Information and Software Technology 54 (2012) 724–738

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Improving the effectiveness of test suite reduction for user-session-based testing of web applications

Sreedevi Sampath [a,*], Renée C. Bryce [b]

[a] Department of Information Systems, University of Maryland Baltimore County, Baltimore, MD 21250, United States
[b] Department of Computer Science, Utah State University, Logan, UT 84322, United States

ARTICLE INFO

Article history:
Received 6 October 2011
Received in revised form 20 January 2012
Accepted 21 January 2012
Available online 1 February 2012

Keywords:
Test suite prioritization
Test suite reduction
Ordering reduced suites
Web application testing
User-session-based testing

ABSTRACT

Context: Test suite reduction is the problem of creating and executing a set of test cases that are smaller in size but equivalent in effectiveness to an original test suite. However, reduced suites can still be large and executing all the tests in a reduced test suite can be time consuming.
Objective: We propose ordering the tests in a reduced suite to increase its rate of fault detection. The ordered reduced test suite can be executed in time constrained situations, where, even if test execution is stopped early, the best test cases from the reduced suite will already be executed.
Method: In this paper, we present several approaches to order reduced test suites using experimentally verified prioritization criteria for the domain of web applications. We conduct an empirical study with three subject applications and user-session-based test cases to demonstrate how ordered reduced test suites often make a practical contribution. To enable comparison between test suites of different sizes, we develop *Mod_APFD_C*, a modification of the traditional prioritization effectiveness measure.
Results: We find that by ordering the reduced suites, we create test suites that are more effective than unordered reduced suites. In each of our subject applications, there is at least one ordered reduced suite that outperforms the best unordered reduced suite and the best prioritized original suite.
Conclusions: Our results show that when a tester does not have enough time to execute the entire reduced suite, executing an ordered reduced suite often improves the rate of fault detection. By coupling the underlying system's characteristics with observations from our study on the criteria that produce the best ordered reduced suites, a tester can order their reduced test suites to obtain increased testing effectiveness.

1. Introduction

Test suite *reduction* and *prioritization* are two different approaches to manage test suites during regression testing. Test suite reduction strategies select a smaller test suite from a larger original suite, such that the reduced suite satisfies all of the requirements that the original suite satisfies. Test suite prioritization modifies the order of test cases within a test suite with the goal of improving the rate of fault detection. These test selection methods are mainly focused on giving the tester the best test cases for execution early on, so that under limited time and cost constraints, increased effectiveness in testing may be achieved. Most of the existing literature studies the two test selection methods disjointly. While reduction techniques generate a smaller set of tests than the original suite, even a reduced test suite can be so large that it cannot be executed completely under time constraints. We propose that the effectiveness of reduced test suites can be further improved by *ordering the reduced set of test cases* and show how such an ordering may be performed. Such an ordering would be beneficial to a tester who when faced with limited time and resources can still complete the testing process.

Only recently have researchers begun to merge reduction and prioritization techniques. Bertolino et al. [2] revisit test suite reduction and prioritization and motivate the need for ordering a reduced suite. They acknowledge that testers can be under pressure and testing could be stopped before all of the tests in a reduced test suite are run, making the ordering important. They advocate that a reduction technique should choose test cases such that the tests meet the coverage of test requirements, and in an order where test cases that are the most effective with respect to fault detection are favored. To this end, they evaluate the effectiveness of reduced suites using a rate of fault detection measure. They apply well known reduction heuristics, on four subject applications and use APFD [27] to measure the effectiveness of reduced test suites. However, they do not propose *how* a reduced suite can be ordered. Further, while the APFD metric works in their paper

* Principal corresponding author. Tel.: +1 410 455 8845; fax: +1 410 455 1073.
E-mail addresses: sampath@umbc.edu (S. Sampath), Renee.Bryce@usu.edu (R.C. Bryce).

0950-5849/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2012.01.007

because they compare test suites of the same sizes, we raise the issue that the APFD metric cannot compare the rate of fault detection for test suites that are different sizes, as is needed for comparing full and reduced test suites.

In this paper, we investigate this idea of ordering a reduced suite for the domain of web applications. In particular, we conduct our study in the domain of user-session-based testing of web systems, where usage logs are converted into test cases and used to test the system. For a frequently used system, web usage logs can be large, thus resulting in a large set of test cases. In previous work, Sampath et al. [31] explored several reduction criteria to create reduced user-session-based test suites. However, some of these reduced test suites were large, e.g., in CPM, one of the subject applications studied by Sampath et al., the reduced suite produced by *name_value* contained close to 50% of the tests from the original test suite. Executing all the tests in the reduced suite can be time consuming. Thus, there is a clear need for ordering the reduced set of test cases so faults can be found early in the test execution cycle. We first apply experimentally verified reduction criteria to reduce an original test suite and obtain a reduced test set. Then, we apply a different set of experimentally verified prioritization criteria to order the reduced test set. Thus, the final ordered reduced test set is created by applying multiple criteria.

Since we are applying multiple criteria, at some level, our work is similar to the work on hybrid criteria that has been proposed previously [4,16,18,19]. Our approach differs from these works, because they focus on creating an optimal reduced or minimized test suite without regard to the order of the test cases in the reduced set.

Furthermore, in previous work, Sampath et al. [31] noted that certain reduction criteria produce effective reduced suites when evaluated using traditional effectiveness measures, such as the number of faults detected and the reduction in number of tests. Reduced suites that are effective under these measures may perform differently when the rate of fault detection and certain costs are considered. Since the goal with an ordered reduced suite is to find fault *quickly*, a measure such as the rate of fault detection should be considered. Factors such as test generation time and test execution time are also important to a tester who is interested in performing testing with minimum time overhead. We measure the cost-effectiveness of the test suites using a metric that accounts for the time taken to create the reduced/prioritized suite, the time taken to execute the reduced/prioritized suite, and the rate of fault detection.

The specific contributions of this paper are:

1. a model that reduces and orders a test set by creating 40 new hybrid test selection criteria,
2. a new metric called *Mod_APFD_C* that allows us to compare test suites of different sizes and that incorporates a cost measure—test generation time,
3. an empirical comparison of the reduced, ordered test suites when compared to the unordered reduction and full prioritization,
4. guidance to web application testers on factors to consider when selecting a test selection technique which technique would be best suited based on the results of our empirical study.

In the remainder of this paper, Section 2 reviews previous work on test suite reduction and prioritization. We also provide an overview of user-session-based testing for web applications since we use this domain to demonstrate our approach that reduces and then prioritizes test suites. Section 3 reviews the extensive set of reduction and prioritization criteria that we use in our experiments, including 5 reduction criteria, 8 prioritization criteria, and 40 combined criteria. In Section 4 we present our experimental setup and the *Mod_APFD_C* metric. Section 5 presents our results and discussion, and we conclude in Section 6.

2. Background and related work

2.1. Test suite reduction

Test suite reduction is a test suite management method where a smaller set of test cases are selected from a larger original suite while maintaining the requirement coverage of the original suite. Harrold et al. [16] formally define test suite reduction as follows:

> *Given* a test suite *TS*, and a set of test case requirements $r_1, r_2, \ldots, r_n$, that must be satisfied to provide the desired testing coverage of the program, and subsets of *TS*, $T_1, T_2, \ldots, T_n$, one associated with each of the r_i's such that any one of the test cases t_j belonging to T_l can be used to test r_i, *find* a representative set of test cases from *TS* that satisfies all of the r_i's.

Test suite reduction has been critically examined by several researchers [4,16,18–22,24,32,36]. These studies typically propose reduction techniques or criteria (e.g., based on code coverage) and evaluate the effectiveness of the reduced test suite. They find that reduction is not only practical, but also quite effective in terms of fault finding effectiveness. On the other hand, some studies raise serious implications in terms of the fault finding effectiveness of reduced test suites. For instance, Heimdahl and George [17] use several coverage criteria (i.e., transition coverage, decision coverage, MC/DC, etc.) to significantly reduce the sizes of test suites, but they also suffer from a decrease in fault finding effectiveness. They encourage readers to be cautious when using structural coverage to reduce a test suite because it may lower the quality of a test suite. Alternatively, they suggest that prioritization may be a more promising alternative to reduction.

Multiple test selection criteria models have also been proposed [4,16,18,19,21,37,38] to address these shortcomings. While Harrold et al. [16] and Jeffrey and Gupta [19] use multiple coverage criteria in series, Black et al. [4], Hsu and Orso [18] and Yoo and Harman [37,38] propose the use of multi-objective functions, where the objective function satisfies a coverage criterion,increases fault detection, or decreases cost. However, none of these approaches address the problem of ordering a reduced test set to improve its effectiveness.

In this paper, we do not seek to provide another validation of whether a particular reduction technique or criterion is effective, but rather we use techniques and criteria that have already been shown to be useful and seek to extend their effectiveness in regression testing by ordering reduced test suites. To our knowledge, the only other work that advocates ordering a reduced suite is by Bertolino et al. [2] where they propose measuring the rate of fault detection with APFD. We will shortly define APFD and show that we extend the measurement beyond test suites of identical sizes.

2.2. Test case prioritization

Test case prioritization is the process of scheduling the execution of test cases according to some *criterion* to satisfy a *performance goal*. Consider the function for test prioritization as formally defined by Rothermel et al. [27].

> Given *T*, a test suite, Π, the set of all test suites obtained by permuting the tests of *T*, and *f*, a function from Π to the set of real numbers, the problem is to find $\pi \in \Pi$ such that $\forall \pi' \in \Pi$, $f(\pi) \geqslant f(\pi')$. In this definition, Π refers to the possible prioritized orderings of *T* and *f* is a function that is applied to evaluate the orderings. The selection of the function *f* leads to many criteria to prioritize software tests.

Engstrom et al. [13] present a systematic review of test prioritization methods that exist in the literature. For instance, prioritization criteria may consider code coverage, fault likelihood, and fault

exposure potential [9,27]. Rothermel et al. [25,27] examine several prioritization techniques and in all of their case studies, with the exception of one, the prioritization techniques detect faults more efficiently than untreated or randomly ordered test suites. Elbaum et al. [9] provide one of the largest sets of empirical studies for test suite prioritization which applies prioritization techniques to eight programs that each have multiple versions. They observe that prioritization is worthwhile, again motivating future work on the topic. Elbaum et al. also examine five prioritization techniques applied to eight programs and identify that the performance of test case prioritization techniques vary based on characteristics of the programs and test suites [11]. Though there exist studies that empirically compare test suite reduction and test suite prioritization [7,15,26], the concept of ordering a reduced test suite and evaluating the effectiveness of the ordered reduced test suite has not been studied before.

Rothermel et al. [27] evaluate the effectiveness of the test suites using the Average Percentage of faults Detected (APFD) metric, which is a measure of the rate of fault detection. AFPD for a test suite is the area under the curve that is plotted between test suite size and percentage of faults detected. The APFD metric fails to capture information such as cost of test case and severity of faults in the evaluation of the test suite's effectiveness. To address this shortcoming, Elbaum et al. [10] introduce the $APFD_C$ metric that incorporates the costs of test cases and faults. We review the $APFD_C$ metric in detail in Section 4.

