Web Application Testing with Customized Test Requirements — An Experimental Comparison Study

Sreedevi Sampath, Sara Sprenkle, Emily Gibson, Lori Pollock
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716
{sampath, sprenkle, gibson, pollock}@cis.udel.edu

Abstract

Test suite reduction uses test requirement coverage to determine if the reduced test suite maintains the original suite's requirement coverage. Based on observations from our previous experimental studies on test suite reduction, we believe there is a need for customized test requirements for web applications. In this paper, we examine usage-based customized test requirements for the test suite reduction problem in web application testing. We conduct an extensive experimental study to evaluate the tradeoffs between five classes of customized requirements with respect to reduced test suite size, program coverage and fault detection effectiveness. Our results show that the reduced suites' program coverage and fault detection effectiveness increases with the context or data associated with the reduction requirement. Based on our experimental results, we provide guidance to testers on the most useful test requirement for web applications in general and provide intuition on factors testers need to consider when selecting test requirements.

1. Introduction

Test requirement coverage is an important and well accepted measure for deciding when to stop testing, selecting test cases, and reducing test suites. Test coverage criteria define rules that impose requirements on a test suite, such that a test suite can be judged by the level at which it satisfies the coverage criterion. A set of test requirements can be described in terms of source code elements, design components, specification modeling elements, or elements of the input space [1]. In addition to traditional coverage-based criteria, test criteria have been proposed for web applications, including data flow criteria [13, 18], criteria based on link transitions [18] and standard graph criteria from a model of the application based on finite state machines [2].

One particular test criterion customized to web applications, and specifically used in test suite reduction, is base request coverage (`base`). A base request for a web application is the request type and resource location without associated data (e.g., *GET /servlets/authentication/Login.jsp*). In test suite reduction, the `base` coverage criterion requires that every base request in the original test suite be covered by the reduced test suite [20]. The test requirements to be satisfied for achieving `base` coverage are generated from the input space of user requests to the application in the form of base requests and optional name-value pairs (e.g., form field data). A reduced suite that provides full `base` coverage indicates that for every base request b covered by the original test suite, there is at least one test case in the reduced suite that covers b. The `base` requirement is used only for reduction. During reduced suite replay, the base request and the associated name-value pairs are replayed.

In our previous work in reduction of test suites formulated from field data, our experiments indicated that `base` can lead to suites with large percent reduction in test suite size while maintaining high program coverage and fault detection effectiveness [20, 25]. However, the `base` reduced suites were not *as effective* as the original suite in terms of program coverage and fault detection effectiveness. Intuitively, `base` test requirements are not as effective because the data associated with each request and the sequence in which users request resources is ignored during reduction. Furthermore, by covering the `base` requirement we cannot estimate the resulting underlying code coverage. For example, a single request can execute several servlets and Java/JSP classes before a response is sent back to the user.

During our experimental studies of test suite reduction for web applications [20, 21, 22, 25], we encountered several indicators prompting us to investigate alternative test requirements and the experimental investigation in this paper. Our experimental results revealed compelling evidence that motivate alternative test requirements for reduction.

First, we used base as the requirement for two different reduction techniques: Concept [20] and HGS [8]. In our studies, we found that neither technique produced reduced suites that covered as many statements or detected as many faults as the original suite [22, 25].

Our second observation is based on using base as a similarity measure for clustering and then reducing the original suite by selecting single test cases from a subset of clusters. For our reduction heuristic to be most effective, the test cases clustered together should cover the same program code. We found test cases clustered by concept analysis (see Section 3.2.3) with base requests as a similarity metric did not always cover the same program code [21]. A similarity metric that includes sequencing or data in addition to base requests may create clusters that have more common program code coverage, thus increasing the effectiveness of the reduction heuristic.

These two observations motivated us to develop more sophisticated test requirements that achieve higher program coverage and fault detection effectiveness than existing test requirements. Our intuition behind the more sophisticated requirements came from additional detailed analysis of our experimental results.

Our analysis of code covered by various reduced suites confirmed that certain code is covered and certain faults detected by a particular *request sequence*. For example, consider the difference in application responses when a user attempts to access a password-protected page. If the user is logged in, the web application immediately returns the requested resource. If not logged in, the user is redirected to a login page before accessing the password-protected page—resulting in a different sequence of requests. Unless the request sequence is considered during reduction, redirection code executed by the latter sequence will not necessarily be executed by the base reduced suite.

We also found that the *data associated with a request* can affect the program coverage and fault detection effectiveness of the reduced suite. Validating user input is important to evaluating the application's robustness with respect to user input and its security [15]. In our analysis, we found our applications perform validations, such as ensuring email addresses contain the symbol '@', checking for null values, trimming blank spaces, and checking for illegal characters. Unless the data associated with the request is considered during reduction, the corresponding validation code and faults located in validation code may not be covered by a base reduced suite.

By using more sophisticated request-based requirements, it is possible to gain program code coverage and fault detection equivalent to the original suite. The sequence in which requests are accessed and the data associated with requests needs to be considered as a requirement for reduction to create effective reduced suites. In this paper, we examine the tradeoffs between five classes of requirements based on request context (sequences) and data (name-value pairs) in terms of reduced suite size, program coverage, and fault detection. We provide guidance to testers by statistically analyzing the results across four subject applications and two reduction techniques and present the most suitable requirement for web applications in general. We were surprised by the requirement that emerged as the best overall requirement, when the desired qualities of reduced suites are considered together. Our results indicate that additional factors, such as the underlying code quality and application domain characteristics should be considered, when a tester is selecting an appropriate test requirement.

