TEST CASE PRIORITIZATION TECHNIQUES

[1] SIRIPONG ROONGRUANGSUWAN, [2]JIRAPUN DAENGDEJ

[1] Autonomous System Research Laboratory, Science and Technology, Assumption University, Thailand

[2] Autonomous System Research Laboratory, Science and Technology, Assumption University, Thailand

ABSTRACT

Software testing has been proven that testing, analysis, and debugging costs usually consume over 50% of the costs associated with the development of large software systems. Many researchers have found several approaches to schedule an order of test execution. Unfortunately, existing test prioritization techniques are failed to prioritize multiple test suites and test cases with same priority values. Consequently, those techniques are inefficient to prioritize tests in the large commercial systems. They incorrectly schedule tests and the cost is overrun during the prioritization process. Thus, this paper proposes two new efficient prioritization methods to address the above issues. The first method aims to resolve the problem of many test cases assigned the same weight values. The second method is developed to effectively prioritize multiple suites. As a result, this paper discusses an ability to reserve high prioritize tests in multiple suites while minimizing a prioritization time.

Keywords: *Test Case Prioritization, Test Prioritization Methods, Test Prioritization Comparison, Multiple Test Prioritization And Test Suite Prioritization*

1. INTRODUCTION

Software testing is a comprehensive set of activities conducted with the intent of finding errors in software. It is one activity in the software development process aimed at evaluating a software item, such as system, subsystem and features (e.g. functionality, performance and security) against a given set of system requirements. Also, software testing is the process of validating and verifying that a program functions properly. Many researchers have proven that software testing is one of the most critically important phases of the software development life cycle, and consumes significant resources in terms of effort, time and cost.

Arden [33] said that "The impact of software errors is enormous because virtually every business in the United States now depends on software for the development, production, distribution, and after-sales support of products and services. Innovations in fields ranging from robotic manufacturing to nanotechnology and human genetics research have been enabled by low-cost computational and control capabilities supplied by computers and software." Also, a study conducted by NIST in 2002 reports that software bugs cost the U.S. economy $59.5 billion annually. More than a third of this cost could be avoided if better software testing was performed [33].

Boris [5] claimed that software testing should take around 40-70% of the time and cost of the software development process. Many approaches have been proposed to reduce time and cost during software testing process, including test case prioritization techniques and test case reduction techniques. For example, [20], [13], [14], [15], [17], [34], [37] and [38]. Also, many empirical studies for prioritizing test cases have been conducted, like [46], [25], [43], [48], [14] and [15].

Furthermore, Gregg Rothermel [15] has proven that prioritizing and scheduling test cases are one of the most critical tasks during the software testing process. He referred to the industrial collaborators reports, which shows that there are approximately 20,000 lines of code, running the entire test cases requires seven weeks. In this situation, test engineers may want to prioritize and schedule those test cases in order that those test cases with higher priority are executed first. Additionally, he [13], [16] stated that test case prioritization methods and process are required, because: (a) the regression testing phase consumes a lot of time and cost to run, and (b) there is not enough time or resources to run

the entire test suite (c) there is a need to decide which test cases to run first.

Test case prioritization techniques prioritize and schedule test cases in an order that attempts to maximize some objective function. For example, software test engineers might wish to schedule test cases in an order that achieves code coverage at the fastest rate possible, exercises features in order of expected frequency of use, or exercises subsystems in an order that reflects their historical propensity to fail. When the time required to execute all test cases in a test suite is short, test case prioritization may not be cost effective - it may be most expedient simply to schedule test cases in any order [13], [16]. When the time required to run all test cases in the test suite is sufficiently long, the benefits offered by test case prioritization methods become more significant.

Although test case prioritization methods have great benefits for software test engineers, there are still outstanding major research issues that should be addressed. The examples of major research issues are: (a) existing test case prioritization methods ignore the practical weight factors in their ranking algorithm (b) existing techniques have an inefficient weight algorithm and (c) those techniques are lack of the automation during the prioritization process.

Section 2 discusses a comprehensive set of existing test case prioritization techniques and prioritization processes, and proposes to divide them into four major groups. Section 3 proposes outstanding research challenges and offers a guide for researchers in the test case prioritization field. Section 3 also determines which research problems have been resolved by which techniques. Section 4 provides the conclusion and research directions for test case prioritization area. The last section represents all source references used in this paper.

2. LITERATURE REVIEW

This section discusses an overview of a software development life cycle (or SDLC) and a general software testing process. It describes a comprehensive set of existing test case prioritization methods researched from 1998 to 2008. In addition, it introduces a new "4C" classification of existing test case prioritization techniques. In general, the SDLC process contains the following phases, which are: requirement gathering, design & analysis, development, testing and maintenance [36]. Those phases can be represented as follows:

Figure 1. A General SDLC Process

From the above, the testing phases [29] contain the following processes: test planning, test development, test execution and evaluation of results. Those processes can be represented as follows:

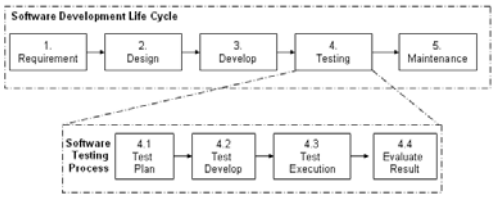

Figure 2. A General Software Testing Process

Software testing has been widely used as a way to help engineers develop high-quality systems. It is an important process that is performed to support quality assurance by gathering information about the nature of the software being studied M. J. Harrold [62]. These activities consist of designing test cases, executing the software with those test cases, and examining the results produced by those executions. Boris [5] indicates that more than fifty percent of the cost of software development is devoted to testing with the percentage for testing critical software being even higher. As software becomes more pervasive and is used more often to perform critical tasks, the importance of its quality will remain high. Unless engineers can find efficient ways to perform effective testing, the percentage of development costs devoted to testing may increase significantly. Software testing is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test [8], with respect to the context in which it is intended to operate. It also provides an objective, independent view of the software to allow the business to appreciate and understand the risks of implementation of the software. Test techniques include the process of executing a program or application with the intent of finding software bugs. It can also be stated as the process of validating and verifying that software meets the business and technical requirements that guided its design and development, so that it works as expected. Software testing can be implemented at any time in the development process; however, the most test effort is employed after the requirements have been defined and coding process has been completed. Software engineers generally save test suites that they develop so that they can easily reuse those suites later as the software evolves. Reusing

test cases in regression testing process is pervasive in the software industry [28] and can save as much as one-half of the cost of software maintenance [5] However, executing a set of test cases in an existing test suite consume a significant amount of time.

Rothermel [13], [14] gave an interesting example as follows: "one of the industrial collaborators reports that for one of its products that contains approximately 20,000 lines of code, running the entire test suite requires seven weeks. In such cases, testers may want to order their test cases so that those test cases with the highest priority, according to some criterion, are run first". This has proven that prioritizing and scheduling test cases are one of the most important tasks during regression testing process.

Additionally, Rothermel [13], [16] mentioned that the test case prioritization process is required for software testing because: (a) the regression testing phase consumes a lot of time and cost to run, and (b) there is not enough time or resources to run the entire test suite, therefore (c) there is a need to decide which test cases to run first.

