An Interaction-Based Test Sequence Generation Approach for Testing Web Applications

Wenhua Wang[1], Sreedevi Sampath[2], Yu Lei[1], Raghu Kacker[3]

[1]*Dept. of Comp. Sci. & Eng.*
U. of Texas at Arlington
Arlington, Texas 76019
{wenhuawang, ylei}@uta.edu

[2]*Dept. of Information Systems*
U. of Maryland, Baltimore County
Baltimore, Maryland 21250
sampath@umbc.edu

[3]*Info. Technology Laboratory*
National Institute of Standards & Technology
Gaithersburg, Maryland 20899
raghu.kacker@nist.gov

Abstract

Web applications often use dynamic pages that interact with each other by accessing shared objects, e.g., session objects. Interactions between dynamic pages need to be carefully tested, as they may give rise to subtle faults that cannot be detected by testing individual pages in isolation. Since it is impractical to test all possible interactions, a trade-off must be made between test coverage (in terms of number of interactions covered in the tests) and test effort. In this paper, we present a test sequence generation approach to cover all pairwise interactions, i.e., interactions between any two pages. Intuitively, if a page P could reach another page P', there must exist a test sequence in which both P and P' are visited in the given order. We report a test sequence generation algorithm and two case studies in which test sequences are generated to achieve pairwise interaction coverage for two web applications. The empirical results indicate that our approach achieves good code coverage and is effective for detecting interaction faults in the subject applications.

1. Introduction

Web applications have proliferated in recent years, as more companies, government agencies and other organizations conduct business on the Internet. Along with this proliferation comes a growing amount of concern about the reliability of those applications. A failure, even partial functionality loss, may cause an entire business to standstill and cost millions of dollars [1, 2]. Also, users' confidence in web applications depends to a large degree on whether their business transactions are handled correctly by the application. Reliability is considered to be the biggest challenge in the further promotion of web applications [3].

Testing is an effective approach to ensuring the quality of web applications [4-6]. One important aspect of web applications is that they often consist of dynamic pages that interact with each other by accessing shared objects. For example, a web application may use session objects to track a user in a sequence of requests. As another example, different pages may access persistent data storage like a database to exchange data. Interactions between dynamic pages need to be carefully tested as they may give rise to subtle faults that cannot be detected by testing individual pages in isolation.

One approach to testing interactions between dynamic pages is to test all possible sequences in which those pages could be visited. The rationale is that each sequence represents a specific way in which those pages interact with each other, i.e., one possible interaction among those pages. There are, however, two problems with this approach. First, because the number of sequences grows exponentially as the size of a web application increases, it is nearly always impractical to test all possible sequences. Second, faults only manifest in a small number of sequences. Thus, it is inefficient to test all possible sequences, many of which do not contribute to fault detection.

In this paper, we propose a test sequence generation approach to address the above problems. We define the research problem as follows: *Given a web application containing dynamic pages, how could we generate, in a systematic manner, a small number of test sequences that are effective for detecting interaction faults*? The key idea of our approach is generating test sequences to cover all pairwise interactions, i.e., interactions between any two pages. In other words, if a page *P* could reach another page *P'*, there must exist one test sequence in which both *P* and *P'* are visited in the given order (but not necessarily in a row). Our approach involves three major steps: First, a graph

model is built to capture the navigation structure of the application under test, where each node represents a web page (or a portion of it), and each edge represents a direct link from one node to another. Second, all pairwise interactions that may occur in the web application are computed from the navigation graph. Finally, a set of paths is selected from the navigation graph to cover all those pairwise interactions. These paths are then used as test sequences to test the web application.

For the purpose of evaluation, we built a prototype tool and applied our approach to two web applications, namely, Book and CPM [5]. We compared our approach with an approach that generated test sequences to cover all the edges in a navigation graph. We refer to our approach as the *AllOrderedPairs* approach and the latter approach as the *AllEdges* approach. The empirical comparison shows that *AllOrderedPairs* approach can effectively detect some interaction faults that cannot be detected by the *AllEdges* approach.

Our approach is inspired by the pairwise testing approach for general software testing [7, 8]. Pairwise testing has been shown very effective for detecting general software faults. We believe that the notion of achieving pairwise coverage will also be effective for detecting faults in web applications, for which our case studies have provided some initial evidence. To the best of our knowledge, our work is the first attempt to apply combinatorial testing to generate test sequences for web applications. More importantly, our work deals with two unique challenges in the context of testing web applications. First, for all the existing approaches for combinatorial testing, the order in which different components appear in a combination is insignificant. This is in contrast to our approach, where the order of the pages being visited in an interaction is important. In particular, page *P* may reach page *P'*, but page *P'* may not reach page *P*. Second, existing work provides limited support for handling constraints. Constraints are used to exclude invalid combinations, based on the domain semantics, from the resulting test set. In our approach, the possible constraints among different pages are implicitly captured in a graph structure. The notion of using a graph to represent interaction constraints, as well as the required algorithmic support, is novel.

We point out that our approach currently does not address the problem of test data generation. That is, the test data required to execute a test sequence are assumed to be supplied manually or using other techniques like domain partitioning [9].

The remainder of this paper is organized as follows: Section 2 briefly surveys related work. Section 3 describes our test generation approach and presents an algorithm that implements the approach. Section 4 describes the prototype tool and reports two case studies. Section 5 concludes this paper and discusses future work.