In Section 2 we describe five classes of test requirements customized to test suite reduction for web applications. In Section 3, we present our experiment to analyze the cost-effectiveness of the requirements for four subject web applications and two reduction techniques. Section 4 presents a thorough analysis of these results and provides guidance to testers on the factors to consider when selecting a requirement for reducing test suites for their web application. Related work is presented in Section 5. We conclude and outline future research directions in Section 6.

2. Customized Test Requirements

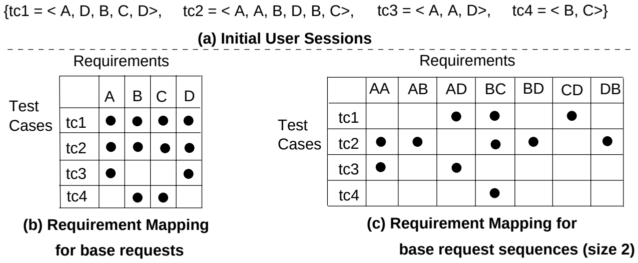
In this section, we examine five classes of requirements customized for web applications and based on field data: base, seqk, name, name_value and seqk_name. For web applications, we expect test cases to be sequences of requests, where a request is of the form: request type (GET/POST), the network data object or service being requested, and associated name-value pairs. In Table 1, we present the form of each customized test requirement and the requirements for the example test case
$\langle$*GET /bookstore/Login.jsp?name=xxx&password=yyy, GET /bookstore/ShoppingCart.jsp?item_no=1&book_name =ccc&price=60*$\rangle$.

The reduction criterion for any of the requirements can be stated as: for every customized 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 requirements can be further extended to consider sequences of requests of size *k* with names and values of input data, but the tradeoff between test suite effectiveness and size of the reduced test suite is likely to make the use of the requirement impractical in practice. For this reason, we consider only size 2 sequences of requests, seq2, and size 2 sequences of URLs and names, seq2_name, in our empirical evaluation.

A requirement mapping between each test case and the requirements that it covers is used by the reduction techniques to create reduced test suites that satisfy the given requirement. For each criterion selected, a mapping is created

Requirement	Form	Example Requirements
base	base request	{*GET /bookstore/Login.jsp,* *GET /bookstore/ShoppingCart.jsp*}
seqk	base request sequences of size *k*, where $k > 1$	{⟨*GET /bookstore/Login.jsp,* *GET /bookstore/ShoppingCart.jsp*⟩}, for *k=2*
name	base request and parameter names	{*GET /bookstore/Login.jsp?name&password,* *GET /bookstore/ShoppingCart.jsp?item_no?book_name?price*}
name_value	base request and parameter names and values	{*GET /bookstore/Login.jsp?name=xxx&password=yyy,* *GET /bookstore/ShoppingCart.jsp?item_no=1&book_name=ccc&price=60*}
seqk_name	size $k > 1$ sequences of base requests and parameter names	{⟨*GET /bookstore/Login.jsp?name&password,* *GET /bookstore/ShoppingCart.jsp?item_no&book_name&price*⟩}, for *k=2*

Table 1. Customized Test Requirements



{tc1 = < A, D, B, C, D>, tc2 = < A, A, B, D, B, C>, tc3 = < A, A, D>, tc4 = < B, C>}

(a) Initial User Sessions

Requirements

Test Cases	A	B	C	D
tc1	●	●	●	●
tc2	●	●	●	●
tc3	●			●
tc4		●	●	

(b) Requirement Mapping for base requests

Requirements

Test Cases	AA	AB	AD	BC	BD	CD	DB
tc1			●	●		●	
tc2	●	●		●	●		●
tc3	●		●				
tc4				●			

(c) Requirement Mapping for base request sequences (size 2)

Figure 1. Test Requirement Mapping

between the requirements and the test cases. In this paper, test cases are user sessions. A *user session* is a sequence of user requests in the form of base requests and name-value pairs (e.g., form field data). To generate user sessions, we use session cookies as identifiers when cookies are available. Otherwise, we say that a user session begins when a request from a new IP address reaches the server and ends when the user leaves the web site or the session times out. In our work, we consider a 45-minute gap between two requests from a user equivalent to a session timing out.

Figure 1 (a) shows an initial set of test cases *tc1, tc2, tc3, tc4*. Figure 1 (b) shows the requirement mapping for the base requirement, while Figure 1 (c) shows the mapping for the requirement seq2. In Figure 1 (b), test cases *tc1* and *tc2* appear to be equivalent, and a reduction algorithm may choose only of them (say, *tc1*) for the reduced suite. On changing the requirement to seq2, the test cases appear very different; this distinction may prove to be very important if *tc2*'s size 2 sequences, AA, AB, BC, BD, or DB, are fault-exposing sequences. However, the size of the requirement mapping increases with the requirement change.

3. Experimental Study

In our experimental study, we examine the use of each test requirement for test suite reduction. We use two reduction techniques HGS [8] and Concept [20]. We will evaluate the tradeoffs between the requirements with respect to reduced suite size, program coverage and fault detection effectiveness, replay costs and reduction time and space costs.