Test case prioritization techniques prioritize and schedule test cases in an order that attempts to maximize some objective function. For example, software test engineers might wish to schedule test cases in an order that achieves code coverage at the fastest rate possible, exercises features in order of expected frequency of use, or exercises subsystems in an order that reflects their historical propensity to fail. When the time required to execute all test cases in a test suite is short, test case prioritization may not be cost effective - it may be most expedient simply to schedule test cases in any order [13], [16]. When the time required to run all test cases in the test suite is sufficiently long, the benefits offered by test case prioritization methods become more significant.

Test case prioritization techniques provide a way to schedule and run test cases, which have the highest priority in order to provide earlier detect faults. This study presents numerous techniques developed, between 2002 and 2008, that can improve a test suite's rate of fault detection.

With existing test case prioritization techniques researched in 1998-2008, this paper introduces and organizes a new "4C" classification of those existing techniques, based on their prioritization algorithm's characteristics, as follows:

1. Customer Requirement-based techniques. Customer requirement-based techniques are methods to prioritize test cases based on requirement documents. Many researchers have researched this area, such as Srikanth [20], Zhang [47] and Nilawar [30]. Also, many weight factors have been used in these techniques, including custom-priority, requirement complexity and requirement volatility.

2. Coverage-based techniques. Coverage-based techniques are methods to prioritize test cases based on coverage criteria, such as requirement coverage, total requirement coverage, additional requirement coverage and statement coverage. Many researchers have researched this area, such as Leon [9], Rothermel [13], [14] and Bryce [35].

3. Cost Effective-based techniques. Cost effective-based techniques are methods to prioritize test cases based on costs, such as cost of analysis and cost of prioritization. Many researchers have researched this area, for instance, Malishevsky [1], Alexey [2], and Elbaum [40].

4. Chronographic history-based techniques. Chronographic history-based techniques are methods to prioritize test cases based on test execution history. A few researchers have researched this area, for example, Kim [26] and Qu [3].

The following sections describe the above techniques in details.

2.1 Customer Requirement-Based Prioritization Techniques

Hema [20] presented the requirements-based test case prioritization approach to prioritize a set of test cases. They built upon current test case prioritization techniques [41] and proposed to use several factors to weight (or rank) the test cases. Those factors are the customer-assigned priority (CP), requirements complexity (RC) and requirements volatility (RV). Additionally, they assigned value (1 to 10) for each factor for the measurement. They stated that higher factor values indicate a need for prioritization of test case related to that requirement.

Weight prioritization (*WP*) measures the important of testing a requirement earlier.

$$WP = \Sigma\ (PF_{value} * PF_{weight}); \text{ PF=1 to } n \qquad (1)$$

Where:

- *WP* denotes weight prioritization that measures the importance of testing a requirement.
- PF_{value} is the value of each factor, like CP, RC and RV.
- PF_{weight} is the weight of each factor, like CP, RC and RV.

Test cases are then ordered such that the test cases for requirements with high WP are executed before others. Recent research demonstrated that using different test case prioritization techniques can significantly affect the rate of fault detection of the test suite [13]. Y.T. Yu [49], [50] proposed that the methodology be enhanced to allow the software test engineer to specify criteria for prioritization so that the ordering of test cases for execution can be automated. CTM was first introduced by Grochtmann [31], and has been successfully applied in many industrial development projects such as in aviation technology, car electronics and commercial data processing applications [12]. However, while the CTM technique aims to provide the tester with a structured framework and methodical guidelines, it still leaves many manual routine tasks that may demand great effort. Yu, Ng and Chan proposed to solve the CTM's limitations by assigning weights to each classification in the classification tree. Their idea is that a test case with classifications or classes of higher weights should have a higher priority. The weight priority value is calculated as the following formula:

$$p = (\Sigma i\ wiWi) / (\Sigma i\ Wi) \qquad (2)$$

Where:

- *p* denotes weight prioritization that measures the importance of test case.
- *wiWi* is a combination of higher weights, which are associated with the classification tree, called CTM.
- *Wi* is a classification weight value that is included in the classification tree.

The tester may then choose to arrange the test cases in descending order of their priority values (with arbitrary ordering in case of ties).

Hema [21] were interested in two particular goals of test case prioritization approaches: (a) to improve user perceived software quality in a cost effective way by considering potential defect severity and (b) to improve the rate of detection of severe faults during system level testing of new code and regression testing of existing code. He presented a value-driven approach to system-level test case prioritization called the Prioritization of Requirements for Test (PORT). PORT prioritizes system test cases based upon four factors: requirements volatility (*RV*), customer priority (*CP*), implementation complexity (*IC*) and fault proneness of the requirements (*FP*).

They proposed the following formula to prioritize test cases:

$$PFV_i = \sum 4j=1(FactorValue_{ij} * FactorWeight_j) \qquad (3)$$

Where:

- PFV_i is the prioritization factor value for requirement *i*.
- $FactorValue_{ij}$ is the value for factor *j* for requirement *i*.
- $FactorWeight_j$ is the factor weight for the *j*th factor for a particular product.

A value-matrix representation of PFV for requirements is shown below where PFV (*P*) is the product of value (*V*) and weight (*w*).

$$P = V_w\ (PFV_1 \ldots PFVn)(n*1) = (RCP_1 \ldots RCP_n\ \ RIC_1 \ldots RIC_n\ \ RRV_1 \ldots RRV_n\ \ RFP_1 \ldots RFP_n)(n*4)\ (WCP\ \ WRC\ \ WFP\ \ WRV)(4*1) \qquad (4)$$

Where:

- PFV_i is prioritization factor value for requirement *i*, which is the summation of the product of factor value and the assigned factor weight for each of the factors.
- *R1..n* is requirements coverage of each test case.
- *WCP* is a weight measurement for *CP* factor.
- *WRC* is a weight measurement for *RC* factor.
- *WFP* is a weight measurement for *FP* factor.
- *WRV* is a weight measurement for *RV* factor.

The computation of PFVi for a requirement is used in computing the Weighted Priority (*WP*) of its associated test cases. WP of the test case is the product of two elements: (a) the average PFV of the requirement(s) the test case maps to and (b) the requirements-coverage a test case provides. Requirements coverage is the fraction of the total project requirements exercised by a test case. Let there be n total requirements for a product/release, and test case j maps to i requirements. WPj is an

indication of the priority of running a particular test case. WPj is represented as below:

$$WPj = (\sum ix=1\ PFV_x/\sum_{nj}=1 PFV_y)*(1/n) \quad (5)$$

Where:

- WP_j is an indication of the priority of running a particular test case.
- PFV_i is prioritization factor value for requirement *i*, which is the summation of the product of factor value and the assigned factor weight for each of the factors.

The test cases are ordered for execution by descending value of *WP* such that the test case with the highest *WP* value is run first.

All the above techniques rely on the assumption that testing requirement priorities and test case costs are uniform, however in practice these can vary widely. For the former, testing requirement priorities can change frequently during software development, and the uniformly categorized testing requirements specification often fail to address stakeholder values [6], [19]. For the latter, test cases usually require different execution time and resources. Obviously, testing requirement priorities and test case costs should have a great impact on the prioritization of those test cases, and so the existing prioritization techniques and the corresponding metrics should be adapted to incorporate them. Xiaofang [47] proposed a new, general test case prioritization technique and associated metric based on varying testing requirement priorities and test case costs. They proposed an algorithm that weights test cases by the following factors: (a) test history (b) additional requirement coverage (c) test case cost and (d) total requirement coverage.