Finally, in our previous work we developed criteria to prioritize Event-Driven Systems, including GUI and web-based applications [5,29]. Bryce et al. [5] provided a single model to prioritize GUI and web test cases building on the criteria proposed by Sampath et al. [29]. Ten event-based prioritization criteria are applied to four GUI and three web-based applications. All of the prioritization criteria were based on data that was extracted from the test cases (i.e., number of events, number of inter-window event interactions, number of unique windows visited, and most frequently invoked sequences of events). These experiments show that selecting test cases that invoke the most events or cover the most inter-window interactions usually provided the best rate of fault detection. In addition, criteria that use "frequently accessed sequences" often worked quite well when tests did not have a high Fault Detection Density (FDD), where FDD is a measure of the number of faults that each test identifies on average [29].

2.3. Web applications and user-session-based testing

In user session-based testing, testers extract user-sessions from their web server logs and use them as test cases for future regression testing [12,32]. Fig. 1 shows an example of the user-session-based testing process as described by Sampath et al. [32] and Sprenkle et al. [33]. A web application is accessible to users through the Internet. Each HTTP POST and GET request that the user requests is saved in a log file. A parser converts the web logs into test cases by using IP addresses, cookies, and time stamps to identify the POST and GET request from each user during their session. In addition to cookies, a time out interval of 45 min of inactivity is also used to start new test cases. A user-session-based test case is a sequence of user requests in the form of base requests and parameter name-value pairs (e.g., form field data). A base request for a web application is the request type and resource location without associated data (e.g., GET /servlets/authentication/HelloWorld.java). After the user-sessions are converted to a suite of test cases, test cases are executed by a replay tool. An oracle then compares the output of the test cases to the expected results and records a fault detection report that indicates whether each test passes or fails. Elbaum et al. [8] show that user-session based testing is quite effective when compared to white box techniques, but these techniques augment each other as they find different faults.

For a frequently used application, an organization may have a large number of users and correspondingly, large numbers of user sessions. If a large pool of test cases have accumulated, testers may not be able to run all test cases and may have to select a smaller effective test set, i.e., reduce the test suite. Further, even if they can run all test cases, they would prefer to execute certain test cases earlier in the testing process if they are more likely to find faults than other test cases, i.e., order the reduced test set.

Sampath et al. [32] reduce test cases using a concept-analysis based reduction technique, ensuring that all base requests are covered and that the use case representation is maintained, where a use case is viewed as a set of base requests. They further presented and evaluated several other test requirements for test suite reduction, that are event-based and black-box in nature, i.e., the execution of the web application is not required to create a mapping between the test cases and the requirements [31]. Though several requirements/criteria exist for reducing test suites, in this work, we select black-box test requirements that are cost-effective when compared with traditional program-coverage based requirements [35].

Sprenkle et al. [35] presented an empirical comparison of two reduction techniques—concept-analysis-based reduction [32] and Harrold, Gupta and Soffa's reduction technique [16] and discuss the tradeoffs. Harrold et al.'s reduction technique first maps each requirement to a set of test cases that cover the requirement. Then, the requirement with the least cardinality test set, i.e., requirement covered by only one test case, is considered and the test case is added to the reduced test suite. As each test is selected, requirements in whose test sets the selected test exists are marked as covered. The process repeats until all the requirements are covered by the reduced test suite. Sprenkle et al. found that though the test suites from concept analysis-based reduction were larger in size, they covered more use cases of the web system. The concept-analysis based reduced test suite is as efficient, and in some cases better, than reduced test suite from the Harrold et al.'s technique with respect to program coverage and fault detection. Therefore, we decided to use only one of the reduction techniques in this work—the concept analysis-based reduction technique. We describe concept-analysis based reduction and the requirements next.

3. Test suite management techniques

Test suite reduction and test suite prioritization are both test suite management techniques. In this section, we review the techniques for our experiments including (1) reduction, (2) prioritization, and (3) criteria to order the reduced suites.

3.1. Test reduction technique and test requirements

Concept analysis is a clustering technique that clusters objects that share common attributes [3]. In previous work, Sampath et al. [32] exploited concept analysis to cluster user-session-based test cases and then apply heuristics to select from these clusters to create a reduced suite. We refer the reader to [32] for details of the concept-analysis-based reduction technique; however, here we present an overview of the different reduction requirements, the selection heuristic and the reduction criterion that we use in our study.

Sampath et al. [31,32] propose several test requirements that we use in this study, namely, *base*, *seq*, *name*, *seq_name*, *name_value*. The reduction criterion for any of the requirements can be stated as:

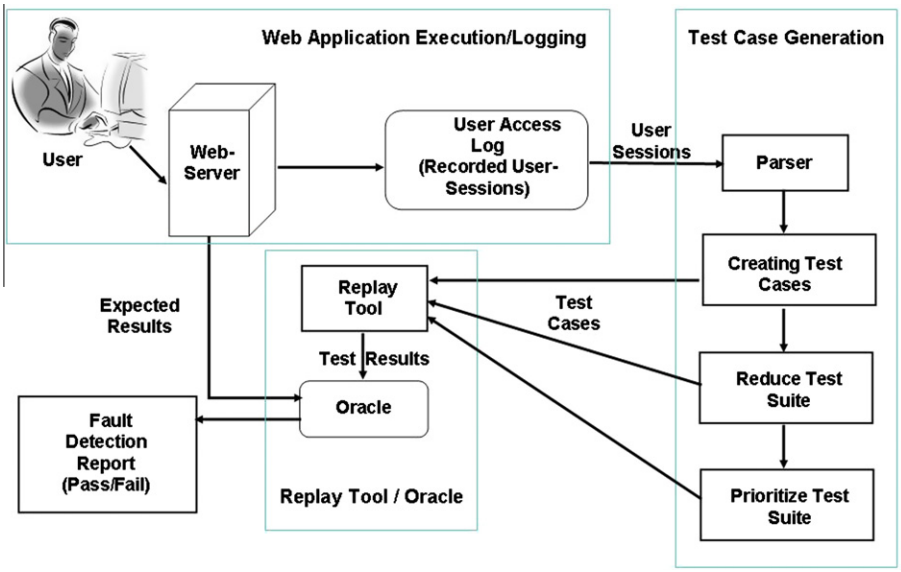


Fig. 1. User-session-based testing process with options for reduction and prioritization.

Table 1
Reduction test requirements.

Original test case	*base*	*seq*
books.jsp?book_type="java" book_record.jsp?book_author="deitel" buy.jsp?item_id="145"	books.jsp book_record.jsp buy.jsp	⟨books.jsp, book_record.jsp⟩ ⟨book_record.jsp, buy.jsp⟩
	name	*name_value*
	books.jsp, book_type book_record.jsp, book_author buy.jsp, item_id	books.jsp?book_type="java" book_record.jsp?book_author="deitel" buy.jsp?item_id="145"
	seq_name	
	⟨book.jsp,book_type, book_record.jsp, book_author⟩ ⟨book_record.jsp,book_author, buy.jsp,item_id⟩	

For every test reduction requirement r that occurs in the original suite T, there is at least one test case $t \in T$ in the reduced suite that covers r.

The heuristic used to select test cases from the clusters created by concept analysis is called the *test-all-exec-requests* heuristic [32]. As per this heuristic, test cases are selected from the different concept analysis clusters such that the resulting reduced test suite is guaranteed to cover all of the test requirements that the original test suite covers, while maintaining different use cases of the application, where a use case is specified as a set of base requests.

We briefly summarize the requirements here:

- *base*: requirement is of the form of a base request,
- *seq*: requirement is of the form of base request sequences of size 2,
- *name*: requirement is of the form of base request and parameter name,
- *name_value*: requirement is of the form of base request, parameter name and parameter value,
- *seq_name*: requirement is of the form of size 2 sequences of base request and parameter name.

Table 1 shows how the test cases are viewed from the perspective of the different test requirements. In Table 1, the first column represents a test case in its original form, a sequence of three requests where *book.jsp, book_record.jsp*, and *buy.jsp* are the base requests, the strings *book_type, book_author, item_id* are the parameter names associated with each base request, and the strings *java, deitel*, and *145* are the values associated with the parameters. (We truncate the URL to remove the server name. For instance, http://www.usu.edu/book.jsp is referred to as "book.jsp" here.) From Table 1, we see that the requirement for reduction by *base* specifies that the reduction requirement is a base request, and the test case is viewed just as a sequence of base requests (without parameter names and values). Similarly, for the *seq* requirement, the reduction requirement is a consecutive sequence of two base requests, so the original test case is viewed as two sequences <*books.jsp, book_record.jsp*> and <*book_record.jsp*, *buy.jsp*>, where the angle brackets denote a sequence. Thus, the test requirements satisfied by the test case in Table 1 for the *seq* requirement are these two sequences. Similarly, the *name*, *name_value*, and *seq_name* requirements are shown in Table 1.

Since each requirement considers different information during reduction, the reduced suites generated by each technique will be of a different size. The *base* requirement creates the smallest reduced suite, since it considers the least information from the test case during reduction, whereas, the other requirements consider incrementally larger pieces of information from the test case (i.e., parameter names, names and values) for reduction, thus creating larger test suites.

We illustrate the working of the reduction requirements with the example test suite shown in Table 2 and concept-analysis-based test suite reduction. We simplify the representation of base requests and parameter-names and values for ease of illustration.

In the test suite shown in Table 2, there are three test cases, *tc1, tc2, tc3* that access the base requests denoted by the letters A, B, C and D. The parameter names associated with a base request are named u_i and the values associated with the parameter-names are named v_i. To explain the abbreviations used in Table 2 in the context of the test case shown in Table 1, consider the first request in *tc1* from Table 2 and the first request in the original test case from Table 1. The *A* in *tc1* would correspond to the base request *book.jsp*, the parameter *u1* would correspond to the parameter *book_type* and the value *v1* would correspond to the value *java* (note that test case *tc1* is not the exact same test case shown in Table 1, we compare the two test cases only to explain the abbreviations).

For the reduction requirement, *base*, the test cases are viewed in terms of their base URLs. A requirement mapping as shown in rows 1 through 3 of Table 3 is created for the test suite. Recall that the *test-all-exec-requests* of concept analysis-based test suite reduction selects test cases that cover all the requirements covered by the original suite, while maintaining different use cases of the application, where a use case is the set of base requests covered by the test case. Concept analysis-based reduction does not produce the minimal test suite because it captures use cases in addition to satisfying the requirements [32]. In this case, even though *tc2* covers all the base requests, *tc1* and *tc3* capture a different use case of the application, since the set of base requests covered by *tc1* (A, B, C) and *tc3* (A, C, D) is different from the set of base requests covered by *tc2* (A, B, C, D), thus *tc1* and *tc3* are also selected for the reduced suite. Thus, based on the *test-all-exec-requests* heuristic, the test cases selected to satisfy the requirement *base* are *tc1, tc2, tc3*.