3.1. Expected Cost-benefit Tradeoffs

Our intuitions based on previous experimental studies led to several predictions about using the various test requirements for test suite reduction. The more context or data the requirement contains, the larger the requirement mapping. In this paper, we equate the complexity of the requirement to the data/context maintained in the requirement, (i.e., the more data associated with a given requirement, the more complex the requirement). The complexity relationship between the requirements is base $\leq$ seq2 $\leq$ seq2_name, base $\leq$ name $\leq$ name_value and name $\leq$ seq2_name. When applied to test suite reduction, requirements with larger mappings probably generate larger reduced suites. The reduced test suite size also depends on the selected requirement. We expect the trend in reduced suite size to follow the complexity relationship between the requirements, i.e., greater the complexity, larger the reduced test suite. We expect a similar trend in program coverage effectiveness and space and time costs for the different requirements. The tradeoffs in fault detection effectiveness are not easy to predict because faults depend on the input data and the oracle. However, we expect the fault detection effectiveness to be similar to the program coverage effectiveness. Since the requirement mappings are input to the test suite reduction techniques, we also expect the size of the requirements to affect time/space costs of the reduction techniques. The reduction techniques and the nature of the subject web application are likely to further impact the tradeoffs in the requirements. However, the focus of this paper is not to compare the reduction techniques but to compare the test requirements.

3.2. Independent Variables

To evaluate our research questions, the *independent variables* in our study are the subject applications, original test suites, requirements, and the test case selection technique.

3.2.1 Subject Applications and Original Test Suites

We used four subject programs with different characteristics: a conference website (Masplas), an open-source, e-

Apps	Classes	Methods	Statements	NCLOC	# Faults Seeded
Masplas	9	42	441	999	29
Book	11	385	5250	7791	39
CPM	75	172	6966	8947	135
DSpace	355	1543	27136	61729	50

Table 2. Subject Program Characteristics

Application	# US	Tot requests	Largest US (# requests)	Avg US (# requests)
Masplas	169	1107	69	7
Book	125	3694	160	29
CPM_ALL	890	12352	585	14
CPM_Y04	261	3719	152	14
CPM_Y05	629	8633	585	14
CPM_A04	58	1326	152	23
CPM_F05	203	2393	97	12
CPM_M05	105	1528	88	15
CPM_A05	168	2240	172	13
CPM_D05	356	4865	585	14
DSpace	1342	16275	1334	12

Table 3. Original Test Suite Characteristics

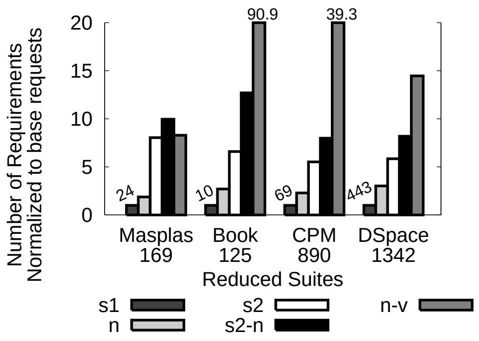


Figure 2. Number of Test Requirements Normalized to the Number of `base`

commerce bookstore (Book) [7], a course project manager (CPM), and a customized digital library (DSpace) [5]. The characteristics of the subject programs are shown in Table 2 and original test suite characteristics are in Table 3. In Table 2, NCLOC refers to the number of non-commented source lines of code. The number of statements in Table 2 also refers to the non-commented source lines of code but excludes code artifacts, such as braces, method and class headers. In Table 3, the second and third column represent the test suite size in terms of number of test cases, and the total number of requests accessed by the test cases, respectively. The last two columns represent the largest and average test case size in terms of requests in each test suite.

Masplas. Masplas is a web application developed by one of the authors 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. We collected 169 test cases when the workshop was held in 2005.

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. To collect 125 test cases, we sent email to local newsgroups and posted advertisements in the University's classifieds web page asking for volunteer users.

CPM. In CPM, 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 its state in a file-based datastore. We collected 890 test cases (CPM_ALL in Table 3) from instructors, teaching assistants, and students using CPM during the 2004-05 and 2005-06 academic years at the University of Delaware. To observe the effect of different test suite sizes in our experiments, we created six additional test suites by partitioning test cases by year (CPM_Y04 and CPM_Y05) and semester (CPM_A04 through CPM_D05).

DSpace. Our research group developed a customized web application for maintaining a digital publications library based on DSpace, an open-source digital repository system [5]. The application automatically generates sorted publications pages from a database that research group members maintain through a web application interface. A user can create dynamic views of publications by searching with different criteria. DSpace is written in both Java and JSP that deliver HTML content to the user and uses a PostGreSQL database and a filestore as the backend. We collected 1342 test cases after publicizing our digital library in August 2005 (DSpace in Table 3).

3.2.2 Test Requirements

Figure 2 shows the number of test requirements normalized over the number of unique base requests. The number of unique base requests for each application are shown in the graph above the `base` reduced suite. The graph indicates the relative size of the requirement mappings. As expected, with an increase in the complexity of the requirement, the size of the mapping increases.