Manish [30] proposed an approach for test case generation for web based applications. One of their generation processes is the prioritization of test cases. They presented a simple approach for test case prioritization through the requirement traceability matrix. The matrix can be produced by mapping from use cases in the Use Case diagram to functional requirements from users. They also proposed simply to use weight values assigned to each requirement by developers. Each requirement is assigned a priority weight from 1 to 10, 10 being highest.

2.2 Coverage-Based Prioritization Techniques

Test coverage analysis is a measure used in software testing known as code coverage analysis for practitioners. It describes the quantity of source code of a program that has been exercised during testing. It is a form of testing that inspects the code directly and is therefore a form of white box testing. The following lists a process of coverage-based techniques: (a) finding areas of a program not exercised by a set of test cases (b) creating additional test cases to increase coverage (c) determining a quantitative measure of code coverage, which is an indirect measure of quality and (d) identifying redundant test cases that do not increase coverage.

The coverage-based technique is a structural or white-box testing technique. Structural testing compares test program behavior against the apparent intention of the source code. This contrasts with functional or black-box testing, which compares test program behavior against a requirements specification. It examines how the program works, taking into account possible pitfalls in the structure and logic. Functional testing examines what the program accomplishes, without regard to how it works internally. The coverage-based techniques are methods to prioritize test cases based on coverage criteria, such as requirement coverage, total requirement coverage, additional requirement coverage and statement coverage.

The following paragraphs present coverage-based prioritization techniques that have been proposed.

Leon [9] presented an empirical comparison of four different techniques for filtering large test suites: test suite minimization, prioritization by additional coverage, cluster filtering with one-per-cluster sampling, and failure pursuit sampling. The first two techniques are based on selecting subsets that maximize code coverage as quickly as possible, while the latter two are based on analyzing the distribution of the tests' execution profiles.

Rothermel [18] have researched and surveyed test case prioritization. They considered nine approaches for prioritizing a set of test cases and reported results measuring the effectiveness of those approaches to improve the capability to reveal faults. They proposed the following techniques: (a) random approaches (b) optimal prioritization (c) total branch coverage prioritization (d) additional branch coverage prioritization (e) total statement coverage prioritization (f) additional statement coverage prioritization (g) total fault-exposing-

potential prioritization and (h) additional fault-exposing-potential prioritization.

Bryce [35] described an algorithm for re-generating prioritized test suites. The generated test suites are a special kind of a covering array called a biased covering array. They began by defining a set of interaction weights for each value of each factor. For each factor the weight of combining it with each other factor is computed as a total interaction benefit. The factors are sorted in decreasing order of interaction benefit and then filled as follows. First, the individual interaction weights for each of the factor's values are computed. This selects the value of the factor that has the greatest value interaction benefit. After all factors have been fixed, a single test has been added, and the benefits for factors are recomputed and the process starts again. The algorithm is complete when all pairs have been covered.

Leon [9] believed that test case filtering is closely related to the field of test case prioritization. The goal of test case filtering is to select a relatively small subset of a test suite which finds a large portion of the defects that would be found if the whole test suite were to be used. In their paper they presented an empirical comparison of four different techniques for filtering large test suites: test suite minimization, prioritization by additional coverage, cluster filtering with one-per-cluster sampling and failure pursuit sampling. Their results indicate that their techniques can be as efficient as or more efficient at revealing defects than coverage-based techniques, but that the two kinds of techniques are also complementary in the sense that they find different defects. Accordingly, some simple combinations of these techniques were evaluated for use in test case prioritization. The results indicate that applying this combination of techniques can produce results more efficiently than applying prioritization by additional coverage alone.

In fact, Elbaum [39], [41], [42] extended the selection technique in order to prioritize the test cases in a test suite, that is, to place the test cases in non-decreasing order with respect to their perceived likelihood of revealing defects. Elbaum's approach, which he called additional coverage prioritization, involves running a greedy coverage maximization algorithm repeatedly on the set of test cases that have not yet been prioritized. The priority of a test case corresponds to the order in which it is selected during this process. The earlier a test case is selected, the higher its priority is. In the sequel, Leon [9] referred to the prioritization technique as repeated coverage maximization.

In Xiao's previous work [45], Xiao examined the prioritization methods of CIT test suites and developed several ways to control the prioritization through weightings. They used methods that utilize code coverage data from prior releases, as well as one that is specification based. Further, they observed that Bryce and Colbourn's prioritization technique [35] is a combination of the generation and prioritization techniques, rather than a pure prioritization method. This is because it regenerates tests each time rather than simply reordering them.

Hyunsook [22] considered seven different test case prioritization techniques, which they classified into three groups: (a) the first group is the control group, containing three "orderings" that serve as experimental controls. The untreated ordering is the ordering in which test cases are originally provided with the object. The optimal ordering represents an upper bound on prioritization technique performance and is obtained by greedily selecting each test case in terms of its exposure of faults not yet exposed by test cases already ordered. The process is repeated until all test cases are ordered. Ties are broken randomly. The random ordering places test cases in a random order (b) the second group is the block level group, containing two techniques: *block-total* and *block-addtl*. By instrumenting a program, they can determine the numbers of basic blocks in that program that are exercised by that test case. The block-total technique prioritizes test cases according to the total number of blocks they cover simply by sorting them by that number. The block-addtl technique prioritizes test cases in terms of the numbers of additional blocks test cases cover by greedily selecting the test cases that cover the most as-yet-uncovered blocks until all blocks are covered, then repeating this process until all test cases have been placed in order and (c) the third group is the method level group, containing two techniques: *methodtotal* and *method-addtl*. These techniques are exactly the same as the corresponding block level techniques just described, except that they rely on coverage measured in terms of numbers of methods rather than numbers of blocks covered.

The test case prioritization techniques studied in [4], [11] are primarily based on variations of the total requirement coverage and the additional requirement coverage of various structural elements in a program. For instance, total statement coverage prioritization orders test cases in decreasing order of the number of statements they exercise.

Additional statement coverage prioritization orders test cases in decreasing order of the number of additional statements they exercise that have not yet been covered by the tests earlier in the prioritized sequence. These prioritization methods do not take into consideration the statements which influenced, or could potentially influence, the values of the program output. Neither do they take into consideration whether a test case traverses a statement or not while prioritizing the test cases. It is intuitive to expect that the output of a test case that executes a larger number of statements that influence the output, or have the potential to influence the output, is more likely to be affected by the modification than tests covering fewer such statements. In addition, tests exercising modified statements should have higher priority than tests that do not traverse any modifications.

Jeffrey [10] presented a new approach for prioritizing test cases that is based not only on total statement coverage (also known in that paper as branch coverage), but that also takes into account the number of statements executed that influence or have potential to influence the output produced by the test case. The set of such statements corresponds to the relevant slice, which is computed on the output of the program when executed by the test case [7]. The approach is based on the following observation: If a modification in the program has to affect the output of a test case in the regression test suite, it must affect some computation in the relevant slice of the output for that test case. Therefore, their heuristic for prioritizing test cases assigns higher weight to a test case with larger number of statements in its relevant slice of the output. They used the following factors in their approach to prioritize test cases: (a) the number of statements in the relevant slice of output for the test case, because any modification should necessarily affect some computation in the relevant slice to be able to change the output for this test case and (b) the number of statements that are executed by the test case but are not in the relevant slice of the output.

