Prioritising test cases by collaborating artefacts of software development life cycle

Md Saeed Siddik*

Institute of Information Technology,
University of Dhaka,
Dhaka, Bangladesh
Email: saeed.siddik@iit.du.ac.bd
*Corresponding author

Md Abdur Rahman

Centre for Advanced Research in Sciences,
University of Dhaka,
Dhaka, Bangladesh
Email: mukul.arahman@gmail.com

Kazi Sakib

Institute of Information Technology,
University of Dhaka,
Dhaka, Bangladesh
Email: sakib@iit.du.ac.bd

Abstract: Test case prioritisation reorders test cases based on faulty module detection capability to detect maximum faults by minimum execution. Since 40% budget is allocated for software testing, prioritisation can reduce that budget by early fault detection. In software development life cycle (SDLC), faults are propagated to the connected phase and continuing until final release, because every phase depends on its previous phase's outcomes. This motivation derived prioritisation approach by focusing collaboration of different SDLC phases named requirements, design and code. Since, each of the SDLC phase has its own impact on test case, unique priority constants are assigned to every phase, which are used for constructing final priorities. The proposed framework was experimented on different projects, and results have been compared to several prominent schemes considering individual phase of SDLC. On average, proposed collaborative approach performs 22.77% and 29.01% better than individually requirements and source code based prioritisation techniques respectively.

Keywords: regression testing; software development life cycle; SDLC; requirement; test case prioritisation.

Reference to this paper should be made as follows: Siddik, M.S., Rahman, M.A. and Sakib, K. (2019) 'Prioritising test cases by collaborating artefacts of software development life cycle', *Int. J. Forensic Software Engineering*, Vol. 1, No. 1, pp.47–72.

Biographical notes: Md Saeed Siddik have been working on software testing and software analysis research where he experimented how software are developed and tested efficiently. He has completed his MSc in Software Engineering, including the highest marked thesis dissertation on software test case prioritisation from the IIT University of Dhaka. He was the first research student of IITDU Optimization Research Group, where he was working on software design migration to enhance modularity and manageability. He is also a member of IEEE, SIGSOFT, and a group advisor of IEEE CS SB at the University of Dhaka.

Md Abdur Rahman received his BSc in Information Technology from the Visva Bharati University, India in 2004. He has completed his Post Graduate Diploma and his Master in Information Technology from the University of Dhaka, Bangladesh in 2008 and 2009 respectively. He is a Senior Computer Scientist in the Centre for Advanced Research in Sciences at the University of Dhaka. His major research interest includes natural language processing, machine learning, deep learning, big data analytics, and software engineering.

Kazi Sakib is a Professor at the Institute of Information Technology (IIT), University of Dhaka, Bangladesh. He received his PhD in Computer Science at the School of Computer Science and Information Technology, RMIT University. His research interests include software engineering, cloud computing, software testing, software maintenance, etc. He is an author of a great deal of research studies published at national and international journals as well as conference proceedings.

This paper is a revised and expanded version of a paper entitled ‘RDCC: an effective test case prioritization framework using software requirements, design and source code collaboration’ presented at the 17th International Conference on Computer and Information Technology (ICCIT), IEEE, Bangladesh, 22 December 2014.

1 Introduction

Test case prioritisation reorders test cases for providing earlier feedback to software testers and managers about faulty modules, which are exigent specially in regression testing. This type of testing revalidates the modified software when new component is included to adapt the change requirements which may adversely impact on existing software. This revalidating procedure considers both old and new test cases to ensure software functionalities, which is costly and time consuming (Rothermel et al., 2001; Srikanth and Williams, 2005; Hasan et al., 2017). Therefore, various test case selection techniques are introduced to improve regression testing efficiency which are categorised as test suite reduction, selection and prioritisation (Rothermel et al., 2001). Although reduction and selection reduce regression testing reliability by omitting test cases, whereas prioritisation selects the appropriate set in terms of fault detection without omitting any test cases. In prioritisation schemes, priority values are assigned to all test cases based on their accuracy of fault detection. Test cases are prioritised before the software release using related information such as requirements, source code, etc. Since, testing is a vital part of software development life cycle (SDLC), other phases of SDLC are intimately related to test cases (Pressman, 2005).

Several test case prioritisation schemes have been proposed to increase regression testing effectiveness in terms of fault detection rate which can be categorised into several groups named as software requirements-based (Arafeen and Do, 2013; Srivastva et al., 2008), code coverage-based (Rothermel et al., 2001; Haidry and Miller, 2013), without considering SDLC information directly (Li et al., 2010; Malhotra and Tiwari, 2013), etc. Collaborating requirements' risk and severity Srivastva et al. (2008) proposed prioritisation approach for tracing the potential test cases. Arafeen and Do (2013) presented a requirement-based prioritisation approach, where similar requirements are grouped together based on their textual similarities. Clusters were created using these similarity values, where similar requirements were assigned to the same cluster. Identifying critical component in software source code Mala and Praba (2011) implemented test case prioritisation approach to improve code coverage where computationally critical components were identified from software and suggested their related test cases for early execution.

Using additional function coverage (AFC) Elbaum et al. (2002) presented a code coverage-based approach for early fault detection where the functions are the major part to assign weights on a test case. Genetic algorithm-based prioritisation tool has been implemented by Islam et al. (2012a) to identify appropriate test case ordering with respect to three different dimensions which are structure, function and cost. Malhotra and Tiwari (2013) proposed a prioritisation framework that emphasised the rate of code coverage by incorporating knowledge about the significance of code blocks.

However, mentioned approaches have been proposed by considering either requirement or source code for prioritisation, which may lead an incomplete view. Since, testing is one of the last phase of SDLC, any previous phases' errors or faults may propagate to this phase. On the other hand, every phase has its' unique view points for representing a software. That is why, all of these schemes would become more appropriate by incorporating every previous SDLC phases together, because every phase has considerably unique impact on test cases (Pressman, 2005).

In order to address the discussed scenario, this paper presents an effective test case prioritisation framework named as requirements, design diagrams, and source code collaboration (RDCC) where all the SDLC phases are considered. For test case mapping, every requirement ID is uniquely identified as RDCC ID. Collaboration from requirements to design diagrams, requirements to source code and requirements to test cases are used to detect priority values. The proposed framework is experimented on different software projects and results are compared to several prominent software test case prioritisation schemes like requirements or source code-based prioritisation technique. On average, proposed collaborative approach performs 22.77% and 29.01% better than other implemented test case prioritisation schemes considering individual phase of SDLC. The initial tasks of this research has been published in Siddik and Sakib (2014). The major contributions of the research are listed below.

1 Requirements, corresponding to each test case, are analysed as textual similarity by calculating term frequency and inverse document frequency. Design diagrams, corresponding to each requirements, are extracted as readable XML format to test the connectivity among software modules. Source code (developed based on design diagram) are parsed as software code metrics (CMs) to calculate code

priority for either classes or functions related to each requirements. Those three priorities are used to assign final priority values to test cases for early fault detection and minimisation of test case execution.

2 Two versions of priority constants are experimented in this research, where the first one is assigning equal priority to every phases of SDLC and another is different priorities based on error propagation among different phases of SDLC. Priority constants of different phases are also normalised for computational simplicity. The cumulative priority values of different phases of SDLC are assigned to related test cases. Finally test cases are sorted in descending order based on their priority values which are assigned by collaborative SDLC information.

3 It has been demonstrated experimentally that the collaborative information from different phases of SDLC are more effective for test case prioritisation than considering any individual phases [e.g., requirements (Srivastva et al., 2008) and source code (Elbaum et al., 2002)]. In addition, assigning different priorities based on error propagation performs better than assigning similar priorities. By integrating those phases, proposed RDCC scheme overcomes the limitations of prioritisation techniques, which analyse individual SDLC phase.

The remainder of this article is organised as follows: Section 2 provides the background studies of this research, whereas a detailed literature survey of test case prioritisation are discussed in Section 3. The details of newly proposed prioritisation framework is described at Section 4. Experiments and result comparisons of proposed framework are described in Section 5. Finally conclusions and future work of this research are presented in Section 6.