2. Related work

Existing research most relevant to our work falls into two main categories: (1) work on testing web applications, which can be further classified into model-based testing and user-session-based testing, and (2) work on general combinatorial testing.

Model-based Testing. In model-based testing an abstract model of the application under test is built, and test sequences are generated from the model to satisfy some coverage goals. Existing model-based testing techniques for web applications extend traditional testing techniques, e.g., those based on control flow and/or data flow, to the web application domain [10, 11, 12, 14, 15]. In particular, Lucca et al. [14] proposed applying several coverage criteria presented by Binder [9] to test web browser interactions. One of those criteria is called *all-transition-k-tuples*, which aims to cover all possible sequences of transitions of size *k* consecutively. This criterion is different from our *AllOrderedPairs* criterion, since we do not require that the two pages in an ordered pair be connected by a direct edge in the graph model of the application. Similarly, the test sequences do not have to contain the pages in an ordered pair consecutively one after another.

Note that our interaction-based testing approach is also a model-based testing approach. The novelty of our work lies in the fact that we generate test sequences to achieve pairwise interaction coverage, which is a concept unexplored in the web testing domain.

User session-based testing. Several techniques have been developed that record real usage data and use the usage data to generate test sequences [6, 16, 17, 18, 19]. In particular, Sampath et al. [19] investigated the effectiveness of reducing test suites with a criterion designed to cover all page sequences of size 2 that occur in the original suite of logged user sessions. This study revealed that certain faults are detected only by the occurrence of a certain sequence of pages in the test case. Note that the approach of covering sequences of size 2 is different from the approach presented in this paper. In the *AllOrderedPairs* approach, we do not require that the two pages in an ordered pair appear consecutively one after another in the test sequence.

Compared with model-based testing, user session-based testing does not need to construct a model, which can be difficult for large and/or complex applications. However, the fault detection ability of user-session-based techniques depends to a large degree on the quality of collected user sessions [20]. In addition, user-session-based techniques require field deployment and extensive user participation, which significantly limits the applicability of those techniques.

General combinatorial testing. Combinatorial testing refers to a general test generation approach which creates tests by combining different parameter values, based on some effective combinatorial strategies. Existing work has mainly focused on how to generate a test set that is as small as possible while still satisfying some coverage goals [7, 8]. Empirical studies have shown that combinatorial testing can be very effective in detecting general software faults [21-23]. We note that Yuan et al. described a combinatorial approach to testing GUI applications in [23]. Their work, however, differs from ours significantly both in terms of the way they define their coverage criteria and the way they handle constraints. An excellent survey on the state-of-the-art of combinatorial testing can be found in [24].

As discussed in Section 1, we believe our work is the first attempt to apply combinatorial testing to the web application testing domain. Furthermore, our work deals with two unique challenges that are related to ordered interaction and constraint handling in the context of web application testing, which is considered to be the main technical contribution of this paper.

3. An interaction-based test sequence generation approach

In this section, we present our interaction-based test sequence generation algorithm and an example to illustrate the approach.

3.1. Basic concepts

We first introduce the notion of a navigation graph. In our approach, a navigation graph is used to represent the navigation structure of a web application. A node in a navigation graph can be a static node, which represents a static page, or a dynamic node, which represents a dynamic page or if a dynamic page has multiple forms, a form in the dynamic page. We distinguish the home page of the web application as a special node called home node. There exists an edge from one node *m* to another node *n* if node *n* can be visited immediately after node *m* through a direct link. Note that a direct link can be a hyperlink in a static page or an action in a dynamic page. Formally, a navigation graph *G* can be denoted as $G = (V, E, n_0)$, where $V = V^s \cup V^d$ with V^s being a set of static nodes, and V^d being a set of dynamic nodes, and $E \subseteq V \times V$ is a set of edges, and n_0 is the home node.

Note that a dynamic page could potentially generate an infinite number of page instances. This is because the content of such a page instance may depend on the user input, which could be potentially infinite. If these page instances were directly represented as individual nodes, the size of a navigation graph would be unbounded. This explains why it is the forms in a dynamic page that are directly represented in a navigation graph. Each form can be considered to represent a group of page instances that may be generated from the same form (with different user inputs). In our approach, a form is identified by the URL of the dynamic page containing the form and the names (but not values) of the input parameters the form can take.

Next, we introduce the notion of pairwise interaction coverage. The term "pairwise interaction" refers to interaction between two dynamic nodes. Let $G = (V, E, n_0)$ be a navigation graph. Formally, a pairwise interaction in *G* is an ordered pair (*m*, *n*), where *m* and *n* are two dynamic nodes, and there exists a path from *m* to *n* in *G*. Note that static nodes do not access shared objects and thus have no interaction with other nodes. (Static nodes are included in a navigation graph to capture the navigation structure, which is needed to generate executable test sequences.) Also note that the order of nodes in a pairwise interaction is significant, as a node *m* may reach a node *n*, but the reverse may not be true. (We only consider navigations through links within a web application. That is, we do not consider navigations due to actions that are performed on the web browser.)