Jeffrey [10] ordered the test cases in decreasing order of test case weight, where the weight for a test is determined as follows:

$$TW = ReqSlice + ReqExercise \qquad (6)$$

Where:

- *TW* denotes a weight prioritization determined for each test case.
- *ReqSlice* is a number of requirements presented in the relevant slice of output for each test case.
- *ReqExercise* is a number of requirements exercised by the test case.

Ties are broken arbitrarily. This criterion essentially gives "single" weight to those exercised requirements that are outside the relevant slice, and "double" weight to those exercised requirements that are contained in the relevant slice.

Jones [24] presented new algorithms for test-suite reduction and prioritization that can be tailored effectively for use with modified coverage (MC) and decision coverage (DC). Most existing techniques from researchers who have been investigating test suite reduction (also referred to as test suite minimization) and prioritization techniques consider a set of test-case coverage criteria such as, statements, decisions, definition user associations and specification items. In their paper, they focused on MC and DC criteria as for test case reduction and prioritization, building on Rothermel's test case prioritization technique [13], [14]. Their approach uses total requirement coverage and the additional requirement coverage to weight and schedule test cases accordingly.

2.3 Cost Effective-Based Prioritization Techniques

Cost effective-based techniques are methods of prioritizing test cases based on costs, such as cost of analysis and cost of prioritization. Many researchers have researched this area. The following paragraphs present existing cost effective-based test case prioritization techniques.

Leung and White presented a cost model for regression test selection in [23]. The proposed model incorporates various costs of regression testing, including the costs of executing and validating test cases and the cost of performing analyses to support test selection, and provides a way to compare tests for relative effectiveness. This model can be appropriately applied to an effective regression test selection techniques [17], which necessarily select all test cases in the existing test suite that may reveal faults.

However, Leung's model does not consider the costs of overlooking faults due to discarded tests. Alexey Malishevsky [1] presented cost models for prioritization that take these costs into account. They defined the following variables to prioritize

test cases: cost of analysis, Ca(T) and cost of the prioritization algorithm, Cp(T).

$$WP = Ca(T) + Cp(T) \qquad (7)$$

Where:

- *WP* is a weight prioritization value for each test case.
- *Ca(T)* includes the cost of source code analysis, analysis of changes between old and new versions, and collection of execution traces.
- *Cp(T)* is the actual cost of running a prioritization tool, and, depending on the prioritization algorithm used, can be performed during either the preliminary or critical phase.

Furthermore, Malishevsky [1] divided the regression testing process into two phases: preliminary phase and critical phase. Preliminary phase activities may be assigned different costs than critical phase activities, since the latter may have greater ramifications for things like release time.

The cost of a test case is related to the resources required to execute and validate it. Additionally, cost-cognizant prioritization requires an estimate of the severity of each fault that can be revealed by a test case. Fault severity may be used to order tests by the same two criteria listed previously. Previous works [14] have defined and investigated various prioritization techniques. Meanwhile, Alexey Malishevsky [2], [40] focus on four practical code-coverage-based heuristic techniques. Those four techniques are: total function coverage prioritization (*fn-total*), additional function coverage prioritization (*fn-addtl*), total function difference-based prioritization (*fn-diff-total*) and additional function difference-based prioritization (*fn-diff-addtl*).

2.4 Chronographic History-Based Prioritization Techniques

Chronographic history-based techniques are methods to prioritize test cases based on test execution history. The following paragraphs present an overview of existing chronographic history-based test case prioritization techniques.

Jung-Min [26] proposed to use information about each test case's prior performance to increase or decrease the likelihood that it will be used in the current testing session. Their approach is based on ideas taken from statistical quality control (exponential weighted moving average) and statistical forecasting (exponential smoothing). Kim [26] defined the selection probabilities of each test case, *TC*, at time, *t*, to be $P_{tc,t}(H_{tc}, \alpha)$, where H_tc is a set of *t*, time-ordered observations $\{h_1, h_2, \ldots h_n\}$ drawn from runs of TC and α is a smoothing constant used to weight individual historical observations. The higher values emphasize recent observations, while lower values emphasize older ones. These values are then normalized and used as probabilities. The general form of:

P is $P_0 = h_1$ and $P_k = \alpha_{hk} + (1- \alpha)P_k\text{-}1$, $0<= \alpha<=1$, $k>=1$. (8)

When testing in a black box environment, source code related information is not available. In such situations, practitioners only have output of test cases and other run-time information available, such as the running time of test cases. Bo Qu [3] proposed a prioritization technique based on this limited information. One general method of prioritization for black box testing is to initialize a test suite using test history, and then adjust the order of the rest of the test cases based on run-time information. To guide the adjusting strategy, a matrix R is used. They defined the matrix, R, to predict the fault detection relationship of test cases, so once a test case revealed regression faults, related test cases can be adjusted to higher priority to achieve a better rate of fault detection. Let T be a test suite, let T′ be a subset of T, and let R be a matrix which describes the fault detection relationship of test cases. Their general process of test case prioritization for black box testing can be described shortly as follows: (a) select T′ from T and prioritize T′ using available test history (b) build a test case relation matrix R based on available information (c) draw a test case from T′, and run it (d) reorder rest test cases using run-time information and test case relation matrix R and (e) repeat from step c until testing resource is exhausted.

3. RESEARCH CHALLENGES

This section provides details of outstanding research issues motivated this study. The literature review reveals that there are many outstanding research issues in the test case prioritization area, such as poor performance of prioritization algorithms, non-practical weight factors and non-commercial prioritization methods.

However, this paper highlights two critical outstanding research issues, which are: (a) existing

prioritization methods ignores prioritizing multiple test suites and (b) existing techniques randomly prioritize all test cases with the same weight values, without any systematic algorithm.

The first issue can lead to the following difficult situations in the commercial industry. In the commercial systems, there is always more than a single test suite, particularly during a system integration testing. The existing prioritize techniques are not applicable if there are many test suites. In addition, it may take longer time to individually prioritize each test suite with existing methods. As a result, it may rapidly increase an amount of time and cost to prioritize test suites.

The second issue may cause to a poor performance problem while prioritization process. The study shows that there is a high probability that a significant number of test cases can be assigned to the same weight values. With those test cases, the existing methods randomly prioritize without any systematic algorithm and weight factor. Consequently, the prioritized test cases may not be accurate with the right weight values.

Finally, researchers should develop an effective test case prioritization method that can be used to schedule many test suites and test cases with the same weight values.

4. PROPOSED METHODS

This section describes a new practical set of prioritization weight factors and two effective methods. The following describes practical factors and the proposed methods in details.

This study proposes practical factors to prioritize and schedule test cases, as follows.

