*, along each path. We assign node *n* to the path where the difference is the least.

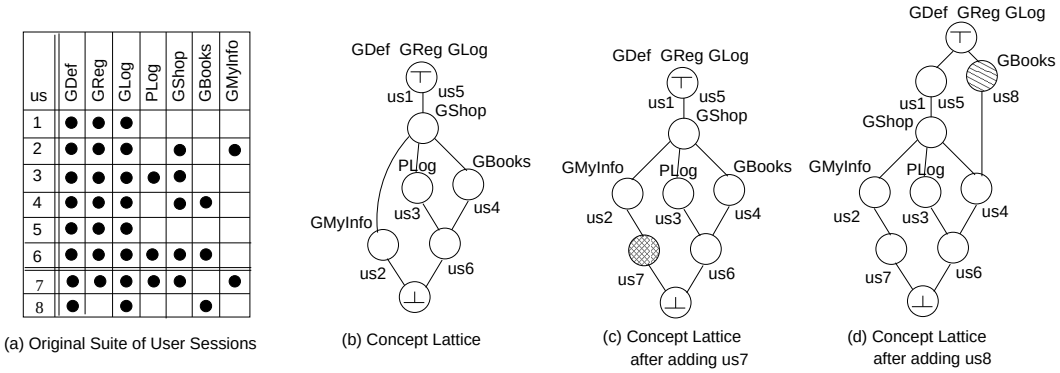
Because of the partial ordering in the lattice, a given node's position in the lattice heavily depends on the nodes preceding the current node. We believe that in cases of conflict, the level of a node can be identified better when associated with the node's predecessors (when traversing the lattice from the $\perp$) in the lattice. In Figure 3 (a), the difference in the attribute set size of *node 1* and *node 2* is less than the difference in attribute set size of *node 1* and *node 3*. Thus, *CLF* places node 1 at the *closest level first*, i.e., in level 2, as seen in Figure 3 (d). In this particular example, *CLF* created a tree similar to *LLF*, so the tree has the same disadvantages as *CLF*. However, we can create examples where the *CLF* would function similar to *HLF*, i.e., nodes with fewer attributes are higher in the lattice, and so are likely to be excluded when selecting test cases up to a certain level *k*.

Values for *k* are selected starting from $\perp$ at $k = 0$ to the node furthest from $\perp$. For $k = 1$, the reduced suite selected by all the approaches is the same as the reduced suite selected by the *test-all-exec-URLs* heuristic. Through our experimental studies we provide intuition on reasonable values of *k* and study the tradeoffs between increasing the number of levels represented in the reduced test suite versus effectiveness of the test suite.

For the example in Figure 2 (b), after converting the lattice into a tree by one of the three approaches described above, if we traverse the lattice bottom-up until $k = 2$ where $\perp$ is at level 0, the sessions selected by the *2-limited* heuristic are *us2, us6, us3, us4*.

D. Expected Cost-Benefits Tradeoffs

By definition, the reduced suites selected by the heuristics *one-per-node* and *k-limited* will contain more user sessions than the *test-all-exec-URLs* heuristic. Program coverage is expected to increase as the number of sessions included in the reduced suite increases. The relative fault detection capabilities of the reduced suites is not easy to predict. The costs of generating the reduced test suites are expected to be relatively small. The *k-limited* heuristics are likely to be more expensive than the other heuristics because they require processing the lattice prior to applying the heuristic. However, the lattice needs to be converted into a tree only once. The tree once created can be used for all subsequent values of *k*. In



(a) Original Suite of User Sessions (b) Concept Lattice (c) Concept Lattice after adding us7 (d) Concept Lattice after adding us8

Fig. 4. (a) Relation table and (b), (c), (d) Concept lattices for test suite reduction and incremental test suite update

Section VII we report on our evaluation of the heuristics in terms of the size of the reduced test suite, the program coverage obtained on executing the reduced suites, and the fault detection capabilities.

V. Incremental Concept Analysis

The key to enabling the generation of a reduced test suite with URL and use case 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' from scratch, and then incrementally update the reduced test suite T to T' with respect to L'.

Our incremental algorithm, shown in Figure 5, utilizes the incremental concept formation algorithm developed by Godin et al. [50]. 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 a specific level *k*, in the lattice may now be raised in the lattice, new nodes may have been created at the the level *k*, but existing internal nodes will never "sink" to the level *k* because the partial ordering of existing internal nodes with respect to the existing nodes at level *k* is unchanged by addition of new nodes [51]. These changes are the only ones that immediately affect the updating of the reduced test suite for all our heuristics. Thus, test cases are not maintained for internal nodes. Thus, all our three heuristics, *one-per-node*, *test-all-exec-URLs* and *k-limited*, require only the test cases from the old reduced test suite and the concept lattice for incremental test suite update. The design of the algorithm in Figure 5 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 reduced set after incremental lattice update.