3.2.3 Test Suite Reduction Techniques

We used two reduction techniques in our experiments, `HGS` and `Concept`. Both reduction techniques initially associate test cases with the requirements met by the test cases.

Harrold et al. (`HGS`) use a heuristic to select a reduced suite that approximates the smallest requirementrepresentative set of test cases, i.e., the minimum cardinality hitting set [8]. The criterion used by HGS is to cover all requirements covered by the original suite. To address non-determinism in HGS's heuristic to break ties, we create 100 reduced suites from the original suite with a particular requirement and select the suite generated most frequently.

In previous work [20], we presented a test case selection technique, Concept, based on clustering test cases by concept analysis. We developed a heuristic based on the concept lattice for selecting a subset of user sessions to be maintained as the current test suite [20]. Our heuristic for test case selection seeks to identify the smallest set of test cases that satisfies a certain requirement, as determined by the original test suite, while representing the set of requirements covered by the original test suite's test cases.

3.3. Dependent Variables

The *dependent variables* in our study are test suite size, program coverage and fault detection effectiveness, replay costs of the reduced suites, and time and space costs of test selection.

3.4. Methodology

Testing Framework. Field data is captured at the server and converted into test cases. We then reduce the original test suite using our Java implementation of HGS. For Concept, we use the concept analysis tool Concepts [12] to cluster test cases, and implemented our heuristic for test suite reduction in Java. The test cases are replayed by a customized version of HTTPClient [9]. Code instrumentation and coverage are measured with the publicly available tool Clover [4]. The test requirements, such as base and name, are used only for reduction. On replay, an entire user session (sequence of base requests and name-value pairs) is converted into a test case and replayed. The experimental framework is described in detail in [19, 24].

Fault Seeding. For fault detection experiments, graduate and undergraduate students familiar with JSP/Java servlets/HTML manually seeded faults in Book, CPM, Masplas and DSpace. In general, five types of faults were seeded in the applications—data store (faults that exercise application code interacting with the data store), logic (application code logic errors in the data and control flow), form (modifications to name-value pairs and form actions), appearance (faults which change the way in which the user views the page), and link (faults that change the hyperlinks location). We also seeded naturally occurring faults that were discovered by users during application deployment.

We seeded faults in CPM in two phases. Initially, we seeded 85 faults arbitrarily. We then added 50 faults to better differentiate the fault detection effectiveness of the test suites from the customized requirements. We compared the coverage of the reduced suites, characterized the differences, and seeded faults in code that may be exercised specifically by a reduced suite from a given requirement. The additional faults are broadly characterized as being exposed by request sequences (sequence-dependent), conditions on names (name-dependent) and conditions on values (value-dependent). If appropriate, we reclassified the initial 85 faults into one of these three categories.

Replay Mechanism and Oracle Comparators. We use the *with_state* replay mechanism, where application state is restored before the replay of every session in the reduced suite [24]. For Book, we use the *diff* oracle, which executes the UNIX utility *diff* on the output HTML pages. For applications CPM, Masplas and DSpace, which have real-time content, we use the *struct* oracle. The *struct* oracle compares the structure of the output HTML pages. Further details on the oracle comparators and replay mechanisms can be found in our previous work [24].

3.5. Threats to Validity

Because of the limited number of subject applications and available test cases, our results may not show significant differences in code coverage and fault detection for the different requirements. In practice the differences between the requirements may be larger than what we find for our subject applications. Further, due to the small number of subject applications, we cannot generalize our results to all web applications.

Although we seeded faults to model naturally occurring faults, our hand-seeded faults may be harder to detect than faults in practice. Andrews et al. [3] found that hand-seeded faults are more difficult to detect than naturally occurring faults. We believe we minimized bias introduced by hand seeding the faults as described in Section 3.4.

We selected our oracle comparator and replay technique to give a conservative estimate of the fault detection effectiveness of the reduced suites for each requirement. The *struct* oracle minimizes the number of false positives detected, at the risk of not detecting some of the faults. Using the *with_state* reduced suite replay strategy ensures the detected faults are caused by the reduced suite executing the fault, and not due to an inconsistent state issue.

4. Results and Analysis

In this section, we present our results and analysis comparing the different test requirements when applied to the problem of test suite reduction. We abbreviate the requirement base as **s1**, seq2 as **s2**, name as **n**, name_value as **n-v** and seq2_name as **s2-n**. In all our figures, unless otherwise specified, the x-axis represents the reduced suites of