Table 1. Proposed Practial Weight Prioritization Factors

Factor	Description
1. Cost Factors	
Execution cost (*EC*)	A total cost of running a set of test cases.
Cost of analysis (*Ca*)	Ca includes the cost of source code analysis, analysis of changes between old and new versions, and collection of execution traces.
Cost of data preparation (*CoDP*)	A total cost of preparing all input values for test cases.
Cost of validation (*CoV*)	A total cost for validating the expected result and actual result.
2. Time Factors	
Execution time (*ET*)	A total time of running a set of test cases.
Time consuming for data preparation (*TCfDP*)	A total time for preparing all input values.
Time consuming for validation (*TCfV*)	A total time for validating the expected result and actual result.
3. Defect Factors	
Defects occurred (*DO*)	An indicator of how many test cases have caused defects after execution.
Defects severity (*DS*)	A dimension for classifying seriousness for defects. This factor contains: showstopper, critical and minor severity.
4. Complex Factors	
Test case complexity (*TCplx*)	The complexity of test cases. It measure how difficult and complex a test case is. In addition, its complexity usually determines the effort required to execute it.
Requirement coverage (*ReqCov*)	A number of requirements covered by test cases. Requirements coverage views can help validate that all requirements are implemented in the system.
Dependency (*Dep*)	A dependency of test cases. This factor describes how many pre-requisite of each test case before execution are required.
Test impact (*TI*)	An impact of test cases. This factor assesses how importance of test cases if test cases are not be executed.

The above factors can be elaborated in details as follows.

Douglas Hoffman [57] addressed many cost related factors, such as cost for test case execution, cost for test results analysis including validation, cost for data preparation, into his cost-benefits analysis model. This can imply that cost related factors proposed in this study (i.e. *EC*, *Ca*, *CoDP* and *CoV*) are important and sensible. Also, Douglas described that the time consuming, for data preparation (or *TCfDP*), factor is one of common factors used in software testing. Therefore, this factor should be applied in prioritizing test cases. Additionally, Douglas described that the dependency, called "*Dep*" in this paper, factor is one of common factors used in software testing. Therefore, researchers should focus on this factor during the prioritization process.

Kalyana [58] addressed that time consuming for validation (or *TCfV*) factor is one of the most important and practical metrics in the software testing field. Also, it is related to the effort required to validate results and find a defect. Also, Kalyana referred the test impact (or *TI*) factor as business impact that effort to end-users. Also, this factor is widely used as software testing metrics. This can represent how important of this factor is. In addition, the impact of ignoring this test may lead to the following problems: a) increasing failures due to poor quality b) increasing software development costs c) increasing time to market due to inefficient

testing and d) increasing market transaction costs [66].

Cadar and Engler [59], from Stanford University, argued that high cost is actually not so important in some sense. They selected the execution time (or ET) to measure their proposed technique in their experiment. This can imply that this factor is a significant factor that should be included.

In general, test case that detects bugs should have higher priority, due to the fact that those bugs will be fixed and are required to re-test again. This factor can be referred to the "defect discovery rate" (or also known as *Do*), one of the most widely used metrics in software testing. The severity level of a defect indicates the potential business impact for the end user (or called *Ds*). The business impact factor is equal to the multiply between effect on the end user and frequency of occurrence. This factor provides indications about the quality of the software under test. A high-severity defect means low product / software quality, and vice versa. At the end of testing phase, this information is useful to make the release decision based on the number of defects and their severity levels. Kalyana [58] addressed that this factor is one of the most important and practical metrics in software testing. In addition, Julie and Mark reported that this factor is widely used in defect measurement system and always recorded in defect report. This can imply that prioritizing test cases with defects severity factor has a significant reason.

Furthermore, this section proposes two methods to resolve the following problems: (a) those existing methods randomly prioritize test cases with the same priority value and (b) existing prioritization techniques assume explicitly that there is always a single test suite. The first method, called MTSSP, aims to improve the ability to prioritize test cases with same weight values. The second method, called MTSPM, is developed to prioritize many test suites and test cases.

The following describes both of MTSSP and MTSPM methods in details.

4.1 A Method for Multiple Test Cases with Same Priority

The following lists reasons why this method has been proposed in this study.

1. Poor weight algorithm can lead to the wrong prioritization. Current prioritization techniques ignore the problem of multiple test cases with the same priority value [23], [24], [25], [26], [27], [28], [29], [44], [52], [53], [54], [55] [56].
2. Most current techniques use a random method to prioritize test cases when they have the same priority. Several empirical studies provided by Rothermel [7], [9], [19], [26], [27], [32] show that random method is not desirable comparing to other prioritization techniques.
3. Saif-ur-Rebman [40] stated that they applied a random method for prioritizing test cases if there are multiple cases with the same weight values. This means that there is a room to improve the ability for test case prioritization for this case.

The following considers improving the capability of weight algorithms in case that there are multiple test cases with the same priority and multiple sets of test cases.

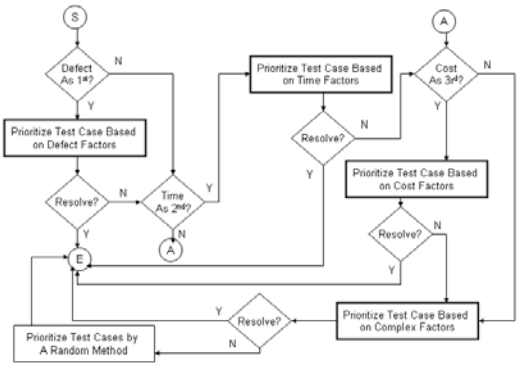

Figure 3. A Flow Chart of MTSSP Method

From the above figure, the procedures can be described as follows:

The beginning process starts with deciding that defect factors are the first priority or not. If those factors are the first priority factors, then assigning, computing and prioritizing test cases by using the proposed prioritization method, in section 4.2, included the following formula:

$$WP = \sum 2i=1(PFValue_i * PFWeight_i) \qquad (9)$$

Where:

- *WP* is weight prioritization for each test case that calculates from two (2) factors, which are: *DO* and *DS*
- $PFValue_i$ is a value assigned to each test case
- $PFWeight_j$ is a weight assigned for DO and DS factor

Test cases can be ordered in accordance with higher priority. However, if the weight priority value is still the same, then go to next step.

In this step, time factors are required to be concerned as second priority. Test cases can be prioritized based on these secondary factors (e.g. *ET*, *TCfDP* and *TCfV*) by using the method addressed in section 4.2 and the following formula:

$$WP = \sum 3i=1(PFValue_i * PFWeight_i) \quad (10)$$

Where:

- *WP* is weight prioritization for each test case that calculates from three (3) factors, which are: *ET*, *TCfDP* and *TCfV*.
- $PFValue_i$ is a value assigned to each test case
- $PFWeight_j$ is a weight assigned for *ET*, *TCfDP* and *TCfV*.

Test cases can be ordered in accordance with higher priority. However, if the weight priority value is still the same, then go to next step.

In this step, cost factors are required to be concerned as third priority. Test cases can be prioritized based on these factors (e.g. *EC*, *Ca*, *CoDP* and *CoV*) by using the method stated in section 4.2 and the following formula:

$$WP = \sum 4i=1(PFValue_i * PFWeight_i) \quad (11)$$

Where:

- *WP* is weight prioritization for each test case that calculates from four (4) factors, which are: *EC*, *Ca*, *CoDP* and *CoV*.
- $PFValue_i$ is a value assigned to each test case
- $PFWeight_j$ is a weight assigned for *EC*, *Ca*, *CoDP* and *CoV*.

Test cases can be ordered in accordance with higher priority. However, if the weight priority value is still the same, then go to next step.