2 Background

Software testing (Avila-George et al., 2013) is an inseparable part of SDLC to validate the final product against users' expectations. Although testing phase is executed at the end of SDLC, about 30% to 40% time and cost are allocated for this phase (Rothermel et al., 2001). By prioritising those test cases, total software development budget can be reduced.

2.1 Test case prioritisation

Test case prioritisation determines the proper re-orderings of test cases so that faults can be detected early (Rothermel et al., 2001; Rahman et al., 2018). It does not always confirm to detect faults early but maximises the probability to detect early faults every time (Islam et al., 2012b). The objectives of test case prioritisation techniques are to identify the optimal order of the test cases that are effective (in terms of capability of early detecting faults) and efficient (in terms of test case execution numbers). More specifically, the test case prioritisation is a process to identify maximum faults in minimum test case execution.

Test case prioritisation is formulated as a computational problem in the very first time (Rothermel et al., 2001). According to their formulated problem, test case prioritisation is defined as:

Given:

- (T): a test suite
- (PT): the set of permutations of T
- (f: PT $\rightarrow$ R): a function from PT to real numbers

Problem:

to find $T' \in PT$ such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geqslant f(T'')]$.

Here, PT represents the set of all possible prioritisation schemes of T and f is a function that, applied to any such ordering, yields an award value for that ordering.

2.2 Traceability matrix

Test cases have a sheer connection between requirements. A traceability matrix is a document usually in tabular form, containing that relationship. Test case IDs (e.g., TC1, ..., TCm) including its corresponding requirement IDs (e.g., Req1, ..., Req n) are denoted by the row of that table. Traceability matrix would be generated for implementing any test case prioritisation schemes, because this is the only matrix where test cases and requirements are mapped. A sample traceability matrix is presented in Table 1, containing m test cases and n requirements.

Table 1 Traceability matrix sample

Test case	*Req1*	*Req2*	*Req3*	*Req4*	*...*	*Req n*
TC 1	*				...	
TC 2				*	...	
TC 3		*	*		...	
...	...	...	...	...	...	
TC m						mn

2.3 Connection between test cases and SDLC phases

Test cases are basically written to validate final product against requirements. On the other hand, modules in design diagrams are drawn based on their corresponding requirement IDs. Finally, source codes are developed for every design module. Sample interconnections among different phases of SDLC are presented in Table 2, where one requirement ID might be connected to multiple design modules or source classes. In this case, several test cases are needed to fulfill that requirement, for example, in Table 2 requirement 1 is connected with design modules 1 and 4, source code class sample1, and test cases 1 and 4.

Table 2 Sample interconnection among different phases of SDLC

Requirement specification	*Design diagram*	*Source code*	*Test case*
Req 1	Module 1, module 4	sample1.class	TC 1, TC 4
Req 2	Module 2	sample2.class	TC 2
Req 3	Module 3	sample3.class	TC 3, TC 4, TC 5
...	...	...	...

3 Literature review of test case prioritisation

Because of its importance and effectiveness in large scale software testing, in recent years, researchers have investigated different approaches of test case prioritisation. Existing researches on test case prioritisation can be divided into different groups such as analysing software requirements to detect faulty modules earlier (Arafeen and Do, 2013; Srivastva et al., 2008), processing source code to get maximum code coverage (Rothermel et al., 2001; Haidry and Miller, 2013; Elbaum et al., 2002), etc. This section describes those grouped prioritisation schemes including their methodologies, experiments and limitations.

3.1 Prioritisation by analysing software requirements

Test cases are usually written based on software requirements, the key part of SDLC. Software requirements reflect the requirement engineers' viewpoints, customers' feedback and priorities, etc. (Sommerville, 2004), which are needed for prioritising test cases. That is why, researchers pointed the use of requirements during software test case prioritisation (Srikanth and Williams, 2005; Arafeen and Do, 2013; Srivastva et al., 2008).

Srivastiva et al. collaborated risk and severity factor (RSF) for tracing the potential test cases. Different values were manually assigned to every software requirements from customers and developers which are added to the risk factor of those requirements. Probability factors were assigned to each requirement based on their importance (Srivastva et al., 2008). This process finally delivered a numeric value of weighted risk and requirements priority by multiplying associated risk and probability values. This process was validated by an in-house devolved software. However, there was also be a lack of integrating different software information such as requirements traceability, source code priority, etc. which might this result questionable. Because, according to Islam et al., without considering source code and requirements, prioritisation approaches can not cover all possible segments (Islam et al., 2012b).

Arafeen et al. presented a requirement-based prioritisation approach, where similar requirements are grouped together based on their textual similarities. Clusters were created using these similarity values, where similar requirements were assigned to the same cluster. To perform prioritisation, several types of information were being used such as requirements-traceability matrix, requirements modification history (Arafeen and Do, 2013), etc. The empirical results showed that clustering approach significantly outperformed the techniques without clustering. However, reported results would be more accurate by considering the corresponding design diagrams for prioritising requirements, because design diagrams have idiosyncratic uniqueness to present a software.

3.2 Prioritisation by analysing software source code

Source code is the most important part of SDLC, because of its programed logic, scripts, etc. (Harman, 2010), which are used to run software test cases. It was researched that, information from source code analysis can lead effective prioritisation approaches (Haidry and Miller, 2013; Elbaum et al., 2002).

In every software, some components become critical and fault prone, because of their high functionality and dependability with other components. On the other hand, poor quality of any software components adversely affects the quality of the overall system (Balsamo et al., 1998). Mala and Praba (2011) proposed test case prioritisation approach using critical component identification in software source code. Their approach identified the computationally critical components from software and suggested their related test cases for early execution.

The AFC is a code-based prioritisation approach for early fault detection where the functions are the major part to assign weights on a test case (Elbaum et al., 2002). This prioritisation technique depends on information relating the test suite to various elements of source code of the original system like statements, classes, functions, etc. (Haidry and Miller, 2013; Elbaum et al., 2002). For example, a particular code-based technique can utilise information about the number of functions executed, or the number of blocks of code executed, by a test. To achieve early fault detection, Elbaum et al. (2002) presented several code-based prioritisation techniques including AFC. This approach was proposed by analysing source code only, where information from source code are used to prioritise test cases.

3.3 Other test case prioritisation schemes

Several prioritisation schemes have also been proposed excluding the SDLC information. Among them heuristic approaches (Li et al., 2010; Islam et al., 2012a), trace events technique (Rajarathinam and Natarajan, 2013), graph theoretic approaches (Ramanathan et al., 2008), etc. are most prominent. Those approaches are basically developed using the combination of test cases to find the best one. Researchers introduced some heuristic algorithms to predict the approximate order of test cases for prioritising such as hill climbing (Li et al., 2010), genetic algorithm (Malhotra and Tiwari, 2013), etc.

Islam et al. (2012a) presented a software tool named multi-objective test case prioritisation technique (MOTCP) to achieve both code and requirements coverage. Genetic algorithm was used to identify appropriate test case ordering with respect to three different dimensions which are structure, function and cost. In this scenario, the structural dimension was related to source code and test cases under analysis. On the other hand, the functional dimension was about how test cases exercise requirements, whereas the cost dimension was concerned to the test execution time. Malhotra and Tiwari (2013) proposed a framework for test case prioritisation that emphasised the rate of code coverage by incorporating knowledge about the significance of blocks of code.

This approach used a newly proposed metric as fitness evaluation function in a genetic algorithm in order to evaluate the effectiveness of a test case sequence. The result would be more efficient by considering software requirement, because requirement has inauguration viewpoint of a software.

Sangaiah et al. (2018) proposed a software risk assessment framework to rank notable software project risks for decision making in SDLC phases using fuzzy multi-criteria approaches. This strategy combined multiple risk evaluation techniques for enhancing software project performance in five major aspects named requirements, estimation, planning and control, team organisation, and project management. Triangular fuzzy numbers are used to quantify fuzzy linguistic variables on a scale of 0 or 1. In this approach, potential software risks will be identified and prioritised at early stage of SDLC phases, which ensures better handle on the improvement of software performance. The investigated result shows this approach can effectively support the decision making during validation of software risk factors compared to existing other methods.

