Integrating Customized Test Requirements with Traditional Requirements in Web Application Testing

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

ABSTRACT

Existing test suite reduction techniques employed for testing web applications have either used traditional program coverage-based requirements or usage-based requirements. In this paper, we explore three different strategies to integrate the use of program coverage-based requirements and usage-based requirements in relation to test suite reduction for web applications. We investigate the use of usage-based test requirements for comparison of test suites that have been reduced based on program coverage-based test requirements. We examine the effectiveness of a test suite reduction process based on a combination of both usage-based and program coverage-based requirements. Finally, we modify a popular test suite reduction algorithm to replace part of its test selection process with selection based on usage-based test requirements. Our case study suggests that integrating program coverage-based and usage-based test requirements has a positive impact on the effectiveness of the resulting test suites.

1. INTRODUCTION

The demand for reliability in web applications is fueled by their growing usage and the potentially huge losses that can be incurred upon unexpected failures [10, 12]. Similar to traditional testing methodology, the *quality* of test suites for web applications can be measured by test requirement coverage, which can be used to decide when to stop testing, which test cases to select, and how to reduce a test suite to eliminate redundant testing effort. In addition to traditional coverage-based criteria, a number of test criteria have been proposed for web applications, including data flow criteria [9, 13], criteria based on link transitions [13], and standard graph criteria from a model of the application based on finite state machines [2].

Designed primarily to be used in test suite reduction, *usage-based test requirements* are generated from the input space, namely, the sequences of user requests to an application in the form of base requests and optional 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/Login.jsp`). The usage-based test requirement, `base`, is derived from the application usage data and customized to web applications. The `base` requirement coverage criterion requires that every base request in the original test suite be covered by the reduced test suite [15]. 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. Unlike usage-based testing or operational-profile testing, where testing focuses on usage scenarios and the probability of the occurrence of usage scenarios [19], *usage-based test requirements* are a set of test requirements based on usage data that the reduced suite is expected to satisfy. Usage-based test requirements are similar to operational profiles since both reflect how the software has been used in practice. However, operational profiles are a quantitative characterization of how a system will be used [11], such as the probability of the occurrence of a certain profile, whereas usage-based test requirements can be used to show the actual system usage.

Recently, we investigated more sophisticated usage-based test requirements for test suite reduction that consider sequences of requests and data associated with the requests [16]. In our previous work [16], we experimentally evaluate the tradeoffs between the different usage-based requirements with respect to reduced test suite size, program coverage and fault detection effectiveness. The study confirmed our intuition that although using usage-based requirements is less expensive than program-based test requirements, the effectiveness of any usage-based requirement depends on the characteristics of the input data, such as the variation in the different values for the data associated with the requests. By covering all usage-based requirements, the tester cannot estimate how much of the underlying code will be covered. For example, a single `base` request can execute several servlets and Java/JSP classes before a response is sent back to the user. Thus, a balance between the cost of generating the reduced suite and the effectiveness of the reduced suite might be achieved by integrating the use of customized usage-based test requirements with coarse-grained, program coverage-based requirements, such as method coverage, within test suite reduction tasks for web applications. Black-box and white-box testing approaches have been integrated to improve the effectiveness of the testing strategy for object-oriented programs [3, 5]; however, to our knowledge little research has been done on combining requirements, particularly program-based and usage-based, for test suite reduction.

The main contributions of this paper are strategies to integrate usage-based and program coverage-based test requirements and case studies of

- empirically comparing program coverage-adequate test suites by using usage-based test requirements to select the test suite that best captures usage coverage
- combining traditional coverage-based requirements with usage-based requirements for test suite reduction and empirically evaluating the effectiveness of the reduced suites
- modifying the test suite reduction algorithm by Harrold et al. (`HGS`) [7], a well known test reduction technique, using program coverage-based requirements for most decisions and utilizing usage-based requirements information to break ties, and empirically evaluating the effectiveness of the reduced suites

Section 2 presents an overview of the usage-based test requirements customized to web applications. We also describe background on user-session-based testing since we perform our comparisons using field data within a user-session-based testing framework. In Section 3 we describe the three strategies for integrating traditional program coverage-based requirements and usage-based requirements. In Section 4 we describe our case studies, which include empirical evaluation and results analysis based on a deployed web application with gathered usage data. We summarize conclusions and describe future directions in Section 5.

2. BACKGROUND

2.1 User-session-based Testing

In this paper, we focus on applying requirements customized for web applications to user-session-based test suite comparison and reduction. In user-session-based testing, the web application is deployed and interactions between the user and the application are captured and replayed as test cases to test the application [6, 15]. Each test case is a *user session*, and a test suite is a set of 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). A user session begins when a new IP address makes a request to the server and ends when the user leaves the web site or the session times out. User sessions are uniquely identified by the user's IP address and requests that arrive from the same IP but after an inactivity interval of 45 minutes are considered a new session.

2.2 Usage-based Test Requirements

Test coverage criteria define rules that impose test 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 this section, we outline the five classes of requirements customized for web applications: `base`, `seqk`, `name`, `name_value` and `seqk_name`. The customized test requirements are presented in detail in [16]. 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 general form of each customized test requirement, and the requirements for the example test case

⟨*GET /bookstore/Login.jsp?name=xxx&password=yyy, GET /bookstore/ShoppingCart.jsp?item_no=1&book_name =ccc&price=60*⟩.