In this step, other factors are required to be concerned as last priority. Test cases can be prioritized based on these factors (e.g. *ReqCov*, *TCplx*, *Dep* and *TI*) by using this method mentioned in section 4.2 and the following formula:

$$WP = \sum 4i=1(PFValue_i * PFWeight_i) \quad (12)$$

Where:

- *WP* is weight prioritization for each test case that calculates from four (4) factors, which are: *TCplx*, *ReqCov*, *Dep* and *TI*.
- $PFValue_i$ is a value assigned to each test case
- $PFWeight_j$ is a weight assigned for *TCplx*, *ReqCov*, *Dep* and *TI*.

Test cases can be ordered in accordance with higher priority. However, if the weight priority value is still the same, then select and prioritize test cases randomly.

It is obvious that this proposed technique is aiming to improve the ability to rank or score test cases when there are multiple cases with the same priority. This can improve the efficient of ranking in test case prioritization techniques.

In addition, the above procedures can apply to prioritize multi sets of test cases. The literature survey reveals that most test case prioritization techniques assume explicitly to prioritize a single set of test case.

4.2 Many Test Suites Prioritization Method

Current test case prioritization techniques explicitly prioritize a single set of test cases. The literature review shows that no techniques addressed a method to prioritize multiple sets of test cases / many test suites [23], [24], [25], [26], [27], [28], [29], [44], [52], [53], [54], [55] [56], [126]. In this study, test suites define as a collection of test cases. Therefore, the following has been proposed to resolve the problem of prioritizing test suites.

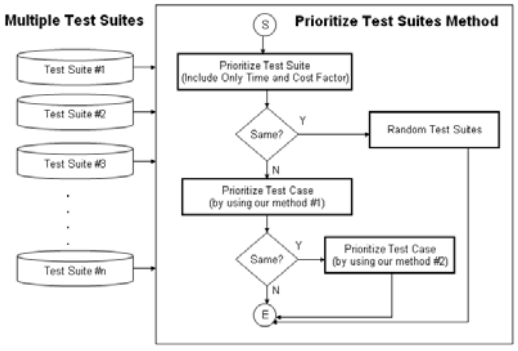

Figure 4. A Flow Chart of MTSPM Method

From the above figure, the procedures can be described as follows:

The beginning starts with prioritizing test suite along with time and cost factors. In this step, the method in section 4.2 has been proposed and used. If there are many test suites with the same priority, then select and order test suites randomly.

In addition, the following lists formula using in this step:

$$WP = \sum 7i=1(PFValue_i * PFWeight_i) \qquad (13)$$

Where:

- *WP* is weight prioritization for each test suite that calculates from seven (7) factors, which are: *EC*, *Ca*, *CoDP*, *CoV*, *ET*, *TCfDP*, and *TCfV*.
- $PFValue_i$ is a value assigned to each test suite
- $PFWeight_j$ is a weight assigned for *EC*, *Ca*, *CoDP*, *CoV*, *ET*, *TCfDP*, and *TCfV*.

If no test suites have the same priority value, then prioritizing test cases in each test suite. If there are many test cases with the same priority value, the method addressed in section 4.3 has been proposed to resolve that problem.

5. EVALUATION

This section describes an experiment design, measurement metrics and results in order to determine the most recommended test case prioritization methods.

5.1 Experiments Design

An evaluation method for this experiment has been proposed in order to compare and assess the proposed method with other current prioritization techniques, as follows:

1. Prepare Experiment Data. Generate randomly 20 test suites with 1,000 test cases. Each test suite contains the following values: test suite id, test suite purpose, a set of test cases, *EC*, *Ca*, *CoDP*, *CoV*, *ET*, *TCfDP*, *TCfV*. Also, each test case contains the following values: test case id, test case description, input, expected output, actual output and pass / fail status.

2. Run Test Suites Prioritization Method. A comparative evaluation method has been made among the following techniques: (a) random method (b) Hema's technique [32] (c) Alexy's cost-effective prioritization method [2] and (d) the "Multi-Prioritization" method presented in previous section. The experiment data used in this experiment are: 20 test suites that contain 1,000 test cases.

3. Evaluate Results. In this step, collecting results from previous steps are included. Also, plotting graph and discussing results are required to evaluate final results for the previous methods, from previous step.

5.2 Measurement Metrics

The section lists the measurement metrics used in the experiment. The section lists the measurement metrics used in the experiment. This paper proposes to use three metrics, which are: (a) percentage of high priority reserve effectiveness (b) size of acceptable test case and (c) total prioritization time.

Test case prioritization techniques aim to prioritize and schedule high-priority test cases. This is because a number of time and cost consumed in the software testing process, particularly during a regression testing process, can be significantly decreased by executing those high-priority cases first [22], [23], [30], [31] Thus, the percentage of high-priority test cases is one of the important metrics used in this experiment [51]. This experiment compares the existing test case prioritization and proposed methods to find out the best methods that reserve a maximum number of high-priority test cases. This paper proposes to use a number of acceptable test cases as another metric, because a size of prioritized test cases has an impact to the effort, time and cost consuming during the execution, particularly during a regression testing phase [7], [22], [23], [30], [31]. Thus, the smaller number of test cases consumes less effort, time and cost. This paper compares a number of reserved acceptable test cases between existing techniques and proposed method. The acceptable test cases in this experiment are test cases with high and medium priority. All low-priority test cases are excluded. Additionally, this paper proposes to use a total prioritization time as last metrics. This is because time-consuming prioritization techniques can produce a huge amount of time during the software testing process. The techniques with the least total prioritization time are desirable. The following describes details of each metric used in this experiment.

1. Percentage of High Priority Reserve Effectiveness: This metric measure the effectiveness of reserving high priority test cases from original test cases [51]. This is because high priority test cases have higher priority value more than lower priority test cases. Therefore, the high percentage of high priority reserve effectiveness is desirable.

In addition, this metric can be calculated as the following formula:

$$\% HPRE = (\# of Reserved / \# of Total)*100$$

Where:

- *% HPRE* is a percentage of high priority reserve effectiveness.
- *# of Reserved* is a number of redundancy test cases removed from original test cases.
- *# of Total* is a total number of test cases.

2. Size of Acceptable Test Cases: This metric is a number of acceptable test cases, in the format of percentage, as follows:

$$\% Size = (\# Size / \# of Total Size)*100$$

Where:

- *% Size* is a percentage of a number of acceptable test cases that exclude all low-priority test cases.
- *# of Size* is a number of test cases that each method generates, excluding low-priority test cases.
- *# of Total Size* is a maximum number of test cases in the experiment, which is assigned to 1,000.

3. Total Prioritization Time: This is a total number of times running the prioritization methods in the experiment. This metric is related to time used during pre-process and post-process of test case prioritization. Therefore, less time is desirable.

It can be calculated as the following formula:

$$TPT = ComT + CalT + RPMT$$

Where:

- *TPT* is a total number of times consuming in running prioritization methods.
- *ComT* is a time to compile source code / binary code in order to prioritize test cases.
- *CalT* is a total number of times consuming in assigning weights, assigning values and computing weight prioritization values.
- *RPMT* is a total time to run the test case prioritization methods including ordering test cases under this experiment.

Also, the following represents a formula that calculates the total time in the format of percentage.

$$\% Time = (\# TPT / \# of Maximum Total Time)*100$$

Where:

- *% Time* is a percentage of total time.
- *# of TPT* is a total time consumed during the generation process.
- *# of Maximum Total Time* is a maximum time in the experiment, which is assigned to 1,000 seconds.