While the old reduced suite may be small for *test-all-exec-URLs*, the other heuristics require storing a larger reduced suite. For the *one-per-node* heuristic, since the heuristic selects from every node in the lattice, maintaining the reduced suite entails maintaining one test case per node in the lattice i.e., the reduced suite before update. As a result, the space saved may not be considerable. For the *k-limited* heuristics, depending on the values of *k* a different size subset of test cases from the original suite is maintained.

A. Example showing reduced test suite update

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 4 (b). Consider

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 labeling $\perp$ = $\oslash$
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.

Fig. 5. Incremental reduced test suite update

the addition of the user session *us7*, which contains all URLs except *GBooks*. Figure 4 (c) 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 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. On applying the *test-all-exec-URLs* heuristic, the incremental update to the reduced test suite is a deletion of the user session *us2* and addition of user session *us7*. The updated reduced suite selected by the *k-limited* heuristic depends on the value of *k*.

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 4 (d) 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 nodes at the *next-to-bottom* level remained unchanged. Thus, the reduced test suite selected by the *test-all-exec-URLs* heuristic remains unchanged, and the new user session is not stored. The *one-per-node* heuristic, however, would include *us8* in the old reduced test suite. Note that our previous studies [47] 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*.

VI. 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 $2^m|O|$, where m is the upper bound on the number of attributes of any single object in O; however, this behavior is not expected in practice [50]. In our application of concept analysis, m is limited by the number of URLs that any given user session would request relating to the web application; since m 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 m. The relation table is not needed during 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(**$2^m|O|$**)**. For a fixed m the batch algorithm can execute in **O(**$|O|$**)**. 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 [50].

Metrics	Book	CPM	MASPLAS
Classes	11	75	9
Methods	319	173	22
Conditions	1720	1260	108
Non-commented LOC	7615	9401	999
Seeded Faults	40	135	29
Total Number of User Sessions	125	890	169
Total Number of URLs Requested	3640	12352	1107
Largest User Session in Number of URLs	160	585	69
Average User Session in Number of URLs	29	14	7

TABLE I
OBJECTS OF ANALYSIS

VII. EXPERIMENTS

A. Research Questions

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

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?

B. Independent and Dependent Variables

The *independent variables* of our study are the objects (user sessions) and attributes (URLs) input to concept analysis and the heuristic applied for test suite reduction. The *dependent variables* are the reduced suite size, program coverage and fault detection effectiveness of the original and reduced suites.

C. Objects of Analysis

Table I shows the characteristics of our three subject programs: an open-source, e-commerce bookstore Book [48], a course project manager (CPM) developed and first deployed at Duke University in 2001,

and a conference registration and paper submissions system (MASPLAS). Book allows users to register, login, browse for books, search for books by keyword, rate books, add books to a shopping cart, modify personal information, and logout. Book uses JSP for its front-end and MySQL database for the backend. To collect the 125 user sessions for Book, we sent email 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. Table I presents the characteristics of the collected user sessions.

In CPM, course instructors login and create *grader* accounts for teaching assistants. Instructors and teaching assistants set up *group* accounts for students, assign grades, and create schedules for demonstration time slots for students. CPM also sends email to notify users about account creation, grade postings, and changes to reserved time slots. Users interact with an HTML application interface generated by Java servlets and JSPs. CPM manages its state in a file-based datastore. We collected 890 user sessions from instructors, teaching assistants, and students using CPM during the 2004-05 academic year and the 2005 fall semester at the University of Delaware. The URLs in the user sessions mapped to the application's 60 servlet classes and to its HTML and JSP pages.

MASPLAS is a web application developed by one of the paper's authors for a regional workshop. Users could register for the workshop, upload abstracts and papers and view the schedule, proceedings, and other related information. Masplas displays front-end pages in HTML, back-end code is implemented in both Java/JSP, and a MySQL database is at the back-end. We collected 169 user sessions. The 8 unique URLs mapped back to 4 JSP's and 4 Java servlets.

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. In general, five types of faults were seeded in the applications—data store faults (faults that exercise 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 which change the way in