Pairwise interaction coverage requires that a set of paths be selected from a navigation graph as test sequences so that every ordered pair is covered in at least one of those test sequences. Let $P = n_1 n_2 \ldots n_l$ be a path in a navigation graph. Let $p = (m, n)$ be an order pair. Then, *p* is covered in *P* if there exists $1 \le i < j \le l$ such that $n_i = m$, and $n_j = n$. Note that in *P*, nodes *m* and *n* must appear in the given order, but they do not need to appear consecutively.

To help better understand the notion of pairwise interaction coverage, let us compare it with edge coverage. The latter requires that every edge in a navigation graph be covered by at least one test sequence. Fig. 1 (a) shows an example navigation graph. Two test sequences, *ABDEG* and *ACDFG*, are

sufficient to cover all the edges in the graph. But these two test sequences fail to cover two pairwise interactions, namely, (*B*, *F*) and (*C*, *E*). If a fault is only triggered by these two interactions, then this fault would be detected by a test set satisfying pairwise interaction coverage, but may not be detected by a test set satisfying edge coverage.

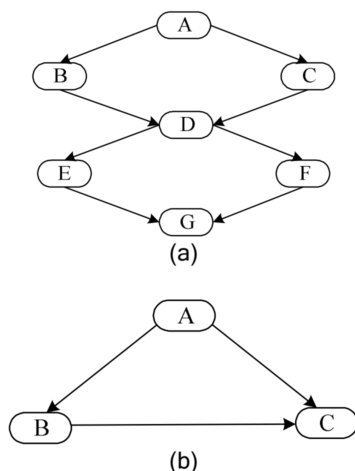

Figure 1. Two example navigation graphs

It is interesting to note that pairwise interaction coverage does not subsume edge coverage. Fig. 1 (b) shows a navigation graph that consists of three nodes *A*, *B*, and *C*, and three edges (*A*, *B*), (*A*, *C*), and (*B*, *C*). In this graph, path *P* = *ABC* satisfies pairwise interaction coverage but does not satisfy edge coverage. This is because (*A*, *C*) as a pairwise interaction is covered by path *P*, since *A* and *C* appear in *P* in the given order, but (*A*, *C*) as an edge is not covered by this path, since *A* and *C* do not appear in *P* in a row.

3.2. An interaction-based test generation algorithm

Fig. 2 shows an algorithm called *Generate-Sequences*, which implements the interaction-based test sequence generation approach. Algorithm *Generate-Sequences* takes as input a navigation graph *G* of the web application under test, and produces as output a set *seqs* of sequences that covers all the ordered pairs in *G*. The algorithm begins by computing all the set *pairs* of ordered pairs in the navigation graph (line 1). Note that we only consider ordered pairs involving two dynamic nodes. We point out that this computation basically requires determining reachability from one node to another, which is a classical problem in graph theory and can be solved using algorithms that have been reported in the literature [25].

Next, the algorithm tries to generate a set of sequences to cover all the ordered pairs computed earlier. This is accomplished by a *while* loop (lines 3 – 12) in which each iteration generates one sequence to cover a set of pairs that have not been covered before until all the pairs are covered. Each iteration of the *while* loop works as follows: First, a list *L* of nodes is built in which every two adjacent nodes is an ordered pair in set *pairs*, i.e., an ordered pair that has not been covered yet (line 4). The purpose of building this list is to guide the creation of a test sequence *S* so that *S* will cover a good number of ordered pairs that have not been covered yet. An optimal approach would build *L* in a way such that the size of the resulting set of test sequences is minimal. (The size of the resulting test sequence set can be measured in different ways, e.g., in terms of the total number of requests if we ignore the cost of test setup and teardown.) It is easy to see that finding an optimal solution is a NP-hard problem, due to the combinatorial nature of the problem. Here we describe a heuristic approach to building *L*. In this approach, we begin by picking an arbitrary pair (*m*, *n*) from *pairs*, and add *m* and *n* into *L* in the given order, i.e., *L* = (*m*, *n*). Next, we try to extend *L* using the following three rules: (1) if there is a pair (*m'*, *m*) in *pairs*, then we add *m'* into the beginning of *L*, i.e., *L* = (*m'*, *m*, *n*); (2) if there exists a pair (*n*, *n'*) in pairs, we add *n'* into the end of *L*, i.e., *L* = (*m*, *n*, *n'*); (3) if there are two pairs (*m*, *o*) and (*o*, *n*), we add *o* into the middle of *L*, i.e., *L* = (*m*, *o*, *n*). We will refer to the three rules as the front-end, back-end, and middle extension, respectively. These three rules can be easily generalized to keep extending *L* until *L* can no longer be extended, i.e., no more nodes can be added into *L*. We note that this approach is implemented in our prototype tool to conduct our empirical studies.

Now we are ready to discuss how to actually create a test sequence *S* out of *L*. This is done by first initializing *S* to be an empty sequence (line 5) and then appending to *S* a shortest path from the first node to the second node, and a shortest path from the second node to the third node, and so on (lines 6 – 9). In other words, *S* is created by adding into *L* a shortest path *P* between every two adjacent nodes to connect them. Note that node list *L* itself is not necessarily a path in navigation graph *G*, and thus cannot be directly used as a test sequence. This is because there may not exist an edge connecting every two adjacent nodes in *L*. Also note that *P* can always be found since *L* is built in a way that every two adjacent nodes, say n_i and n_{i+1},

where $1 \le i < k$, is an ordered pair, implying that there must exist at least one path from n_i to n_{i+1}.