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, `seqk_name_value`, but the tradeoff between the marginal improvement in test suite effectiveness and the large size of the reduced test suite is likely to make use of the `seqk_name_value` requirement impractical in practice. For the same reasons as above, we consider only size 2 sequences of URLs, `seq2`, and size 2 sequences of URLs and names, `seq2_name`, in our empirical evaluation.

3. STRATEGIES FOR INTEGRATION

In this section, we present strategies for integrating customized test requirements with traditional test requirements to increase the effectiveness of test suites for web applications.

3.1 Test Suite Comparison

The customized usage-based requirements can be used to compare test suites shown to be adequate by a traditional program coverage requirement, such as statement, method, or branch coverage, to identify the test suite that better portrays the usage information represented by the usage-based requirements. For example, if two test suites, *T1* and *T2*, are statement-coverage adequate, the two suites can be distinguished by their coverage of the customized usage-based requirements. If *T1* satisfies more of the usage-based requirements than *T2*, then *T1* is a better test suite for testing the application because it satisfies both the program-based coverage criterion and represents application usage better than *T2*.

Comparing test suites (not necessarily reduced test suites) based on usage-based requirements is also useful when the tester has multiple test suites at her disposal. Since gathering usage-based requirements in less expensive than collecting program coverage-based requirements, if the tester has time to execute only one test suite, the tester can compare the usage-based requirements covered by the different test suites and select the suite that covers the most requirements. We believe that by covering more customized requirements, the test suite is likely to detect more faults than test suites with lesser customized requirements coverage.

Another scenario where customized requirements are useful is in software engineering experimentation. In our previous experiments [18], we implemented and evaluated the effectiveness of three reduction techniques, `HGS` [7], `Greedy` and randomly generating program coverage adequate test suites. Due to non-determinism in `HGS`, `Greedy` and random selection, we reduced the original suite 100 times. We executed all of the 100 reduced suites and computed program coverage and fault detection effectiveness. In the interest of time, however, it is important to have the ability to select a

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

representative suite from these 100 suites. In previous work, we have used statistical approaches to select the representative suite, such as selecting the mode test suite (suite that occurs most frequently), the minimum suite (suite with the least sessions), or the maximum suite (suite with the most sessions). We believe that customized requirements can be used to select an effective representative reduced suite from a large pool of sessions. The test suite that covers the most usage-based requirements is likely to be the best reduced suite.

By selecting the suite that better represents usage information, the tester can combine both traditional requirements and usage information in one reduced suite. We believe a program coverage requirement-adequate test suite that covers more of the customized requirements is likely to detect more faults than a suite that covers less of the requirements.

3.2 Combining Requirements for HGS

Black et al. [4] suggest that multi-criteria decision making allows evaluating multiple objectives based upon several, usually conflicting, criteria. By combining traditional program coverage-based requirements with customized usage-based requirements, we can create a reduced test suite that meets both sets of requirements and reflects both the desired underlying program coverage as well as actual web application usage. However, combining two sets of requirements will result in the creation of a large number of requirements that need to be met by the reduced test suite. The increased number of overall requirements will most likely create a larger reduced test suite than reducing with program coverage-based requirements alone. A tester needs to consider the tradeoff between integrating usage requirements and test suite size. By integrating customized usage-based requirements and traditional program coverage-based requirements, at the potential cost of increased suite size, the reduced test suite should detect more faults and cover more code than a test suite reduced by traditional program coverage metrics alone. For requirements-based reduction techniques the input is the union of the customized and program coverage-based requirements.

3.3 HGS with Usage-based Tie-breaking

In our third approach to integrating customized usage-based test requirements with traditional program coverage-based requirements, we modify Harrold et al.'s [7] reduction technique (`HGS`) to incorporate the use of usage-based test requirements into its nondeterministic component.

Harrold et al. [7] developed a test suite reduction technique (`HGS`) with a heuristic that selects a representative test set from the original test suite by approximating the optimal reduced set. Determining the optimal reduced set is an NP-complete problem [7]. Before the algorithm can reduce the test set, test cases must be associated with the requirements they meet. The requirement's cardinality is the number of test cases that covers the requirement. After a test case is added to the reduced set, the algorithm marks the test case's covered requirements. The `HGS` algorithm then selects the most frequently occurring test case among the unmarked test cases with lowest requirement cardinality, i.e., the test case that covers the most unmarked requirements. `HGS` stops selecting test cases when each test requirement is covered by at least one test case in the reduced set.

Traditionally, in the case of ties, the algorithm chooses the test case that occurs most frequently at the next higher requirement cardinality. Breaking ties repeats until cardinality equals the maximum cardinality, at which point the algorithm randomly selects from the tied test cases. We implemented an algorithm, `Modified HGS`, which modifies `HGS`'s tie-breaker component to use the usage-based requirement information.

When a test case is selected for the reduced set, `Modified HGS` marks the test case's covered program coverage-based and usage-based test requirements. In case of ties, instead of considering the test cases that occur most frequently at the next higher program coverage-based cardinality, `Modified HGS` computes the number of usage-based requirements covered by each tied session and selects the session that covers the most uncovered usage-based requirements. If all tied sessions cover the same number of usage-based requirements, a session is selected at random. We expect `Modified HGS` to produce more effective suites than `HGS` when a large number of ties are encountered.

4. CASE STUDY

We evaluate the test suite comparison and reduction strategies with a case study on one subject web application.