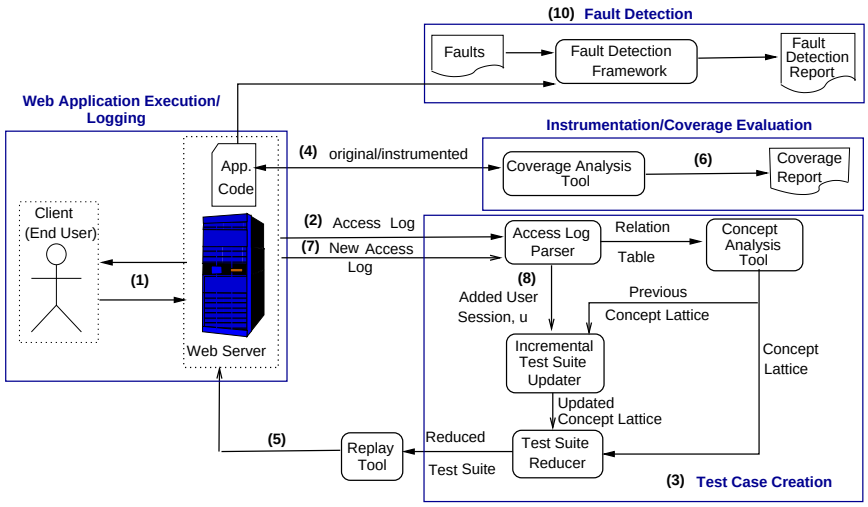


Fig. 6. Current Prototype Framework

which the user views the page), and link faults (faults that change the hyperlinks location).

D. Experimental Framework

Based on the framework in Figure 6, we constructed an automated system that enables the incremental generation of a reduced test suite of user sessions representing use cases for web-based applications. The reduced set of user sessions is automatically replayed and provided as input to a coverage analysis tool and an automated oracle to generate test coverage and fault detection reports, respectively. Following the labeled edges in Figure 6, the process begins with the collection of user-session data (Step (1)). We augmented the web server's access logging class to record both GET and POST requests. For each user request, we collect the originating IP address, time stamp, URL requested, 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 build an initial reduced test suite for the initial set of user-session data, while in Step (8) we incrementally update the reduced test suite as new user sessions (from the new access log in Step (7)) are processed. 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 [52], to construct a concept lattice[1]. The heuristic for identifying the reduced test suite from the lattice is embedded in the *Test Suite Reducer*.

[1]We need not create the lattice explicitly for reduction. The information required for reduction is derivable from the relation table. However, information about the concepts and the partial order are required for incremental test suite update.

In Step (4), the applications' Java source files are instrumented by the *Coverage Evaluator*, Clover [53], for test coverage analysis. Step (6) depicts the generation of the coverage report. With web applications written in JSP, the server typically creates a Java servlet from the JSP file. The Java servlet is instrumented and used by Clover to compute the coverage reports. Since the Java servlet created by the server contains a lot of server-dependent code that does not depend on the application code, we treat the server-dependent code as similar to libraries in traditional programs. To accurately report the coverage results, 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.

Step (5) is composed of the replay tool. We implemented a customized replay tool using HTTP-Client [54], which handles GET and POST requests, file upload and maintains the client's session state. Our replay tool takes as input the user sessions in the form of sequences of URL requests and name-value pairs and replays the sessions to the application, while maintaining the state of the user session and the application accurately. We adopt the `with_state` replay strategy during reduced suite replay. In `with_state` replay, the framework restores the database state of the system before each user session in the reduced suite is replayed. Our current implementation of `with_state` replays the original suite and records the data state after each session. When the corresponding session appears in the reduced suite, the database state is restored to the state before the current session. Further details on the replay techniques are presented in our previous work [15].

Steps (7), (8), and (9) represent the incremental update of the reduced test suite. We implemented the incremental concept analysis algorithm [50] in the *Incremental Test Suite Updater*. The updater takes as input a new user session and the lattice, to update the lattice. The lattice is input to the *Test Suite Reducer* to generate the reduced test suite.

In Step (10), we present the *Fault Detection Phase* of our framework. Faults are seeded in the application (one fault per version), and the original/reduced suites are replayed through the application. We compare the output (HTML pages) from the non-faulty or "clean" version of the application (expected output) to

the output from the faulty version of the application. We developed several oracle comparators to compare the expected and actual output [15]. In this paper, we present results for two oracle comparators, the **diff** oracle and the **structure** oracle. The **diff** oracle compares the structure and content of the HTML pages, and the **structure** oracle compares the structure of the output HTML pages, in terms of HTML tags.

E. Threats to Validity

The relatively small number of user sessions and the limited nature of the original suite of user sessions could be considered an *internal threat to validity*. In addition, our applications may not be complex enough to show large differences in program coverage and fault detection when comparing the reduction heuristics. Our first subject application Book is an open-source code, while CPM and MASPLAS were each developed and modified consistently by a single developer—thus threats that arise from different implementation styles are not a threat to internal validity of our study. Since we do not consider the severity of the faults, and evaluation of the reduced suites with respect to the severity of the faults, our experiments are liable to a *construct threat to validity*. Manually seeded faults may be more difficult to expose than naturally occurring faults [55]. Though we tried to model the seeded faults as closely as possible to naturally occurring faults—and even included naturally occurring faults from previous deployments of CPM and MASPLAS, some of the seeded faults may not be accurate representations of natural faults, resulting in an *external threat to validity*. One *conclusion threat to validity* is that we conducted our experiments on three applications, and generalizing our results to all web applications may not be fair.