After sequence S is created, the set *covered* of pairs that are covered by sequence S is computed (line 10) and then removed from set *pairs* (line 11). Note that the indices i and j in the computation of set *covered* do not have to be adjacent. This is because an ordered pair (m, n) is covered in a path if m and n appear in the path in the given order (not necessarily in a row).

We comment that the test sequences generated by algorithm *Generate-Sequences* do not necessarily start from the home node n_0. In practice, some applications may require that every test sequence start from the home node. For example, an application may require the user to log in before any other page is visited. In this case, if a sequence does not start from the home node, it is necessary to add into the beginning of the sequence a shortest path from the home node n_0 to the first node of the sequence.

Now we consider the time complexity of algorithm *Generate-Sequences*. Set *pairs* can be computed (line 1) using a classic algorithm like the Floyd-Warshall algorithm [25], which takes $O(|V|^3)$. Next we consider the time complexity of the *while* loop (lines 3 – 14). Assume that the heuristic approach described earlier is used to build the node list L (line 4). The application of each rule to extend L at a certain point (front end, middle, or back end) takes $O(|V|)$, as we only need to look up ordered pairs involving one or two nodes at the extension point. Since the length of L is $O(|V|)$, L can be built in $O(|V|^2)$. If we pre-compute and store a shortest path for every two nodes using an algorithm like the Floyd-Warshall algorithm, sequence S can be built in a time complexity that is linear to the length of S, which is $O(|V|)$. Note that the pre-computation of the shortest paths can be merged with the computation of set *pairs*, and thus does not incur additional time. The computation of set *covered* (line 10), together with the removal of set *covered* from *pairs* (line 11), takes $O(|V|^2)$. This means that the time complexity of each iteration is $O(|V|^2)$. Since the size of set *pairs* is $O(|V|^2)$, the time complexity of the entire *while* loop is $O(|V|^4)$. This derives that the time complexity of the entire algorithm is $O(|V|^4)$.

3.3. An example scenario

We demonstrate how algorithm *Generate-Sequences* works by using an example scenario from Book, one of the two applications used in our experiments. Fig. 3 shows a portion of the navigation graph for the Book application, where the Default node is the home node. For the ease of reference, each page is identified by a name, instead of its URL. We first generate all the ordered pairs in the navigation graph (line 1 in Fig. 2). Those pairs are shown in Table 1. Note that in the navigation graph, every node can reach itself (through other nodes). Therefore, there exists an ordered pair from each node to itself, e.g., D7 = (Default, Default), A7= (AdvSearch, AdvSearch), and so on.

Algorithm *Generate-Sequences*

Input: A navigation graph $G = (V, E, n_0)$ of the web application under test

Output: A set *seqs* of paths covering all the ordered pairs in G

1. $pairs = \{ (m, n) \mid m$ and n are dynamic nodes in G, and there exists a path from m to n in G }
2. let *seqs* be an empty set (of test sequences)
3. **while** (*pairs* is not empty) {
4. build a list $L = (n_1, n_2, ..., n_k)\}$ of nodes such that $k \le |V|$ and for $1 \le i < k$, $(n_i, n_{i+1}) \in pairs$
5. let S be an empty sequence (of nodes)
6. **for** ($1 \le i < k$) {
7. let P be a shortest path from n_i to n_{i+1}
8. $S = S \bullet P$
9. }
10. $covered = \{(n_i, n_j) \mid 1 \le i < j \le k, n_i, n_j \in L\}$
11. $pairs = pairs - covered$
12. add S into *seqs*
13. }
14. **return** *seqs*

Figure 2. Algorithm *Generate-Sequences*

Next, we try to generate test sequences to cover all the ordered pairs in Table 1 (lines 2 – 13 in Fig. 2). We first try to build a node list L (line 4), using the heuristic approach described in Section 3.2. Assume that we first pick D1 = (Default, AdvSearch), and add Default and AdvSearch into L (and remove D1 from Table 1):

L = {Default, AdvSearch}

Now we try to extend L using the three extension rules, i.e., the front-end, back-end, and middle extension. Without loss of generality, assume that we first apply back-end extension, where we try to find an ordered pair whose first node is the last node of L, i.e., AdvSearch. Note that A1 = (AdvSearch, Books) is one such pair. Thus, we add Books into the end of L (and remove A1, as well as D2, which is also covered by L, from Table 1):

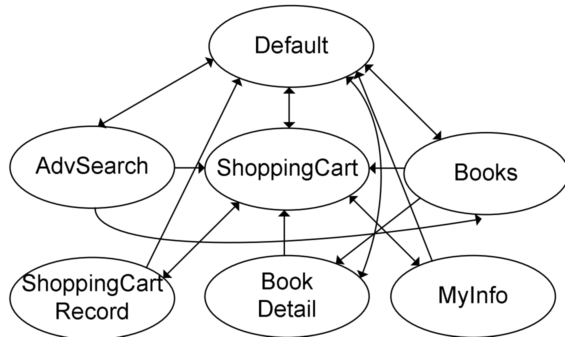

Figure 3. Navigation graph for Book

L = {Default, AdvSearch, Books}

Similarly, as B1 = (Books, BookDetail) is an ordered pair, we add BookDetail into the end of L (and remove B1, as well as D3 and A2, which are also covered by L, from Table 1):