4.1 Research Questions and Hypotheses

Comparing Test Suites

Question 1. What is the relation between fault detection effectiveness of the compared suites that were reduced by program coverage-based requirements and the number of covered usage-based requirements? The answer to this question will indicate if a set of reduced suites that meet the same program coverage requirements can be distinguished by usage-based requirements. If the reduced suites can be distinguished, can a tester select effective suites from this comparison?

Hypothesis 1. Given two test suites that meet the same program coverage-based test requirements, the test suite that covers the most usage-based test requirements will have the highest fault detection effectiveness.

Combining Test Requirements
Question 2. How do the reduced test suite size, program coverage and fault detection effectiveness of a reduced suite from the combined test requirements compare to reduced suites created from using either the traditional program coverage-based requirement or the customized usage-based requirement in isolation?

Hypothesis 2. The test suite size and program coverage effectiveness of the reduced test suites generated to satisfy the combined requirements will be larger than the suites from traditional program-based criteria. The size and program coverage effectiveness of suites from combined requirements will be larger than suites from usage-based criteria alone. Fault detection effectiveness is not easy to predict because it depends on the adopted fault seeding strategy. However, we expect trends similar to program coverage effectiveness.

`Modified HGS`
Question 3. How do the reduced suites created by `Modified HGS` perform with respect to reduced test suite size, program coverage and fault detection effectiveness when compared to `HGS` program-adequate reduced suites or the reduced suites from the customized requirements alone?

Hypothesis 3. By incorporating usage-based requirements as tie-breakers, `Modified HGS` creates test suites with better program coverage and fault detection effectiveness than `HGS`. The trends in test suite size are harder to predict, but we believe test suite size will remain the same for the `Modified HGS` and `HGS` reduced suites. `Modified HGS` reduced suites will be smaller than suites from the customized requirements and more effective.

4.2 Independent and Dependent Variables

The *independent variable* in the comparing test suites study is the test suite selection mechanism. The *independent variable* for the combining test requirements study is the program-based and usage-based test requirements and the *independent variable* for the `Modified HGS` study is the reduction technique (i.e., `HGS` and `Modified HGS`). The *dependent variables* in our study are test suite size, program coverage and fault detection effectiveness.

4.3 Subject Application and Original Test Suite

We deployed a course project manager (CPM) and collected user sessions for our experiments. The application characteristics and the test suite characteristics are presented in Table 2. 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. CPM also sends emails to notify users about account creation, grade postings, and changes to reserved time slots. Users interact with an HTML application interface generated by Java servlets and JSPs. CPM manages its state in a file-based datastore. We collected 890 test cases from instructors, teaching assistants, and students using CPM during the 2004-05 and 2005-06 academic years at the University of Delaware.

Number of Classes	75
Number of Methods	172
Number of Statements	6966
NCLOC	8947
Number of Seeded Faults	135
Number of User Sessions	890
Total Number of URLs	12352
Largest Session Size (number of URLs)	585
Average Session Size (number of URLs)	14

Table 2: CPM Application and Test Suite Characteristics

4.4 Experimental Setup

Fault Seeding. Graduate students familiar with JSP, Java servlets, and HTML manually seeded faults in CPM. 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 that change the way the page appears to the user), and link (faults that change the hyperlinks location). We also seeded naturally occurring faults that were discovered by users during application deployment. Based on our experience, we found that randomly seeding faults in object-oriented code is difficult. Because of the object-oriented nature of the code, a randomly selected location in code may not necessarily be suitable for a fault. Thus, we seeded faults arbitrarily in the application.

We seeded faults in two phases. Initially, we seeded 85 faults arbitrarily into the application. To better differentiate the fault detection effectiveness of the test suites from the customized requirements, we then added 50 more faults. 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.

For the fault detection and coverage experiments we used the framework from our previous studies [14, 17]. We describe experiment-specific methodology in later subsections.

Replay Mechanism and Oracle Comparator. We implemented a customized replay tool using HTTPClient [8], which handles GET and POST requests, file uploads and maintains the client's session state. The replay tool takes as input the test cases in the form of sequences of base requests and name-value pairs and replays the sessions to the application, while maintaining the client state. We adopt the `with_state` replay mechanism during reduced suite replay. In our current implementation of `with_state` replay, the original suite is replayed and the datastore state after each session is stored. When the corresponding session appears in the reduced suite, the state of the application is restored to the state before the current session. In addition, the initial state of the application is stored before any user sessions are logged. Before replay, the state of the application is set to the initial state.