For the *seq* requirement, the test cases are viewed as shown in rows 4 to 9 of Table 3. The reduced suite will contain the test cases *tc2* and *tc3*, because the use case and requirements covered by *tc1*, i.e., ($\langle A,B\rangle$, $\langle B,C\rangle$), are also covered by *tc2*. Test case *tc3* covers the requirement $\langle A,C\rangle$ that is not covered by *tc2* and is thus selected for the reduced suite. For the *name* requirement, the requirement mapping is created as shown in rows 10 to 15 of Table 3. Each test case captures a different use case of the application, and thus the reduced suite contains *tc1, tc2, tc3*.

For the *seq_name* requirement, whose mapping is shown in rows 16 to 21 of Table 3, the reduced suite will contain all the three test cases, *tc1, tc2* and *tc3*, because *tc1* satisfies two requirements not covered by the other two test cases, and similarly, *tc3* satisfies two requirements not covered by the other two test cases. We do not show the requirement mapping for the *name_value* requirement, because the mapping will be the same as the original test case. For the *name_value* requirement all the test cases *tc1, tc2, tc3* will be selected for the reduced suite because no two test cases in this test suite are identical in terms of their base requests, parameter-names and values or use cases. In the remainder of this paper, we use the terms reduction requirement and reduction technique interchangeably.

Table 2
Example test suite.

Test cases		
tc1	*tc2*	*tc3*
A, u_1,v_1	A, u_1,v_5	A, u_1,v_1
B, u_2,v_2	A, u_6,v_6	C, u_3,v_3,u_4,v_{10}
C, u_3,v_3,u_4,v_4	B, u_2,u_7	D, u_5,v_{11}
	B, u_2,v_8	
	C, u_3,v_9,u_4,v_{10}	
	D, u_5,v_{11}	

Table 3
Requirement Mapping.

Row no.		A	B	C	D
base requirement					
1	*tc1*	*	*	*	
2	*tc2*	*	*	*	*
3	*tc3*	*		*	*
base requirement reduced suite: *tc1, tc2, tc3*					
seq requirement					
		$\langle A,B\rangle$	$\langle B,C\rangle$	$\langle C,D\rangle$	$\langle A,A\rangle$
4	*tc1*	*	*		
5	*tc2*	*	*	*	*
6	*tc3*			*	*
		$\langle B,B\rangle$	$\langle A,C\rangle$		
7	*tc1*				
8	*tc2*	*			
9	*tc3*		*		
seq requirement reduced suite: *tc2, tc3*					
name requirement					
		A u_1	B u_2	C u_3,u_4	D u_5
10	*tc1*	*	*	*	
11	*tc2*	*	*	*	*
12	*tc3*	*		*	*
		A u_6			
13	*tc1*				
14	*tc2*	*			
15	*tc3*				
name requirement reduced suite: *tc1, tc2, tc3*					
seq_name requirement					
		$\langle$A u_1, B $u_2\rangle$	$\langle$B u_2, C u_3,$u_4\rangle$	$\langle$A u_1, A $u_6\rangle$	$\langle$A u_6, B $u_2\rangle$
16	*tc1*	*	*	*	
17	*tc2*			*	*
18	*tc3*				
		$\langle$B u_2, B $u_2\rangle$	$\langle$B u_2, C u_3,$u_4\rangle$	$\langle$C u_3, u_4, D $u_5\rangle$	$\langle$A u_1, C u_3,$u_4\rangle$
19	*tc1*				
20	*tc2*	*	*	*	
21	*tc3*			*	*
seq_name requirement reduced suite: *tc1, tc2, tc3*					

3.2. Test prioritization criteria

Though several test prioritization criteria exist, the criteria that we use in this work are tailored and shown effective for event-driven software, and web applications in particular. In this section, we describe the criteria that we use next and illustrate the traces of these prioritization criteria with the example test suite from Table 2. In all of our prioritization criteria, whenever two or more test cases satisfy a prioritization criterion equally, the tie is broken at random.

3.2.1. Count-based criteria

This set of criteria prioritize test cases based on the number of HTTP requests or the number of parameter-values in the test. The criterion *Req-LtoS* (Requests-Long_to_Short) orders tests by the number of HTTP requests in the test in decreasing order. For the example test suite in Table 2, the prioritized test order by *Req-LtoS* is *tc2, tc1, tc3*. In this case, *tc1* and *tc3* are randomly ordered, because they cover the same number of requests (three). The criterion *Req-StoL* (Requests-Short_to_Long) orders test cases by the number of HTTP requests in ascending order. In the example test suite, the prioritized test order is *tc1, tc3, tc2*. The criterion *PV-LtoS*

(Parameter-values-Long_to_Short) orders tests in descending order of the number of parameter-values in the test. The criterion *PV-StoL* (Parameter-values-Short_to_Long) orders tests in ascending order of the number of parameter-values in the tests. In the example test suite the prioritized order by *PV-LtoS* is *tc2, tc1, tc3*. For *PV-StoL*, the prioritized test order is *tc3, tc1, tc2*.

3.2.2. Frequency-based criteria

The next set of criteria are based on the frequency of occurrence of certain window sequences in the test. The criterion *MFAS* (Most Frequently Accessed Sequence), we first identify the most frequently present sequence of HTTP requests in the test suite, and then order test cases in decreasing order of occurrence of this sequence in the test.

For the example test suite, first a frequency of access table is constructed that maintains a count of the number of times every consecutive sequence of base requests appears in the test suite. We consider consecutive sequences of size 2 in this work (larger sequences are an area of future work). For the example test suite in Table 2, the frequency of access table is shown in the first two columns of Table 4. Using the first two columns in the table, the *MFAS* criterion, selects test cases based on the occurrence of the most frequently accessed sequence in a test case. In this example, the sequences $\langle A,B\rangle$, $\langle B,C\rangle$, $\langle C,D\rangle$ occur the same number of times, therefore, one of them is selected at random and used by the criterion. Assume $\langle A,B\rangle$ is selected as the most frequently accessed sequence in the test suite. Then, test cases are ordered based on the number of times the sequence $\langle A,B\rangle$ appears in them. For the example test suite, the prioritized test order is *tc1, tc2, tc3*. Since both *tc1* and *tc2* have the sequence $\langle A,B\rangle$ the same number of times, the tie is broken at random.

The *AAS* (All Accessed Sequence) criterion uses the frequency of access of all sequences to order the test suite. For each sequence of request, starting with the most frequently accessed sequence, test cases that have maximum occurrences of these sequences are selected for execution before other test cases in the test suite. In Table 4, the third column lists the test cases that have maximum occurrence of the sequence in the corresponding first column. For example, both *tc1* and *tc2* have the sequence $\langle A,B\rangle$ once in them, and this is the maximum number of times the sequence $\langle A,B\rangle$ appears in the test suite. Based on this information, the prioritized test order selected by *AAS* is *tc1, tc2, tc3*.

3.2.3. Combinatorial-based criteria

In this set of criteria, the focus is on systematic coverage of interaction between parameter values. In *1-way*, the next test that covers the most uncovered parameter-values is selected. In the example test suite, *tc2* covers 7 unique parameter-values and *tc1* and *tc3* cover 4. The prioritized test order by *1-way* for the test suite in Table 2 is *tc2, tc1, tc3*.

In *2-way*, the next test that covers the most uncovered 2-way interactions between windows is selected. In the example test suite, for the sake of illustration, we assign unique IDs to each unique parameter-value combination in the test cases in Table 2. This is shown in Table 5. Table 6 then shows the pairwise combinations of parameter-values between windows. Test case, *tc2* is selected first as it covers 17 unique pairs. Notice that the pair (9,10) is in bold because it is covered in both test cases *tc2* and *tc3*. Since *tc2* was chosen as the first test case for the prioritized test suite and it covered this pair, test case *tc3* now only offers 5 pairs that have not been covered in the previously selected test case. Therefore, *tc1* is chosen second as it covers 6 pairs that were not covered in the previously selected test case. Test case *tc3* is chosen last and it covers 4 previously uncovered pairs. For the example test suite in Table 2, the prioritized test order by *2-way* is *tc2, tc1, tc3*.

Table 4
Frequency of access for example test suite.

Sequence	Frequency	Test cases with max. occurrences of sequence
⟨A,B⟩	2	*tc1*, *tc2*
⟨B,C⟩	2	*tc1*, *tc2*
⟨C,D⟩	2	*tc2*, *tc3*
⟨A,A⟩	1	*tc2*
⟨B,B⟩	1	*tc2*
⟨A,C⟩	1	*tc3*

3.2.4. Random

We also use a random permutation on the tests as a control in our experiments.

3.2.5. Logged order

The original logged ordering is derived from timestamps of the user-sessions and serves as a second control.

3.3. Criteria to order reduced suites

While reduction techniques create a small, highly effective reduced suite [31,32], a reduction technique traditionally does not specify any ordering mechanisms for the reduced test cases. Our main motivation behind ordering reduced suites is to determine if we can further improve the effectiveness of the reduced suites by ordering them using the prioritization criteria described in the previous section. Such an ordering could be useful when working with very large original test suites that will create large reduced suites. Since time to execute the tests is always a concern during the testing process, an ordering of the reduced set of test cases provides testers an effective mechanism by which they can execute their reduced test sets under limited time situations.

Sampath et al. [31] studied the effect of test suite reduction on four subject applications and found that *name* and *base* are the most effective reduction requirements. In addition, they state that suites reduced by a requirement such as *seq* found faults that were caused because of the presence of a unique sequence of base requests, and *name_value* reduced suites found faults that were caused by the presence of a unique value in the test case. Thus, each reduction requirement creates a reduced suite that detects a certain type of fault.

Also, Bryce et al. [5] found that prioritization criteria such as *2-way*, the frequency-based criteria (*AAS*, *MFAS*), *Req-LtoS* and *PV-LtoS* found a large number of faults quickly, i.e., have high values for APFD. Thus, the different prioritization criteria create effective test orders, while targeting unique aspects of the web applications, such as two-way interactions (*2-way* prioritization) or covering large amount of code as denoted by the length of the test case (*Req-LtoS*, *PV-LtoS* prioritization).

The observation that different test prioritization and reduction criteria target different characteristics of the underlying system, coupled with the time constraints under which testing is usually conducted, motivated us to evaluate the concept of ordering a reduced suite, where a reduced set is first created using reduction criteria, and then ordered using prioritization criteria.

Table 5
Unique IDs assigned to each unique ⟨Parameter, value⟩ pair.

u_1	u_2	u_3	u_4	u_5	u_6
(1) v_1	(3) v_2	(6) v_3	(8) v_4	(10) v_{11}	(11)v_6
(2) v_5	(4) v_7	(7) v_9	(9) v_{10}		
	(5) v_8				

Table 6
2-way interaction coverage of ⟨Parameter, value⟩ pairs between pages.