Pan et al. (2018) proposed a test case prioritisation approach combining unit test methods and test definition use associations (DU-chain) coverage to detect faults early at regression testing. This method quantified test cases using DU-chain coverage and fault detection capability to prioritise test cases. The experimental result showed that, the prioritisation technique can improve error detection rate at early of regression testing. However, for experimentation this technique manually implanted error, which leads to biasness in result.

Dahiya et al. (2016) presented an approach for regression test selection using class, sequence and activity diagrams. This work compared old and new versions of UML diagrams to categorise test cases into several category to find out the change operations, which provided significant increase in accuracy. Ramanathan et al. (2008) introduced graph-theoretic framework for test case prioritisation named PHALANX. This approach has been presented by addressing the limitations of implementation complexity in testing process. It considered the dissimilarity of a test case with others in the test suite, by executing dissimilar test cases upfront, different aspects of software are tested earlier that will cover whole software activities (Ramanathan et al., 2008).

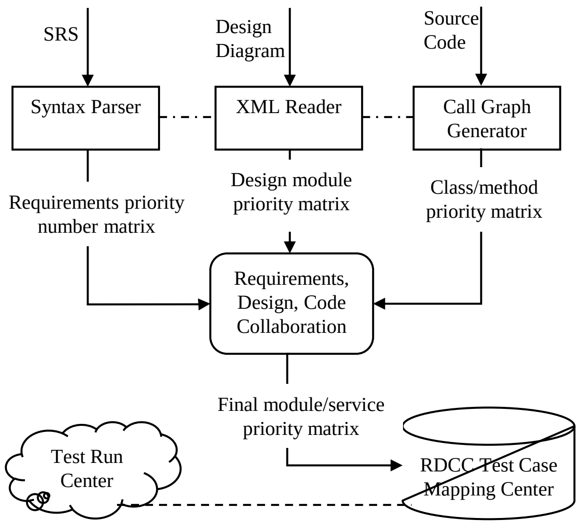
3.4 Summary

The review of the existing literature has shown that various prioritisation schemes have been proposed for software testing like source code analysis, heuristics approach, etc. Very few researchers addressed the software requirement information for test case prioritisation, which are incomplete because, none of the work directly propose any method that incorporates all phases of SDLC named as software requirements, design and source code, where every phase has unique view points.

4 Proposed collaborative framework for test case prioritisation

RDCC is the proposed prioritisation framework, integrating every phase of SDLC to detect faulty modules earlier. In SDLC, requirement engineers assemble SRS documents using direct interaction with customers or end-users. Software designers prepare design diagrams for the development phase on the basis of these SRS documents. Finally, software developers develop source code based on design diagrams. This section describes the detail RDCC framework including it's architecture, activities, processing algorithms and prioritisation approach.

Figure 1 Internal interaction of proposed framework


SRS
Design Diagram
Source Code
Syntax Parser
XML Reader
Call Graph Generator
Requirements priority number matrix
Design module priority matrix
Class/method priority matrix
Requirements, Design, Code Collaboration
Final module/service priority matrix
Test Run Center
RDCC Test Case Mapping Center

The internal interactions among different layer activities including their dependencies and connectivities are presented in Figure 1. Various modules such as syntax parser, XML reader and call graph generator, etc. are used to process inputs to the starting layer of this framework. These processed information are individually prioritised and finally collaborated with each others. RDCC test case mapping centre (presented in Figure 1) uses these collaborative values to generate a prioritised list of test cases, which plays in the principle layer of this framework.

4.1 Layer 1: input layer

The top layer of RDCC framework is named as RDCC input layer, because it takes input from end users for whole RDCC framework. In this layer, requirements, design diagrams, source code and test cases are collected as input from customers, designers, developers and test engineers respectively, and prepared those as output (e.g., text or XML form). Those outputs are usually used as the processing elements of RDCC service layer.

4.2 Layer 2: service layer

RDCC service layer is the principle processing layer of this framework, which takes different SDLC phase information as input from RDCC input layer and calculates their priority values as output. The whole process is divided into four major activities which are listed below.

4.2.1 Requirement prioritisation

SRS documents usually contain the listed software requirements, which are provided by the customers. A list of well documented requirements is mandatory for this phase to parse the representative information (Wiegers, 1999). Because according to Wiegers (1999) without writing quality requirements, a software project may fail to achieve its' success. How to process and prioritise requirements for test case prioritisation, are described below.

Requirement processing

Requirements are analysed for prioritisation, because test cases are written based on software requirements. The requirement priorities are calculated using textual similarities (Aggarwal and Zhai, 2012).

$$tf(t,r) = \frac{f(t,r)}{max\left\{f(w,r) : w \in r\right\}} \tag{1}$$

$$idf(t,r) = log\frac{N}{|\{r \in R : t \in r\}|} \tag{2}$$

$$tfidf(t,r,R) = tf(t,r) \times idf(t,r) \tag{3}$$

$$Req_i = \sum_{j=1}^{t} TermID_{ij} \tag{4}$$

$$ReqPriority_i = \frac{Req_i}{max(Req_0, Req_1, ..., Req_{|R|})} \qquad \forall i = 1, 2, 3, ..., |R| \tag{5}$$

Term document matrix creation and prioritisation

Each requirement is considered as a pool of structured and non structured words. Requirements are inputted as string type which are split into words. The stop words that have no specific meaning or that are not related to RDCC are eliminated, and the only significant words are used to create a term document matrix (Salton and Buckley, 1988) to identify related words. In this matrix the rows and columns correspond to the requirements and the distinct terms respectively. The multiplication of term frequency [equation (1)] and inverse document frequency [equation (2)] calculates the $tfidf$ value [equation (3)]. Those frequency values of all terms are used for requirements prioritisation.

In this framework, every requirement has a requirement ID, which is uniquely used as RDCC ID. The priority of each RDCC ID is calculated by the division of term priority and the sum of all priorities, which is presented by equation (4), where i and j represent the requirement ID and term ID number respectively. Final values are normalised for the implementation simplicity which are presented in equation (5).

Algorithm 1 Algorithm for assigning requirement priorities

```
Require: Set of requirements $R$, priority list $\mathcal{P}1$, priority function: $R_i \to \mathcal{P}1_i$
Ensure: Assigned priority list for requirements $\mathcal{P}1$
1: Begin
2: $R \leftarrow filterstopword(R)$
3: $\mathcal{P}1 \leftarrow \{\}$
4: for each requirements $r_i \in R$ do
5:     $sum \leftarrow 0$
6:     for each term $t \in r_i$ do
7:         $tf$ = calculatetf($t, r$) using equation (1)
8:         $idf$ = calculateidf($t, r$) using equation (2)
9:         $tfidf = tf \times idf$
10:        $sum += tfidf$
11:    end for
12:    $\mathcal{P}1_i \leftarrow sum$
13: end for
14: for each requirement priority $rp_i \in \mathcal{P}1$ do
15:    $rp_i \leftarrow \frac{rp_i}{max(\mathcal{P}1)}$
16: end for
17: End
```

Algorithm for assigning requirement priorities

Algorithm 1 presents the requirements processing activities which takes the set of requirements R, priority function: $R_i \to \mathcal{P}1_i$ as input and returns an assigned priority list as output. It stores the requirement priority values to priority list $\mathcal{P}1$. In the very beginning of this process, every requirement is split into words including stop words. After removing the stop words, only important words are listed for execution, where proper nouns and verb base forms denote the class names and function or method names respectively. These words are usually named as terms for creating term document matrix. Initially, for every requirements, the priority list $\mathcal{P}1$ is initialised to 0 (presented in line 3).

The term list which is generated from the stop word filtering process, are used to calculate term frequency [equation (1)] presented in line 7. Inverse document frequency is also calculated using equation (2) for every requirement in the document presented in line 8. Finally the summation of $tfidf$ values are calculated using equation (3) and stored in priority list $\mathcal{P}1$. The term document matrix is generated using the normalised $tfidf$ values, which is presented in lines 7–11.

4.2.2 Design diagram prioritisation

Software design prioritisation is also an important phase, because design diagrams contain the designers view points, which is unique to any other phases of SDLC. To process the design diagrams, information from different diagrams (e.g., state transition model) are extracted. Every design diagram may have different representational views, but the underlying connection among design modules remain the same.

