Journal of Software Engineering

ISSN 1819-4311

Academic Journals Inc.

www.academicjournals.com

Journal of Software Engineering 4 (3): 193-214, 2010
ISSN 1819-4311

A Test Case Prioritization Method with Practical Weight Factors

Siripong Roongruangsuwan and Jirapun Daengdej
Autonomous System Research Laboratory, Faculty of Science and Technology,
Assumption University, Thailand

Abstract: Statistics gathered in past research show that testing, analysis and debugging costs usually consume over 50% of the costs associated with the development of large software systems. Specifically, regression testing has been shown to be a critically important phase of software testing and many techniques have been proposed that reduce effort, time and cost of testing, such as test case prioritization techniques, regression selection techniques and test case reduction methods. This study concentrates on a survey of test case prioritization techniques. This study classifies and organizes existing test case prioritization techniques researched since 1998 into four categories: (a) customer requirement-based techniques (b) coverage-based techniques (c) cost effective-based techniques and (d) chronographic history-based techniques. Also, this study resolves the following research problems: (a) ignoring practical weight factors (b) inefficient test case prioritization methods and (c) ignoring the size of test cases. In brief, the contributions are to: (a) collate a comprehensive set of test case prioritization techniques (b) compare these test case prioritization techniques and identify the limitations of each technique (c) propose a new classification of test case prioritization techniques (d) introduce a new continuous test case prioritization process (e) propose a new test case prioritization method along with a practical set of weight factors and (f) define specific research issues and guide future research of test case prioritization methods.

Key words: Test prioritization, prioritization method, practical prioritization method, prioritization weight factor and prioritization process

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.

Bement (NIST, 2002) 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

Corresponding Author: Siripong Roongruangsuwan, Autonomous System Research Laboratory, Faculty of Science and Technology, Assumption University, Thailand

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 US economy $59.5 billion annually. More than a third of this cost could be avoided if better software testing was performed (NIST, 2002).

Beizer (1990) 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, Srikanth (Srikanth and Williams, 2005; Rothermel *et al*., 1999-2001a; Tonella *et al*., 2006) and (McMaster and Memon, 2005-2006). Also, many empirical studies for prioritizing test cases have been conducted, like (Qu *et al*., 2008; Kim *et al*., 2000; Clempner and Medel, 2006; Graves *et al*., 2001; Yu *et al*., 2008; Rothermel *et al*., 1998-1999).

Furthermore, Rothermel (Rothermel and Harrold, 1996) gave an interesting justification of test case prioritization 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". They have proven that prioritizing and scheduling test cases are one of the most important tasks during software testing process.

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 (Rothermel *et al*., 2002). 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 lacking automation during the prioritization process.

Software testing has been widely used as a way to help engineers develop high-quality systems. Testing is an important process that is performed to support quality assurance by gathering information about the nature of the software being studied (Harrold, 2000). These activities consist of designing test cases, executing the software with those test cases and examining the results produced by those executions (Beizer, 1990) 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 (Kaner, 2006), with respect to the context in which it is intended to operate. Software Testing 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.

The next sections present techniques to reduce effort, time and cost during the software testing phase, including test case prioritization and test case reduction techniques to help testers reduce the time and cost required for running test cases.

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 (Onoma *et al.*, 1998) and can save as much as one-half of the cost of software maintenance (Beizer, 1990) However, executing a set of test cases in an existing test suite consume a huge amount of time.

Rothermel *et al.* (1999-2001a) 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.

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. 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 earliest in order to provide earlier feedback to software testing engineers and earlier detect faults. This study presents numerous techniques developed, between 2002 and 2008, that can improve a test suite's rate of fault detection.

Rothermel *et al.* (2002) 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.

This study introduces a new 4C classification of test case prioritization techniques researched in 1998-2008, based on their characteristics, as follows:

- **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 and Williams, 2005; Zhang *et al.*, 2007; Nilawar and Dascalu, 2003). Also, many weight factors have been used in these techniques, including custom-priority, requirement complexity and requirement volatility

- **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 and Podgurski, 2003; Rothermel *et al*., 1999-2001a; Bryce and Colbourn, 2006)
- **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 *et al*., 2002, 2006; Elbaum *et al*., 2004)
- **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 and Porter, 2002; Qu *et al*., 2007b)

The following sections describe the above techniques in detail.

Customer Requirement-Based Prioritization Techniques

Srikanth and Williams (2005) presented the requirements-based test case prioritization approach to prioritize a set of test cases. They built upon current test case prioritization techniques (Elbaum *et al*., 2002) 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 measures the important of testing a requirement earlier

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

Where:

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

Srikanth *et al*. (2005) 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. They 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:

$$PFV_i = \sum_{j=1}^{4} (FactorValue_{ij} * FactorWeight_j)$$

Where:

- PFV_i represents the prioritization factor value for requirement i
- $FactorValue_{ij}$ represents the value for factor j for requirement i
- $FactorWeight_j$ represents the factor weight for the jth factor for a particular product. PFV is a measure of the importance of testing a requirement

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

$$P = Vw \ (PFV_1 \ldots PFV_n)_{(n*1)} = (R^{CP}{}_1 \ldots R^{CP}{}_n \ R^{IC}{}_1 \ldots R^{IC}{}_n \ R^{RV}{}_1 \ldots R^{RV}{}_n \ R^{FP}{}_1 \ldots R^{FP}{}_n)_{(n*4)} \ (W_{CP} \ W_{RC} \ W_{FP} \ W_{RV})_{(4*1)}$$

Where:

- PFV_i represents 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
- $R_{1..n}$ represents requirements coverage of each test case
- W_{CP} represents a weight measurement for CP factor
- W_{RC} represents a weight measurement for RC factor
- W_{FP} represents a weight measurement for FP factor
- W_{RV} represents 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:

$$WPj = (\sum^{i}_{x=1} PFV_x / \sum^{n}_{j=1} PFV_y)*(1/n)$$

Where:

- WPj is an indication of the priority of running a particular test case
- PFV_i represents 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 (Boehm and Huang, 2003; Karlsson and Ryan, 1997; Mogyorodi, 2001). 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. Zhang *et al.* (2005) 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.