5.3 Results and Discussion

This section discusses an evaluation result of the above experiment. This section presents a graph that compares the above proposed method to other three existing test case prioritization techniques, based on the following measurements: (a) high priority reserve effectiveness (b) size of acceptable priority and (c) total time. Those three techniques are: (a) random approach (b) Hema's method and (c) Alexey's method. There are two dimensions in the following graph: (a) horizontal and (b) vertical axis. The horizontal represents three measurements whereas the vertical axis represents the percentage value.

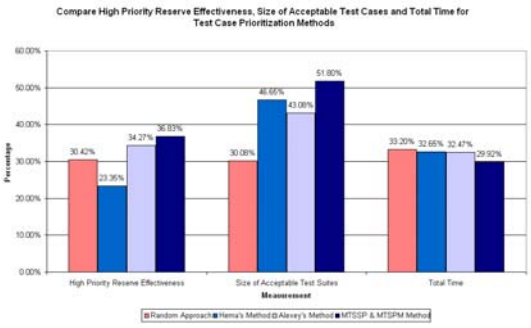

Figure 5. An Evaluation Result of Prioritization Methods

The above graph shows that the MTSSP & MTSPM method generates the highest high priority reserve effectiveness. It is calculated as 96% where as the other techniques is computed less than 70%. Those techniques reserve the less number of test cases with high priority.

Also, the graph shows that the "MTSSP & MTSPM" method consumes the least total time during a prioritization process, comparing to other techniques. It used only 80%, which is slightly less than others.

Finally, the graph presents that the "MTSSP & MTSPM" method is the best technique to reserve the acceptable priority test cases.

The following table ranks test case prioritization techniques used in the experiments, based on the

above measurements, by 1 is the first, 2 is the second, 3 is the third and 4 is the last.

Table 2. Test Case Generation Techniques Ranking

Methods	High Priority Reserve Effectiveness	Size of Acceptable Test Cases	Total Time
Random Approach	3	4	4
Hema's Method	4	3	3
Alexey's Method	2	2	2
MTSSP & MTSPM Method	1	1	1

In the conclusion, the MTSSP and MTSPM method are the most recommended methods to reserve the large number of high priority test cases with the second runner of total time, during a prioritization process.

6. CONCLUSION

Test case prioritization is a method to prioritize and schedule test cases. The technique is developed in order to run test cases of higher priority in order to minimize time, cost and effort during software testing phase. The literature review shows that many researchers propose many methods to prioritize and reduce the effort, time and cost in the software testing phase, such as test case prioritization methods, regression selection techniques and test case reduction approaches. This paper concentrates on test case prioritization techniques only. This paper shows that there are many prioritization techniques researched between 1998 and 2008. With the existing techniques, this study introduces a new "4C" type of test case prioritization techniques, which are: (a) customer requirement-based techniques, (b) coverage-based techniques, (c) cost effective-based techniques and (d) chronographic history-based techniques. First, the customer requirement-based techniques are methods to directly prioritize test cases from requirement specifications. Second, the coverage-based techniques are structural white-box testing techniques. They compare test program behavior against the apparent intention of the source code. This contrasts with functional black-box testing, which compares test program behavior against a requirements specification. In addition, they examine how the program works, taking into account possible pitfalls in structure and logic. Third, the cost effective-based techniques are methods of prioritizing test cases based on only cost factors, such as cost of analysis and cost of prioritization. Last, the chronographic history-based techniques are methods to prioritize test cases based on test execution history factors.

This paper reveals that there are many research challenges and gaps in the test case prioritization area. Those challenges and gaps can give the research direction in this field. However, the research issues that motivated this study are:

1. No existing prioritization techniques address the problem of multiple cases with same weight values. The existing test case prioritization techniques use a random approach to prioritize those cases to resolve that problem. The problem may lead to a poor performance of an ability to prioritize and schedule test cases.

2. Existing test case prioritization techniques assume explicitly that there is only a single test suite. The test suite is a collection of a set of test cases. There are no prioritization techniques to resolve the problem of multiple test suites.

This paper proposes two methods to resolve the above research issues. The first method aims to improve the ability to prioritize a set of test cases in case that there are multiple cases with the same priority weight values. The second method is developed to prioritize multiple test suites, which they contains a set of test cases. As the evaluate result, it is appeared that the above two methods are the best to reserve the large number of high priority test cases with the second runner of total time, during a prioritization process. Also, those proposed methods can resolve the issues of duplicated priority weight values and multiple test suites. However, this paper suggests the following future works to improve the capability of those two methods: (a) apply the practical prioritization weight values for commercial systems (b) improve the ability to automatically find duplicate test cases with the same values (c) improve the ability to automatically prioritize multiple large test suites with real commercial data.

REFRENCES:

[1] A. Srivastava, A. Edwards, and H. Vo., "Vulcan: Binary Transformation in a Distributed Environment", Microsoft Research Technical Report, MSR-TR-2001-50, 2001.

[2] Alexey G. Malishevsky, Gregg Rothermel and Sebastian Elbaum, "Modeling the Cost-Benefits

Tradeoffs for Regression Testing Techniques", *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, 2002.

[3] Alexey G. Malishevsky, Joseph R. Ruthruff, Gregg Rothermel and Sebastian Elbaum, "Cost-cognizant Test Case Prioritization", Technical Report TR-UNL-CSE-2006-0004, Department of Computer Science and Engineering, University of Nebraska – Lincoln, 2006.

[4] Amitabh Srivastava and Jay Thiagarajan, "Effectively Prioritizing Tests in Development Environment", In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 97-106, 2002.

[5] Jefferson Offutt, Jie Pan and Jeffery M. Voas, "Procedures for Reducing the Size of Coverage-based Test Sets", 1995.

[6] Barry W. Boehm, "A Spiral Model of Software Development and Enhancement", TRW Defense Systems Group, 1998.

[7] Boris Beizer, "Software Testing Techniques, Van Nostrand Reinhold", Inc, New York NY, 2nd edition. ISBN 0-442-20672-0, 1990.

[8] Bo Qu, Changhai Nie, Baowen Xu and Xiaofang Zhang, "Test Case Prioritization for Black Box Testing", 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), 2007.

[9] Bogdan Korel and Ali M. Al-Yami, "Automated Regression Test Generation", ISSTA98, 1998.

[10] Boris Beizer, "Software Testing Techniques", Van Nostrand Reinhold, Inc, New York NY, 2nd edition. ISBN 0-442-20672-0, 1990.

[11] B. Boehm, "Industrial Metrics Top 10 List", IEEE Software, pp. 84-85, 1987.

[12] B. Boehm, "Value-Based Software Engineering", ACM SIGSOFT Software Engineering Notes, 18(2):3-7, 2003.

[13] B. Korel and J. Laski, "Algorithmic software fault localization", Annual Hawaii International Conference on System Sciences, pages 246–252, 1991.

[14] Cem Kaner, "Exploratory Testing", Florida Institute of Technology, Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL, 2006.

[15] David Leon and Andy Podgurski, "A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases", *Proc. Int'l Symp. Software Reliability Eng.*, pp. 442-453, 2003.