F. Data and Analysis

1) Reduced Test Suite Size: Figures 7, 8, and 9 show the size of the reduced test suites for the different heuristics and our three subject applications. The x-axis (in the figures) represents the different heuristics. We abbreviate the heuristics as follows: 1-lim refers to the *test-all-exec-URLs*, 1-per refers to the *one-per-node* heuristic, k-cl, k-ll, and k-hl refer to the *k-limited* heuristic based on the *CLF*, *LLF* and the *HLF* approaches to converting the lattice into a tree, respectively, for different *k* values. We use the same x-axis

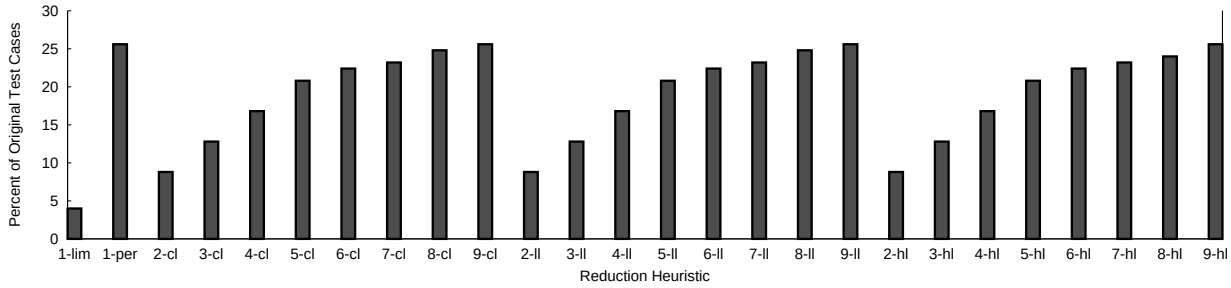


Fig. 7. Bookstore: Reduced Test Suite Size

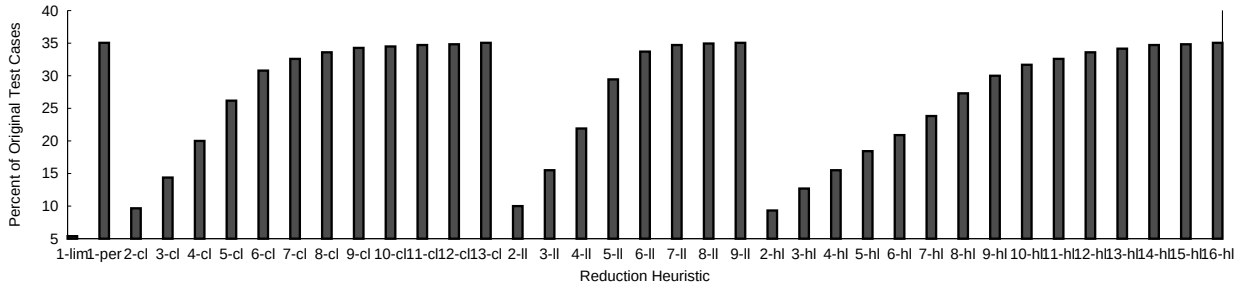


Fig. 8. CPM: Reduced Test Suite Size

labels for all the graphs in this paper. The y-axis represents the reduction in test suite size as a percent of the original test suite.

We observe that for all the subject applications, *test-all-exec-URLs* (1-lim in figures) selects the smallest reduced suite, and *one-per-node* heuristic selects the largest test suite. As the *k* value increases, the *k-limited* heuristic for each of *CLF, LLF,* and *HLF* select larger reduced suites. As expected, *one-per-node* provides least reduction in test suite size since the heuristic selects one user session from each concept node. The *k-limited* heuristics, for higher values of *k* perform similar to the *one-per-node* heuristic. Heuristic *test-all-exec-URLs* selects only from the *next-to-bottom* nodes and represents the least use cases among all the different heuristics. Thus, *test-all-exec-URLs* heuristic produces the smallest reduced test suite.

2) Program Coverage Effectiveness: Figures 10 and 11 present the results of program coverage effectiveness of the reduced suites for Book and CPM. The y-axis presents the number of statements that are covered by the original suite but lost by the reduced suite. We do not show results for MASPLAS because the reduced suite selected by the *test-all-exec-URLs* loses one statement covered by the original suite. All the other reduced suites cover all the statements covered by the original suite. Hence, we did