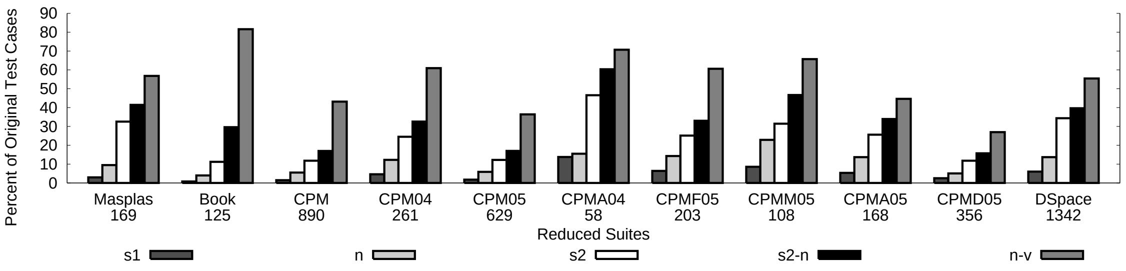


Figure 3. HGS: Reduction in Test Suite Size

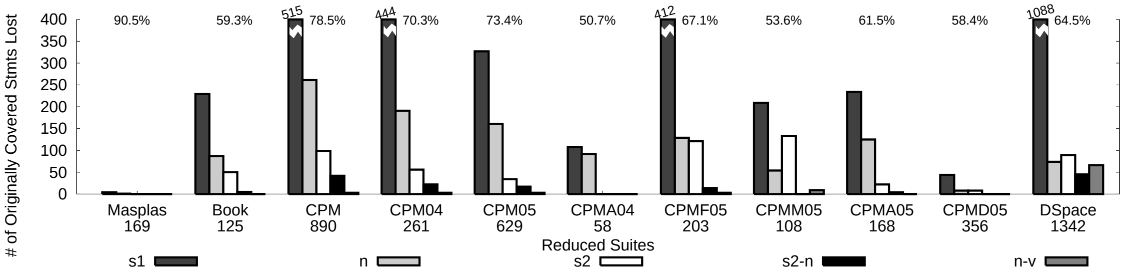


Figure 4. HGS: Statement Coverage Loss

the different subject applications. Due to space constraints, we do not present the graphs of reduced test suite size, program coverage, or fault detection for `Concept` reduced suites because all the trends are similar to those of `HGS`.

4.1. Reduced Test Suite Size

In Figure 3, the y-axis represents the percent reduction in test suite size by applying `HGS`. Figure 3 shows that `base` selects the smallest reduced suite for all the applications and test suites. As expected, with an increase in the complexity of the requirement, the reduced test suite size also increases. The test suite sizes follow the trends: **s1** $\leq$ **s2**$\leq$ **s2-n**, **s1** $\leq$ **n**$\leq$ **n-v**and **n**$\leq$ **s2-n**.

4.2. Program Coverage Effectiveness

Figure 4 shows `HGS` reduced suites' program coverage effectiveness. The y-axis shows the loss in statement coverage between the original suite's coverage and the number of actual statements covered by both the original and the reduced suite. We present statement coverage loss rather than method or conditional coverage loss because statement coverage is the finest coverage granularity we measured and best illustrates differences between requirements. The number above each cluster of reduced suites is the percent program code covered by the corresponding original suite. For both program coverage and fault detection effectiveness (Figures 4 and 5), if a reduced suite does not appear in the figure, the program coverage/fault detection of the suite is 100% of the original suite (i.e., no loss).

From Figure 4, `base` loses the most program code, as expected. For the more complex requirements, such as `name_value` and `seq2_name`, the reduced suites cover all code covered by the original suite, except in DSpace.

Across requirements, Masplas has the least difference in statement loss. Because `base` covers almost as many statements as the original suite, there is little benefit to using the other requirements. The Masplas test suite contains some infrequently accessed requests. We found that sessions that access these requests also covered error code—therefore `base` could cover almost all the application code just by covering all the requests. As expected, as the number of test cases in the reduced suite increases, the number of statements lost decreases. The difference between requirements is more prominent for CPM's larger original test suites.

In CPM, reduced suites covered different code. Consider an *if-then-else* statement that operates on parameter names from the request. The *then* branch is the default condition when the parameter names are correct. The *else* branch performs error checking, when the names are incorrect. In CPM we observed that the reduced suite from the `base` requirement sometimes failed to cover the *else* branch of the statement. However, both the *then* branch and the *else* branch were executed by the reduced suite from the `name` requirement. Since `base` considers only the base requests and not the names when reducing, it is likely to miss the test case with the incorrect parameter name.