Parameter-values in test cases, listed by unique IDs from Table 5	Pairs covered
tc1	*6 pairs covered*
Page A: 1	(1,3) (1,6) (1,8) (3,6) (3,8) (6,8)
Page B: 3	
Page C: 6, 8	
tc2	*19 pairs covered:*
Page A: 2, 11	(2,4) (2,5) (2,7) (2,9) (2,10)
Page B: 4, 5	(4,11) (4,7) (4,9) (4,10)
Page C: 7, 9	(5,11) (5,7) (5,9) (5,10)
Page D: 10	(7,9) (7,10) (7,11) (**9,10**) (9,11) (10,11)
tc3	*6 pairs covered*
Page A: 1	(1,6) (1,9) (1,10) (6,9) (6,10) (**9,10**)
Page C: 6, 9	
Paged D: 10	

In previous work [5,29], though certain prioritization criteria perform better than others with respect to rate of fault detection, the effectiveness measures did not consider factors such as test generation time and test execution time. For example, the *2-way* criterion is among the best prioritization criteria for event-driven applications [5], but in that work, we measure the rate of fault detection in terms of APFD, which does not consider test execution and generation costs. We hypothesize that when test generation and execution costs are considered, test suites could demonstrate different behavior. Furthermore, effectiveness of reduction requirements in previous work [31] is measured with respect to number of faults detected. Their effectiveness may differ when measured with a rate of fault detection measure, where the goal is to find faults quickly. Therefore, in this paper, we study all combinations of each reduction requirement with each prioritization criterion.

This led to the creation of 40 *hybrid test selection criteria*. We use the terms ordered reduced suites and hybrid test suites interchangeably in the remainder of this paper. Table 9 presents all the hybrid criteria that we evaluate in our study. The first row in Table 9 identified by the mnemonic *seq_Req-LtoS*, indicates that the test suite is first reduced to satisfy the *seq* requirement, and then prioritized by the *Req-LtoS* technique to create an ordered reduced test suite.

4. Experimental setup

4.1. Research questions

Our empirical study examines the following research questions.

1. What effect do the different test reduction-test prioritization combinations have on fault detection rates?
2. How does test generation time impact the measured relative fault detection rate?
3. Do certain test reduction-test prioritization combinations work best, in general, for all applications and test suites?

4.2. Independent variables

The independent variables in our study are the subject applications, the original test suites, and the test selection criteria, e.g., the prioritization criteria, the reduction requirements, and the hybrid criteria, and the reduction and prioritization algorithms.

4.2.1. Subject applications and test suites

We use three web-based applications and their pre-existing test suites, where test suites are previously recorded user-sessions in experiments by Sampath et al. [32] and Sprenkle et al. [33]. The subject programs include an open-source, e-commerce bookstore (Book) [14], a Course Project Manager (CPM), and the web application used for the Mid-Atlantic Symposium on Programming Languages and Systems (Masplas). Table 7 shows the subject programs and test suite characteristics. The smallest application is Masplas, with only 999 lines of code and the fewest classes, methods, and conditions in relation to Book and CPM. CPM is the largest application in regard to lines of code and classes, but Book has more conditions and methods.

Book. 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. Since our interest was in testing consumer functionality, we did not include the administration code in our experiments. Book uses JSP for its front-end and a MySQL database back-end. In previous work, Sampath et al. [32] collected 125 test cases by sending emails to local newsgroups and by posting advertisements in the University of Delaware's classifieds web page asking for volunteer users. Table 7 shows the characteristics of the test cases, such as the length of the longest test case and the average test case length for all three applications.

CPM. CPM is a course project manageer where course instructors login and create *grader* accounts for teaching assistants. Instructors and teaching assistants create *group* accounts for students, assign grades, and create schedules for demonstration time slots. Users interact with an HTML application interface generated by Java servlets and JSPs. CPM manages state in a file-based datastore. In previous work, Sampath et al. [32] and Sprenkle et al. [33] collected 890 test cases from instructors, teaching assistants, and students using CPM during the 2004–2005 and 2005–2006 academic years at the University of Delaware.

Masplas. Masplas is a web application developed at the University of Delaware for a regional workshop. Users can register for the workshop, upload abstracts and papers, and view the schedule, proceedings, and other related information. Masplas is written using Java, JSP, and MySQL. In previous work, Sampath et al. [32] and Sprenkle et al. [34] collected 169 test cases that we use in our experiments.

Table 7 shows the number of unique parameter-values in the test cases for each of the three applications, and the number of *2-way* interactions covered by the test cases. These metrics provide an insight into the complexity of the test cases, with respect to the number of parameters they contain. Table 7 also shows the num-

Table 7
Subject applications and test suite characteristics.

Metrics	Book	CPM	Masplas
Classes	11	75	9
Methods	319	173	22
Conditions	1720	1260	108
Non-commented lines of code	7615	9401	999
Test suite characteristics			
Total number of user sessions	125	890	169
Total number of requests accessed	3640	12,352	1107
Number of unique requests	10	69	24
Largest user session in number of requests	160	585	69
Average user session in number of requests	29	14	7
Number of unique parameter-values	1,415	4,146	645
% of 2-way parameter-value interactions covered in pre-existing test suite	92.5%	97.8%	96.2%
Seeded faults characteristics			
No. of seeded faults	40	135	29
Fault Detection Density (FDD)	.59	.056	.19
Minimum no. faults found in a test	6	0	1
Average no. faults found in a test	21.43	4.67	4.62
Maximum no. faults found in a test	32	33	15

ber of user sessions that we converted into test cases for our applications. The CPM test suite is the largest with 890 test cases, followed by MASPLAS with 169 test cases, and Book with 125 test cases. The test cases vary in complexity and length. For instance, the table shows that the users of Book make 29 requests on average in their user-sessions, as opposed to 14 for CPM and 7 for MASPLAS. We also see that the largest test case for CPM includes 585 requests, while the largest for Book includes 160 requests and the largest for MASPLAS includes 69 requests. In terms of the uniqueness of requests, the user-sessions for CPM include 69 unique requests. Those for MASPLAS include 24 unique requests and the test suite for Book include 10. The last set of rows in Table 7 describe the characteristics of the seeded faults. CPM, our largest application in terms of code, has 135 seeded faults, Book has 40 seeded faults, and our smallest application, MASPLAS, has 29 seeded faults. The seeded faults and the Fault Detection Density values in the table are explained later in this section.

4.2.2. Reduction and prioritization criteria

Table 8 summarizes the reduction requirements, the prioritization criteria, and the controls that we use in our study. We refer to the reduction and prioritization criteria in Table 8 as *Pure Reduction* and *Pure Prioritization* techniques in the rest of the paper. Table 9 shows the 40 hybrid criteria that we applied. For instance, H1 (*seq_Req-LtoS*) first reduces a test suite by the *seq* criteria and then prioritizes the remaining test suite by *Req-LtoS*.

Table 10 shows the size of the reduced test suites created using the different test requirements described in Section 3.1. For our three subject applications, the *base* reduction technique reduces the test suites to the smallest size, followed by the *name* and *seq* techniques. The *name* and *seq_name* techniques create the largest reduced test suites. The size of the hybrid test suites are only as large as the initial reduced test suites, whereas, the non-hybrid prioritized test suites will contain the entire original test suite.

Table 8
Non-hybrid test selection criteria.

Label	Mnemonic	Description
Reduction requirements		
R1	*seq*	Reduced while covering all base request sequences of size 2
R2	*base*	Reduced while covering all base requests
R3	*name*	Reduced while covering all base request and parameter names
R4	*seq_name*	Reduced while covering all base request sequences of size 2 and parameter names
R5	*name_value*	Reduced while covering all base requests, parameter names and values
Prioritization techniques		
P1	*1-way*	Prioritized by 1-way interaction coverage of parameters
P2	*2-way*	Prioritized by 2-way interaction coverage of parameters
P3	*AAS*	Prioritized by covering all accessed base request sequences count
P4	*Req-LtoS*	Prioritized by test case length in terms of base requests (long to short)
P5	*Req-StoL*	Prioritized by test case length in terms of base requests (short to long)
P6	*MFAS*	Prioritized by most frequently accessed base request sequence count
P7	*PV-LtoS*	Prioritized by test case length in terms of parameter-values (long to short)
P8	*PV-StoL*	Prioritized by test case length in terms of parameter-values (short to short)
Control techniques		
C1	*Logged order*	Unordered (or in order as logged when original capture)
C2	*Random*	Random selection

4.3. Dependent variables and measures

The primary goal of ordering a reduced suite is to find faults quickly, which is best measured by a metric that measures the rate of fault detection, such as APFD [27]. However, when comparing the effectiveness of ordered reduced suites, we encounter the situation where the reduced suite generated by criterion H1 is different in size from the suite generated by criterion H2. APFD is not adequate to compare test suites of different sizes, as we explain later in this section. Therefore, we propose the *Mod_APFD_C* metric. The *Mod_APFD_C* metric also incorporates the two costs.

- The time taken to generate a reduced/prioritized suite: because test generation time affects the effectiveness of the selection technique.
- The time taken to execute the reduced/prioritized suite: because reduced suites are typically much smaller than the prioritized suites.

Test generation time is important because, criteria vary in the time taken to create the ordered reduced suite. Factors that affect the test generation time are the complexity of the requirements used by the criteria and the algorithm implementation. For example, since branch coverage is more complex than statement coverage, a criterion that uses branch coverage to generate the ordered test set could take longer than a criterion that uses statement coverage, especially, if the coverage matrix mapping the tests to the requirements is built as the ordering algorithm creates the test order. The time it takes to execute the test suite is also important, because, during regression testing, a primary concern is time availability to execute the test suite. Thus, considering test execution time during the evaluation of effectiveness of regression test orders is a natural extension.

Next, we illustrate how we incorporate these measures into *Mod_APFD_C* that extends APFD_C metric [10].

Elbaum et al. [10] raise the practical consideration that not all test cases or faults are of equal cost. They develop the APFD_C metric that accounts for test case cost and fault severity. Intuitively, APFD_C measures the area under the curve that plots the percentage test case cost incurred on the x-axis and the percentage of cumulative faults detected on the y-axis.

However, both the metrics APFD [27] and APFD_C [10] were originally designed to compare test suites of equal length, which is often the case in test case prioritization. In its current form, APFD_C favors test suites that are longer in size, even if the latter test cases are not detecting any new faults. For example, consider the fault matrix in Fig. 2a, where test cases *t1* and *t2* detect all four faults, while test cases *t3* and *t4* do not detect any faults. For simplicity of illustration, in this example, we assume that each test case is of equal cost and each fault is equally severe. Assume that one reduced suite is *t1, t2, t3, t4*, and another reduced suite contains the test cases *t1, t2*.

In this case, a reduced test suite that contains all four test cases (shown in Fig. 2b), has a higher APFD_C (shaded area under the curve) than the reduced test suite which contains only two test cases (as shown in Fig. 2c), even though both suites have the same rate of fault detection. The longer test suite gets a higher APFD_C only because of the number of tests in the suite (4 test cases, as against the reduced suite with 2 test cases), though it is clear from the fault matrix that the two test suites are equally effective at detecting faults. Thus, when comparing test suites of unequal length, larger test suites are favored by the APFD_C metric. Qu et al. [23] address the issue of unequal test lengths with the APFD metric by limiting the size of the test suite to the smallest test suite and examining the APFD curve only until that point. For example, if two techniques are compared where the test suites are such that

Table 9
Hybrid test selection techniques.