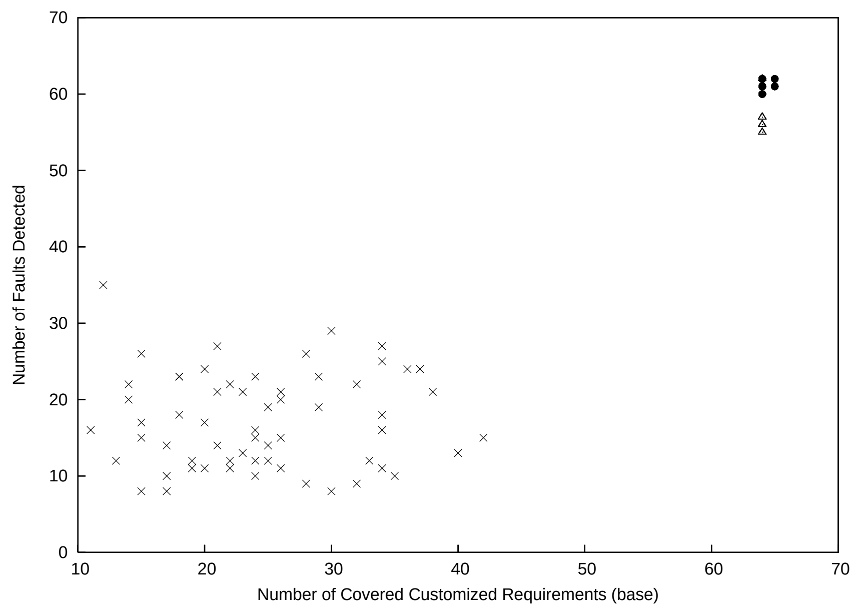

Figure 1: Comparing Reduced Test Suites For `base`

Faults are seeded in the application (one fault per version), and the original and reduced suites are replayed. The non-fault seeded version of the application is considered to be the clean version of the application. Execution of the non-fault seeded version of the application creates the expected output. The actual output is gathered on executing the fault seeding versions of the application. We use the *struct* oracle comparator, which compares the structure of the output HTML pages. Further details on the oracle comparator and replay mechanism can be found in our previous work [17].

4.5 Threats to Validity

We cannot generalize our current results because we are using only one subject application. The number of test cases and their sizes used in our study is also relatively small when compared to commercially deployed web applications. The original test suite is the originally logged user sessions. All test suite reductions are applied on the usage-data and therefore the adequacy measure used is with respect to the original suite. Our faults may not be seeded evenly across all the classes because of the arbitrary nature of the fault seeding approach, therefore, the fault detection results may not mirror the program coverage results. The fault seeding strategy we adopted for the 50 faults seeded in the second fault seeding phase is a threat to the validity of our fault detection experiments as described in Section 4.4.

4.6 Methodology and Results

4.6.1 Comparing Test Suites

Methodology. We executed two reduction techniques, `Greedy` and `HGS`, each 100 times to generate a total of 200 reduced suites. The average sizes of the reduced test suite created by `Greedy` and `HGS` were 9 and 10, respectively. We then created 60 test suites by randomly selecting test cases such that 30 test suites were of size 9, and the other 30 test suites were of size 10. We will refer to this technique as `Random`. The details of the reduction techniques are described in [18]. We computed the fault detection effectiveness of each of the 260 test suites. Method-adequacy, the least expensive program coverage adequacy measure, was used in our experiments. Since the original suite of test cases is created from field data, the reduced suite is method adequate with respect to the original suite based on field data. The `Random` suites are not method-adequate.

Results and Analysis. Figures 1, 2, and 3 show the results for comparing reduced suites for all the reduction techniques. The x-axis shows the number of customized requirements covered by each test suite, and the y-axis shows the number of detected faults. The original suite detects 84 faults. Figure 1 shows results for comparing suites with usage-based requirement `base`, Figure 2 shows comparison with `seq2`, and Figure 3 shows comparison with `name`. The solid circles represent `Greedy` reduced suites, the triangles represent `HGS` reduced suites and the x's represent the `Random` suites.

In Figure 1, for all the suites, particularly the `Random` test suites, we observe that as the number of covered `base` requirements increases, the number of detected faults increases, in general. The range of detected faults for `HGS` and `Greedy` suites with the same number of covered `base` requirements is due to the coarse granularity of the `base` requirement. Since the `base` requirement is coarse-grained, multiple test suites that cover the same number of `base` requirements vary in their fault detection effectiveness. This is expected, since different name-value pairs associated with a request may expose different faults. The fault detection effectiveness is thus affected by the data associated with a request.

Figures 2 and 3 show the expected trends more prominently. As the number of covered `seq2` and `name` requirements increases, the number of detected faults increases. We further investigated the unique clustering of the results in these graphs. `HGS` selected test suites that detected between