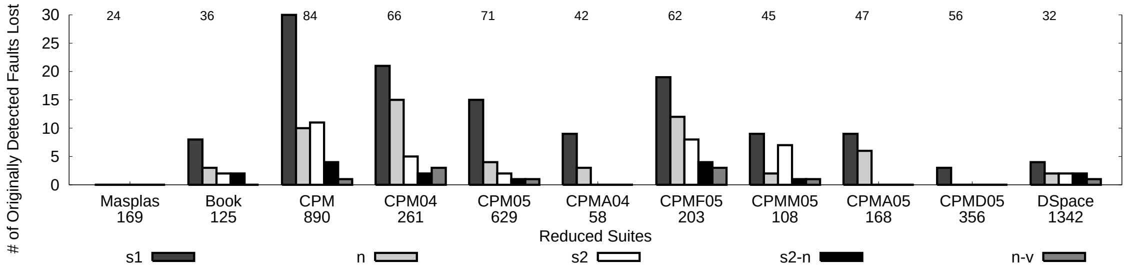


Figure 5. HGS: Fault Detection Loss

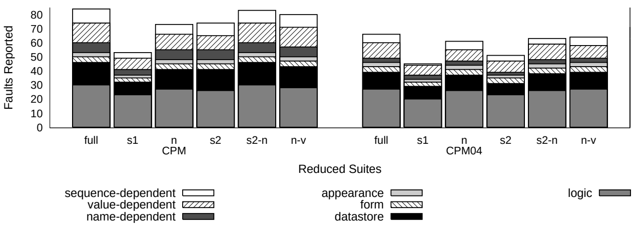


Figure 6. HGS: CPM Fault Detection Characterization

For DSpace, `base` loses the most statements when compared to the original suite. Contrary to the usual trend, we note that the `name` suite loses fewer statements than `seq2`. We attribute this behavior to certain application characteristics. In DSpace, users can search for publications with different search options. The search options result in sending different parameter names to the server code for processing. As a result, `name` covers code that handles the different parameter names. `seq2` loses more code than `name` because of the length of typical security-enabled sequences in DSpace. Most sequences that result in covering different code are longer than size 2. Thus, our `seq2` requirement is too coarse-grained to capture suites that cover error code, which is typically covered as a result of long sequences of requests. As the complexity of the requirement increases, the statement loss decreases. Because we have some problems maintaining state of the application during reduced suite replay for DSpace, the reduced suites cover different code from the original suite, such as error code. We plan to address the incorrect state problem in the future.

4.3. Fault Detection Effectiveness

In Figure 5, we present the results of our fault detection study. The y-axis is the number of faults detected by the original suite that are lost by the reduced suite. The number of faults detected by the original suite is represented above each cluster of reduced suites. From Figure 5, the trends in fault detection loss do not always mirror the coverage loss (suites CPM, CPM04, DSpace); for example, in suite CPM from Figure 4, `name` covers less code than `seq2`, but from Figure 5, `name` detects more faults than `seq2`. Even though a large area of additional code was covered, there may be no faults seeded in these areas.

The faults lost by the reduced suites decrease as the complexity of the requirement increases. Eventually, the most complex requirement has no loss in fault detection when compared to the original suite. As expected, Masplas did not show any difference in fault detection with the suites from the various requirements because the difference in statements covered was minimal. An example of a CPM fault found by `seq2` and not by `base` is only exhibited when a user attempts to access a password-protected page without first logging in. Unless one of the test cases selected to satisfy the `base` requirement contains the sequence of requests from a password-protected page to the login page, `base` will not detect the fault.

Unlike coverage, requirements that generate larger suites do not necessarily uncover more faults. For suite CPM in Figure 5, `seq2` missed more faults than `name`, even though `seq2` includes more test cases. To help explain the differences in fault detection, we analyzed the types of seeded faults uncovered by the suites generated by each requirement. In Figure 6 we show the types of faults detected by suites reduced from CPM and CPM04. Between the reduced suites there is little difference between the logic, datastore, form, and appearance faults detected. `base` does not detect as many faults that depend on request execution sequences, names, values, and logic faults, since their detection depends on data entered by the user. In CPM04, `seq2` detects as many sequence-dependent faults as the original suite but misses a name-dependent fault caught by `name`. We believe that `name` and `name_value` catch most of the sequence-dependent faults because different names or values can cause the user to access pages in different orders, thus exposing the sequence faults.

Grouping		Mean fom	Requirement
	A	76.988	name (**n**)
B	A	72.762	base (**s1**)
B	C	61.639	seqk (**s2**)
	C	54.361	seqk_name (**s2-n**)
	D	41.786	name_value (**n-v**)
df=105 MSE=200.2773 Critical Value of T=2.86756 Minimum Significant Difference=12.236 (α=.05)			

Table 4. Bonferroni means separation tests The mean is the mean figure of merit ($fom = redux * cvg * fd$) for each requirement across reduction technique and subject application (higher is better than lower). Requirements of the same group are not significantly different.

4.4. Time and Space Costs

The time and space to select test cases increases with the complexity of the requirements. For HGS the average time to generate reduced suites was 2 seconds. Figure 2 is indicative of the time and space costs for test suite reduction. In most cases the time and space requirements follow the relations, **s1** $\leq$ **s2** $\leq$ **s2-n**; **s1** $\leq$ **n** $\leq$ **n-v** and **n** $\leq$ **s2-n**.

We measured the speedup in replay time for the reduced suites as compared to the original suite. Due to space restrictions, we do not show speedup time here. Trends similar to test suite size were observed for replay speedup. The speedup ranged from 12.5 for CPM with base requirement to 1 for the seq2_name requirement.

4.5. Statistical Analysis Across Subject Applications

In this study, our goal is to analyze the cost-effectiveness of the reduced suites created by each requirement in terms of percent reduction ($redux$), percent coverage (cvg), and percent fault detection (fd), where the percent refers to percentage of the original suite. Intuitively, reduction by the ideal requirement produces reduced suites maximizing the percent coverage and fault detection of the original suite, as well as maximizing the percent reduction (minimizing the percent size) of the original suite.

We began our data analysis by plotting each dependent variable ($redux$, cvg, fd) against the test requirements. We noticed a positive correlation between the requirement complexity and the percent coverage and fault detection, as well as a negative correlation between the requirement complexity and the percent reduction[1]. These correlations indicate a tradeoff between the amount of reduction and the effectiveness, in terms of coverage and fault detection, of a requirement. To better analyze this tradeoff, we define a figure of merit, $fom = redux * cvg * fd$, as a means of comparing the cost-effectiveness of the requirements in terms of $redux$, cvg, and fd. The higher the fom for a requirement, the more cost-effective are the reduced suites created by that requirement.