Nilawar and Dascalu (2003) 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.

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. Structural testing 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 and Podgurski (2003) 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 (Leon and Podgurski, 2003).

Rothermel *et al.* (1999-2001a) 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 and Memon (2007) 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 and Podgurski (2003) 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 study 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.

The test case prioritization techniques studied (Korel and Al-Yami, 1998; Brottier *et al*., 2006) 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 and Gupta (2006) presented a new approach for prioritizing test cases that is based not only on total statement coverage (also known in that study 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 (Agrawal *et al.,* 1993; Korel and Laski, 1991; Gyimothy *et al*., 1999). 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 and Gupta (2006) ordered the test cases in decreasing order of test case weight, where the weight for a test is determined as follows:

$$TW = \text{Req slice} + \text{Req exercise}$$

Where:

- TW represents a weight prioritization determined for each test case
- ReqSlice represents a number of requirements presented in the relevant slice of output for each test case
- Req exercise represents 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 and Harrold (Jones and Harrold, 2001) 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 study, they focused on MC and DC criteria as for test case reduction and prioritization, building on Rothermel's test case prioritization technique (Rothermel *et al*., 1999-2001b). Their approach uses total requirement coverage and the additional requirement coverage to weight and schedule test cases accordingly.

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 (1991) presented a cost model for regression test selection. 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 (Rothermel and Harrold, 1997), 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 G. Malishevsky, Gregg Rothermel and Sebastian Elbaum (Malishevsky *et al*., 2002) 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).

$$WP = Ca(T) + Cp(T)$$

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 (2002) 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 (Rothermel *et al*., 1999) have defined and investigated various prioritization techniques. Meanwhile, Malishevsky *et al*. (2006) and Elbaum *et al*. (2000) 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).

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.

Kim and Porter (2002) 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 and Porter (2002) 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 {h1, h2, …hn} drawn from runs of TC and á 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:

$$P \text{ is } P_0 = h_1 \text{ and } P_k = \alpha h_k + (1 - \alpha)P_{k-1},\ 0 <= \alpha <= 1,\ k >= 1$$

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. Qu *et al*. (2007a) 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.

In the conclusion, 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, customer requirement-based techniques are methods to directly prioritize test cases from requirement specifications. Second, the coverage-based technique is a structural white-box testing technique. Third, cost effective-based techniques are methods to prioritize 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. All existing test case prioritization techniques have their own advantages and disadvantages. Outstanding research issues are addressed in next section.

RESEARCH CHALLENGES

Here, provides details of the research issues related to test case prioritization techniques that motivated this study:

- **Ignore Practical Weight Prioritization Factors:** Existing test case prioritization methods consider a set of test-case coverage criteria (e.g., statements, decisions, definition-use associations, or specification items), other criteria such as risk or fault-detection effectiveness, or combinations of these criteria. They ignore the complex practical criteria such as resource constraints, time and resource consumption, configuration of software, customization of the application under test and other computer language paradigms. Examples of existing techniques that ignore the practical factors are: Srikanth's method (Srikanth *et al*., 2005), Williams's technique (Srikanth *et al*., 2005) and (Zhang *et al*., 2008)
- **Inefficient Ranking Algorithm for Test Case Prioritization:** Existing test case prioritization techniques propose a simple ranking algorithm. Some of them use a random method for ranking. Some of them ignore the relevant knowledge for their ranking. For example, Hema Srikanth (2005), Laurie Williams (Srikanth *et al*., 2005) and Gregg Rothermel and Sebastian Elbaum (Malishevsky *et al*., 2002)
- **Ignore Size of Test Case:** One of the purposes of test case prioritization and reduction techniques is to minimize size of test cases as much as possible. Small test suites that retain high fault detection are desirable

PROPOSED METHODS

Here, proposes a new 2R-2S-3R continuous test case prioritization process. Also, this section discusses a proposed method that resolves the above research problems. The proposed method aims to: (a) include practical weight prioritization factors (b) improve the ability to rank and schedule test cases during the prioritization process and (c) reserve a large number of test cases with high priority.