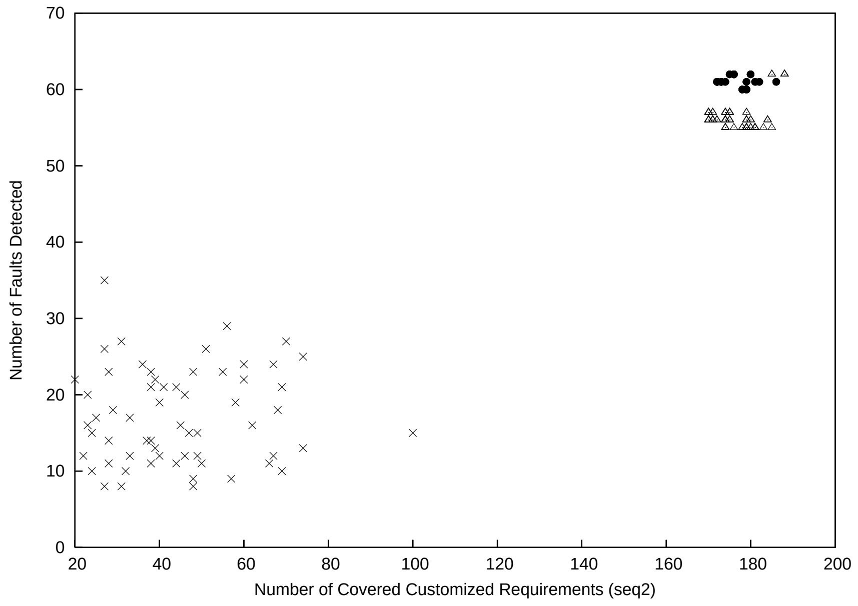

Figure 2: Comparing Reduced Test Suites For `seq2`

55 and 60 faults and a few suites detected 65 faults, `Greedy` selected test suites between 60 and 65 faults, and `Random`'s fault detection varies between 9 and 35 faults. From Figures 2 and 3 we observe from the fault detection effectiveness for `Random` that as the number of covered customized requirements increases, the fault detection of the suite increases. These results suggest that the nature of the test suite in terms of its constituent test cases has an effect on the results of our study. Test suites generated by `Random` vary greatly in terms of the actual sessions in the suites and are more likely to show the expected trend (i.e., greater the number of covered usage-based test requirements, greater the number of faults detected). Since the 200 reduced suites produced by `HGS` and `Greedy` have little variance in terms of the test cases in the suites (i.e., for the 100 reduced suites generated by `HGS`, 70% of the test cases appeared in all the 100 reduced suites) the effect is not as pronounced. Contrary to our hypothesis, for `HGS` and `Greedy` reduced suites, more customized test requirement coverage does not necessarily relate to an increase in fault detection because the techniques generate reduced suites with little variation in constituent test cases.

The trend remains the same across all the usage-based requirements (we show `seq2` and `name` here). The similarity in trends suggests that the type of usage-based requirement does not have an effect on the detected faults. However, the lack of trends in different requirements could also be due to our fault seeding approach. If we had faults that would be detected only by a certain requirement, say `seq2`, then `seq2` is likely to show a different trend than the other test requirements. Since our fault categories are not strict partitions, i.e., a fault detected by suites from `seq2` requirement can also be detected by suites from `name_value` requirement, we do not observe a difference in the trends in these experiments. The lack of trends between requirements could also depend on the oracle comparator used. We used the *struct* oracle in our experiments [17], which has no false positives for our subject application, but can have false negatives. If there were a more precise oracle, we may see trends in different requirements. Figures 1, 2 and 3 also suggest that method coverage adequacy is not effective for web applications, since `HGS` and `Greedy` reduced suites detected at least 25% fewer faults than the original suite for our subject application.

4.6.2 *Combining Test Requirements*

Methodology. We combined the method requirement with our customized requirements and input the complete requirement set into the `HGS` reduction technique. We generated 100 reduced suites and computed the reduced test suite size and fault detection effectiveness of the suites. We computed the program coverage effectiveness of the mode test suite, i.e., the test suite that occurs most frequently among the 100 suites.

Results and Analysis. For all the figures in this experiment (Figures 4, 5 and 6), the x-axis shows the reduced suites created by using the traditional coverage requirement—statement (S), conditional (C) and method (M), with `HGS`. The x-axis also shows reduced suites created by combining the customized requirement with the traditional method requirement in `HGS`. The reduced suites created by the customized requirements alone are also shown here. We abbreviate the customized requirement `base` as *S1*, `seq2` as *S2*, `name` as *N*, `name_value` as *N-V* and `seq2_name` as *S2-N*. To show a combination of the traditional and customized test requirements, the above abbreviations are separated by a plus sign (e.g., "M+S1" represents the combination of method and `base` requirements). The lines on each bar show the range of coverage for the 25th and 75th percentiles of nondeterministic `HGS` reduced suites. We use the same x-axis labeling for all graphs in this experiment. The results of using customized requirements alone are presented in previous work [16].

Figure 4 shows the reduced test suite size for the different requirements. In Figure 4, the left y-axis shows the reduced

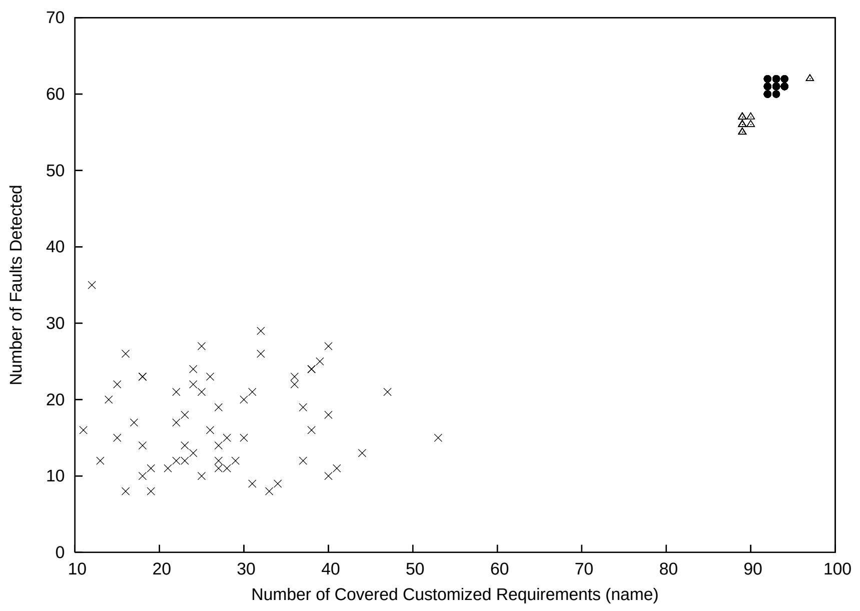

Figure 3: Comparing Reduced Test Suites For `name`

test suite size as a percent reduction in number of test cases and the right y-axis shows the reduced test suite size in terms of total number of requests in the reduced suite. As expected, a combination of method and the customized requirement creates reduced suites larger than `HGS`-method. As the complexity of the requirement increases, the size of the combined reduced test suite increases. Similarly, with an increase in the complexity of the customized requirement, the size of the reduced test suite created by using the customized requirement alone increases.