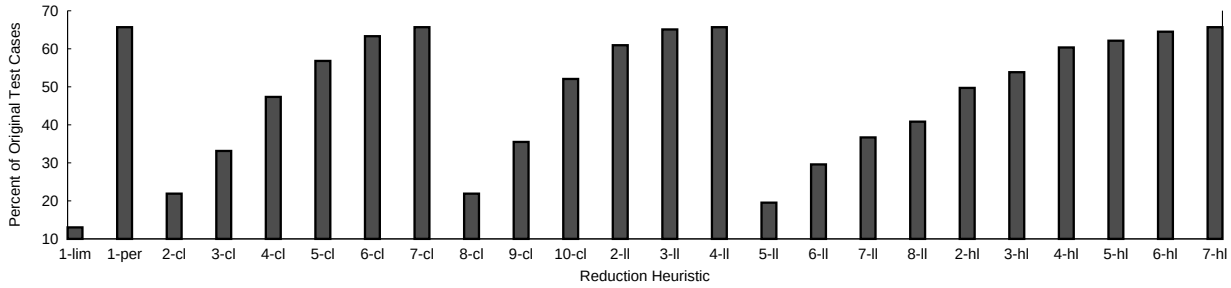


Fig. 9. MASPLAS: Reduced Test Suite Size

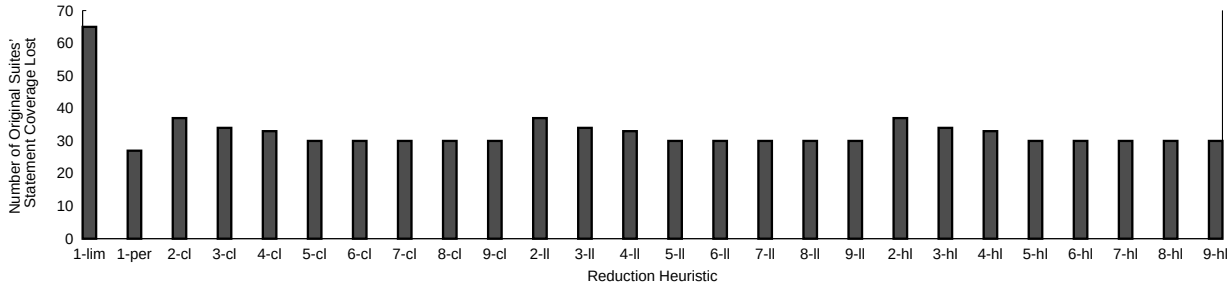


Fig. 10. Bookstore: Program Coverage Effectiveness

not observe any further improvement by using the other heuristics. The original suite for Book covered 56.99% of the statements, and the original suite for CPM covered 78.5% of the statements.

From Figures 10 and 11 we observe that as the reduced test suite size increases, the loss in program coverage decreases, as expected. We note that the program code coverage lost by the reduced suite selected by the heuristic *test-all-exec-URLs* is much larger than the the program code coverage loss by the reduced suites selected by the *2-limited* heuristics. In addition, we observe that there is a small difference in the program coverage effectiveness of the reduced suites generated by the *k-limited* heuristic with increasing values of *k* (especially for $k >$2 for Book and $k > 5$ for CPM).

In Figure 11, we observe that approximately 20 statements are not covered by any of the *k-limited* heuristics. On examining the code, we observed that certain sessions that cover these statements in the code were clustered together with sessions that did not cover this code. Since the clustering is based on the base URL (i.e., excludes name-value pairs and sequence of URLs when clustering), our clustering technique considered the session similar to other sessions in the cluster and the heuristic selected a session at random from the cluster. We believe that by changing the attribute to include more information, such