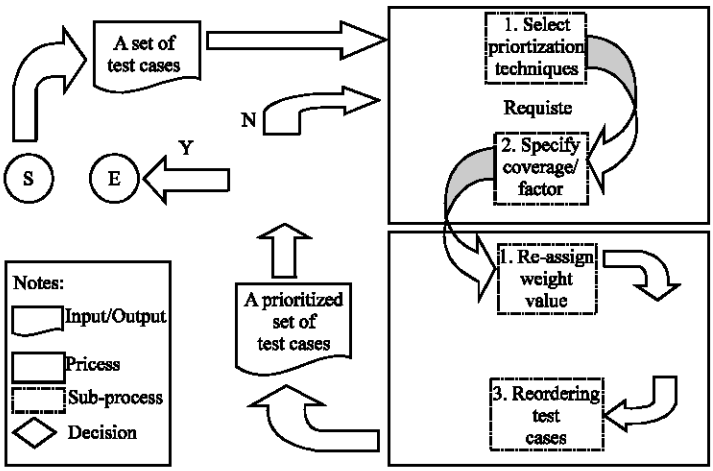

Fig. 1: A 2R-2S-3R test case prioritization process

Test Case Prioritization Process

This section introduces a new 2R-2S-3R continuous process to prioritize and schedule test cases introduced by using the above literature review and previous works (Kosindrdecha and Roongruangsuwan, 2007; Roongruangsuwan and Daengdej, 2009). Also, the new process includes a re-prioritization sub-process in order to ensure that the result of prioritization is satisfied.

Figure 1 shows a proposed test case prioritization process. The proposed process is a continuous process that allows users to continuously prioritize test cases until they are satisfied with the result. However, it starts with a large number of test cases needed to be prioritized. The process begins with a requisite process and follows with a reordering process. It simply prioritizes test cases based on given prioritization technique, coverage factors, weight value and priority value. The following elaborates the continuous process in details.

From the Fig. 1, there are two processes in the test case prioritization technique, which break down briefly as follows:

- **Requisite:** This contains two sub-processes, which are:
  - **Select Prioritization Techniques:** This sub-process is used to select type of test case prioritization techniques. There are four types of techniques: (a) customer requirement-based techniques (b) coverage-based techniques (c) cost effective-based techniques and (d) chronographic history-based technique)
  - **Specify Coverage or Factors:** This sub-process is used to identify type of factors. There are many weight factors used in the test case prioritization techniques, such as requirement, statement-coverage, code-coverage, function-coverage and cost-benefit
- **Reordering:** This consists of three sub-processes described as below:
  - **Re-assign Weight Value:** This sub-process is used to assign weight value for each test case in order to calculate weight prioritization value
  - **Re-calculate Priority Value:** This sub-process is used to calculate priority value based on assigned weights and values for each weight prioritization factors
  - **Re-order Test Cases:** This sub-process is used to schedule test cases based on the higher priority value

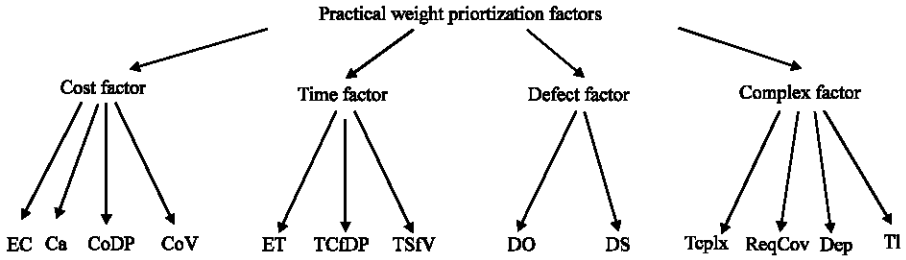

Fig. 2: Practical weight prioritization factors

However, this study proposes to include a re-prioritize process in case that the result of prioritization is not satisfied. All existing test case prioritization techniques do not include that process. Those techniques assume explicitly that the result is always satisfied.

Practical Weight Factors

This study proposes the following practical factors to prioritize test cases.

Figure 2 introduces a comprehensive set of practical weight factors for test case prioritization. This study proposes 13 factors that software test engineers should not ignore while running a test case prioritization process. Those factors are valuable and key criteria of a success of testing activities. The following describes them in details.

Figure 2, there are four groups of practical weight prioritization factors, which are: (a) cost factors (b) time factors (c) defect factors and (d) other factors. Those factors can be described in table below.

Table 1 shows practical factors for test case prioritization activities along with their explanation. The following describes the above factors in details: The above factors can be elaborated as follows:

Hoffman (1999) addressed many cost related factors, such as cost for test case execution, cost of test results analysis including validation, cost of 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 consumption factor, for data preparation (or TCfDP), is a common factor used in software testing and should therefore be applied in prioritizing test cases. Additionally, Douglas described that the dependency factor, called Dep in this study, is also a common factor used in software testing and therefore researchers should focus on this factor during the prioritization process.

Kalyana (2005) described that the time consumption for validation (or TCfV) factor is one of the most important and practical metrics in the software testing field. It is related to the effort required to validate results and find a defect. Also, Kalyana referred to the test impact (or TI) factor as a business impact that affects end-users. This factor is widely used as software testing metric. Ignoring this test may lead to the following problems: a) increased failures due to poor quality b) increased software development costs c) increased time to market due to inefficient testing and d) increased market transaction costs (NIST, 2002).

Cadar and Engler (2005), 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 cases that detect 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

Table 1: Proposed practical 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 (Malishevsky *et al*., 2002)
Cost of data preparation (CoDP)	A total cost of preparing all input values for test cases
Cost of validation (CoV)	A total cost of 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 acknowledged defects after execution.
Defects severity (DS)	A dimension for classifying seriousness for defects. Possible values are: showstopper, critical and minor severity.
4. Complex factors	
Test case complexity (TCplx)	The complexity of test cases. Measures how difficult and complex a test case is (Srikanth *et al*., 2005). In addition, its complexity usually determines the effort required to execute it (Aranha and Borba, 2006; Tai, 1980; Tsui *et al*., 2008)
Requirement coverage (ReqCov)	Number of requirements covered by test cases (Srikanth and Williams, 2005)..Requirements coverage views can help validate that all requirements are implemented in the system (Lormans and van Deursen, 2005)
Dependency (Dep)	A dependency of test cases. This factor describes how many pre-requisites are required for each test case before execution
Test impact (TI)	Impact of test cases. This factor assesses the importance of test cases, to determine if test cases are not executed

defect discovery rate (also known as DO), one of the most widely used metrics in software testing (Rajib, 2006).