L = {Default, AdvSearch, Books, BookDetail}

We keep applying back-end extension to L until we get the following sequence:

L = {Default, AdvSearch, Books, BookDetail, ShoppingCart, ShoppingCartRecord, Default, MyInfo, ShoppingCart, AdvSearch}

At this point, we cannot find any ordered pair whose first node is AdvSearch. Next we apply front-end extension, where we try to find an ordered pair whose second node is the first node of L. We find M3 = (MyInfo, Default) to be one such pair. Thus, we add MyInfo to the beginning of L (and remove M3, as well as M4, M5, M6 and M7, which are also covered by L, from Table 1):

L = {MyInfo, Default, AdvSearch, Books, BookDetail, ShoppingCart, ShoppingCartRecord, Default, MyInfo, ShoppingCart, AdvSearch}

Note that all the ordered pairs whose second node is MyInfo, namely, D6, A6, B5, BD4, S4, R2, and M7, are already covered in L, implying that those pairs have been removed from Table 1. Thus, at this point, we cannot find any ordered pair to extend L in the front end.

Next we try to apply middle extension. However, none of the remaining ordered pairs, i.e., B7, BD6, BD7, S6, S7, R5, and R6, satisfies the condition for middle extension. At this point, we finish building L.

Now we generate a test sequence out of L such that every two adjacent nodes in L are connected via a shortest path between the two nodes (lines 5 – 9 in Fig. 2). It turns out that most adjacent nodes in L have a direct edge between them, except for adjacent nodes Default and MyInfo, which can be connected by a shortest path (Default, ShoppingCart, MyInfo), and adjacent nodes ShoppingCart and AdvSearch, which can be connected by path (ShoppingCart, Default, AdvSearch). Thus, we generate the following test sequence:

S = {MyInfo, Default, AdvSearch, Books, BookDetail, ShoppingCart, ShoppingCartRecord, Default, ShoppingCart, MyInfo, ShoppingCart, Default, AdvSearch}

Table 1. The pairs set for Book

Pair	Default	Pair	AdvSearch Node
D1	Default, AdvSearch	A1	AdvSearch, Books
D2	Default, Books	A2	(AdvSearch, BookDetail)
D3	Default, BookDetail	A3	(AdvSearch, ShoppingCart)
D4	(Default, ShoppingCart)	A4	(AdvSearch, ShoppingCartRecord)
D5	(Default, ShoppingCartRecord)	A5	AdvSearch, Default
D6	Default, MyInfo	A6	AdvSearch, MyInfo
D7	Default, Default	A7	(AdvSearch, AdvSearch)
Pair	**Books**		**BookDetail**
B1	Books, BookDetail	BD1	(BookDetail, ShoppingCart)
B2	Books, ShoppingCart	BD2	(BookDetail, ShoppingCartRecord)
B3	(Books, ShoppingCartRecord)	BD3	BookDetail, Default
B4	Books, Default	BD4	BookDetail, MyInfo
B5	Books, MyInfo	BD5	(BookDetail, AdvSearch)
B6	Books, AdvSearch	BD6	(BookDetail, BookDetail)
B7	Books, Books	BD7	BookDetail, Books
Pair	**ShoppingCart**	**Pair**	**ShoppingCartRecord**
S1	(ShoppingCart, ShoppingCartRecord)	R1	(ShoppingCartRecord, Default)
S2	(ShoppingCart, Default)	R2	(ShoppingCartRecord, MyInfo)
S3	(ShoppingCart, MyInfo)	R3	(ShoppingCartRecord, ShoppingCart)
S4	(ShoppingCart, ShoppingCart)	R4	(ShoppingCartRecord, AdvSearch)
S5	(ShoppingCart, AdvSearch)	R5	(ShoppingCartRecord, Books)
S6	ShoppingCart, Books	R6	(ShoppingCartRecord, Books)
S7	(ShoppingCart, BookDetail)	R7	(ShoppingCartRecord, ShoppingCartRecord)
Pair	**MyInfo**		
M1	(MyInfo, ShoppingCart)		
M2	MyInfo, AdvSearch		
M3	MyInfo, Default		
M4	MyInfo, Books		
M5	MyInfo, BookDetail		
M6	(MyInfo, ShoppingCartRecord)		
M7	MyInfo, MyInfo		

Note that sequence S does not cover all the ordered pairs. For example, BD7 = (BookDetail, Books) is not yet covered. The same process can be repeated to generate additional test sequences until all the ordered pairs are covered, which is not explained for the purpose of brevity.

4. Experiments

Our experiments are designed to answer the following research questions.

1. How does the *AllOrderedPairs* test sequence generation approach compare with *AllEdges* test generation approach with respect to program coverage?
2. How does the *AllOrderedPairs* test sequence generation approach compare with *AllEdges* test generation approach with respect to fault detection effectiveness?

We measure the effectiveness of the two test generation strategies by measuring statement coverage and number of faults detected.

4.1. Experimental setup

Subject Applications: We used two applications, Book and CPM [5], in our experiments. Book is an online e-commerce application that users can use to browse, search and buy books [26]. CPM is a course project manager developed at Duke University. CPM allows course instructors to create grader accounts for teaching assistants. Instructors and teaching assistants can create student accounts, post student grades and post available time slots for students to demonstrate their course projects. Students can view their grades and sign up for specific demo time slots with a grader. New grader/student/course accounts can be created and deleted as necessary. More details on the applications are presented in previous work by Sampath et al. [5]. Table 2 presents the main characteristics of the two applications.