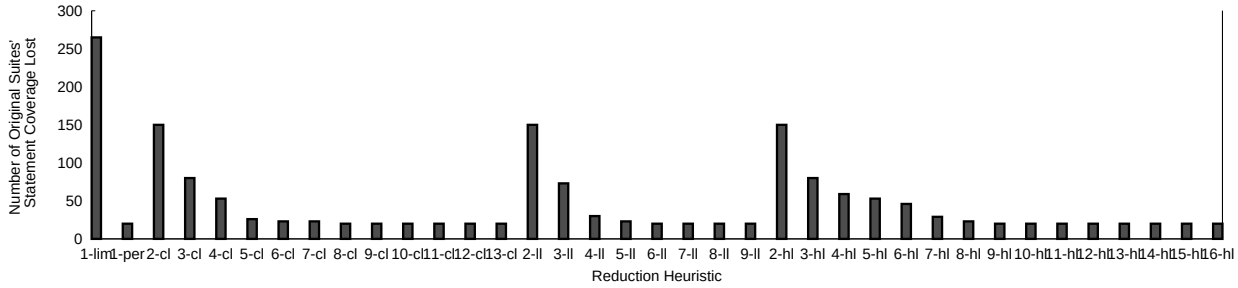


Fig. 11. CPM: Program Coverage Effectiveness

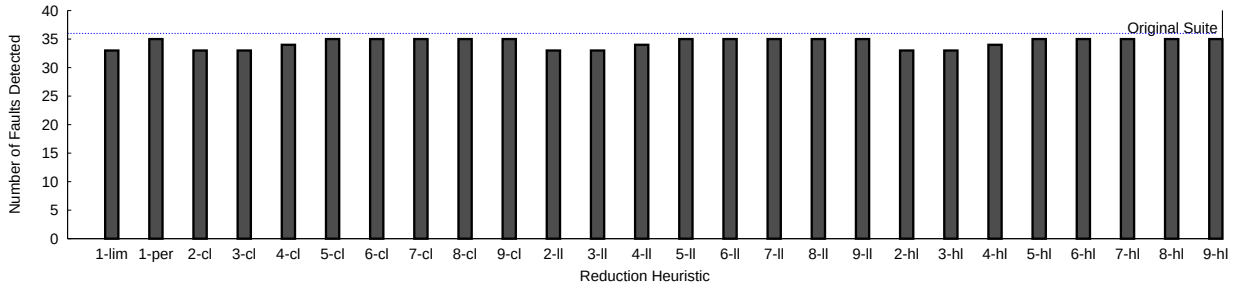


Fig. 12. Bookstore: Fault Detection Effectiveness with **Diff** Oracle

as sequences of URLs or the data traveling with the URL, we can cover the statements missed by the suite generated from the current attribute used for clustering.

3) Fault Detection Effectiveness: Figures 12 and 13 present the results of fault detection effectiveness of the reduced suites for Book with the **diff** and **struct** oracle. Figure 14 shows the results for fault detection effectiveness of CPM with the **struct** oracle. The y-axis presents the number of faults detected by the reduced suite. The horizontal line parallel to the x-axis shows the number of faults detected by the original suite. We do not show results for MASPLAS because the reduced suite selected by the *test-all-exec-URLs* detects all the faults detected by the original suite. Hence, we did not observe any further improvement by using the other heuristics. We believe the reduced suite selected by the *test-all-exec-URLs* heuristic detects all the faults for MASPLAS due to the simple nature and limited functionality in the application. The results of CPM with the **diff** oracle are not presented because the *1-lim* heuristic selected a reduced suite that detected all the faults detected by the original suite. Hence, the other heuristics did not provide any improvement in the fault detection capability. The total number of faults seeded in each application are presented in Table I.

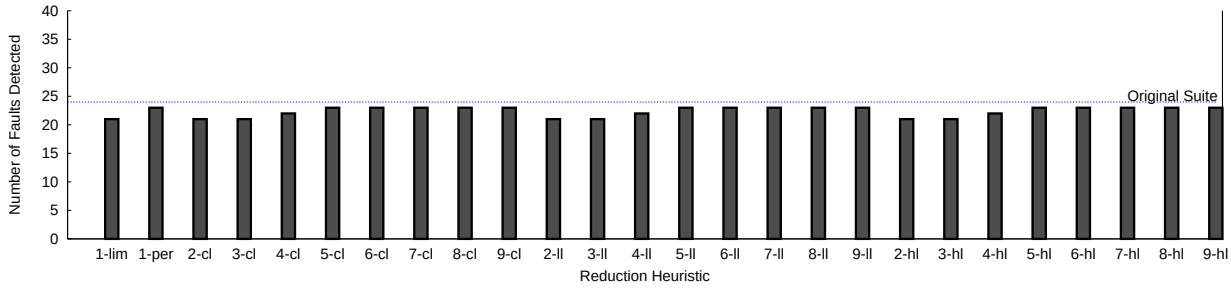


Fig. 13. Bookstore: Fault Detection Effectiveness with **Struct** Oracle

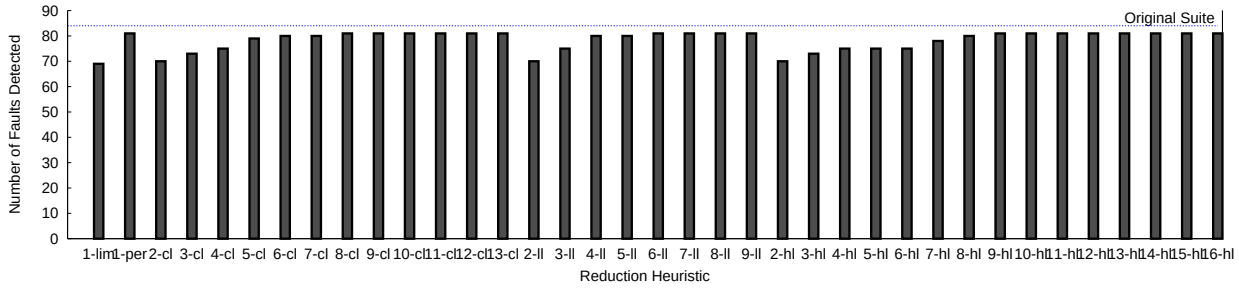


Fig. 14. CPM: Fault Detection Effectiveness with **Struct** Oracle

In Section IV, when evaluating the expected cost-benefit tradeoffs, we said we could not predict the fault detection effectiveness of the suite since the faults detected depends on (1) the number of sessions present in the reduced suite and (2) the actual session present in the reduced suite. As expected, different reduced suites show different fault detection effectiveness in our results. In Figures 12 and 13, we observe that the reduced suite generated by *test-all-exec-URLs* detects the least number of faults. As the value of *k* increases, the reduced suites selected by the *k-limited* heuristics detect more faults. However, for both the applications, some faults are missed by all the reduced suites.