The severity level of a defect indicates the potential business impact for the end user (also known as DS). The business impact factor is equal to the effect on the end user multiplied by the 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 (Konda 2005) stated 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 is always recorded in defect reports (Offutt *et al*., 1995). This consensus implies a significant rationale for prioritizing test cases by defect severity level.

The literature reviews (Aranha and Borba, 2006; Tai, 1980; Tsui *et al*., 2008) reveal that the complexity of a test case is one of the most important factors. The complexity determines the effort required to execute test cases. Also, the literature review (Lormans and van Deursen, 2005) reveals that requirement coverage, called ReqCov in this study, views can help validate that all requirements are implemented in the system.

Multi Prioritization Method

This study proposes a new test case prioritization method with the above practical weight factors, called Multi-Prioritization, in order to improve the ability to rank and prioritize test cases. The following lists steps proposed in this study.

Assign weight for each factor (e.g., EC, Ca and DO) in each group (e.g., cost factors, time factors, defect factors and other factors) by using 100-point method (Leffingwell and Widrig,

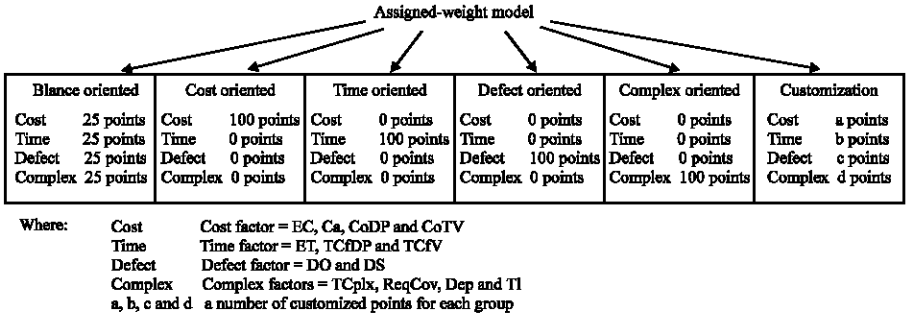

Fig. 3: Assigned-weight model

Table 2: Assigned-value model

Factors	Scale assignment
EC	A range between 1-5 by: 5 = Best, 4 = Very Good, 3 = Good, 2 = Poor and 1 = Bad.
Ca	
CoDP	
CoV	
ET	High, Medium and Low
TCfDP	
TCfV	Where High denotes the scale from 80 to 100 points, Medium denotes the scale from 40 to 70 points and Low denotes the scale from 10 to 30 points.
DO	True and False Where True denotes 100 points and False denotes 0 point. The 100-points method from (Leffingwell and Widrig, 2003) is applied to this scale assignment.
DS	Showstopper, Critical and Minor Where Showstopper denotes 50 points, Critical denotes 40 points and Minor denotes 10 points. The 100-points method from (Leffingwell and Widrig, 2003) is applied to this scale assignment.
ReqCov	Scale from 1 to 10 where 1 is the minimum value and 10 is maximum value.
TCplx	True and False
Dep	Where True denotes 100 points and False denotes 0 point. The 100-points method from (Leffingwell and Widrig, 2003) is applied to this scale assignment.
TI	High, Medium and Low Where High denotes the scale from 80 to 100 points, Medium denotes the scale from 40 to 70 points and Low denotes the scale from 10 to 30 points.

2003). Due to the fact that auto weight algorithm is beyond the scope of this study, however, the structure of weight assignment is proposed as follows:

Figure 3 presents an assigned-weight model used in the proposed prioritization method in this study. There are six approaches to assign weights for each factors: (a) balance oriented (b) cost oriented (c) time oriented (d) defect oriented (e) complex oriented and (f) customization.

Balance oriented model assigns 25 points for each group (e.g., cost, time, defect and complex). Cost oriented model focuses on only cost factors. Time oriented model assigns 100 points for all time factors. Defect oriented model also assigns 100 points for each defect factor. Complex oriented model gives 100 points for complex factors as well. The last model allows users to customize and give their own points for each group.

Assign a value for each test case. Due to the fact that auto-assigned value algorithm is very complex and beyond the scope of this study, the following assigned-value model is proposed.

Table 2 presents an approach of how to assign value for each factor in this study. This study simply use three groups: (a) range between 1 and 5 (b) high, medium and low and (c) 100 points technique proposed by Leffingwell.

Compute weight prioritization (WP) value for each test case as follows:

$$WP = \sum_{i=1}^{13}(PF\ value_i * PF\ weight_i)$$

Where:

- WP is weight prioritization for each test case calculated from 13 factors
- $PFValue_i$ is a value assigned to each test case
- $PFWeight_i$ is a weight assigned for each factor

Order test cases by WP, such that higher WP gives a test case higher priority.

EVALUATION

Here, describes the experiments design, measurement metrics and results.