The design priority assignment is illustrated in Algorithm 2, which takes XML design diagram D as input, and stores their priorities to design element priority list $\mathcal{P}2$. It generates a weighted priority list for design elements corresponding to each requirement. XML Parser is used to retrieve information from design XML. Before executing Algorithm 2, priority list $\mathcal{P}2$ is initialised to 0 for all design elements. This algorithm calculates the relationship between every pair of elements (e.g., state in state-transition). Relationships are calculated among all states in design diagrams which is presented at lines 4 and 5. If there is a relation between any two elements, the priority value is updated for both elements which is denoted at lines 7 and 8.

Algorithm 2 Algorithm for assigning design diagram priorities

```
Require: Design diagram XLM D, priority list $\mathcal{P}2$, priority function: $D_i \rightarrow \mathcal{P}2_i$
Ensure: Assigned priority list for design diagram $\mathcal{P}2$
 1: Begin
 2: $E \leftarrow XMLParser(D)$
 3: $\mathcal{P}2 \leftarrow 0$
 4: for each element $e \in E$ do
 5:     for each element $d \in E \setminus \{e\}$ do
 6:         if $hasRelation(e, d)$ then
 7:             $\mathcal{P}2_e \leftarrow \mathcal{P}2_e + 1$
 8:             $\mathcal{P}2_d \leftarrow \mathcal{P}2_d + 1$
 9:         end if
10:     end for
11: end for
12: for each design priority $dp_i \in \mathcal{P}2$ do
13:     $dp_i \leftarrow \frac{dp_i}{max(\mathcal{P}2)}$
14: end for
15: End
```

Algorithm 3 Algorithm for assigning source code priorities

```
Require: Source Code C, priority list $\mathcal{P}3$, priority function: $C_i \rightarrow \mathcal{P}3_i$
Ensure: Assigned priority list for requirements $\mathcal{P}3$
 1: Begin
 2: $CM_{ij} \leftarrow 0$, $\mathcal{P}3 \leftarrow 0$
 3: if Source Code follows OOP then
 4:     for each class $c \in C$ do
 5:         $CM_{ij} \leftarrow calculateCodeMetrics(c)$
 6:     end for
 7: else
 8:     for each function $f \in C$ do
 9:         $CM_{ij} \leftarrow calculateCodeMetrics(f)$
10:     end for
11: end if
12: for $j = 0$ to $len(CM_j)$ do
13:     $CM_{max} \leftarrow max(CM_j)$
14:     for $i = 0$ to $len(CM_{ij})$ do
15:         $CM_{ij} \leftarrow \frac{CM_{ij}}{CM_{max}}$
16:     end for
17: end for
18: for each source code priority $cp_i \in \mathcal{P}3$ do
19:     $cp_i \leftarrow$ average($CM_{ij}$)
20: end for
21: End
```

4.2.3 Software source code prioritisation technique

Source code is prioritised based on CMs, which is a set of quantitative measures that provides overall description about a software (Kaner and Bond, 2004). RDCC framework analyses both object oriented programming (OOP) and functional programming (FP). In both, OOP and FP, classes and functions are the idiosyncratic code elements accordingly. Code metrics [e.g., lines of code, McCabe's cyclomatic complexity, nested block depth, etc. (Fenton and Bieman, 2014)] are used to calculate source code priority.

Algorithm 3 presents the whole source code processing and prioritising approach to detect vulnerable code sections. It takes source code C as input, and stores their priorities to source code priority list $\mathcal{P}3$. A sorted source code priority list is generated as output to investigate error prone modules earlier. The CM values and priority list are initialised to 0 (Algorithm 3 lines 2–3). OOP and FP metrics are individually calculated for classes and functions accordingly (lines 4–12). The calculated CMs are normalised by dividing their maximum value of their priority list. This process is executed for every CM in the priority list which is presented at lines 13–15. Finally the average value of all normalised CMs are assigned to the source code priority list $\mathcal{P}3$ which is denoted in lines 16–18.

4.2.4 RDCC module integration and prioritisation

Previously prioritised RDCC modules named as requirements, design diagrams and source code are integrated and prioritised in this phase, where every requirement ID is considered as RDCC ID. Equation (6) explains the collaborative viewpoints by calculating the values of every RDCC ID priorities, and the symbols used in equation (6) are listed in Table 3. The calculated values from requirement specifications, design diagrams and source code are multiplied by their priority constants $\alpha, \beta,$ and γ respectively. The priority constants are positive numbers, and the sum of all three constants must be equal to 1.

$$W_i = \alpha \times \mathcal{P}1 + \beta \times \mathcal{P}2 + \gamma \times \mathcal{P}3 \quad \forall_{i=1,2,3,\ \ldots,\ n} \tag{6}$$

$$\begin{aligned} Subject\ to:\ & 0 \le \alpha \ \le 1 \\ & 0 \le \beta \ \le 1 \\ & 0 \le \gamma \ \le 1 \\ \alpha + & \beta + \gamma = 1 \end{aligned}$$

If any requirement is connected to multiple design module or source code elements, cumulative priorities will be considered for that requirement. For example, ID requirement 4 is connected to design modules 4 and 5, then $\mathcal{P}2$ will be the sum of design priority $\mathcal{P}2[4]$ and $\mathcal{P}2[5]$. This paper proposed two types of constant value assignment techniques for RDCC module integration, which are described in the following subsections.

Priority type 1: equal priority based on equal importance

Requirement, design diagram and source code are uniquely related to software test cases by generation, inter-connection and execution respectively. Those phases of SDLC also have distinct point of views to represent the whole software. According to Pressman (2005), all phases of SDLC are equally important to represent a software development. Hence, equal values are assigned to every priority constant of requirement (α), design diagram (β) and source code (γ). According to equation (6), the sum of all priority constants α, β and γ is equal to 1. Hence, the individual priority of these constants will be 1/3, because 1/3 + 1/3 + 1/3 = 1.

Table 3 Symbols used in equation (6)

Symbol	*Description*
W_i	Final weight of RDCC ID$_i$
n	Number of RDCC IDs
$\mathcal{P}1$	Set of requirement priorities (0...1)
α	Requirement prioritisation constant
$\mathcal{P}2$	Set of design priorities (0...1)
β	Design prioritisation constant
$\mathcal{P}3$	Set of source code priorities (0...1)
γ	Source code prioritisation constant

Priority type 2: weighted priority based on error propagation

In SDLC, every phase depends on it's previous phase output (Sommerville, 2004). If any phase triggers an error, that will be propagated to next phases and continued until the final release. For example, errors of requirements engineering phase will be propagated to the design and development phase. An example of error propagation procedure is presented in Table 4, containing three errors (e1, e2 and e3) and three considered RDCC phases (requirement, design and source code). Based on this hypothesis, the impacts of requirements, design and source code will be 3, 2 and 1. So, according to equation (6), the normalised value of α, β and γ is 3/6, 2/6 and 1/6 respectively.

Table 4 Different priorities based on error propagation

	e1 (initiated at requirements)	*e2 (initiated at design)*	*e3 (initiated at code)*
Requirements	✓	×	×
Design	✓	✓	×
Source code	✓	✓	✓
Impact of errors	3	2	1
Normalised impact	$\frac{3}{6}$	$\frac{2}{6}$	$\frac{1}{6}$

The sum of all multiplied weighted values calculates the final weight. This calculation are executed until all the RDCC IDs are processed.

4.3 Layer 3: test case processing layer

This layer takes final priority value from RDCC service layers and generates an ordered list of test cases using those values for the prioritisation. Test cases are written based on the customers' requirements to verify the expected final products. That is why, every test case must be related to at least one requirement, which are mapped in traceability matrix. Using the values from that matrix and prioritised SDLC phases, final priorities are assigned to test cases.

A requirements traceability matrix is generated in test case generation phase including requirements and their corresponding test cases to point related words. Since, requirement IDs are used as RDCC IDs, that means the 'RDCC – test cases mapping resolution' can be found in that matrix and those requirement priorities are used for their assigned test cases. If one test case is assigned to multiple requirements, the cumulative priorities are set to that test case. On the other hand, if one requirement is assigned to multiple test cases, that requirement priority is set to all assigned test cases. The priority value of a test case is proportional to it's ability to fault detection. That means, test cases containing higher priority values have higher fault detecting possibilities.