We noted that one fault missed by all reduced suites in Book affects the server state of the system. During replay, we reset the database state to the time before the user session execution, but we ignored the server state. In previous work [15], we presented the challenges and reasoning for ignoring server state.

In CPM (Figure 14), we observed that certain sessions that detect the faults in the code were clustered together with sessions that failed to detect the fault. By selecting just one session from a cluster, we missed selecting the session that detects the fault. We noticed that often the fault was detected by a user

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

TABLE II
INCREMENTAL VERSUS BATCH

session that followed a unique sequence of URLs. Using an attribute such as sequences of URLs would ensure the session was clustered separately and hence selected for the reduced suite.

4) Incremental vs. 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 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` [52].

Table II presents the results of evaluating the effectiveness of incremental versus batch concept analysis for our three subject applications for the *test-all-exec-URLs* heuristic. The batch algorithm required the complete original set of user sessions to generate the reduced test suite. The incremental concept analysis algorithm requires only the reduced test suite and the lattice when generating an updated reduced test suite. We do not show the space requirements for the lattice in Table II, because the space required depends on the lattice implementation. For example, the sparse representation of the lattice contains all the information in the full lattice representation with much less space requirements. From Table II we note that for all the subject applications, space savings greater than 82% is obtained by using incremental concept analysis. A batch analysis of all the sessions yields the same reduced suite as the batch analysis of an initial subset of the sessions followed by an incremental analysis of the remaining sessions [51].

Thus, the incremental reduction process saves space costs considerably by not maintaining the original suite of user sessions. 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 the space by keeping a continually reduced suite.

5) Analysis Summary: From our results, we note that concept analysis-based reduction with the *test-all-exec-URLs* 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 various fault detection and program coverage effectiveness, as expected. Hence, a tester can choose an appropriate heuristic to obtain the desired suite characteristics. From our results, it also appears that the approach followed to convert the lattice to a tree has little impact on the effectiveness of the reduced suite: any of the three approaches could be followed prior to applying the *k-limited* heuristic.

Our results suggest that a tradeoff exists between test suite size and effectiveness. We observed that the *2-limited* heuristic selects a test suite that is slightly larger in size than the *test-all-exec-URLs* 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. Our results also suggest that using different attributes for clustering sessions could detect faults and cover code missed by clustering based on the current attribute, URL only. While *one-per-node* performs as effectively as the original suite in terms of program code coverage and fault detection for all the applications, the tradeoff between the reduced test suite size and the effectiveness of the suite needs to be considered. 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.

VIII. Conclusions and Future Work

By applying concept analysis to cluster user sessions and then 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. We presented three heuristics for test suite selection. Our experiments show that 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 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 test suite size that covers all URLs and maintains 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 URLs as attributes clusters sessions that uniquely identify a fault together with sessions that have similar URL coverage. The results motivate clustering by concept analysis with other attributes, such as URL with the data traveling with the URL and URL sequences.

To our knowledge, incremental approaches are not present for the existing requirements-based reduction techniques [12], [26], [27]. 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 [56], we found that for the domain of web applications, reduced suites based on reduction by URL 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 user sessions 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 evaluate concept analysis-based reduction with different attributes to target faults and code missed by the reduced suites generated from the current clustering technique.

REFERENCES

[1] "Web site test tools and site management tools," <http://www.softwareqatest.com/qatweb1.html>, 2006.

[2] "Cactus," <http://jakarta.apache.org/cactus/>, 2006.

[3] "Junit," <http://www.junit.org>, 2006.

[4] "HttpUnit," <http://httpunit.sourceforge.net/>, 2006.

[5] 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.

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

[7] F. Ricca and P. Tonella, "Analysis and testing of web applications," in *International Conference on Software Engineering*, 2001.

[8] A. Andrews, J. Offutt, and R. Alexander, "Testing Web Applications by Modeling with FSMs," *Software and Systems Modeling*, vol. 4, no. 3, August 2005.

[9] S. Elbaum, G. Rothermel, S. Karre, and M. F. II, "Leveraging user session data to support web application testing," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 187–202, May 2005.

[10] "Rational Robot." <http://www-306.ibm.com/software/awdtools/tester/robot/>, 2006.