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:

- **Prepare Experiment Data:** The literature survey shows that most researchers prepare experiment data to evaluate test case prioritization methods consisting of around 1,000-8,000 test cases. Generate randomly 1,000 test cases with general format such as test case id, test case description, input data and expected result
- **Run Prioritization Method:** Prioritize those 1,000 test cases by using prioritization methods. A comparative evaluation method has been made among the following techniques: (a) random method (b) Hema's technique (Srikanth *et al*., 2005) (c) Alexy's cost-effective prioritization method (Malishevsky *et al*., 2002) and (d) the Multi-Prioritization method presented in previous section. The experiment data used in this experiment is 1,000 test cases
- **Evaluate Results:** In this step, the above comparative methods are executed to rank and prioritize 1,000 test cases, for 10 times. This is because this study concentrates on the average percentage of high priority reserve effectiveness, size of acceptable test cases and total prioritization time. In total, there are 10,000 test cases executed in this experiment

Measurement Metrics

The section lists the measurement metrics used in the experiment. This study 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 identify and schedule high-priority test cases. This is because the 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 (Rothermel *et al*., 1999, 2001b). Thus, the percentage of high-priority test cases is one of the important metrics used in this experiment (Rajib, 2006). This experiment compares the existing test case prioritization techniques and proposed methods to find the methods that reserve a maximum number of high-priority test cases. This study proposes to use the number of acceptable test cases as

another metric, because the size of prioritized test cases has an impact on the effort, time and cost consumed during the execution, particularly during the regression testing phase (Beizer, 1990; Rothermel *et al*., 1999-2001b). Thus, a smaller number of test cases consumes less effort, time and cost. This study 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 study proposes to use the total prioritization time as a final metric. This is because time-consuming prioritization techniques can consume 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.

Percentage of High Priority Reserve Effectiveness

This metric measures the effectiveness of reserving high priority test cases from the set of original test cases (Rajib, 2006). 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. This metric can be calculated as the following formula:

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

Where:

- % HPRE is a percentage of high priority reserve effectiveness
- No. of reserved is the number of redundant test cases removed from the set of original test cases
- No. of total is the total number of test cases

Size of Acceptable Test Cases

This metric is the number of acceptable test cases, expressed as a percentage, as follows:

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

Where:

- % size is the number of acceptable test cases, expressed as a percentage
- No. of size is the number of test cases that each method generates, excluding low-priority test cases
- No. of total size is the total number of test cases in the experiment, which is assigned 1,000

Total Prioritization Time

This is the total number of times the prioritization methods are run in the experiment. This metric is related to the 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:

$$\text{TPT} = \text{ComT} + \text{CalT} + \text{RPMT}$$

Where:

- TPT is the total amount of time consumed in running the prioritization methods
- ComT is the time to compile source code in order to prioritize test cases
- CalT is the total amount of time consumed in assigning weights, assigning values and computing weight prioritization values
- RPMT is the total time to run the test case prioritization methods including ordering test cases

RESULTS AND DISCUSSION

Here, 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 4 represents an evaluation result of comparing an effectiveness of high priority reservation, a number of acceptable priority cases and total prioritization time. The above graph showed that the above proposed method generated the highest high priority reserve effectiveness. It was calculated as 46.76% where as the other techniques was computed less than 40%. Those techniques reserved the less number of test cases with high priority. Also, the graph showed that the proposed method consumes the least total time during a prioritization process, comparing to other techniques. It used only 43.30%, which is slightly less than others. Finally, the graph presented that the proposed method is the second best technique to reserve the acceptable priority test cases.

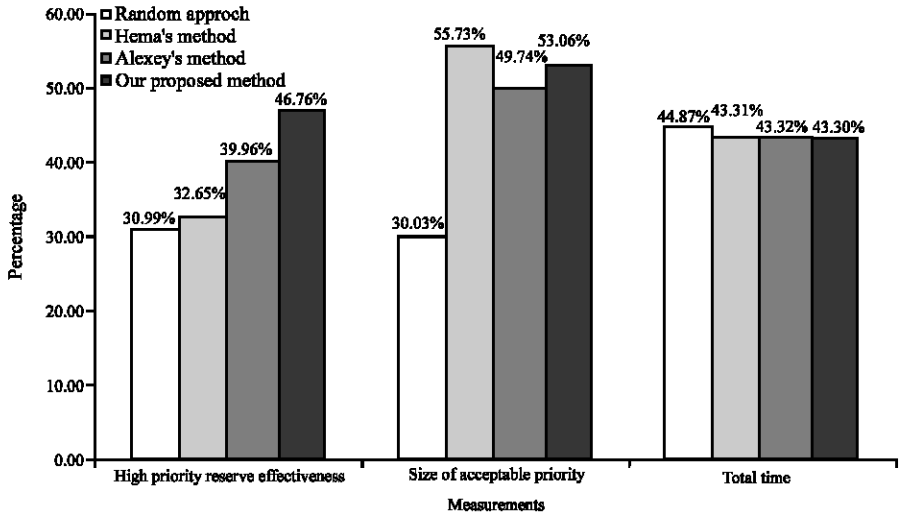

Fig. 4: An evaluation result of test case prioritization methods

DISCUSSION

This section discusses the previous evaluation result. The following table ranked 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 3 shows a ranking of each comparative test case prioritization method. In the table, it is concluded that our proposed method is the most recommended prioritization technique to reserve a large number of high priority test cases while minimizing a total prioritization time. Also, it shows that our proposed method is not worst than other techniques in term of preserving a number of acceptable test cases.