5 Experimental setup and result analysis

This section presents the implementation approaches of the proposed test case prioritisation framework along with its evaluation on early fault detection. The RDCC framework is tested with six applications which are developed by undergraduate Software Project Lab at IIT, University of Dhaka [uploaded at Github (Project Lab for Test Case Prioritization, 2015), and listed in Table 5]. The performance of RDCC is compared to several most prominent test case prioritisation approaches named as natural orders, RSF (Srivastva et al., 2008) and AFC (Elbaum et al., 2002). Details regarding the experimental setup of those schemes are also provided in this section. Finally, the performance of RDCC is evaluated on the basis of early fault detection and percentage of test case execution to detect faults. The weighted variations of detected faults are not considered in this experiment, where every faults are assumed as equal critical.

Table 5 Dataset information for test case prioritisation

DS ID	*DS name*	*Number of requirements*	*Design diagram*	*LOC*	*Number of test cases*
DS1	News-A: an online news portal	14	Yes	356	74
DS2	Scientific calculator	8	Yes	636	37
DS3	Sparrow: file reading software	20	Yes	752	25
DS4	Amghotok: a platform of marriage	13	Yes	953	20
DS5	Painter: a canvas for painting freely	12	Yes	1021	16
DS6	POAS: program office automation software	18	Yes	4,037	62

Source: Project Lab for Test Case Prioritization (2015)

5.1 Implemented test case prioritisation approaches

Several test case prioritisation schemes [e.g., RSF (Srivastva et al., 2008), AFC (Elbaum et al., 2002), etc.] are considered for the experimental analysis. Among those, three are existing prominent approaches and rest are introduced in this paper (Table 6). In this experiment, AFC (Elbaum et al., 2002) and RSF (Srivastva et al., 2008) are used as the representative of existing source code and requirements-based prioritisation approaches.

5.1.1 Existing prioritisation schemes

In RSF the risk and severity values of every requirement are collected empirically from requirement engineers and assigned the average value of that requirement for calculating requirement priority. In this approach, two different prioritisation factors are assigned for each requirement, which are collected from customers, developers or managers (value range 0 to 10) (Srivastva et al., 2008). The first one is the requirement priority which are collected from customers, developers and managers. Another one is risk factors for every requirement which are collected from developers.

In AFC prioritisation the test with the maximum number of functions covered is selected for early fault detection. If more than one test has the maximum number of functions coverage, a test is selected randomly from that test suite. Since this AFC prioritisation technique is implemented in Java programming language, Java rand function is used for random selection. The selected test cases and covered functions are not considered for further selection in AFC approach.

5.1.2 Proposed prioritisation scheme

In the RDCC scheme, requirements are split into words using java split function. Stopwords are filtered using WordNet database (Miller, 1995) in every requirements. Verbs are replaced with their base form using WordNet verb list. Term document matrix is generated based on those remaining words or terms using term frequency – inverse document frequency. The final requirement priority values are normalised by dividing with the maximum priority value which are presented in equation (5).

Design diagrams are processed to retrieve design modules or state corresponding to each requirements. Diagrams need to be converted as readable XML format, because RDCC uses the underlying inter-connectivity among design modules. Hence, any design diagram can be used to calculate design priority [e.g., the experimental datasets (DSs) are designed using state transition diagram]. In the beginning of this process, design priority list is initialised to 0. If there is a connection between two states, these state positions in the priority list are incremented. This process is being continued until all the connections are processed. The design priority values are normalised as same as requirement priority normalisation presented in Algorithm 2.

Source code are processed using four CMs named as, lines of code, McCabe's cyclomatic complexity, weighted methods per class (Fenton and Bieman, 2014), and nested block depth (Fenton and Bieman, 2014). The proposed Eclipse metric plug-in is used to export CMs values into a XML file. Those metrics values are calculated for every classes in OOP to initialise code priority. The individual class of function metric values are normalised by dividing the maximum metric value. The average of these normalised metrics is used as source code priority. Any other CMs might be used, but the average of all normalised metrics lead the code priority.

Two types of collaboration approaches equal priority (RDCC-v1) and weighted priority (RDCC-v2) are experimented in this research. One experimental prioritisation schemes are also noticed, which is the byproduct of RDCC approach named as using design diagrams only (RD1CC). All of the six prioritisation schemes with their acronyms and descriptions are listed in Table 6.

Table 6 Implemented test case prioritisation approaches

Priority scheme ID	*Acronym*	*Description*
P1	Natural	Natural order for test case execution, which is the generation order of test cases.
P2	Random	Random order for test case execution, average of 50 iteration results are used for test case priority values.
P3	RSF	Risk and severity factor (RSF): a requirements-based prioritisation (Srivastva et al., 2008).
P4	R1DCC	A portion of proposed RDCC, where test cases are prioritised based on only requirements prioritisation in a new way.
P5	RD1CC	A portion of proposed RDCC, where test cases are prioritised based on only design diagrams prioritisation.
P6	AFC	Additional function coverage (AFC): a code-based prioritisation (Elbaum et al., 2002).
P7	RDC1C	A portion of proposed RDCC, where test cases are prioritised based on only source code prioritisation only.
P8	RDCC-v1	Proposed RDCC approach with equal priority constants of requirements, design and source code, in the basis of equally importance.
P9	RDCC-v2	Proposed RDCC approach with different priority constants of requirements, design and source code based on error propagation flow.

Table 7 Experimental setup and determined constant values for implementing different prioritisation techniques

Prioritisation parameters	*Values*
Upper limit of single prioritisation parameter	1
Prioritisation range	0 to 1
Requirement priority constant (α)	RDCC-v1: $\frac{1}{3}$, RDCC-v2: $\frac{3}{6}$
Design priority constant (β)	RDCC-v1: $\frac{1}{3}$, RDCC-v2: $\frac{2}{6}$
Source code priority constant (γ)	RDCC-v1: $\frac{1}{3}$, RDCC-v2: $\frac{1}{6}$
Implemented language	Java
Numbers of requirements in dataset	Varied from 8 to 20
Numbers of line of code in dataset	Varied from 636 to 4,037
Numbers of test cases in dataset	Varied from 18 to 74

5.2 Assumptions and priority constants

The priority constants of requirements, design diagrams and source code are α, β and γ respectively, which determine their impacts on test cases. The sum of those three constants must be equal to 1 for normalisation (details presented in Subsection 4.2.4). For implementing RDCC framework, those constants are assigned based on two different types of priority.

Priority type 1	Equal priority: The value of α, β and γ are equal, based on similar importance on test cases to calculate average priority. That means $\alpha = \beta = \gamma = 1/3$.
Priority type 2	Weighted priority: The value of α, β and γ are different based on error propagation. Here, $\alpha = 3/6$, $\beta = 2/6$ and $\gamma = 1/6$.

All the assumptions and priority constant values for implementation are listed in Table 7.

5.3 Performance analysis and comparison

To measure the efficiency of test case prioritisation technique, fault detection rates and test case execution percentage both are considered as metrics. Those parameters represent the performance of prioritisation techniques in two different point of views.

Definition and experimental results of those two performance analysis views are given below.

1 View 1: Percentage of detected faults with varying number of test execution.

3 View 2: Percentage of test case execution to detect different numbers of faults.

According to view 1, the higher the fault detection, the better the prioritisation approach is. On the other hand (for view 2), the prioritisation approach executes the lower number of test cases to detect faults is better than other approaches. For both cases, the experimental setup and assumed constants are remaining the same which are listed in Table 7.

5.3.1 View 1: percentage of detected faults with varying number of test case execution

One of the major goals of test case prioritisation is to detect faults earlier, which is the effectiveness of a prioritisation scheme. The percentage of fault detection shows what number of faults are detected after a certain number of test case execution. This experimental process is divided into three phases, based on percentages of test case execution named as first quarter (25%), half (50%) and third quarter (75%). Full test case (100%) execution results are not considered, because after executing 100% test cases, all the faults should be detected by all prioritisation schemes. Whereas the goal of test case prioritisation is to find the important faults earlier.