To show the differences between the requirements clearly, in Figure 5, the y-axis shows the number of statements covered by the mode reduced suites. The horizontal line in Figures 5 and 6 represents the program coverage/fault detection of the `HGS`-method reduced suite. The original suite covered 5468 statements. Obviously, the `HGS`-statement suite will cover all statements covered by the original suite. `HGS`-conditional also creates a reduced suite with little loss in program coverage effectiveness. However, using `HGS`-method creates a reduced suite that loses around 450 statements. By using a combination of both method and the customized requirements, the number of statements covered increases. Interestingly, combined requirements have lower statement coverage than using just the customized requirement. However, program coverage is consistent with the respective reduced suite sizes. Apart from increasing statement coverage with an increase in complexity of the requirement, we do not observe any relation between the type of requirement and the program coverage effectiveness of the requirement.

Our results suggest that if a tester is planning on using `HGS`-method to create reduced suites, the tester may benefit more from using the customized requirement alone, and avoid the expense of computing method coverage for the original suite. If the tester is interested in computing more expensive requirements, such as statement and conditional coverage, then she would probably not want to combine the traditional requirement-adequate suites with the customized requirements.

Figure 6 presents the fault detection effectiveness results. The y-axis shows the percent faults detected by the reduced suites. The percentage is based on the total number of faults detected by the original suite. The original suite detected 84 of the 135 seeded faults. `HGS`-statement and `HGS`-conditional reduced suites have the best fault detection. `HGS`-method has the least fault detection. On augmenting `HGS`-method with our customized requirements, the fault detection effectiveness of the reduced suite increases. However, the augmented requirements perform similar to the customized requirement alone, suggesting that if the tester intended to use method coverage adequacy, she may instead benefit more from using the customized requirements. As the complexity of the customized requirements increases, the fault detection effectiveness of the combined requirements suites' increases.

Our hypothesis that combined requirements create suites better than a `HGS`-method reduced suite was supported by both our program coverage and fault detection results, except for `base`. However, the combined requirements suites had lower program coverage and fault detection than the usage-based requirements alone.

4.6.3 Augmenting `HGS` with the Tie Breaker

Methodology. We augmented our Java implementation of `HGS` with the customized requirements-based tie-breaker to create the `Modified HGS` algorithm. We executed the `HGS` and `Modified HGS` algorithms 100 times and collected the test suite size and fault detection results for the 100 suites. To measure the program coverage effectiveness, we executed the mode suite, i.e., the test suite that occurs most frequently among the 100 suites.

Results and Analysis. Figure 7 shows the reduced test suite size for reduced suites generated by `Modified HGS` and `HGS`. The x-axis for all the figures in this experiment represents the reduced test suites. We show results for reducing the CPM original suite by statement (S), conditional (C) and method (M) adequacy criteria with `HGS`. When the cus-

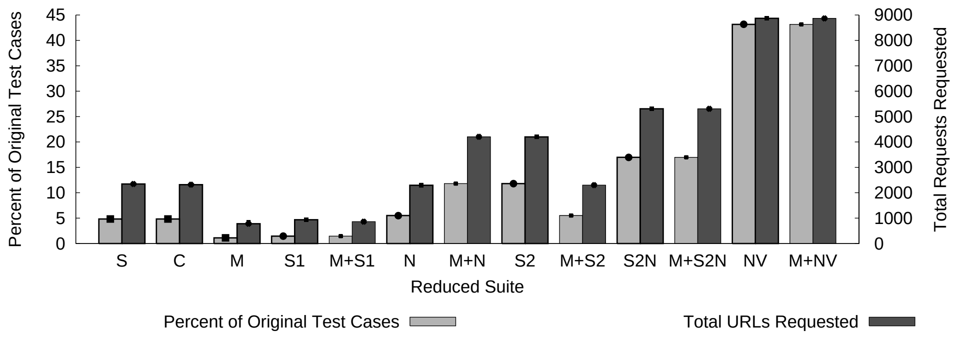

Figure 4: Combining Traditional and Customized Requirements: Reduced Test Suite Size

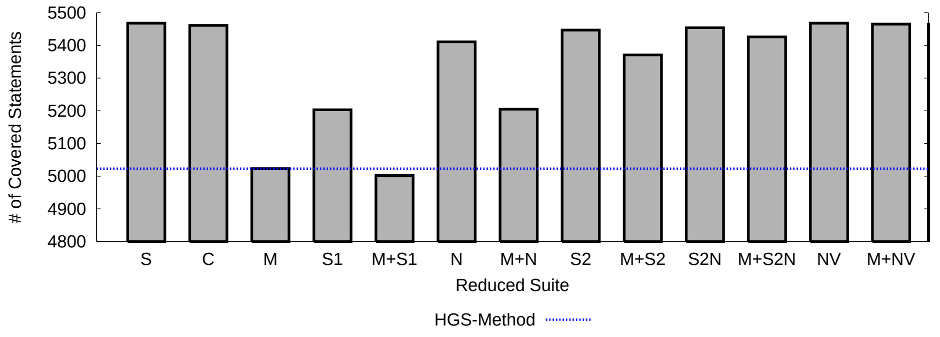

Figure 5: Combining Traditional and Customized Requirements: Program Coverage Effectiveness