Navigation Graph and Test Case Characteristics: Table 2 also shows the characteristics of the navigation graph (number of nodes and number of edges of the navigation graph) and Table 3 shows characteristics of test cases for each application. From the last row of Table 3, we see that on average, the length of test cases generated by both *AllEdges* and *AllOrderedPairs* test generation strategies is the same (around 8 requests), except for Book's *AllOrderedPairs* test cases. The *AllOrderedPairs* test generation algorithm is designed to (a) cover as many ordered pairs as possible in a test sequence, and (b) find the shortest paths between two consecutive nodes in the test sequence. Thus, the length of a generated test case depends on these two factors above. Since Book's navigation graph has high connectivity—each node is connected to several other nodes, and since the algorithm is designed to find the shortest path between every two consecutive nodes, long test cases are created, however, the number of test cases is small—7 *AllOrderedPairs* test cases. Also, since *AllOrderedPairs* does not subsume *AllEdges* (as described in Section 3.1), we find that the *AllOrderedPairs* test cases for our subject applications do not necessarily cover all the edges in the graph (*AllOrderedPairs* test cases cover 60.7% of edges in Book, and 78.4% of edges in CPM).

Experimental Framework: We used the framework presented in Sprenkle et al. [6] for capturing program coverage and fault detection information. The framework has three main components: a customized tool for replaying the test cases, Clover [27] for instrumenting and measuring program coverage, and a fault detection component that allows insertion of hand-seeded faults into the application, application of oracles to determine if a test case detects a fault or not, and creation of fault detection reports based on the faults detected by a test suite [6]. We augmented faults seeded by Sampath et al. [5] with faults that are likely to occur when two pages interact with each other. Table 2 presents the number of seeded faults in each application.

For the fault detection study, we use both the *diff* and the *struct* oracle, presented by Sprenkle et al. [6, 28]. The *diff* oracle applies the Unix utility 'diff' on the HTML responses returned on executing the test cases on the clean and faulty versions of the application and reports any difference between the HTML responses as a failure. Since the *diff* oracle considers any difference in the HTML as a failure, differences in real-time content, e.g., current date, are flagged as a failure by the oracle, thus leading to false positives. The *struct* oracle is more conservative—it filters the HTML responses and reports only differences in the HTML tags. The obvious disadvantage of the *struct* oracle lies in its inability to capture faults that arise from differences in the content of the HTML page. Sprenkle et al. [28] discuss more about the oracles and the trade-offs. In this paper, we present results from both *diff* and *struct* oracles.

We implemented the *Generate-Sequences* algorithm in our prototype tool to generate test sequences. This tool generates test sequences that cover all pairwise interactions (*AllOrderedPairs*) and all edges

(*AllEdges*). After generating the test sequences, our tool also verifies whether all the pairwise interactions are covered by test sequences for covering all pairwise interactions and whether all the edges are covered by test sequences for covering all edges. It also shows us statistics of comparison between the *AllOrderedPairs* approach and *AllEdges* approach. Our test sequence generation algorithm generates only the base requests for the test cases and ensures that pairwise interactions are covered by the *AllOrderedPairs* test cases. To execute the test cases correctly, we manually augment the requests with name-value pairs. This is similar to how testers provide test input to ensure correct execution of test cases in traditional programs. We also use an initial data store state that is reset before each test case is executed, to avoid cascading faults.

4.2. Results and discussion

From Table 4, for Book application, we observe that both *AllEdges* approach and *AllOrderedPairs* approach have the same code coverage, 85.32%, but *AllOrderedPairs* detects 6 to 8 more faults than the *AllEdges* approach. By design, in Book, certain methods are included in every page of the application (through an include JSP statement), even though these methods are never called by the other methods in the page—these methods are designed to be called by a user with different privileges (an admin user), instead of an end-user. In this paper, since our tool focused on covering pairwise interactions of end-user accessible pages and functions, we report program coverage results for Book after removing such repeated code from the coverage report generated by Clover. The program coverage is same for *AllOrderedPairs* and *AllEdges* because the same pages are accessed with the same parameters and values, thus resulting in same code coverage. That means sequences for *AllEdges* approach and *AllOrderedPairs* approach should have the same detection ability for faults in unit testing. The primary advantage of the *AllOrderedPairs* test cases is that they can guarantee pairwise interaction coverage, while the *AllEdges* approach cannot.

From our experiments, we observe that all the faults detected by the *AllEdges* test cases are also detected by the *AllOrderedPairs* test cases. One example of a fault that is caught by *AllOrderedPairs* but missed by *AllEdges* is presented here: we found that a fault is exposed when the Login page is accessed the second time in a test sequence. Since (Login, Login) was an ordered pair for Book, the ordered pair (Login, Login) appeared in one of the *AllOrderedPairs* test sequences, and the fault was detected by the *AllOrderedPairs* test case. However, since there was no direct edge from Login to Login, the *AllEdges* test cases were not required to generate a test sequence with two occurrences of the Login page in them, thus failing to detect the fault.