Label	Mnemonic	Description
H1	*seq_Req-LtoS*	Reduce by *seq* and prioritize by *Req-LtoS*
H2	*seq_Req-StoL*	Reduce by *seq* and prioritize by *Req-StoL*
H3	*seq_PV-LtoS*	Reduce by *seq* and prioritize by *PV-LtoS*
H4	*seq_PV-StoL*	Reduce by *seq* and prioritize by *PV-StoL*
H5	*seq_MFAS*	Reduce by *seq* and prioritize by *MFAS*
H6	*seq_AAS*	Reduce by *seq* and prioritize by *AAS*
H7	*seq_1-way*	Reduce by *seq* and prioritize by *1-way*
H8	*seq_2-way*	Reduce by *seq* and prioritize by *2-way*
H9	*base_Req-LtoS*	Reduce by *base* and prioritize by *Req-LtoS*
H10	*base_Req-StoL*	Reduce by *base* and prioritize by *Req-StoL*
H11	*base_PV-LtoS*	Reduce by *base* and prioritize by *PV-LtoS*
H12	*base_PV-StoL*	Reduce by *base* and prioritize by *PV-StoL*
H13	*base_MFAS*	Reduce by *base* and prioritize by *MFAS*
H14	*base_AAS*	Reduce by *base* and prioritize by *AAS*
H15	*base_1-way*	Reduce by *base* and prioritize by *1-way*
H16	*base_2-way*	Reduce by *base* and prioritize by *2-way*
H17	*name_Req-LtoS*	Reduce by *name* and prioritize by *Req-LtoS*
H18	*name_Req-StoL*	Reduce by *name* and prioritize by *Req-StoL*
H19	*name_PV-LtoS*	Reduce by *name* and prioritize by *PV-LtoS*
H20	*name_PV-StoL*	Reduce by *name* and prioritize by *PV-StoL*
H21	*name_MFAS*	Reduce by *name* and prioritize by *MFAS*
H22	*name_AAS*	Reduce by *name* and prioritize by *AAS*
H23	*name_1-way*	Reduce by *name* and prioritize by *1-way*
H24	*name_2-way*	Reduce by *name* and prioritize by *2-way*
H25	*seq_name_Req-LtoS*	Reduce by *seq_name* and prioritize by *Req-LtoS*
H26	*seq_name_Req-StoL*	Reduce by *seq_name* and prioritize by *Req-StoL*
H27	*seq_name_PV-LtoS*	Reduce by *seq_name* and prioritize by *PV-LtoS*
H28	*seq_name_PV-StoL*	Reduce by *seq_name* and prioritize by *PV-StoL*
H29	*seq_name_MFAS*	Reduce by *seq_name* and prioritize by *MFAS*
H30	*seq_name_AAS*	Reduce by *seq_name* and prioritize by *AAS*
H31	*seq_name_1-way*	Reduce by *seq_name* and prioritize by *1-way*
H32	*seq_name_2-way*	Reduce by *seq_name* and prioritize by *2-way*
H33	*name_value_Req-LtoS*	Reduce by *name_value* and prioritize by *Req-LtoS*
H34	*name_value_Req-StoL*	Reduce by *name_value* and prioritize by *Req-StoL*
H35	*name_value_PV-LtoS*	Reduce by *name_value* and prioritize by *PV-LtoS*
H36	*name_value_PV-StoL*	Reduce by *name_value* and prioritize by *PV-StoL*
H37	*name_value_MFAS*	Reduce by *name_value* and prioritize by *MFAS*
H38	*name_value_AAS*	Reduce by *name_value* and prioritize by *AAS*
H39	*name_value_1-way*	Reduce by *name_value* and prioritize by *1-way*
H40	*name_value_2-way*	Reduce by *name_value* and prioritize by *2-way*

one test suite has 100 test cases and the other has 300 test cases, Qu et al. study the area under the curve of only 100 tests cases for both the techniques. Our approach differs from Qu et al.'s approach because we consider all the tests generated by a technique, when comparing techniques that generate test suites of unequal length.

In our evaluation, to allow comparison between suites of unequal sizes, when calculating the rate of fault detection for the different reduced test suites, we use the cost of *all* the test cases in the original suite, even though those test cases are not present in the reduced suite execution. This limits the bias that the original APFD_C formula has towards longer test suites. The effect of including all test case costs is shown in Fig. 2c. By including all the costs, both the reduced test suite and the longer test suite has the same APFD_C value of 80%, thus showing that both test suites are equally effective.

Adding all test cases cost does not negatively affect the reduced test suites effectiveness in any way, e.g., when measuring the area under the curve, the added cost does not make a reduced suite less effective than without the added cost. Instead, this addition allows the comparison of test suites with unequal number of test cases.

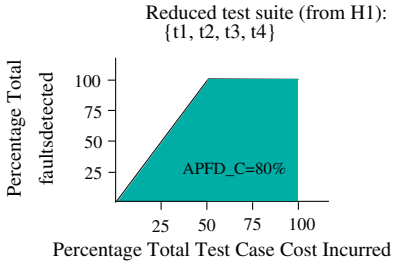
We modified the APFD_C metric further to incorporate test generation cost. We treat the generation cost as a pseudo test case that does not detect any faults. Intuitively, each technique's suite is weighted at the beginning with the time the technique took to generate the prioritized/reduced suite. This has the effect of penalizing those test suites which have higher generation costs, everything else being the same between the test suites. Fig. 2e and f show the effect of adding test generation time as a pseudo test case. In Fig. 2e, we illustrate with a generation cost, *t_gen* of 1*C for the prioritization technique, where C is the cost of one test case. The generation cost, *t_gen* is treated as a test case that detects no faults, and thus, is represented in the first 20% of the x-axis. After this, the next test case, *t1* detects 2 faults, which is plotted after 40% of the cost is incurred. The fault detection of the remaining faults is similar to Fig. 2b, only pushed to the right by the test generation time of the technique. When the test suite is plotted in this manner, the *Mod_APFD_C* of the test suite evaluates to 60%. Similarly, in Fig. 2f, where the reduced suite generation time is 4* C, where C is the cost of one test case, the *Mod_APFD_C* value of the reduced test suite evaluates to 30%. Thus, treating test generation time as a pseudo test case penalizes techniques that take longer to generate the prioritized/reduced test suite.

Table 10
Sizes of the original and reduced test suites for the Book, CPM, and MASPLAS web applications.

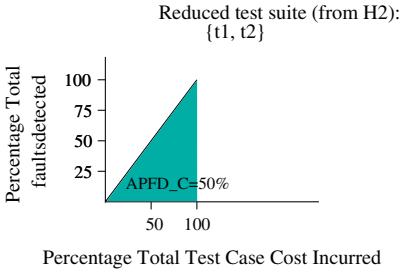
	Book	CPM	MASPLAS
Original test suites			
	125	890	169
Reduced test suites			
seq	66	270	60
base	5	48	21
name	27	111	42
name_value	113	439	100
seq_name	96	367	108

test	fault			
	f1	f2	f3	f4
t1	×	×		
t2			×	×
t3				
t4				

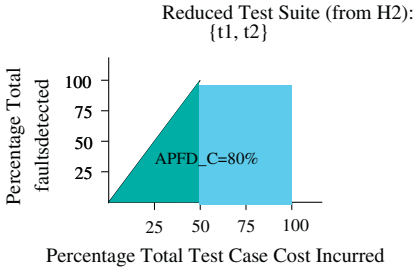
(a) Fault Matrix (assume each test has equal cost, C, and each fault is equally severe)



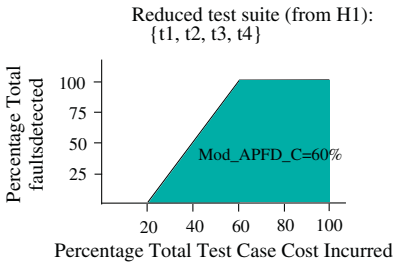
(b) APFD_C



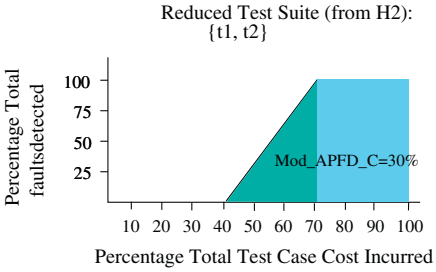
(c) APFD_C with reduced test suite



(d) Our modification to APFD_C for reduced test suites of varying sizes



(e) With test generation time included (assume: test generation time = C)



(f) With test generation time included (assume: test generation time = 4*C)

Fig. 2. Example showing APFD_C's bias towards longer test suites.

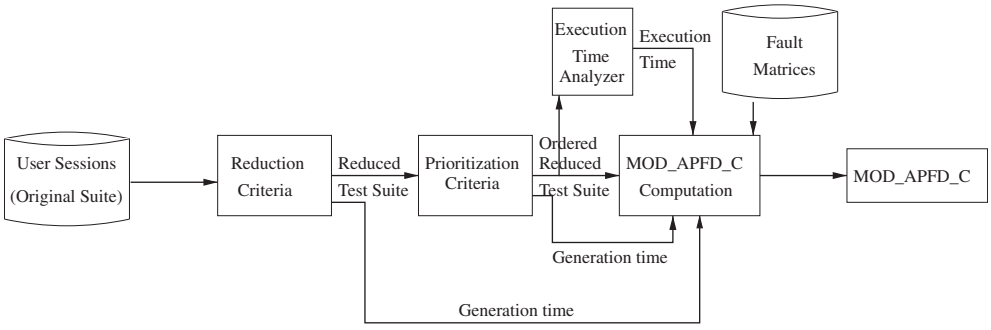


Fig. 3. Overview of experimental process.

We define the modified APFD_C metric, *Mod_APFD_C* as follows:

For a test suite, T with n test cases, and execution costs $t_1, t_2, t_3, \ldots, t_n$, if F is a set of m faults detected by T with severities $f_1, f_2, f_3, \ldots, f_m$, then let TF_i be the position of the first test case t in T', where T' is an ordering of T, generated by a technique, G with generation time, t_{gen}, that detects fault i. Then, the $APFD_C$ metric for T' is given as

$$Mod\_APFD\_C = \frac{\sum_{i=1}^{m}\left(f_i \times \left(\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i}\right)\right)}{\left(\sum_{i=1}^{n} t_i + t_{gen}\right) \times \sum_{i=1}^{m} f_i} \quad (1)$$

In this work, all faults are treated as equally severe. Elbaum et al. [10] note that the APFD_C metric reduces to APFD, when all test costs and fault severities are identical. Therefore, in our experiments, the lack of fault severities data does not effect the validity of the *Mod_APFD_C* metric.