Figure 4, this study determines and ranks the above comparative methods into five ranking: 5-Excellent, 4-Very good, 3-Good, 2-Normal and 1-Poor. This study uses a maximum and minimum value to find an interval value for ranking those methods.

For an effectiveness of high priority test cases reservation, the maximum and minimum percentage is 46.76% and 30.99%. The different between maximum and minimum value is 15.77%. An interval value is equal to a result of dividing the different values by 5. As a result, the interval value is 3.154. Thus, it can be determined as follows: 5-Excellent (since 43.606 to 46.76%), 4-Very good (between 40.452 and 43.605%), 3-Good (between 37.298 and 40.451%), 2-Normal (between 34.144 and 37.2988%) and 1-Poor (from 30.99 to 34.143%).

For a number of acceptable test cases, the maximum and minimum percentage is 55.73 and 30.03%. The different value is 25.7%. The interval value is 5.14. Therefore, it can be determined as follows: 5-Excellent (since 50.59 to 55.73%), 4-Very good (between 45.45 and 50.58%), 3-Good (between 40.31 and 45.44%), 2-Normal (between 35.17 and 40.30%) and 1-Poor (from 30.03 to 35.16%).

For a total prioritization time, the maximum and minimum percentage is 44.87 and 43.30%. The different between maximum and minimum value is 1.57%. An interval value is equal to a result of dividing the different values by 5. As a result, the interval value is 0.314. Thus, it can be determined as follows: 5-Excellent (since 43.3 to 43.614%), 4-Very good (between 43.614 and 43.928%), 3-Good (between 43.928 and 44.242%), 2-Normal (between 44.242 and 44.556%) and 1-Poor (from 44.556 to 44.87%).

Therefore, the experiment result of those four comparative methods can be shown in Table 4.

The above result suggests that our proposed method is perfectly suitable for a scenario that concentrates on reserving a large number of high priority test cases, preserving acceptable cases and minimizing total prioritization time. Our proposed method is by far better than other three methods in term of high priority reserve effectiveness. Hema's method

Table 3: Test case generation techniques ranking table

Methods	High priority reserve effectiveness	Size of acceptable priority	Total time
Random approach	4	4	4
Hema's method	3	1	2
Alexey's method	2	3	3
The proposed method	1	2	1

Table 4: A comparison of test case reduction methods

Algorithm	High priority reserve effectiveness	No. of acceptable test cases	Total time
Random method	1	1	1
Hema's method	1	5	5
Alexey's method	3	4	5
Our proposed method	5	5	5

and our method are top two excellent prioritization methods for reserving medium priority test cases. Finally, the random approach consumes the greatest prioritization time comparing to other three methods.

CONCLUSION AND FUTURE WORK

This study proposes a new test case prioritization process, called 2R-2S-3R. The new process contains two processes, named 2R: (a) requisite and (b) reordering. The first process consists of two sub-processes, called 2S, which are: (a) select test case prioritization technique and (b) specify coverage or factors. The second process is composed of three sub-processes, called 3R, included as follows: (a) re-assign weight value (b) re-calculate priority value and (c) re-order test cases. This study reveals that there are many research challenges and gaps in the test case prioritization area. However, this study focus on solving the following research issues: (a) a lack of practical weight factors (b) an inefficient ranking algorithm used in the prioritization process and (c) ignore to reserve the high priority test cases. This study introduces a new practical set of weight factors used in the test case prioritization process. The new set is composed of four groups: (a) cost (b) time (c) defect and (e) complex. Also, this study proposes to improve the ability to weight and rank test cases with practical factors. This study compares the proposed method to other existing test case prioritization methods, which are: (a) random approach (b) Hema's technique and (c) Alexey's work. Consequently, this study reveals that the proposed method is the most recommended method to reserve the large number of high priority test cases with the least total time, during a prioritization process. However, there is an improvement to maintain and reserve the acceptable numbers of test cases, carried out in the future works.

REFERENCES

Agrawal, H., J.R. Horgan, E.W. Krauser and S.A. London, 1993. Incremental regression testing. Proceedings of the IEEE International Conference on Software Maintenance, Sep. 27-30, Montreal, Quebec, Canada, pp: 348-357.

Aranha, E. and P. Borba, 2006. Measuring test execution complexity. Proceedings of the International Workshop on Predictor Models in Software Engineering, (PMSE'06) Informatics Center Federal University of Pernambuco, pp: 1-2.

Beizer, B., 1990. Software Testing Techniques. 2nd Edn., Van Nostrand Reinhold, New York, ISBN: 0-442-20672-0, pp: 550.

Boehm, B. and L.G. Huang, 2003. Value-based software engineering: A case study. Computer, 36: 33-41.

Brottier, E., F. Fleurey, J. Steel, B. Baudry and Y.L. Traon, 2006. Metamodel-based test generation for model transformations: An algorithm and a tool. Proceedings of 17th International Symposium on Software Reliability Engineering, Nov. 7-10, Raleigh, pp: 85-94.

Bryce, R.C. and C. Colbourn, 2006. Prioritized interaction testing for pair-wise coverage with seeding and constraints. J. Inform. Software Technol., 48: 960-970.

Bryce, R.C. and A.M. Memon, 2007. Test suite prioritization by interaction coverage. Proceedings of the Workshop on Domain Specific Approaches to Software Test Automation: In Conjunction with the 6th ESEC/FSE Joint Meeting, Sept. 4, ACM, New York, USA., pp: 1-7.