Table 2. Characteristics of subject applications

	Book	CPM
Technologies	JSP, MySQL	Java servlets, File-based data store, HTML
Non-commented LOC	7615	9401
Number of classes	11	75
Number of Methods	319	173
Number of Seeded Faults	72	197
Number of Nodes	41	64
Number of Edges	63	125

Table 3. Characteristics of test cases

	AllEdges		AllOrderedPairs	
	Book	CPM	Book	CPM
Number of test cases	15	41	7	261
Percent of ordered pairs covered	53.12%	15.03%	100%	100%
Percent of edges covered	100%	100%	60.7%	78.4%
Total number of requests	133	330	154	2273
Longest test case length	20	25	40	105
Shortest test case length	4	3	12	3
Average Test Case length	8.87	8.05	22	8.71

Table 4. Book: Effectiveness Metrics

	All Edges	All Ordered Pairs
Total Faults	72	72
Detected Faults	*Diff* oracle: 61 *Struct* oracle:45	*Diff* oracle: 69 *Struct* oracle:51
Statement Coverage	85.32%	85.32%

Table 5. CPM: Effectiveness Metrics

	All Edges	All Ordered Pairs
Total Faults	197	197
Detected Faults	*Diff* oracle: 37 *Struct* oracle: 28	*Diff* oracle: 124 *Struct* oracle: 49
Statement coverage	62.8%	67.5%

Table 4 presents the program coverage and fault detection results for CPM. Test sequences from the *AllOrderedPairs* approach have higher program coverage than sequences from *AllEdges* approach. However, we see a large difference between the numbers of faults detected by each approach. From our experiments, we observe that all the faults detected by the *AllEdges* test cases are also detected by the *AllOrderedPairs* test cases. Fault detection by the *AllOrderedPairs* test cases improved by a factor of 2.57 over the *AllEdges* test cases. From Table 2, we see that the *AllEdges* test cases for CPM cover only 15.03% of the pairwise interactions, whereas the *AllOrderedPairs* test cases cover 100% of the interactions. We also found that test cases that cover the most ordered pairs (2273 and 1523 ordered pairs), detect the most faults in the application (35 and 33 faults, respectively). Thus, we observe that there is a relation between the number of ordered pairs covered in a test case and the number of faults it detects.

Also, CPM application is more complex in logic than the Book. There are many pairwise interactions through data storage. For example, the pairwise interaction (CreateCourseServlet, CatchGroupSignupServlet) is not covered by the *AllEdges* test cases. If a fault exists in storing the course name of a new course in the CreateCourseServlet page, the *AllEdges* test cases will fail to detect it. But, since the interaction is present in the *AllOrderedPairs* test cases, such a fault can be detected by test cases that contain the pairwise interaction (CreateCourseServlet, CatchGroupSignupServlet).

Another reason for the difference is because of the complex logic in sequences for *AllOrderedPairs* test cases and the name-value pairs supplied to the test cases. For example, the request for creating a grader may just occur once in a test case generated by the *AllEdges* approach. But in a test case created by the *AllOrderedPairs* approach, the same request may occur multiple times because we are trying to cover all pairwise interactions. Thus, when an existing grader is created again by a request that appears later in the test case, the error tolerance code will be covered because that grader already exists. This is also another reason for the higher code coverage for *AllOrderedPairs* approach when compared to the *AllEdges* approach.

A disadvantage of the *AllOrderedPairs* test cases is that there are more *AllOrderedPairs* test cases (261) than *AllEdges* (41) test cases—thus, the *AllOrderedPairs* test cases take longer to execute and require more resources. However, we believe the trade-off in improved fault detection effectiveness is worth the increased test execution time. Also, it is important to note that the fault detection of the *AllOrderedPairs* test cases is still only 36.5% of the total seeded faults (with the struct oracle). But, this is expected because our test generation algorithm only generates the base requests—the name-value pairs to the request are still manually supplied. The particular name-value pairs used in the test cases have a significant impact on code coverage and fault detection. In the future, we plan to implement strategies to systematically generate name-value pairs for web application requests.

4.3. Threats to validity

One important threat to validity of our results is that we conducted our experiments on only two subject programs. Though the subject applications are fairly large-sized programs, we cannot generalize our results to all web applications. We also manually generated parameter-values to the requests generated by our tool—the effectiveness of the test case largely depends on the parameter-values used in the test case. In the future, we will closely investigate the problem of test input generation for web applications. The techniques were tested on applications with hand-seeded faults. Also, we do not report on the time to generate test cases and the time to execute the test cases for fault detection effectiveness—these are measures that we plan to evaluate in the future.

5. Conclusion

In this paper, we presented a new test sequence generation approach, called *AllOrderedPairs*, for web applications. This approach tries to test all pairwise interactions in a web application. Our experimental results indicate that for applications that involve complex interactions between dynamic pages, the *AllOrderedPairs* approach can be significantly more effective than the *AllEdges* approach.

There are a number of venues to continue our work. First, at present, a navigation graph is built manually, which can be time consuming, especially for complex web applications. We plan to develop an approach to automatically or semi-automatically explore the navigation structure of a web application. Second, we plan to address the problem of test data generation. In particular, we want to explore if it is possible to apply the notion of combinatorial testing as well. Finally, we want to conduct more case studies to thoroughly evaluate the effectiveness of our approach. The automated or semi-automated approaches to building

navigation graphs and generating test data will allow us to study more complex applications.