[11] Parasoft WebKing, <http://www.parasoft.com>, 2004.

[12] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 3, pp. 270–285, July 1993.

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

[14] 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.

[15] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock, "Automated replay and failure detection for web applications," in *International Conference of Automated Software Engineering*. IEEE/ACM, November 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," in *Proceedings of the Automated Software Engineering Conference*, September 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, 2001.

[18] D. C. Kung, C.-H. Liu, and P. Hsia, "An object-oriented web test model for testing web applications," in *Proceedings of the First Asia-Pacific Conference on Quality Software*, 2000, pp. 111–120.

[19] J. Sant, A. Souter, and L. Greenwald, "An exploration of statistical models of automated test case generation," in *Proceedings of the Third International Workshop on Dynamic Analysis*, May 2005.

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

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

[22] C. Fu, B. Ryder, A. Milanova, and D. Wonnacott, "Testing of Java Web Services for Robustness," in *International Symposium on*

Software Testing and Analysis, 2004.

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

[24] M. Benedikt, J. Freire, and P. Godefroid, "VeriWeb: Automatically testing dynamic web sites," in *Proceedings of the Eleventh International Conference on World Wide Web*, May 2002.

[25] 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.

[26] 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.

[27] J. A. Jones and M. J. Harrold, "Test suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, March 2003.

[28] D. Jeffrey and N. Gupta, "Test suite reduction with selective redundancy," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*. IEEE, 2005.

[29] S. M. Master and A. Memon, "Call stack coverage for test suite reduction," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*. IEEE, 2005.

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

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

[32] M. Krone and G. Snelting, "On the inference of configuration structures from source code," in *International Conference on Software Engineering*, 1994.

[33] G. Snelting and F. Tip, "Reengineering class hierarchies using concept analysis," in *SIGSOFT Foundations on Software Engineering*, 1998.

[34] G. Ammons, D. Mandelin, and R. Bodik, "Debugging temporal specifications with concept analysis," in *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 2003.

[35] T. Kuipers and L. Moonen, "Types and concept analysis for legacy systems," in *International Workshop on Program Comprehension*, 2000.

[36] C. Lindig and G. Snelting, "Assessing modular structure of legacy code based on mathematical concept analysis," in *International Conference on Software Engineering*, 1997.

[37] P. Tonella, "Concept analysis for module restructuring," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 351–363, Apr 2001.

[38] M. Siff and T. Reps, "Identifying modules via concept analysis," in *International Conference on Software Maintenance*, 1997.

[39] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 210–224, Mar 2003.

[40] T. Ball, “The concept of dynamic analysis,” in *European Software Engineering Conference / SIGSOFT Foundations on Software Engineering*, 1999, pp. 216–234.

[41] S. Tallam and N. Gupta, “A concept analysis inspired greedy algorithm for test suite minimization,” in *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*, 2005.

[42] A. Podgurski, W. Masri, Y. McCleese, F. G. Wolff, and C. Yang, “Estimation of software reliability by stratified sampling,” *ACM Transactions on Software Engineering and Methodology*, vol. 8, no. 3, pp. 263–283, 1999.

[43] 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 held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press, 2001, pp. 246–255.

[44] C.-H. Lung, “Software architecture recovery and restructuring through clustering techniques,” in *Proceedings of the Third International Workshop on Software Architecture*, 1998, pp. 101–104.

[45] 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*, 2002, pp. 431–438.

[46] T. Wiggerts, “Using clustering algorithms in legacy systems remodularization,” in *Fourth Working Conference on Reverse Engineering (WCRE ’97)*, 1997.

[47] 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.

[48] “Open source web applications with source code,” <http://www.gotocode.com>, 2006.

[49] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. Souter, “Analyzing clusters of web application user sessions,” in *Proceedings of the Third International Workshop on Dynamic Analysis*, May 2005.

[50] 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.

[51] S. Sampath, “Cost-effective techniques for user-session-based testing of web applications,” Ph.D. dissertation, University of Delaware, 2006.

[52] C. Lindig, “Concepts tool,” <http://www.st.cs.uni-sb.de/ lindig/src/concepts.html>, 2006.

[53] “Clover: Code coverage tool for Java.” <http://www.cenqua.com/clover/>, 2006.

[54] “HTTPClient V0.3-3,” <http://www.innovation.ch/java/HTTPClient/>, 2006.

[55] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?” in *ICSE ’05: Proceedings of the 27th International Conference on Software Engineering*. New York, NY, USA: ACM Press, 2005, pp. 402–411.

[56] S. Sprenkle, S. Sampath, E. Gibson, L. Pollock, and A. Souter, “An empirical comparison of test suite reduction techniques for user-session-based testing of web applications,” in *International Conference on Software Maintenance (ICSM05)*. IEEE, September 2005.