To quantify which requirement is most cost-effective in terms of this figure of merit, we applied the Bonferroni multiple comparison method to contrast the mean fom for each requirement.[2] Table 4 shows the results of this analysis. Requirements are grouped together if the mean fom of the requirements are not significantly different. Groups higher in the table (with a higher mean fom) are more cost-effective than lower groups. For example, it is clear from Table 4 that the name and base requirements (group A in Table 4) are more cost-effective than name_value (group D in Table 4), because group A is higher in the table than group D. However, we cannot make any claims about the difference in cost-effectiveness between base and seq2 because they are in the same group, B.

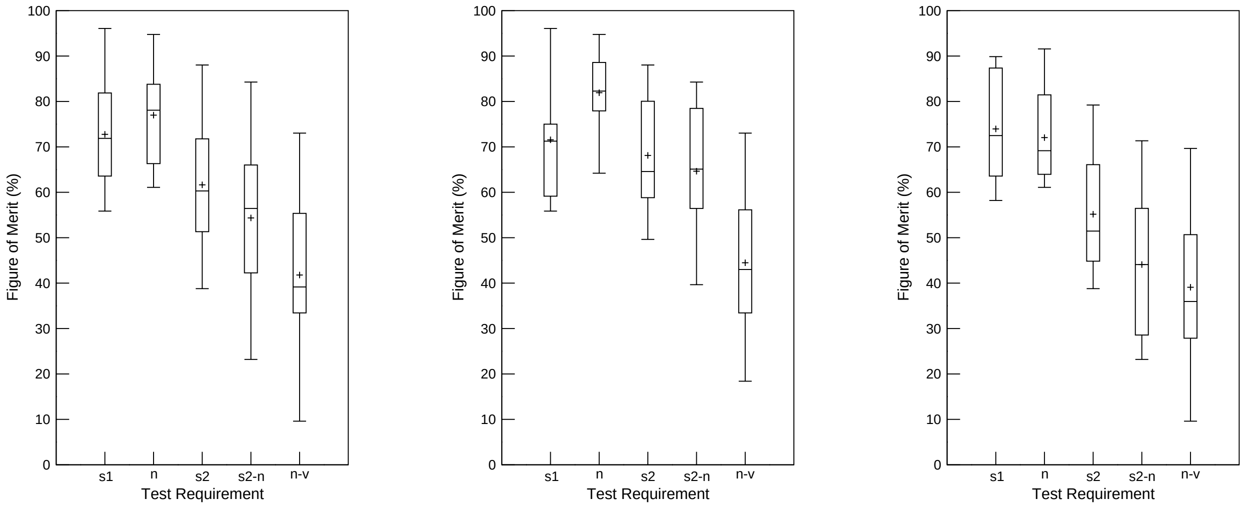
To better understand how the requirements relate, we plotted the mean fom of each requirement independent of subject application and reduction technique in Figure 7(a).[3] The center horizontal line within each box denotes the median, $+$ represents the mean, and outliers are represented by $\circ$. Based on the means and medians in Figure 7(a), we see the trends of Table 4—that base and name are more cost-effective than seq2, seq2_name, or name_value. Looking more closely at the requirements base and name, we note that requirement name has a slight advantage over base in terms of variability—namely, *name is more consistent in producing cost-effective reduced suites in terms of percent reduction, coverage, and fault detection than base.* Although we cannot verify this claim with statistical significance (because base and name are within the same group in Table 4), Figure 7(a) gives a strong indication that name is the most cost-effective technique across type of application or reduction technique employed.

To verify our conclusion that name is more cost-effective than base independent of reduction technique, we plotted the fom for Concept and HGS in Figures 7(b) and 7(c), respectively. For HGS, name has a clear advantage over every other requirement—the inner quartile range, mean, and median are clearly higher than the other requirements. However, for Concept, it is inconclusive which technique (base or name) is superior—the name distribution is skewed lower than that of base.

[1] The correlation coefficients for the requirement complexity and cvg, fd, $redux$ are .6640, .6208, and $-.7859$, respectively.

[2] We first used an analysis of variance F-test to verify that the mean fom for the requirements was significantly different before performing the contrasts. We chose the Bonferroni test to limit the experimentwise error rate.

[3] A boxplot is a standard statistical plot used in exploratory data analysis to display the symmetry, variability, and central tendency of a distribution [16]. The box represents 50% of the data and spans the width of the inner quartile range (IQR), with each whisker extending $1.5*$IQR beyond the top and bottom of the box.



(a) All Reduction Techniques

(b) HGS Reduction Only

(c) Concept Reduction Only

Figure 7. Comparison of figures of merit (cost-effectiveness) of each requirement across applications (+ represents the mean, outliers are represented by ○)

4.6. Observations

We cannot quantitatively evaluate the effect application characteristics have on the cost-effectiveness of each requirement because we have insufficient data to compare requirements across applications. However, we can offer some qualitative insights into the interaction of these application characteristics and the test requirements from our exploratory data analysis.

We have found that requirements that capture context and data, such as name, name_value and seq2_name, increase reduced suite effectiveness for applications that exhibit significant control dependence on names and values. For example, if the control flow in the underlying application code is *not* dependent on the presence of particular names or values, then the tester can use base to obtain reasonably good program coverage and fault detection. However, if the form field names and values affect the application control flow, such as causing the application to execute validation code depending on parameter names and values, a requirement such as name or name_value will be necessary to achieve high levels of program coverage and fault detection effectiveness. An example can be seen in our DSpace application, where the same base request displays different information based on the search parameters specified in name-value pairs.