4.4. Experimental setup

Fig. 3 presents an overview of the experimental setup. The reduction criteria are applied on the original suite to create the reduced test suite. The reduced suite is then ordered by applying the

prioritization criteria. Generation times are computed as the reduction and prioritization implementations execute. The test execution time is measured for the ordered reduced suite, and is fed into the *Mod_APFD_C* calculator along with the generation time and fault matrices.

4.4.1. Generation time computation

Note that because of random tie breaking, we execute the reduced/prioritization algorithms 30 times and report the average generation time. The time taken to generate the reduced suites is shown in Table 11. The generation time for the ordered reduced suites shown in Table 11 includes the time to reduce and then prioritize the reduced test suite. The generation time varies based on the size of the test suite. For instance, the test suite for CPM is our largest test suite with 890 test cases and requires longer generation time for reduction and prioritization when compared to Book and Masplas. Most of the reduction, prioritization, and hybrid generations take less than 2 min, but some take almost 10 min, such as the *2-way* prioritization and the *name_value_2-way* hybrid criteria applied to CPM. Random ordering is the fastest technique, taking between.02 to.06 seconds in our experiments. This fast execution time will give random an advantage in our experiments since we consider the generation time as part of the costs in our experiments. Similarly, *Logged Order* has a generation time of 0 s because no processing is required to derive the log order.

4.4.2. Execution time computation

We replay the entire original test suite in log order and compute the average time per request by dividing the total execution time of original suite by the total number of requests in the test suite. We then use the average time per request to compute the execution time of each test case. We assume that the execution time of each request is equal. Thus, execution times here are, in part, an abstraction of the length of the test cases. Table 12 shows the average, minimum, and maximum execution time for the test cases of our subject applications. These execution times represent the time taken to replay a set of test cases in the logged order and not the time taken for fault detection replay. (See Table 13).

4.4.3. Fault matrices

In this work, we use pre-existing fault matrices that show how many faults are detected by each test case. The fault matrices in this paper use the *diff* oracle for *Book* and the *struct* oracle for *CPM* and *Masplas* [33]. While the *diff* oracle compares the entire HTML response returned on each request, the *struct* oracle reduces false positives from real-time data by only comparing the HTML tags of the response page. The comparison of the actual and expected results is done by using the Unix utility *diff*. A non-fault seeded version of the application is used as the gold standard for expected results. When the fault detection experiments were conducted, only one fault exists in the application for one run of the test suite. Table 7 show the total number of seeded faults in the application, and the minimum, maximum, and average number of faults detected by the test suites of the three applications. The seeded faults have been used in previous work [32,33,31,30].

In previous work [29], we defined a metric, the **fault detection density (FDD)**, to understand the spread of how many faults are detected by the test cases. FDD is defined as the ratio of the sum of the *total number of faults detected by each test case* and *the total number of test cases*, normalized with respect to the *total number of faults detected*. Given a set of test cases, $t_i \in T$ and a set of faults F detected by test cases in T, let t_{f_i} be the number of faults detected by t_i, then the fault detection density,

$$FDD = \frac{tf_1 + tf_2 + \cdots + tf_n}{|T| * |F|} \quad (2)$$

Table 11
Generation time (in seconds).

	Book	CPM	Masplas
Reduction criteria			
base	0.58	4.25	0.60
seq	4.69	16.54	0.85
name	1.82	4.13	0.70
name_value	1.19	4.27	0.66
seq_name	13.96	22.03	0.85
Prioritization criteria			
AAS	0.33	3.60	0.27
Req-StoL	0.09	0.58	0.58
Req-LtoS	0.09	0.48	0.08
MFAS	0.28	3.52	0.25
1-way	0.20	3.17	0.17
2-way	181.91	1334.38	14.29
PV-LtoS	0.12	0.21	0.13
PV-StoL	0.21	0.44	0.16
Random	0.003	0.01	0.003
Hybrid criteria			
seq_AAS	5.03	20.19	1.12
seq_MFAS	4.97	20.06	1.10
seq_Req-LtoS	4.69	16.59	0.85
seq_Req-StoL	4.69	16.55	0.85
seq_2-way	72.68	412.95	3.43
seq_1-way	4.70	16.92	0.86
seq_PV-LtoS	4.69	16.54	0.85
seq_PV-StoL	4.69	16.54	0.85
base_AAS	0.92	7.85	0.88
base_MFAS	0.86	7.77	0.86
base_Req-LtoS	0.58	4.26	0.60
base_Req-StoL	0.58	4.29	0.61
base_2-way	2.63	114.15	1.16
base_1-way	0.59	4.29	0.61
base_PV-LtoS	0.58	4.25	0.60
base_PV-StoL	0.58	4.25	0.60
name_AAS	2.16	7.75	0.97
name_MFAS	2.10	7.66	0.96
name_Req-LtoS	1.82	4.16	0.70
name_Req-StoL	1.82	4.16	0.70
name_2-way	20.66	275.45	2.70
name_1-way	1.84	4.22	0.70
name_PV-LtoS	1.82	4.14	0.70
name_PV-StoL	1.82	4.14	0.70
name_value_AAS	1.59	7.92	0.94
name_value_MFAS	1.47	7.81	0.92
name_value_Req-LtoS	1.20	4.35	0.67
name_value_Req-StoL	1.20	4.35	0.67
name_value_2-way	121.14	636.76	9.31
name_value_1-way	1.23	5.23	0.68
name_value_PV-LtoS	1.19	4.28	0.66
name_value_PV-StoL	1.19	4.28	0.66
seq_name_AAS	14.31	25.66	1.14
seq_name_MFAS	14.25	25.57	1.11
seq_name_Req-LtoS	13.97	22.09	0.86
seq_name_Req-StoL	13.97	22.09	0.86
seq_name_2-way	122.26	569.66	6.48
seq_name_1-way	13.99	22.71	0.87
seq_name_PV-LtoS	13.97	22.04	0.85
seq_name_PV-StoL	13.97	22.04	0.85

FDD is the ratio of the sum of the *total number of faults detected by each test case* and *the total number of test cases*, normalized with respect to the *total number of faults detected*. A fault detection density of 1 for a test suite indicates that each test case in the suite detects every fault. Table 7 shows the FDD values for the three subject applications. CPM has the lowest FDD (.056) in our experiments, which means that each test case on average finds fewer faults than those for Masplas and Book that have higher FDD values.

The fault matrices were gathered using the *with_state* replay mechanism presented by Sprenkle et al. [33]. In *with_state* replay, the original suite is executed in log order, and the application state after each test case is executed is stored. Then, when a test case in

Table 12
Execution time of test cases (in seconds).

Application name	Minimum	Average	Maximum
Book	0.33	27.79	223.74
CPM	0.072	2.22	46.15
Masplas	0.066	1.22	11.88

Table 13
Number of faults detected by reduced suites.

	Book	CPM	Masplas
Faults detected by original suite	36	84	24
Faults detected by *seq*	35	80	24
Faults detected by *base*	34	69	24
Faults detected by *name*	34	79	24
Faults detected by *name_value*	36	84	24
Faults detected by *seq_name*	36	82	24

the prioritized order or the reduced suite is executed, the state of the application is reset to the stored state of the test case from the original suite execution. This way, the test case is replayed under the same conditions as when it was logged originally and allows for accurate replay. We refer the reader to previous work for further details on how the test cases were replayed, the descriptions of the different oracles used in the experiments and the categories of faults seeded in the applications [29,32,33].

4.5. Threats to validity

First, our results may be affected by the fault seeding methodology used (manually seeding faults) and the distribution of the faults in the programs. Andrews et al. [1] found that hand seeded faults are harder to detect than mutation faults. By using hand seeded faults in our programs, we believe we have reduced the threat. As mentioned earlier by Sampath et al. [32], the faults seeded into the applications were closely modeled after naturally occurring faults—some faults in CPM and Masplas where indeed naturally occurring that were then seeded into the applications for the experiments.

A second issue is that our reduction and prioritization algorithms use random tie-breaking when multiple tests meet a criteria for selection. To address the variation caused by the nondeterminism, we executed each test selection technique 30 times and report the average generation times and *Mod_APFD_C* values.

Threats to construct validity arise when the measurements are not adequate to measure the concepts studied. We propose the *Mod_APFD_C* metric which augments Elbaum et al.'s [10] APFD_C metric by accounting for test generation time. In addition, to allow a fair comparison between test suites of unequal lengths, we modify the manner in which cost is computed for the reduced suites. We have shown by examples that this modification does not adversely affect the reduced suite's effectiveness. In this study, we do not consider fault severities. However, the *Mod_APFD_C* metric is similar to APFD_C [10] in that it reduces to the traditional APFD [27] metric when test case cost and fault severity are not considered. Though it would be interesting to evaluate the effectiveness of the techniques when fault severities are considered, the lack of fault severities does not affect the validity of the *Mod_APFD_C* metric. Finally, Do et al. [6,7] proposed a cost-benefit measure where they identify several costs and benefits that are encountered during the prioritization process and propose a measure that is the difference between the costs and benefits. We did not have access to the costs and benefits proposed in their study and thus could not apply their effectiveness measure. Finally, we define costs as generation and execution time of test suites, but we do not consider the human time to analyze the results or the costs of individual faults. Future work may consider more extensive costs and fault severities.

Threats to external validity are factors that may impact our ability to generalize our results to other situations. The main threat to our external validity is that we examine only 3 web-based applications and their existing user-session-based test suites, which limits the extent to which we can generalize our results. We attempt to reduce this bias by selecting medium-sized web applications from different domains—an e-commerce bookstore, a course project manager, a conference registration system. An experiment that would be more readily generalized would include a larger pool of web applications.

5. Results and discussion

To answer our research questions, we examine the *Mod_APFD_C* values of the ordered reduced test suites.

Tables 14–16 show the *Mod_APFD_C* values of the ordered reduced suites in tabular form. The reported *Mod_APFD_C* values are the average of 30 runs of each test selection algorithm. The columns are named by the prioritization criterion and the rows are named by the reduction criterion. The right-most column presents the *Mod_APFD_C* values for *Pure Reduction* techniques named in first column. The last row presents the *Mod_APFD_C* values for the *Pure Prioritization* techniques corresponding to the column heading. All other cells present the *Mod_APFD_C* values of the hybrid criteria identified by the ⟨row name, column name⟩. For example, in the Table 14, the entry in the cell at the second column, first row, 81.17, is the *Mod_APFD_C* value of the hybrid *seq_1-way*. The last two rows of the tables show the *Mod_APFD_C* values for *Logged Order* and *Random*. Both *Logged Order* and *Random* contain all the test cases from original suite.

In the tables, to denote when hybrids perform better than *Pure Prioritization* or *Pure Reduction* we use the following notation— if a hybrid is better than the *Pure Reduction* technique, then the hybrid's *Mod_APFD_C* is **bold-faced**. If a hybrid is better than the corresponding *Pure Prioritization* technique, then the hybrid's *Mod_APFD_C* is *italicized*. If a hybrid is better than both the corresponding *Pure Prioritization* and *Pure Reduction* technique, then the *Mod_APFD_C* value is ***bold-italicized***.