Acknowledgements. We would like to thank the University of Delaware Software Testing Research Group and their collaborators at Drexel University for creating the fault seeded versions and for sharing with us the original and fault seeded versions of the Book and CPM web applications. This work is partly supported by a grant (Award No. 60NANB6D6192) from the Information Technology Lab (ITL) of National Institute of Standards and Technology (NIST).

Disclaimer. Any mention of commercial products within this article is for information only; it does not imply recommendation or endorsement by NIST.

6. References

[1] Web Application Development–Bridging the Gap between QA and Development. http://www.stickyminds.com.
[2] S. Pertet, and P. Narsimhan, "Causes of Failures in Web Applications", CMU-PDL-05-109, Carnegie Mellon University, 2005.
[3] A. Stout, "Testing a Website: Best Practices", The Revere Group, 2001.
[4] G. Di Lucca, A. Fasolino, F. Faralli, and U.D. Carlini, "Testing Web Applications", In *18th IEEE International Conference on Software Maintenance*, pp. 310-319, 2002.
[5] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A.S. Greenwald, "Applying Concept Analysis to User-session-based Testing of Web Applications", IEEE Transactions on Software Engineering, 33, (10), pp. 643-658, 2007.
[6] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock, "Automated Replay and Failure Detection for Web Applications", In *20th International Conference of Automated Software Engineering*, pp. 253-262, 2005.
[7] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton, "The Combinatorial Design approach to Automatic Test Generation", IEEE Software, 13, (5), pp. 83-88, 1996.
[8] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOG-D: Efficient Test Generation for Multi-way Combinatorial Testing", Software Testing, Verification and Reliability, 2007.
[9] R. Binder, "Testing Object-Oriented Systems" (Addison Wesley. 2000).
[10] C.-H. Liu, D.C. Kung, P. Hsia, and C.-T. Hsu, "Structural Testing of Web Applications", In *11th International Symposium on Software Reliability Engineering*, pp. 84-96, 2000.
[11] Y. Qi, D. Kung, and E. Wong, "An Agent-based Testing Approach for Web Applications", In *29th Annual International Computer Software and Applications Conference*, pp. 45-50, 2005.
[12] F. Ricca, and P. Tonella, "Analysis and Testing of Web Applications", In *23rd International Conference on Software Engineering*, pp. 25-34, 2001-05, 2001.
[13] P. Tonella, and F. Ricca, "A 2-layer Model for the White-box Testing of Web Applications", In *6th IEEE International Workshop on Web Site Evolution*, pp. 11-19, 2004.
[14] G.A. Di Lucca, and M.D. Penta, "Considering Browser Interaction in Web Application Testing", In *5th International Workshop on Web Site Evolution*, pp. 74-81, 2003.
[15] A. Andrews, J. Offutt, and R. Alexander, "Testing Web Applications by Modeling with FSMs", Software and Systems Modeling, 4, (3), pp. 326-345, 2005.
[16] S. Elbaum, G. Rothermel, S. Karre, and M.F. II, "Leveraging User Session Data to Support Web Application Testing", IEEE Transactions on Software Engineering, 31, (3), pp. 187-202, 2005.
[17] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock, "A Case Study of Automatically Creating Test Suites from Web Application Field Data", In *Workshop on Testing, Analysis, and Verification of Web Services and Applications*, pp. 1-9, 2006.
[18] S. Sprenkle, S. Sampath, E. Gibson, L. Pollock, A. Souter, "An Empirical Comparison of Test Suite Reduction Techniques for User-session-based Testing of Web Applications", In *21st International Conference on Software Maintenance*, pp. 587-596, 2005.
[19] S. Sampath, S. Sprenkle, E. Gibson, and L. Pollock, "Web Application Testing with Customized Test Requirements—An Experimental Comparison Study", In *17th International Symposium on Software Reliability Engineering*, pp. 266-278, 2006.
[20] S. Elbaum, S. Karre, and G. Rothermel, "Improving Web Application Testing with User Session Data", In *25th International Conference on Software Engineering*, pp. 49-59, 2003.
[21] D.R. Kuhn, and M.J. Reilly, "An Investigation of the Applicability of Design of Experiments to Software Testing", In *27th NASE/IEEE Software Engineering Workshop*, pp. 91-95, 2002.
[22] D.R. Kuhn, D.R. Wallace, and A.M. Gallo, "Software Fault Interactions and Implications for Software Testing", IEEE Transactions on Software Engineering, 30, (6), pp. 418-421, 2004.
[23] X. Yuan, M. Cohen, and A.M. Memon, "Covering Array Sampling of Input Event Sequences for Automated GUI Testing", In *22nd IEEE/ACM international conference on Automated Software Engineering*, pp. 405-408, 2007.
[24] M. Grindal, J. Offutt, and S.F. Andler, "Combination Testing Strategies: A Survey", Journal of Software Testing, Verification and Reliability, 15, (2), pp. 97-133, 2005.
[25] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, "Introduction to Algorithms" (The MIT Press, 2001, Second edn.), pp. 629-632.
[26] Open Source Web Applications with Source Code. http://www.gotocode.com, 2006, last access.
[27] Clover: Code Coverage Tool for Java. http://www.cenqua.com/clover/, 2006, last access.
[28] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott, "Automated Oracle Comparators for Testing Web Applications", In *18th IEEE International Symposium on Software Reliability*, pp. 117-126, 2007.