Cadar, C. and D. Engler, 2005. Execution generated test cases: How to make systems code crash itself. Proceeding of the 20th ACM Symposium on Operating Systems Principles, March 25, Stanford University, USA., pp: 1-14.

Clempner, J. and J. Medel, 2006. Prioritizing information systems implementation using the amalgamation of lattice structures. Inform. Technol. J., 5: 74-82.

Elbaum, S., A.G. Malishevsky and G. Rothermel, 2000. Prioritizing test cases for regression testing. Software Eng. Notes, 25: 102-112.

Elbaum, S., A. Malishevsky and G. Rothermel, 2002. Test case prioritization: A family of empirical studies. IEEE Trans. Software Eng., 28: 159-182.

Elbaum, S., G. Rothermel, S. Kanduri and A.G. Malishevsky, 2004. Selecting a cost-effective test case prioritization technique. Software Qual. J., 12: 185-210.

Graves, T.L., M.J. Harrold, J.M. Kim, Ad. Porter and G. Rothermel, 2001. An empirical study of regression test selection techniques. ACM Trans. Software Eng. Methodol., 10: 184-208.

Gyimothy, T., A. Beszedes and I. Forgacs, 1999. An efficient relevant slicing method for debugging. Proceedings of ACM/SIGSOFT Foundations of Software Engineering, November 1999, New York, USA., pp: 303-321.

Harrold, M.J., 2000. Testing: A roadmap. Proceedings of the International Conference on Software Engineering, June 04-11, Limerick, Ireland, pp: 61-72.

Hoffman, D., 1999. Cost benefits analysis of test automation. Proceedings of the SoftwareTesting Analysis and Review Conference, (STARC '99) Orlando, FL., USA., pp:1-14.

Jeffrey, D. and N. Gupta, 2006. Test case prioritization using relevant slices. Proc. 30th Ann. Int. Comp. Software Appl. Conf., 1: 411-420.

Jones, J.A. and M.J. Harrold, 2001. Test-suite reduction and prioritization for modified condition/decision coverage. Proceedings of the 17th IEEE International Conference on Software Maintenance, Nov. 07-09, IEEE Computer Society Washington, DC, USA., pp: 92-92.

Kaner, J.D.C., 2006. Exploratory testing. Proceeding of the Quality Assurance Institute Worldwide Annual Software Testing Conference, Nov. 17, Orlando, FL., pp: 1-47.

Karlsson, J. and K. Ryan, 1997. A cost-value approach for prioritizing requirements. IEEE Software, 14: 67-74.

Kim, J.M., A. Porter and G. Rothermel, 2000. An empirical study of regression test application frequency. Proceedings of the 22nd International Conference on Software Engineering, June 04-11, ACM, New York, USA., pp: 126-135.

Kim, J.M. and A. Porter, 2002. A history-based test prioritization technique for regression testing in resource constrained environments. Proceedings of the 24th International Conference on Software Engineering, May 19-25, ACM Press, pp: 119-129.

Konda, K.R., 2005. Measuring defect removal accurately. Software Test Performance, 2: 35-39.

Korel, B. and A.M. Al-Yami, 1998. Automated regression test generation. Software Eng. Notes, 23: 143-152.

Korel, B. and J. Laski, 1991. Algorithmic software fault localization. Proc. 24th Ann. Hawaii Int. Conf. Syst. Sci., 20: 246-252.

Kosindrdecha, N. and S. Roongruangsuwan, 2007. Reducing test case created by path oriented test case generation. Proceedings of the AIAA Conference and Exhibition, (AIAACE'07), Rohnert Park, California, USA., pp: 1-1.

Leffingwell, D. and D. Widrig, 2003. Managing Software Requirements: A Use Case Approach. 2nd Ed., Addison-Wesley, Boston, MA., pp: 124–125.

Leon, D. and A. Podgurski, 2003. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. Proceedings of the 14th International Symposium on Software Reliability Engineering, Nov. 17–21, IEEE Computer Society Washington, DC, USA., pp: 442-453.

Leung, H.K.N. and L. White, 1991. A cost model to compare regression test strategies. Proceedings Conference on Software Maintenance, Nov. 15, IEEE Computer Society Press, pp: 201-208.

Lormans, M. and A. van Deursen, 2005. Reconstructing requirements coverage views from design and test using traceability recovery via LSI. Proceedings of the 3rd International Workshop on Traceability in Emerging forms of Software Engineering, Nov. 08, ACM New York, USA., pp: 37-42.

Malishevsky, A., G. Rothermel and S. Elbaum, 2002. Modeling the cost-benefits tradeoffs for regression testing techniques. Proceedings of the International Conference on Software Maintenance, Oct. 03-06, IEEE Computer Society, Washington, DC., USA., pp: 204-204.

Malishevsky, A.G., J.R. Ruthruff, G. Rothermel and S. Elbaum, 2006. Cost-cognizant test case prioritization. Technical Report TR-UNL-CSE-2006-0004, Department of Computer Science and Engineering, University of Nebraska-Lincoln. http://citeseerx.ist.psu.edu/viewdoc/summary? doi=10.1.1.112.9150.

McMaster, S. and A. Memon, 2005. Call stack coverage for test suite reduction. Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), Sept. 26-29, Budapest, Hungary, pp: 539-548.