5.1.. RQ1: Effect of test reduction-prioritization combinations on rate of fault detection

5.1.1. CPM

From Table 14, we see that several hybrid test suites perform better than the corresponding *Pure Reduction* criterion as indicated by the large number of bold-faced *Mod_APFD_C* values. The *Mod_APFD_C* results in Table 14 for the five hybrids that include a reduction technique with *1-way* are between 76.19 and 82.78, while *Pure Reduction* ranges from 74.62 to 82.65. The hybrids with *1-way*, *AAS*, *Req-LtoS*, and *PV-StoL* are in general, better than *Pure Reduction*, although within 5%. We note that 55% of the hybrids perform better than *Pure Reduction*.

Hybrids involving *MFAS*, *PV-StoL* and *Req-StoL* always improve the *Mod_APFD_C* when compared to *Pure Prioritization*, whereas, for the other prioritization criteria (such as *1-way*, *AAS*, *Req-LtoS*), only hybrids with stricter reduction criteria (such as *name_value*, *seq_name*) produce test suites that are more effective than *Pure Prioritization*. We observe that 60% of the hybrids perform better than *Pure Prioritization* and 30% of the hybrids perform better than both *Pure Reduction* and *Pure Prioritization*. We see that *Logged Order* and *Random* perform poorly when compared to most of the better hybrids.

Table 14
CPM: The *Mod_APFD_C* for the 40 hybrids, *Pure Reduction* and *Pure Prioritization*. Bold-face indicates hybrid better than *Pure Reduction*, Italics indicates hybrid better than *Pure Prioritization*, Bold-italics indicates hybrid better than both *Pure Reduction* and *Pure Prioritization*.

	1-way	*2-way*	*AAS*	*Req-LtoS*	*PV-LtoS*	*MFAS*	*Req-StoL*	*PV-StoL*	*Pure Reduction*
seq	**81.17**	67.44	**82.98**	***84.08***	77.85	*76.53*	*77.61*	***80.59***	80.19
base	**76.19**	71.98	**78.07**	**77.72**	**75.01**	***76.03***	***76.11***	***77.3***	74.62
name	82.19	72.02	**84.35**	***85.49***	*79.79*	*79.57*	***82.69***	***86.88***	82.65
name_value	***82.78***	62.1	***85.32***	***85.82***	*78.11*	*75.6*	*76.7*	*81.18*	82
seq_name	**81.54**	63.49	**83.14**	***83.8***	77.19	*74.61*	*76.68*	***80.82***	80.77
Pure Prioritization	82.84	82.12	84.62	81.25	77.91	73.48	70.91	70.19	–
Logged order					80.54				
Random					77.96				

Table 15
MASPLAS: The *Mod_APFD_C* for the 40 hybrids, *Pure Reduction*, and *Pure Prioritization*. Bold-face indicates hybrid better than *Pure Reduction*, Italics indicates hybrid better than *Pure Prioritization*, Bold-italics indicates hybrid better than both *Pure Reduction* and *Pure Prioritization*.

	1-way	*2-way*	*AAS*	*Req-LtoS*	*PV-LtoS*	*MFAS*	*Req-StoL*	*PV-StoL*	*Pure Reduction*
seq	***92.09***	**91.73**	***92.82***	***96.54***	***85.72***	***83.97***	**83.23**	***93.64***	82.71
base	***95.75***	***95.23***	***95.63***	***96.71***	***95.14***	***94.85***	***97.22***	***97.73***	93.41
name	***92.39***	***92.41***	***93.37***	***96.68***	***86.05***	***82.93***	***84.02***	***96.61***	83.75
name_value	**88.66**	**88.43**	**91.75**	***96.58***	**81.12**	***74.19***	64.37	***94.27***	71.33
seq_name	***90.51***	**89.86**	***92.38***	***96.33***	***84***	***78***	**80.68**	***90.7***	75.5
Pure Prioritization	89.07	92.39	91.78	90.7	81.38	73.83	83.53	87.57	–
Logged order					70.90				
Random					90.20				

Table 16
Book: The *Mod_APFD_C* for the 40 hybrids, pure reduction, and pure prioritization. Bold-face indicates hybrid better than *Pure Reduction*, Italics indicates hybrid better than *Pure Prioritization*, Bold-italics indicates hybrid better than both *Pure Reduction* and *Pure Prioritization*.

	1-way	*2-way*	*AAS*	*Req-LtoS*	*PV-LtoS*	*MFAS*	*Req-StoL*	*PV-StoL*	*Pure Reduction*
seq	*91.16*	*87.68*	92.05	*93.56*	*89.07*	*90.08*	93.23	95.78	96.1
base	*92.14*	*92.07*	**92.46**	***92.55***	*92.15*	*92.15*	**93.32**	**93.41**	92.32
name	*89.15*	*88.7*	92.77	91.83	88.5	*91.27*	**93.63**	93.41	93.54
name_value	*91.42*	*86.78*	93.35	*93.42*	*88.95*	*89.69*	94.49	***96.57***	96.28
seq_name	*91.17*	*86.9*	93.14	*93.55*	88.75	89.66	94.47	***97.24***	96.48
Pure Prioritization	88.97	85.38	93.36	86.89	88.97	89.66	96.03	96.09	–
Logged order					96.27				
Random					95.34				

The best *Pure Prioritization* criterion is *AAS* with a *Mod_APFD_C* value of 84.62 and the best *Pure Reduction* technique is *name* with a *Mod_APFD_C* value of 82.65. We observe that a hybrid of *Req-LtoS* and *name_value* produces a test suite with a better *Mod_APFD_C* value (85.82), thus suggesting that ordered reduced suites can be more effective than *Pure Prioritization* and *Pure Reduction*.

5.1.2. MASPLAS

From Table 15, we see that majority of the hybrid techniques (97.5% of the hybrids) perform better than *Pure Reduction*, within a 10% range of the *Mod_APFD_C* values. However, the hybrids do not necessarily perform better than the *Pure Prioritization* (77.5% of the hybrids perform better than *Pure Prioritization*). Hybrids with *Req-LtoS* have *Mod_APFD_C* values ranging from 96.33 to 96.71 while *Pure Reduction* ranges from 71.33 to 93.41 and *Pure Prioritization* ranges from 73.83 to 92.39. In general, hybrids with *AAS*, *Req-LtoS*, *PV-LtoS*, *MFAS* consistently perform better than both *Pure Reduction* and *Pure Prioritization*. We note that 77.5% of the hybrids perform better than both *Pure Reduction* and *Pure Prioritization*.

When compared to *Logged Order*, we see that majority of the hybrids perform better than *Logged Order*. *Random*, however, is better than several of the reduced suites. We attribute this to the high FDD of Masplas (from Table 7). A high FDD denotes that each test case detects a large number of the faults. Therefore, *Random* has a greater chance of selecting a test that covers a large number of faults.

The best *Pure Prioritization* technique is *2-way* with a *Mod_APFD_C* value of 92.39 and the best *Pure Reduction* technique is *base* with a *Mod_APFD_C* value of 93.41. The hybrid of *base_PV-StoL* produces the test order with the highest *Mod_APFD_C* (97.73).

5.1.3. Book

From Table 16, we see that hybrids with *2-way*, *1-way*, *MFAS*, *PV-LtoS* generally perform better than the *Pure Prioritization*. We note that 57.5% of the hybrids perform better than *Pure Prioritization*. Hybrids with *AAS*, *Req-LtoS*, *Req-StoL*, and *PV-StoL* perform better than *Pure Reduction*. Overall, 17.5% of the hybrids perform better than *Pure Reduction* and 7.5% of the hybrids perform better than both *Pure Reduction* and *Pure Prioritization*.

Both *Logged Order* and *Random* create effective test suites for Book. The high FDD of Book (0.59) from Table 7 explains these results. An FDD of 1 indicates that every test case detects every fault. With a high FDD, in Book, every test case is detecting a large number of faults, therefore, *Logged Order* and *Random* tend to perform well. Hybrids of *seq_name_PV-StoL* has the highest *Mod_APFD_C* (97.24) when compared to the best *Pure Prioritization* (96.09) and *Pure Reduction* (96.28) technique.

In all three subject applications, we see that a hybrid criterion where a reduced suite is ordered has the highest effectiveness value when compared to the pure techniques.

5.2. RQ2: Effect of test generation time on relative fault detection rate

For this research question, we focus on the *2-way* criterion and its hybrids because the effect of test generation time is most pronounced in these cases. The *2-way* criterion in our subject applications (from Table 12) has the largest execution time compared to the other criteria. In previous work, prioritizing by *2-way* criterion created the most effective test order for our subject applications. In that work, we measured the effectiveness of the test orders using APFD [5], which assumes that all tests have equal cost and all faults are equally severe. In this paper, we consider the factors of test generation time and test execution time, both of which are important for a tester. When these factors are incorporated into the effectiveness measure, *Mod_APFD_C* we observe that *2-way* is no longer the most effective criterion.

For CPM, Table 14, we note that hybrids with *2-way* criterion do not perform well when compared to *Pure Reduction* and *Pure Prioritization*, primarily because of the longer generation time of *2-way* that results in a low *Mod_APFD_C* value. For Masplas, *2-way*'s performance is comparable to *Pure Prioritization* and *Pure Reduction* even though the generation time of *2-way* is larger than the other generation times (from Table 12). For Book, hybrids with *2-way* perform better than *Pure Prioritization* but are poor when compared to *Pure Reduction*. This difference between the subject applications is expected, because from Table 12, generation time of *2-way* is much larger for CPM and Book when compared to the criterion with the least generation time, than for Masplas.

5.3. RQ3: Test reduction-prioritization combinations that are effective for all subject applications

From the tables, we note that hybrid combinations between *base*, *seq*, and *name* reduction criteria and the prioritization criteria *AAS*, *Req-LtoS*, *Req-StoL*, and *PV-StoL* create reduced suites that are more effective than *Pure Reduction* for all the three subject applications. In CPM and Masplas, the hybrids created with these combinations are mostly better than *Pure Reduction*. In Book, the hybrids created with these criteria are better than *Pure Reduction* when combined with the *base* reduction criterion, and in one case with the *name* criterion. In prior work, Sampath et al. [31], found that *base*, *seq* and *name* are among the better reduction criteria with respect to number of faults detected for three applications. By applying the above prioritization criteria, we are able to increase the effectiveness of the reduced test suites further. Thus, in our experiments, there is a clear benefit to creating hybrids of reduced suites versus executing a reduced suite in logged order. Note that the hybrid test suites include the time to reduce the original test suite and the time to prioritize the reduced suite. Despite the additional time overhead, these hybrid test suites perform better than the pure techniques in several instances.

5.4. Guidance to testers

From our results, we find that we can improve the effectiveness of reduced suites created by *base*, *name* and *seq* by ordering them based on certain frequency (*AAS*) and length-based prioritization (*Req-LtoS*, *Req-StoL*, *PV-StoL*) criteria. Our controls, *Random* and *Logged Order*, create effective test orders when a test suite has a high Fault Detection Density (FDD). However, we believe that though *Random* appears to do well in in our experiments, it is not a reliable technique, because the different random orderings often result in substantial variation in the *Mod_APFD_C* values.