The fault detection results are listed in Table 8, indicating the percentage of fault detection with varying number test case execution for individual DS. According to that table, collaborative approaches (RDCC-v1 and RDCC-v2) performs better than any other approaches. In first quarter (25%), RDCC-v1, RDCC-v2 and AFC detect 36.41%, 32.66%, and 28.97% faults respectively, which are better than average. After 50% for test case execution, RDCC-v2 detects 71.67% faults on average, which is the best among all others, and RDCC-v1, RDCC-v2, RD1CC perform better than average. In this quarter the highest and lowest fault detection rates are 71.47% and 36.47%, which are performed by RDCC-v2 and natural order respectively. In the third quarter, other schemes except RDCC-v1 and RDCC-v2 are only the below average of fault detection accuracy. In this quarter, the highest and lowest detection percentages are 88.12% (by RDCC-V2) and 61.33% (by natural order) respectively.

Figure 2 Percentage of detected faults with varying number test case execution, (a) after 25% of test case execution (b) after 50% of test case execution (c) after 75% of test case execution

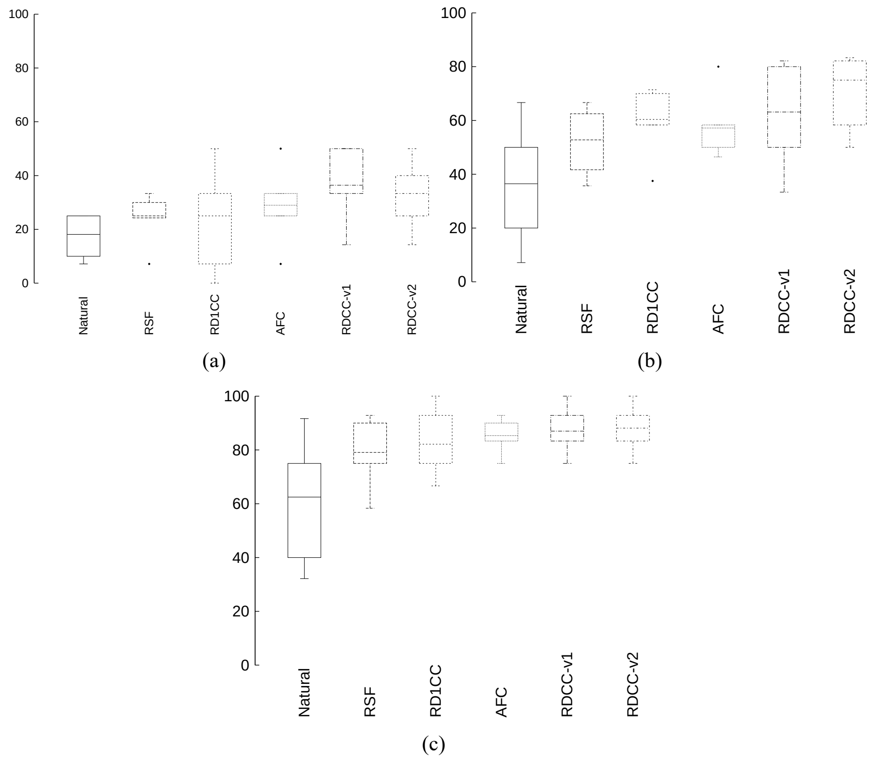


Table 8 Percentage of fault detection with varying number test case execution

% of TCs	*DS*	*P1*	*P2*	*P3*	*P4*	*P5*	*P7*	*P6*	*P8*	*P9*
25%	DS1	25	16.67	25	33.33	25	25	25	33.33	33.33
	DS2	25	8.33	25	41.67	0	16.67	33.33	50	50
	DS3	25	50	25	37.5	25	25	25	37.5	25
	DS4	7.14	17.86	7.14	14.29	7.14	7.14	7.14	14.29	14.29
	DS5	16.67	50	33.33	33.33	33.33	33.33	33.33	33.33	33.33
	DS6	10	20	30	40	50	40	50	50	40
	AVG	18.13	27.14	24.25	33.35	23.41	24.52	28.97	36.41	32.66

Table 8 Percentage of fault detection with varying number test case execution (continued)

% of TCs	*DS*	*P1*	*P2*	*P3*	*P4*	*P5*	*P7*	*P6*	*P8*	*P9*
50%	DS1	41.67	50	41.67	58.33	58.33	58.33	58.33	58.33	58.33
	DS2	66.67	33.33	66.67	58.33	58.33	58.33	58.33	75	75
	DS3	50	87.5	62.5	62.5	37.5	37.5	50	50	50
	DS4	7.14	42.86	35.71	57.14	71.43	67.86	46.43	82.14	82.14
	DS5	33.33	50	50	33.33	66.67	50	50	33.33	83.33
	DS6	20	50	60	70	70	70	80	80	80
	AVG	36.47	52.28	52.76	56.61	60.38	57	57.18	63.13	71.47
75%	DS1	75	66.67	58.33	83.33	83.33	75	83.33	83.33	83.33
	DS2	91.67	75	75	75	75	75	75	75	75
	DS3	62.5	80	75	75	75	75	87.5	87.5	87.5
	DS4	32.14	60.71	92.86	92.86	92.86	92.86	92.86	92.86	92.86
	DS5	66.67	66.67	83.33	83.33	66.67	83.33	83.33	83.33	100
	DS6	40	80	90	100	100	100	90	100	90
	AVG	61.33	71.51	79.09	84.92	82.14	83.53	85.34	87	88.12

The percentage of detected faults are graphically visualised in Figure 2 using box-whisker plot, where X-axis and Y-axis represents the prioritisation approaches and percentages of detected faults (0% to 100%) respectively. In that figure, the average line, minimum and maximum points are depicted for every prioritisation scheme. According to Figures 2(b) and 2(c), approach that detected faults earlier is RDCC-v2, because its average line and maximum outlier are higher than any other approaches. On the other hand, RDCC-v1 performs better than others in the first quarter [Figure 2(a), after 25% of test case execution], because it detects maximum faults in this quarter.

5.3.2 View 2: percentage of test case execution to detect different number of faults

Maximum fault detection in minimum test case execution is one of the major goals of the test case prioritisation. The percentage of test case execution results can measure the efficiency of prioritisation approaches in terms of early fault detection by executing certain amount of test cases. The lower the number of test case execution is, the higher the efficiency of a prioritisation technique will be. This measuring approach is experimented into four different phases of fault detection percentages named as 25%, 50%, 75% and 100%. All faults are divided into those four different quarters to measure the prioritisation schemes' efficiency in step by step.

The test case execution results to detect different quarters of faults are presented in Table 9. Detecting 25% faults, RDCC-v1 and natural order execute 17.52% and 31.82%, which are the lowest and highest value respectively. In this phase AFC, RDCC-v1 and RDCC-v2 execute 22.73%, 17.52% and 20.98% test cases which perform better than average result (24.03%). In the rest of the quarter (e.g., 50%, 75% and 100% fault detection) RDCC-v2 executes 34.74%, 50.65%, and 88.42% test cases respectively to detect faults, which are better than any other schemes. Because, RDCC-v2 considers all possible view points about a software.