[16] Dennis Jeffrey and Neelam Gupta, "Test Case Prioritization Using Relevant Slices", In *Proceedings of the 30th Annual International Computer Software and Applications Conference*, Volume 01, 2006, pages 411-420, 2006.

[17] D. Binkley, "Using semantic differencing to reduce the cost of regression testing", In the Intl. Conf. on Software Maintenance, pages 41–50, 1992.

[18] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: an approach to testing based on combinatorial design", IEEE Transactions on Software Engineering, 23(7):437–444, 1997.

[19] Dr. Mark Austin, "Test Suite Analysis: Minimization and Ordering", 2004.

[20] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry and Yves Le Traon, "Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool", 17th International Symposium on Software Reliability Engineering (ISSRE'06), 2006.

[21] E. Lehmann and J. Wegener, "Test case design by means of the CTE XL", In Proc. of the 8th European International Conf. on Software Testing, Analysis & Review (EuroSTAR 2000), 2000.

[22] Gregg Rothermel, Roland H. Untch, Chengyun Chu and Mary Jean Harrold, "Prioritizing Test Cases for Regression Testing", IEEE Transactions on Software Engineering, 2001.

[23] Gregg Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study", In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 179-188, Oxford, England, UK, 1999.

[24] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin and Christie Hong, "An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites", In *Proceedings of IEEE International Test Conference on Software Maintenance (ITCSM'98)*, Washington D.C., pp. 34-43, 1998.

[25] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne and Christie Hong, "Empirical Studies of Test-Suite Reduction", In Journal of Software Testing, Verification, and Reliability, Vol. 12, No. 4, 2002.

[26] Gregg Rothermel and Mary Jean Harrold, "A Safe, Efficient Regression Test Selection Technique", ACM Transactions on Softw. Eng. And Methodology, 6(2): 173-210, 1997.

[27] Gregg Rothermel and Mary Jean Harrold, "Analyzing Regression Test Selection Techniques", IEEE Transactions on Software Engineering, 22(8):529-551, 1996.

[28] Gregg Rothermel and Mary Jean Harrold, "A Framework for Evaluating Regression Test Selection Techniques", 1994.

[29] Gregg Rothermel, Sebastian Elbaum, Alexey Malishevsky, Praveen Kallakuri and Brian Davia, "The Impact of Test Suite Granularity on the Cost-Effectiveness of Regression Testing", 2001.

[30] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Test Case Prioritization", IEEE Transactions on Software Engineering, vol. 27, pp. 929-948, 2001.

[31] G. Mogyorodi, "Requirements-Based Testing: An Overview, Starbase Corporation", 2001.

[32] Hema Srikanth and Laurie Williams, "Requirements-Based Test Case Prioritization", North Carolina State University, ACM SIGSOFT Software Engineering, pages 1-3, 2005.

[33] Hema Srikanth, Laurie Williams and Jason Osborne, "System Test Case Prioritization of New and Regression Test Cases", In *Proceedings of the 4th International Symposium on Empirical Software Engineering (ISESE)*, pages 62–71. IEEE Computer Society, 2005.

[34] Hyunsook Do and Gregg Rothermel, "On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques", IEEE Transactions on Software Engineering, V. 32, No. 9, pages 733-752, 2006.

[35] Hyunsook Do and Gregg Rothermel, "A Controlled Experiment Assessing Test Case Prioritization Techniques via Mutation Faults", *Proceedings of the IEEE International Conference on Software Maintenance*, pages 411-420, 2005.

[36] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London, "Incremental regression testing", IEEE International Conference on Software Maintenance, pages 348–357, 1993.

[37] H. K. N. Leung and L. White, "A cost model to compare regression test strategies", In *Proc. Conf. Softw. Maint.*, pages 201–208, 1991.

[38] H. Do, G. Rothermel, and A. Kinneer, "Prioritizing JUnit Test Cases: An Empirical Assessment and Cost-Benefits Analysis," Empirical Software Eng.: An Int'l J., vol. 11, no.1, pp. 33-70, March 2006.

[39] H. Leung and L. White, "Insights into regression testing", In *Proceedings of the International Conference on Software Maintenance*, Miami, Florida, U.S.A., pages 60-69, 1989.

[40] James A. Jones and Mary Jean Harrold, "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage", In *Proceedings of the International Conference on Software Maintenance*, 2001.

[41] Jeffery von Ronne, "Test Suite Minimization: An Empirical Investigation", 1999.

[42] Jennifer Black, Emanuel Melachrinoudis and David Kaeli, "Bi-Criteria Models for All-Uses Test Suite Reduction", *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, 2004.

[43] Jung-Min Kim, Adam Porter and Gregg Rothermel, "An Empirical Study of Regression Test Application Frequency", ICSE2000, 2000.

[44] Jung-Min Kim and Adam Porter, "A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments", In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 119–129. ACM Press, 2002.

[45] J. Lee and X. He, "A methodology for test selection", Journal of Systems and Software, 13(3):177–185, 1990.

[46] John Joseph Chilenski and Steven P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing", Software Engineering Journal, Vol. 9, No. 5, pp.193-200, 1994.

[47] K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression testing in an industrial environment", Comm. Of the ACM, 41(5):81–86, 1988.

[48] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer and R. S. Roos, "Time-Aware Test Suite Prioritization", In *Proceedings of the International Symposium on Software testing and Analysis*, pages 1-12, 2006.

[49] Londesbrough, I., "Test Process for all Lifecycles", IEEE International Conference on Software Testing Verification and Validation Workshop, ICSTW'08, 2008.

[50] Lu Luo, "Software Testing Techniques: Technology Maturation and Research Strategies", Carnegie Mellon University, USA, 1999.

[51] Rajib, "Software Test Metric", QCON, 2006.

[52] Sreedevi Sampath, Sara Sprenkle, Emily Gibson and Lori Pollock, "Web Application Testing with Customized Test Requirements – An Experimental Comparison Study", 17th International Symposium on Software Reliability Engineering (ISSRE'06), 2006.

[53] Xiaofang Zhang, Baowen Xu, Changhai Nie and Liang Shi, "An Approach for Optimizing Test Suite Based on Testing Requirement Reduction", Journal of Software (in Chinese), 18(4): 821-831, 2007.

[54] Xiaofang Zhang, Baowen Xu, Changhai Nie and Liang Shi, "Test Suite Optimization Based on Testing Requirements Reduction", International Journal of Electronics & Computer Science, 7(1): 9-15, 2005.

[55] Xiaofang Zhang, Baowen Xu, Zhenyu Chen, Changhai Nie and Leifang Li, "An Empirical Evaluation of Test Suite Reduction for Boolean Specification-based Testing", The Eighth International Conference on Quality Software, 2008.

[56] Xiaofang Zhang, Changhai Nie, Baowen Xu and Bo Qu, "Test Case Prioritization based on Varying Testing Requirement Priorities and Test Case Costs", *Proceedings of Seventh International Conference on Quality Software (QSIC'07)*, 2007.

[57] Douglas Hoffman, "Cost Benefits Analysis of Test Automation", STARW, Software Quality Methods LLC., 1999.

[58] Kalyana Rao Konda, "Measuring Defect Removal Accurately", July 2005.

[59] Cristian Cadar and Dawson Engler, "Execution Generated Test Cases: How to Make Systems Code Crash Itself", Stanford University, USA, 2005.