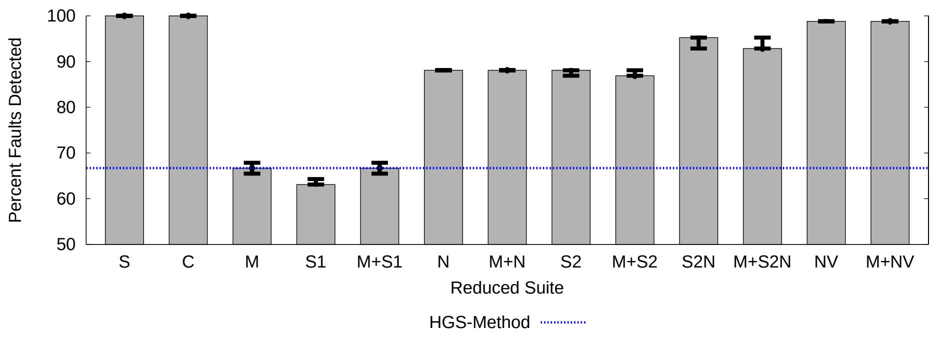

Figure 6: Combining Traditional and Customized Requirements: Fault Detection Effectiveness

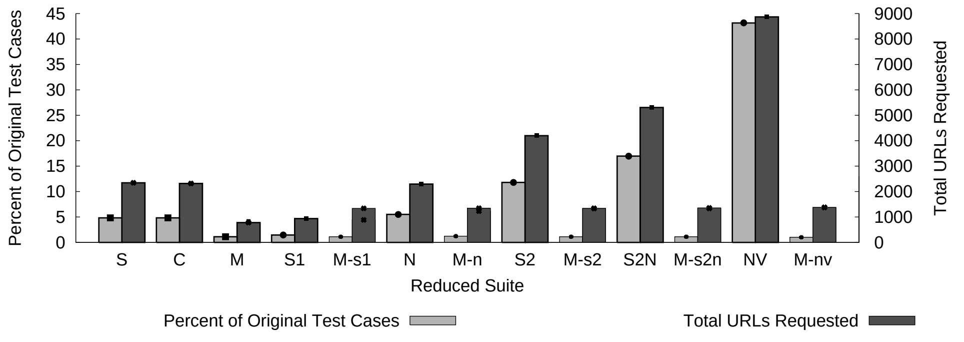

Figure 7: Modified HGS: Reduced Test Suite Size

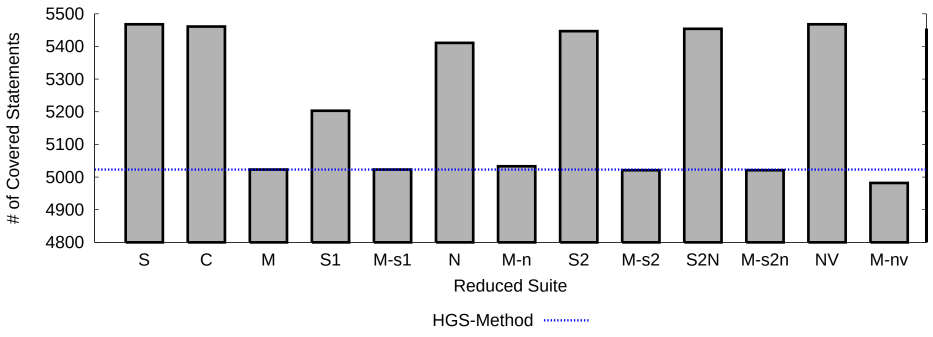

Figure 8: Modified HGS: Program Coverage Effectiveness

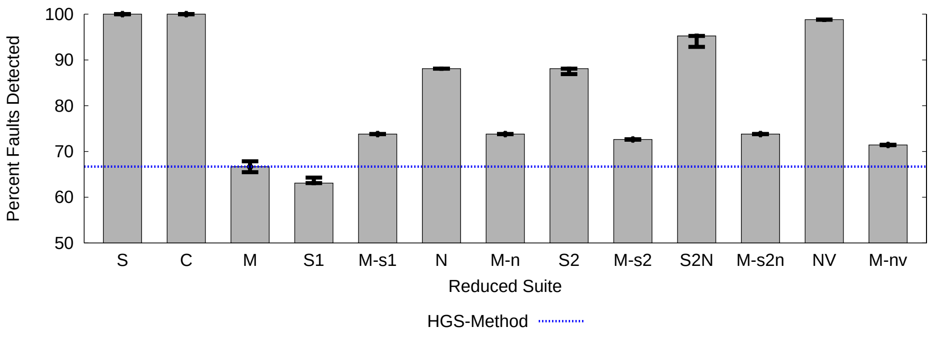

Figure 9: Modified HGS: Fault Detection Effectiveness

tomized requirement is used to break ties we abbreviate using lower case letters as follows: `base` as `s1`, `seq2` as `s2`, `name` as `n`, `name_value` as `nv` and `seq2_name` as `s2n`. Suites from `Modified HGS` are shown by the method adequacy used (M) and the customized requirement used to break ties, separated by a hyphen (e.g., "M-s2").

The vertical lines on each bar represent the 25th and 75th percentile of the nondeterministic `HGS` and `Modified HGS` approaches. The suites created by using customized requirements alone are abbreviated as described earlier in Section 4.6.2. We use the same x-axis labeling for all our figures in this experiment.