Figure 3 Percentage of test case execution to detect different number of faults, (a) for 25% fault detection (b) for 50% fault detection (c) for 75% fault detection (d) for 100% fault detection

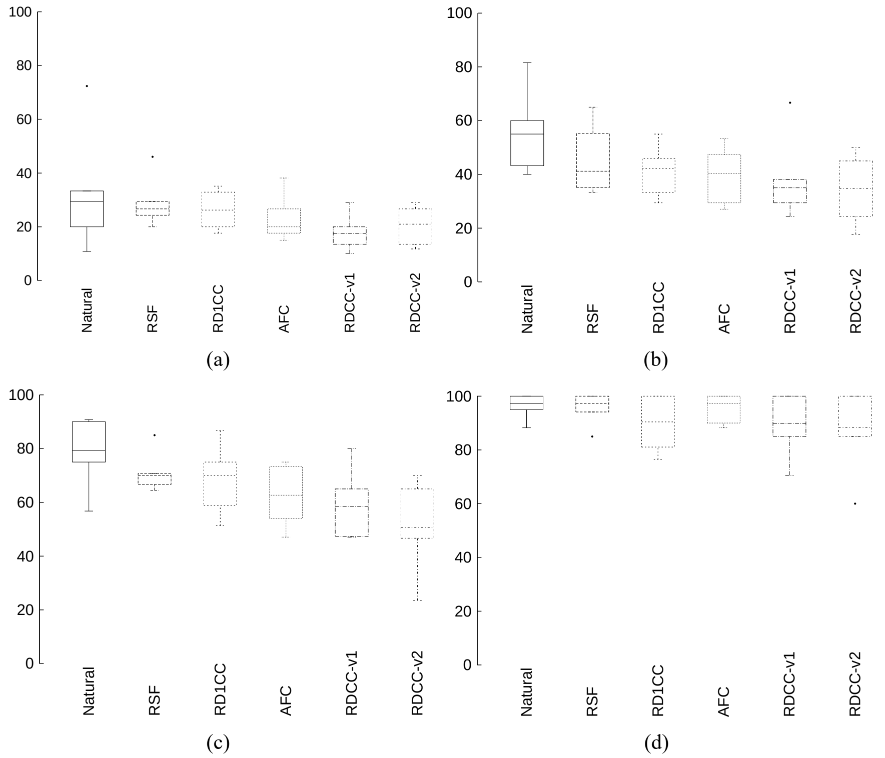


Table 9 Percentage of test case execution to detect different number of faults

% of TCs	*DS*	*P1*	*P2*	*P3*	*P4*	*P5*	*P7*	*P6*	*P8*	*P9*
25%	DS1	25	30	25	15	25	25	20	15	20
	DS2	10.81	43.24	24.32	13.51	35.14	35.14	18.92	13.51	13.51
	DS3	20	15	20	10	20	20	15	10	25
	DS4	72.37	34.21	46.05	28.95	32.89	32.89	38.16	28.95	28.95
	DS5	33.33	13.33	26.67	20	26.67	26.67	26.67	20	26.67
	DS6	29.41	41.18	29.41	23.53	17.65	23.53	17.65	17.65	11.76
	AVG	31.82	29.49	28.58	18.5	26.22	27.2	22.73	17.52	20.98
50%	DS1	55	50	65	35	45	45	45	35	45
	DS2	43.24	59.46	35.14	24.32	45.95	45.95	27.03	24.32	24.32
	DS3	40	25	40	35	55	55	40	35	50
	DS4	81.58	61.84	55.26	38.16	42.11	42.11	47.37	38.16	38.16
	DS5	60	26.67	33.33	66.67	33.33	46.67	53.33	66.67	33.33
	DS6	52.94	52.94	41.18	35.29	29.41	29.41	29.41	29.41	17.65
	AVG	55.46	45.98	44.98	39.07	41.8	44.02	40.36	38.09	34.74

Table 9 Percentage of test case execution to detect different number of faults (continued)

% of TCs	*DS*	*P1*	*P2*	*P3*	*P4*	*P5*	*P7*	*P6*	*P8*	*P9*
75%	DS1	75	80	85	60	70	70	70	60	70
	DS2	56.76	72.97	67.57	51.35	72.97	72.97	54.05	51.35	51.35
	DS3	90	45	70	60	75	75	75	65	65
	DS4	90.79	85.53	64.47	55.26	51.32	51.32	56.58	47.37	47.37
	DS5	86.67	86.67	66.67	80	86.67	60	73.33	80	46.67
	DS6	76.47	70.59	70.59	47.06	58.82	58.82	47.06	47.06	23.53
	AVG	79.28	73.46	70.72	58.95	69.13	64.69	62.67	58.46	50.65
100%	DS1	95	96	100	100	95	100	100	100	100
	DS2	97.3	89.19	97.3	91.89	81.08	97.3	91.89	97.3	97.3
	DS3	100	90	85	95	90	90	90	85	85
	DS4	100	92.5	100	100	100	100	100	100	100
	DS5	100	88.5	100	86.67	100	100	80	86.67	60
	DS6	88.24	98	94.12	76.47	76.47	88.24	76.47	70.59	88.24
	AVG	96.76	92.36	96.07	91.67	90.43	95.92	89.73	89.93	88.42

Figure 3 depicts the execution results of test case using box-whisker plot, where where X-axis and Y-axis represents the prioritisation approaches and test case execution percentages (0% to 100%) respectively. According to that figure, the average results of RDCC-v1 is the best for detecting 25% faults [Figure 3(a)]. According to Figures 3(b), 3(c) and 3(d) prioritisation scheme RDCC-v2 outperforms other implemented schemes in terms of test case execution. Because the average line in whisker-box, and the maximum percentage of execution both are lower than any other prioritisation schemes, listed in Table 6.

5.4 Result analysis and discussion

Since, the minimisation of test case execution can reduce testing time and cost, detecting a specific percentage of faults, prioritisation approaches should need minimum numbers of test case execution. RDCC-v2 scheme executes minimum number of test cases to detect similar percentage of faults and performs better than any other implemented schemes (column 8 in Tables 8 and 9).

RDCC-v2 approach outperforms existing techniques in terms of average fault detection (Subsection 5.3.2) and test case execution (Subsection 5.3.1). However, after executing specific percentage of test cases, it performs similar results with AFC (e.g., Table 8 DS5 in 25% test execution, and Table 9 DS1 and DS4 in 100% faults detection). It may happen in small experiments, because collaborative information from every phases of SDLC may not generate extra supporting issues for test case prioritisation (such as incorporating requirements and code, requirements and design, etc.). In those cases, proposed prioritisation technique may perform similar results to requirements and source code analysis only, but not worst. For example, after 25% of test case execution, RDCC-v1 and RDCC-v2 approaches produce similar results for experiment DS3. However, most of the real life applications usually contains more modules.

Since early fault detection is one of the major goals of test case prioritisation, RDCC-v2 approach performs 61.89%, 24.93%, 21.82% and 7.39% faster on average of all other prioritisation schemes for 25%, 50%, 75% and 100% fault detection respectively. Those percentages present a descending order of performance, because RDCC approach already detects faults in early phases. Since majority faults are detected in early phases of execution, there should be no significance in the later phases of fault detection (e.g., 7.39% improvement in last quarter). That means collaborative information from different phases of SDLC can increase the efficiency of test case prioritisation technique by detecting early faults, which reduces the time and cost in testing phases.

In a nutshell, RDCC-v2 outperforms other prioritisation schemes for all experiments considering average fault detection and percentage of test case execution, specially for large datatset. This scheme performs 13.414% and 13.893% better results on average than any other implemented test case prioritisation approaches in terms of fault detection and test case execution respectively. However, software requirements, design diagrams and source code are mandatory for this prioritisation framework. Excluding any of these three phases, RDCC framework works unexpectedly. Finally, it can be concluded that prioritisation approach using the collaborative information from different phases of SDLC can reduce time and cost in testing phase by increasing the fault detection and minimising the test case execution rates.

5.5 Threats to validity

This section describes the external and internal threats to the validity of this experiment. The approaches which have been taken to reduce those threats are also be added here.

- *Threats to internal validity:* The accuracy of proposed prioritisation scheme depends on the correctness of requirement analysis, design diagrams and source code developments. Incorrect design module connections or invalid requirement analysis may miss lead the prioritisation objectives. Hence, in this study, the greatest concern for internal validity involves the effectiveness of selecting software artefacts for prioritising test cases. To control this threat, the normalised average values are used to calculate any artefacts' priority.
- *Threats to external validity:* The experimental study has been performed using real life software with requirement documents and design diagrams, which were small and medium in size. While the results from this experiment cannot be interpreted in the context of industrial applications, different types of applications are used to reduce this threat that came from various sources such as Android apps, Java desktop apps, open source, etc. The proposed prioritisation approach may not always outperform in every real life DS.

6 Conclusions and future work

Test case prioritisation approaches reduce software development time and cost by maximising the percentage of fault detection and minimising the percentage of test case execution. To achieve an effective prioritisation technique, information from different phases of SDLC which are requirements, design diagrams and source code are needed to be collaborated. Because those phases are uniquely connected to test cases by test case generation, test modules' connection and test case execution, respectively including individual view points. This hypothesis is justified and proved throughout the paper and an effective prioritisation framework is proposed using the collaborative information of all SDLC phases. The comparative study of the proposed framework supports the hypothesis.