Context-dependent requirements (seq2, seq2_name), on the other hand, increase reduced suite effectiveness for applications that have specific user patterns intended by the developer. Although the application might not behave as anticipated, a web application user can access any valid URL of an application at any time. Some applications have predefined URL sequences that users follow if they follow links displayed by the application. For example, to view the details of a publication in DSpace, a user must first visit the search results page. Reducing with the seq2 or seq2_name requirements will include any unanticipated sequences of user requests and, therefore, potentially cover more code and find more faults.

5. Related Work

Multiple strategies exist to test web applications, such as link and form testers [23], structural testing tools [2, 13, 14, 15, 18] and user-session-based testing techniques [6, 10, 17]. Other techniques such as [10] emulate portions of browser behavior and test correctness of returned pages. When used in conjunction with JUnit [11], HttpUnit allows writing test cases for web applications. Web-King [17], a capture-replay tool for testing web applications, allows testers to record interactions with the application and replay the recorded tester input as test cases.

Offutt et al. [15] present an approach to generate test cases with the goal of uncovering faults and security vulnerabilities in server software due to user's ability to bypass client-side input validation. They model web applications and present techniques to generate bypass test cases for value level, parameter level, and control flow level bypass testing. Some of our requirements, such as name_value and seq2, are similar to value and control flow bypass testing, respectively. Bypass testing is complementary to user-session-based testing. Our user-session-based testing approach uses field data to generate test cases that represent common application usage in the testing of later versions of the server software, whereas bypass testing focuses on users who bypass the client-side input validation, and thus the robustness and security holes in the server applications from less normal use. Our test cases are synonymous with typical usage of the application, whereas, bypass testing generates malicious test cases.

6. Conclusions and Future Work

In this paper, we defined a set of customized requirements for testing web applications and evaluated their applicability to the problem of test case selection. Our results indicate that requirements that have more data or context associated with them cover more statements and detect more faults. The faults detected by the various reduced suites change with the requirement used to select the test suite. We found that `base` and `name` select reduced suites that are consistently small in size and effective. A tester might want to choose `name` because it is more consistent than `base`. `seq2_name` might be a good alternative if test suite size is not a concern to the tester, but maximizing program coverage and fault detection are more important. A tester can also choose the applicable requirement by the application usage and code properties. The tradeoff however is the test suite size and the time and space requirements to generate and replay the test suite. In the future we plan to apply our requirements to other web applications and investigate additional uses. We also plan to further evaluate the applicability of our requirements to certain types of web applications by classifying applications based on their usage.

Acknowledgments. We thank Frank Zappaterrini for building the replay tool and customizing DSpace, and Stacey Ecott for seeding faults in Masplas. We also thank Dr. Allen Gibson from the Stillman School of Business at Seton Hall University for his assistance with the statistical analysis.

References

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. In preparation, 2006.

[2] A. Andrews, J. Offutt, and R. Alexander. Testing web applications by modeling with FSMs. *Software and Systems Modeling*, 4(3):326–345, July 2005.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering*, 2005.

[4] Clover: Code coverage tool for Java. `http://www.cenqua.com/clover/`, 2006.

[5] Dspace federation. `http://www.dspace.org/`, 2006.

[6] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II. Leveraging user session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, May 2005.

[7] Open source web applications with source code. `http://www.gotocode.com`, 2006.

[8] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[9] HTTPClient V0.3-3. `http://www.innovation.ch/java/HTTPClient/`, 2006.

[10] HttpUnit. `http://httpunit.sourceforge.net`, 2006.

[11] Junit. `http://www.junit.org`, 2006.

[12] C. Lindig. Concepts tool. `http://www.st.cs.uni-sb.de/~lindig/src/concepts.html`, 2006.

[13] C.-H. Liu, D. C. Kung, and P. Hsia. Object-based data flow testing of web applications. In *The First Asia-Pacific Conference on Quality Software*, 2000.

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

[15] J. Offutt, Y. Wu, X. Du, and H. Huang. Bypass testing of web applications. In *International Symposium on Software Reliability and Engineering*, 2004.

[16] R. L. Ott and M. Longnecker. *An Introduction to Statistical Methods and Data Analysis*. Duxbury, Fifth edition, 2001.

[17] Parasoft WebKing. `http://www.parasoft.com`, 2004.

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

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

[20] S. Sampath, V. Mihaylov, A. Souter, and L. Pollock. A scalable approach to user-session based testing of web applications through concept analysis. In *International Conference on Automated Software Engineering*, 2004.

[21] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. Souter. Analyzing clusters of web application user sessions. In *Third International Workshop on Dynamic Analysis*, May 2005.

[22] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. Souter. Applying concept analysis to user-session-based testing of web applications. Technical Report 2006-329, University of Delaware, 2006.

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

[24] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *International Conference of Automated Software Engineering*, 2005.

[25] S. Sprenkle, S. Sampath, E. Gibson, L. Pollock, and A. Souter. An empirical comparison of test suite reduction techniques for user-session-based testing of web applications. In *International Conference on Software Maintenance*, 2005.