In Figure 7, the left y-axis represents the percent reduction in test suite size, in terms of number of test cases and the right y-axis represents the total number of requests in the reduced test suite. As expected, `HGS`-statement and `HGS`-conditional create suites bigger than `HGS`-method. Reduced suites generated by `Modified HGS` are similar in size to suites generated by `HGS`-method, decreasing slightly for the `name_value` tie-breaker. Reduced test suites from the customized requirements alone are larger in size compared to all other test suites. With an increase in complexity of the customized requirement, the reduced test suite size created by using the customized requirement alone increases. The suite size results suggest that augmenting `HGS` with the customized requirements is acceptable, since the reduced test suite increase in size is negligible.

Figure 8 shows the program coverage effectiveness for all the mode reduced suites. The y-axis shows the number of statements covered by each suite. The horizontal line in Figures 8 and 9 represents the program coverage/fault detection of the `HGS`-method reduced suite. In general, the program coverage effectiveness of `Modified HGS` reduced suites was equivalent to the coverage of `HGS`-method reduced suites (we observed a difference of 2 statements for some of the `Modified HGS` reduced suites). Since `Modified HGS` with `name_value` generates a reduced suite that has one less test case than the other `Modified HGS` reduced suites, we believe `Modified HGS` with `name_value` loses some program coverage (around 0.3% of the statements covered by original suite).

Figure 9 shows the fault detection effectiveness of `HGS`, `Modified HGS`, and the customized test requirements' reduced suites. The y-axis shows the percent of faults detected by the reduced suites. Statement and conditional adequate suites generated by `HGS` detect all the faults. `Modified HGS` reduced suites perform better than using `HGS`-method reduced suites. We note there is variation in the fault detection effectiveness of the `HGS`-method adequate suites. However, the variation decreased when the customized requirements are used to break ties, suggesting that the 100 suites created are consistent in their fault detection capabilities. For the `Modified HGS` algorithm with *M-s1*, the reduced suites detected more faults than the customized requirement *S1* alone. However, *M-s1* is the only suite that exhibited this behavior. The `Modified HGS` suites remain consistent in their fault detection effectiveness across all the customized requirements. As the complexity of the customized requirement increases, the fault detection effectiveness of reduced suites created by using the customized requirement alone also increases. The fault detection effectiveness results are consistent with the respective reduced test suite sizes.

The program coverage and fault detection effectiveness results support our hypothesis that incorporating usage-based requirements as tie-breakers in `HGS` results in more effective test suites than suites that are generated by using program coverage-based requirement alone. However, the type of the customized requirement did not appear to have an effect on the fault detection effectiveness of the `Modified HGS` suites.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we presented three applications of customized test requirements for web applications. We applied the test requirements to compare test suites and to augment existing test suite reduction strategies. We conducted an experimental study with one subject web application and field data collected from two academic years of use at the University of Delaware.

Our results suggest that the customized test requirements are most applicable when the compared test suites vary widely in the constituent test cases. If a tester is using method adequacy for test suite reduction, an inexpensive criterion compared to statement and conditional adequacy, the tester may want to use our customized requirements for reduction instead. Using our customized requirements in isolation creates more effective reduced suites than using `HGS`-method or using the combination of the requirements. Except for `base`, combined requirements also performed better than `HGS`-method. By augmenting `HGS` with customized requirements as tie breakers, in general, we were able to create reduced test suites that were more effective than `HGS`-method reduced suites in terms of program coverage and fault detection.

In the future, we plan to extend our studies to other applications. Extending our study with more subject web applications and test suites will aid in generalizing our results. We believe in some cases we were unable to see distinctions between using customized test requirements versus traditional requirements because of our fault seeding strategy. We plan to modify our fault seeding strategy so that the fault categories are strict partitions, i.e., a fault of category 1 can be detected only by suites that exhibit a certain behavior. In the future, we plan to address the difficulty of randomly seeding faults in object-oriented programs.

All our reduction techniques were based on usage data. The traditional coverage criteria were applied to test cases from field data. In the future, we plan to create test cases from application models and apply traditional coverage metrics. We believe the importance of usage-based requirements will be more significant if we augment traditional coverage adequate reduced suites from static models of the application with customized requirements from usage data.

6. REFERENCES

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. In preparation, 2003.

[2] A. Andrews, J. Offutt, and R. Alexander. Testing Web Applications by Modeling with FSMs. *Software and Systems Modeling*, 4(3), August 2005.

[3] S. Beydeda, V. Gruhn, and M. Stachorski. A graphical class representation for integrated black- and white-box testing. In *Proceedings of the International Conference on Software Maintenance*. IEEE, 2001.

[4] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proceedings of International Conference on Software Engineering*, May 2004.

[5] H. Y. Chen, T. Tse, F.T.Chan, and T. Chen. In black and white: An integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 7(3):250 –295, July 1997.

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

[7] 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.

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

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

[10] Michal Blumenstyk. Web Application Development - Bridging the Gap between QA and Devel opment. <http://www.stickyminds.com>.

[11] J. D. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, 10(2):14–32, March/April 1997.

[12] S. Pertet and P. Narsimhan. Causes of failures in web applications. Technical Report CMU-PDL-05-109, Carnegie Mellon University, 2005.

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

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

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

[16] S. Sampath, S. Sprenkle, E. Gibson, and L. Pollock. Web Application Testing with Customized Test Requirements—An Experimental Comparison Study. Technical Report 2006-330, University of Delaware, 2006.

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

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

[19] T. Thelin, P. Runeson, and C. Wohlin. Prioritized use cases as a vehicle for inspections. *IEEE Software*, 20(4):30–33, July/August 2003.