The particular hybrid criterion a tester selects for their web application depends on two main factors (a) characteristics of the underlying web application, and (b) the test generation time for the hybrid criteria. Consider the first factor, characteristics of the underlying web system—we have shown in previous work that systems with large number of parameter-values benefit from interaction-based criteria or length-based on parameter-values criteria, while systems that have associations between base URLs and large amounts of underlying code benefit from length-based criteria [5]. Similarly, for reduction criteria, systems with large number of parameter names benefit from a criterion that targets parameter names, e.g., *name*, while a system where sequences of requests lead to faults benefit from sequence-based criteria, e.g., *seq* [31,32].

The second factor a tester should consider is the time taken to generate the test set. In our experiments, we see that *2-way* (which is an effective criterion [5]) and hybrids involving *2-way* perform poorly when evaluated with the *Mod_APFD_C* metric. This is because *2-way* tends to take longer to create the reduced test set and thus negatively affects the effectiveness of the metric.

Given these two factors, a couple of options exist for the tester. If the underlying application is such that the sequences in which pages are accessed is important, a hybrid between the prioritization criterion *AAS* and the reduction criterion of *seq* could be most successful. However, *base* and *name* are likely to have less generation time than *seq* because of the pre-processing involved to create the reduced test suites. Therefore, another option for the tester is to combine *AAS* with *base* or *name* to create effective test orders, instead of using a *Pure Reduction* technique.

Similarly, as an alternative to *2-way*, hybrids of *1-way* or *PV-LtoS* can be considered, since they target the same underlying characteristic as *2-way*, i.e., a system that has a large number of parameter-values, while taking less time to order the test suite. Thus, a tester could use a hybrid between *PV-LtoS* (which was seen as effective across all our subject applications) with the *name* (which also targets parameter-names associated with a URL) reduction criterion to create an effective reduced test order.

The tester must therefore evaluate the tradeoff between characteristics of the underlying web system and the generation time of a particular hybrid criterion to determine the most effective hybrid criterion for their system.

6. Conclusions

Though test reduction techniques create test suites that are smaller than the original suite, the reduced suite themselves can be very large. In this paper, we investigate several ways in which reduced test suites can be ordered. We reduce user-session-based test suites using reduction criteria and prioritize them by applying experimentally verified prioritization criteria. We conduct an empirical study where we evaluate the effectiveness of these ordered reduced suites by studying their rate of fault detection. To enable comparison between reduced suites of different lengths we propose the *Mod_APFD_C* metric. This metric also incorporates cost measures such as test generation and execution time. Our studies reveal that in each of our three subject applications, some of the ordered reduced suites have a better *Mod_APFD_C* value than both *Pure Prioritization* and *Pure Reduction*. We also find that in several instances, the reduce and reorder technique performs better than either *Pure Reduction* or *Pure Prioritization* alone. Given the practical consideration that a tester will use a reduced test suite, the approach that also prioritizes the reduced test suite can improve the rate of fault detection. We find these results promising and anticipate that future empirical studies may extend our work to reveal whether this improvement scales for larger applications that also have larger test suites. Our process of reducing and

reordering test suites is repeatable and may also be used in other domains.

Acknowledgements

This research is supported by NIST Grant No. 70NANB10H048. Reference to commercial products or trademarks does not imply endorsement by NIST, nor that such products are necessarily best suited to any purpose. Our tool, CPUT: Combinatorial-based Prioritization for User-session-based Testing, provides functionality to prioritize test suites such as those in this paper [28].

References

[1] J.H. Andrews, L.C. Briand, Y. Labiche, A.S. Namin, Using mutation analysis for assessing and comparing testing coverage criteria, Transactions on Software Engineering 32 (8) (2006) 608–624.
[2] Antonia Bertolino, Emanuela Cartaxo, Patrýcia Machado, Eda Marchetti, Joao Felipe Ouriques, Test suite reduction in good order: comparing heuristics from a new viewpoint, in: International Conference on Testing Software and Systems: Short Papers, October 2010, pp. 13–18.
[3] G. Birkhoff, Lattice Theory, American Mathematical Society, vol. 5, Colloquium Publications, 1940.
[4] Jennifer Black, Emanuel Melachrinoudis, David Kaeli, Bi-criteria models for all-uses test suite reduction, in: International Conference on Software Engineering, 2004, pp. 106–115.
[5] R. Bryce, A. Memon, S. Sampath, Developing a single model and test prioritization strategies for event-driven software, Transactions on Software Engineering 37 (1) (2011) 48–64.
[6] Hyunsook Do, Siavash Mirarab, Ladan Tahvildari, Gregg Rothermel, An empirical study of the effect of time constraints on the cost-benefits of regression testing, in: International Symposium on Foundations of Software Engineering, November 2008, pp. 71–82.
[7] Hyunsook Do, Gregg Rothermel, An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models, in: International Symposium on Foundations of Software Engineering, 2006, pp. 141–151.
[8] S. Elbaum, S. Karre, Gregg Rothermel, Improving web application testing with user session data, in: International Conference on Software Engineering, September 2003, pp. 49–59.
[9] Sebastian Elbaum, Alexey G. Malishevsky, Gregg Rothermel, Test case prioritization: a family of empirical studies, Transactions on Software Engineering 28 (2) (2002) 159–182.
[10] Sebastian Elbaum, Alexy Malishevsky, Gregg Rothermel, Incorporating varying test costs and fault severities into test case prioritization, in: The International Conference on Software Engineering, May 2001, pp. 329–338.
[11] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, Alexey G. Malishevsky, Selecting a cost-effective test case prioritization technique, Software Quality Journal 12 (3) (2004) 185–210.
[12] Sebastian Elbaum, Gregg Rothermel, Srikanth Karre, Marc Fisher, Leveraging user session data to support web application testing, Transactions on Software Engineering 31 (3) (2005) 187–202.
[13] Emelie Engström, Per Runeson, Mats Skoglund, A systematic review on regression test selection techniques, Information & Software Technology 52 (1) (2010) 14–30.
[14] Open source web applications with source code. <http://www.gotocode.com> (accessed on 19.12.11).
[15] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, Gregg Rothermel, An empirical study of regression test selection techniques, Transactions on Software Engineering Methodology 10 (2) (2001) 188–197.
[16] M.J. Harrold, R. Gupta, M.L. Soffa, A methodology for controlling the size of a test suite, Transactions on Software Engineering and Methodology 2 (3) (1993) 270–285.
[17] Mats P.E. Heimdahl, Devaraj George, Test-suite reduction for model based tests: Effects on test quality and implications for testing, in: International Conference on Automated Software Engineering (ASE), September 2004, pp. 176–185.
[18] H.-Y. Hsu, A. Orso, MINTS: a general framework and tool for supporting test-suite minimization, in: International Conference on Software Engineering, May 2009, pp. 419–429.
[19] Dennis Jeffrey, Neelam Gupta, Test suite reduction with selective redundancy, in: International Conference on Software Maintenance, September 2005, pp. 549–558.
[20] James A. Jones, Mary Jean Harrold, Test suite reduction and prioritization for modified condition/decision coverage, Transactions on Software Engineering 29 (3) (2003) 195–209.
[21] J. Lin, C. Huang, Analysis of test suite reduction with enhanced tie-breaking techniques, Information and Software Technology 51 (11) (2008) 679–690.
[22] Scott McMaster, Atif M. Memon, Call-stack coverage for GUI test-suite reduction, Transactions on Software Engineering 34 (1) (2008) 99–115.
[23] Xiao Qu, M.B. Cohen, K.M. Woolf, Combinatorial interaction regression testing: a study of test case generation and prioritization, in: International Conference on Software Maintenance, October 2007, pp. 255–264.
[24] G. Rothermel, M.J. Harrold, J. von Ronne, C. Hong, Empirical studies of test suite reduction, Journal of Software Testing, Verification, and Reliability 4 (2) (2002) 219–249.
[25] G. Rothermel, R. Untch, C. Chu, M. Harrold, Test case prioritization: an empirical study, in: International Conference on Software Maintenance, September 1999, pp. 179–188.
[26] Gregg Rothermel, Sebastian Elbuam, Alexey Malishevsky, Praveen Kallakuri, Xuemei Qiu, On test suite composition and cost-effective regression testing, Transactions on Software Engineering and Methodology 13 (3) (2004) 277–331.
[27] Gregg Rothermel, Roland H. Untch, Chengyun Chu, Mary Jean Harrold, Prioritizing test cases for regression testing, Transactions on Software Engineering 27 (10) (2001) 929–948.
[28] Sreedevi Sampath, Renee Bryce, Sachin Jain, Schuyler Manchester, A tool for combinatorial-based prioritization and reduction of user-session-based test suites, in: International Conference on Software Maintenance: Tool Demo Track, September 2011, pp. 574–577.
[29] Sreedevi Sampath, Renee Bryce, Gokulanand Viswanath, Vani Kandimalla, A. Gunes Koru, Prioritizing user-session-based test cases for web application testing, in: International Conference on Software Testing, Verification and Validation, April 2008, pp. 141–150.
[30] Sreedevi Sampath, Valentin Mihaylov, Amie Souter, Lori Pollock, A scalable approach to user-session based testing of web applications through concept analysis, in: International Conference on Automated Software Engineering, September 2004, pp. 132–141.
[31] Sreedevi Sampath, Sara Sprenkle, Emily Gibson, Lori Pollock, Web application testing with customized test requirements—an experimental comparison study, in: International Symposium on Software Reliability Engineering, November 2006, pp. 266–278.
[32] Sreedevi Sampath, Sara Sprenkle, Emily Gibson, Lori Pollock, Amie Souter Greenwald, Applying concept analysis to user-session-based testing of web applications, Transactions on Software Engineering 33 (10) (2007) 643–658.
[33] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, Lori Pollock, Automated replay and failure detection for web applications, in: International Conference of Automated Software Engineering, November 2005, pp. 253–262.
[34] Sara Sprenkle, Lori Pollock, Holly Esquivel, Barbara Hazelwood, Stacey Ecott, Automated oracle comparators for testing web applications, in: International Symposium on Software Reliability Engineering, November 2007, pp. 253–262.
[35] Sara Sprenkle, Sreedevi Sampath, Emily Gibson, Lori Pollock, Amie Souter, An empirical comparison of test suite reduction techniques for user-session-based testing of web applications, in: International Conference on Software Maintenance, IEEE Computer Society, 2005, pp. 587–596.
[36] W. Eric Wong, Joseph R. Horgan, Saul London, Aditya P. Mathur, Effect of test set minimization on fault detection effectiveness, in: International Conference on Software engineering, May 1995, pp. 41–50.
[37] Shin Yoo, Mark Harman, Pareto efficient multi-objective test case selection, in: International Symposium on Softare Testing and Analysis (ISSTA), July 2007, pp. 140–150.
[38] Shin Yoo, Mark Harman, Using hybrid algorithm for pareto efficient multi-objective test suite minimisation, Journal of Systems and Software 83 (4) (2010) 689–701.