In the proposed prioritisation scheme, requirements are analysed based on textual similarity by filtering the stop words. The remaining terms of every requirement are prioritised using term frequency – inverse document frequency to detect requirements priority. Those priority values are normalised in a specific range for computational simplicity. Design diagrams (corresponding to each requirement) are extracted as readable XML format to detect the dependencies and connectivities among design elements. Each of the elements in diagrams are prioritised based on their number of detected connectivities. Source code are prioritised using various CMs such as line of code, cyclomatic complexity, weighted methods per class, etc. Classes and functions related to each requirement are used as the idiosyncratic values of source code for prioritisation.

Every requirement ID is uniquely identified as RDCC ID in this framework, which are calculated by using the relationships between requirement IDs and design modules, and requirement IDs and classes. Final RDCC priorities are calculated by the multiplication of those calculated priorities and priority constants. The priority constants may be equal or different (both of those are experimented in this research). If any of those phases is missing, the priority constant of that phase is assigned to 0. The calculated final priority values are used to assign priority of related test cases. If a test case is related to more than one requirement IDs, the cumulative priorities are assigned for that test cases. Finally, the test cases are sorted in descending order for early fault detection.

It has experimentally been shown that proposed RDCC scheme has a lower percentage of test case execution and higher percentage of fault detection rate. In terms of early fault detection according to the experiment, collaborative information from different phases of SDLC performs 22.77% on average faster than any other implemented prioritisation schemes. RDCC framework also outperforms in terms of number of test case execution and minimises 29.01% test cases for detecting specific percentage of faults. Those results infer that proposed RDCC scheme can lead to an effective prioritisation approach by maximising the fault detection rate and minimising the percentage of test case execution.

The efficiency of test case prioritisation schemes are closely related to test case generation process. Hence, irrelevant test cases can negatively affect on prioritisation performance. Before prioritising test cases, test generation processes need to be verified. Hence, appropriate and smell free test case generation could be a future research direction to improve the performance of test case prioritisation schemes.

References

Aggarwal, C. and Zhai, C. (2012) 'A survey of text clustering algorithms', in *Mining Text Data*, pp.77–128, Springer, USA.

Arafeen, M. and Do, H. (2013) 'Test case prioritization using requirements-based clustering', in *Sixth International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, pp.312–321.

Avila, G.H., Torres, J.J., Gonzalez, H.L. and Hernandez, V. (2013) 'Metaheuristic approach for constructing functional test-suites', *IET Software*, Vol. 7, No. 2, pp.104–117.

Balsamo, S., Inverardi, P. and Mangano, C. (1998) 'An approach to performance evaluation of software architectures', in *Proceedings of the 1st international workshop on Software and Performance*, ACM, pp.178–190.

Dahiya, S., Bhatia, R.K. and Rattan, D. (2016) 'Regression test selection using class, sequence and activity diagrams', *IET Software*, Vol. 10, No. 3, pp.72–80.

Elbaum, S., Malishevsky, A.G. and Rothermel, G. (2002) 'Test case prioritization: a family of empirical studies', *IEEE Trans. on Software Engineering*, Vol. 28, No. 2, pp.159–182.

Fenton, N. and Bieman, J. (2014) *Software Metrics: A Rigorous and Practical Approach*, CRC Press, USA.

Haidry, S. and Miller, T. (2013) 'Using dependency structures for prioritization of functional test suites', *IEEE Transactions on Software Engineering*, Vol. 39, No. 2, pp.258–275.

Harman, M. (2010) 'Why source code analysis and manipulation will always be important', in *10th Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE Computer Society, pp.7–19.

Hasan, A., Rahman, A., Siddik, M.S. (2017) 'Test case prioritization based on dissimilarity clustering using historical data analysis', in *International Conference on Information, Communiation and Computing Technology*, Springer, pp.269–281.

Islam, M.M., Marchetto, A., Susi, A., Kessler, F.B. and Scanniello, G. (2012a) 'MOTCP: a tool for the prioritization of test cases based on a sorting genetic algorithm and latent semantic indexing', in *28th International Conference on Software Maintenance (ICSM)*, IEEE, pp.654–657.

Islam, M.M., Marchetto, A., Susi, A. and Scanniello, G. (2012b) 'A multi-objective technique to prioritize test cases based on latent semantic indexing', in *16th European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE, pp.21–30.

Kaner, C. and Bond, W. (2004) 'Software engineering metrics: what do they measure and how do we know', in *10th Int. Software Metrics Symposium*, IEEE, pp.1–12.

Li, S., Bian, N., Chen, Z., You, D. and He, Y. (2010) 'A simulation study on some search algorithms for regression test case prioritization', in *10th International Conference on Quality Software (QSIC)*, IEEE, pp.72–81.

Mala, D.J. and Praba, M.R. (2011) 'Critical components identification and verification for effective software test prioritization', in *Third International Conference on Advanced Computing (ICoAC)*, IEEE, pp.181–186.

Malhotra, R. and Tiwari, D. (2013) 'Development of a framework for test case prioritization using genetic algorithm', *ACM SIGSOFT Software Engineering Notes*, Vol. 38, No. 3, pp.1–6.

Miller, G.A. (1995) 'WordNet: a lexical database for English', *Communications of the ACM*, Vol. 38, No. 11, pp.39–41.

Pan, L., Wang, T., Qin, J. and Xiang, X. (2018) 'A dynamic test prioritisation based on du-chain coverage for regression testing', *International Journal of Embedded Systems*, Vol. 10, No. 2, pp.113–119.

Pressman, R. (2005) *Software Engineering: A Practitioner's Approach*, 6th ed., McGraw-Hill, Inc., New York, NY, USA.

Project Lab for Test Case Prioritization (2015) [online] https://github.com/iitprojectlab (accessed 18 August 2018).

Rahman, M.A., Hasan, M.A. and Siddik, M.S. (2018) 'Prioritizing dissimilar test cases in regression testing using historical failure data', *Int. Journal of Computer Applications*, Vol. 180, No. 14, pp.1–8.

Rajarathinam, K. and Natarajan, S. (2013) 'Test suite prioritisation using trace events technique', *IET Software*, Vol. 7, No. 2, pp.85–92.

Ramanathan, M.K., Koyuturk, M., Grama, A. and Jagannathan, S. (2008) 'Phalanx: a graph-theoretic framework for test case prioritization', in *ACM Symposium on Applied Computing*, ACM, pp.667–673.

Rothermel, G., Untch, R.H., Chu, C. and Harrold, M.J. (2001) 'Prioritizing test cases for regression testing', *IEEE Transactions on Software Engineering*, Vol. 27, No. 10, pp.929–948.

Salton, G. and Buckley, C. (1988) 'Term-weighting approaches in automatic text retrieval', *Information Processing & Management*, Vol. 24, No. 5, pp.513–523.

Sangaiah, A.K., Samuel, O.W., Li, X., Abdel-Basset, M. and Wang, H. (2018) 'Towards an efficient risk assessment in software projects–fuzzy reinforcement paradigm', *Computers & Electrical Engineering*, Vol. 71, pp.833–846.

Siddik, M.S. and Sakib, K. (2014) 'RDCC: an effective test case prioritization framework using software requirements, design and source code collaboration', in *17th Int. Conf. on Computer and Information Technology (ICCIT)*, IEEE, pp.75–80.

Sommerville, I. (2004) *Software Engineering*, 7th ed., Pearson Addison Wesley, New York, NY, USA.

Srikanth, H. and Williams, L. (2005) 'On the economics of requirements-based test case prioritization', in *ACM SIGSOFT Software Engineering Notes*, Vol. 30, pp.1–3.

Srivastva, P.R., Kumar, K. and Raghurama, G. (2008) 'Test case prioritization based on requirements and risk factors', *ACM SIGSOFT Software Engineering Notes*, Vol. 33, No. 4, p.7.

Wiegers, K. (1999) 'Writing quality requirements', *Software Development*, Vol. 7, No. 5, pp.44–48.