McMaster, S. and A. Memon, 2006. Call stack coverage for GUI test-suite reduction. Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering, Nov. 7-10, Raleigh, NC., pp: 33-44.

Mogyorodi, G., 2001. Requirements-based testing: An overview. Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems, July 29-Aug. 03, Santa Barbara, California, pp: 286-295.

NIST., 2002. The economic impacts of inadequate infrastructure for software testing. http://www.nist.gov/director/planning/upload/report02-3.pdf.

Nilawar, M. and S. Dascalu, 2003. A UML-based approach for testing web applications. University of Nevada, Reno, http://www.imamu.edu.sa/DContent/IT_Topics/A%20UML-Based%20Approach%20for%20Testing%20Web%20Applications.pdf.

Offutt, A.J., J. Pan and J.M. Voas, 1995. Procedures for reducing the size of coverage-based test sets. Proceedings of the 12th International Conference on Testing Computer Software, June 1995, Washington, DC., pp: 111-123.

Onoma, K., W.T. Tsai, M. Poonawala and H. Suganuma, 1998. Regression testing in an industrial environment. Comm. ACM, 41: 81-86.

Qu, B., C. Nie, B. Xu and X. Zhang, 2007a. Test case prioritization for black box testing. Proc. 31st Ann. Int. Comp. Software Appl. Conf., 10: 465-474.

Qu, X., M.B. Chohen and K.M. Woolf, 2007b. Combinatorial interaction regression testing: a study of test case generation and prioritization. Proceedings of the IEEE International Conference on Software Maintenance, Oct. 2-5, University of Nebraska-Lincoln, Lincoln, pp: 255-264.

Qu, X., M.B. Cohen and G. Rothermel, 2008. Configuration-aware regression testing: an empirical study of sampling and prioritization. Proceedings of the 2008 International Symposium on Software Testing and Analysis, July 20-24, ACM, New York,USA., pp: 75-86.

Rajib, R., 2006. Software test metric. QCON.

Roongruangsuwan, S. and J. Daengdej, 2009. Test case reduction. Technical Report 25521, Assumption University, Thailand.

Rothermel, G. and M.J. Harrold, 1996. Analyzing regression test selection techniques. IEEE Trans. Softw. Eng., 22: 529-551.

Rothermel, G. and M.J. Harrold, 1997. A Safe, efficient regression test selection technique. ACM Trans. Softw. Eng. Methodol., 6: 173-210.

Rothermel, G., M.J. Harrold, J. Ostrin and C. Hong, 1998. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In Proceedings of the 14th IEEE International Test Conference on Software Maintenance, Nov. 16-20, Bethesda, Maryland, pp: 34-43.

Rothermel, G., M.J. Harrold, J. von Ronne and C. Hong, 2002. Empirical studies of test-suite reduction. J. Software Test. Verificat. Reliability, 12: 219-249.

Rothermel, G., R.H. Untch, C. Chu and M.J. Harrold, 1999. Test case prioritization: An empirical study. In Proceedings of the 15th IEEE International Conference on Software Maintenance, Aug. 30-Sept. 03, Oxford, England, pp: 179-188.

Rothermel, G., R.H. Untch, C. Chu and M.J. Harrold, 2001a. Prioritizing test cases for regression testing. IEEE Trans. Software Eng., 27: 929-948.

Rothermel, G., S. Elbaum, A. Malishevsky and P. Kallakuri, 2001b. The impact of test suite granularity on the cost-effectiveness of regression testing. Proceedings of the International Conference Software Engineering, May 2001, University of Nebraska-Lincoln, pp: 230-240.

Srikanth, H. and L. Williams, 2005. On the economics of requirements-based test case prioritization. Proceedings of the 7th International Workshop on Economics-Driven Software Engineering Research, May 15-15, ACM New York, NY, USA., pp: 1-3.

Srikanth, H., L. Williams and J. Osborne, 2005. System test case prioritization of new and regression test cases. Proceedings of the 4th International Symposium on Empirical Software Engineering, Nov. 17-18, IEEE Computer Society, pp: 10-10.

Tai, K.C., 1980. Program testing complexity and test criteria. IEEE Trans. Software Eng., SE-6: 531-538.

Tonella, P., P. Avesani and A. Susi, 2006. Using the case-based ranking methodology for test case prioritization. Proceedings of the 22nd IEEE International Conference on Software Maintenance, Sept. 24-27, Philadelphia, Pennsylvania, pp: 123-133.

Tsui, F., O. Karam and S. Iriele, 2008. A test complexity metric based on dataflow testing technique. Internal Report, School of Computing and Software Engineering, Southern Polytechnic State University, July, 2008.

Yu, Y., J.A. Jones and M.J. Harrold, 2008. An empirical study of the effects of test-suite reduction on fault localization. Proceedings of the 30th International Conference on Software Engineering, May 10-18, ACM, New York, pp: 201-210.

Zhang, X., B. Xu, C. Nie and L. Shi, 2005. Test suite optimization based on testing requirements reduction. Int. J. Electronics Comput. Sci., 7: 9-15.

Zhang, X., B. Xu, C. Nie and L. Shi, 2007. An approach for optimizing test suite based on testing requirement reduction. J. Software, 18: 821-831.

Zhang, X., B. Xu, Z. Chen, C. Nie and L. Li, 2008. An empirical evaluation of test suite reduction for boolean specification-based testing. Proceedings of the 8th International Conference on Quality Software, Aug. 12-13, IEEE Computer Society, Washington, DC., USA., pp: 270-275.