DYNAMIC SMOKE TESTING
DYNAMIC REGRESSION TEST CASE SELECTION AND PRIORITIZATION

Yasitha Nuwan Mallawaarachchi

(168245A)

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

May 2020

DYNAMIC SMOKE TESTING
DYNAMIC REGRESSION TEST CASE SELECTION AND PRIORITIZATION

Yasitha Nuwan Mallawaarachchi

(168245A)

Thesis submitted in partial fulfilment of the requirements for the degree Master of Science in Computer Science

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

May 2020

DECLARATION

I declare that this is my own work and this thesis does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to the University of Moratuwa the non-exclusive right to reproduce and distribute my thesis, in whole or in part in print, electronic or other media. I retain the right to use this content in whole or part in future works.

Yasitha .... 22/05/2020 ....

Yasitha Nuwan Mallawaarachchi Date

The above candidate has carried out research for the Masters/MPhil/PhD thesis/ dissertation under my supervision.

....

Dr. Charith Chitraranjan Date

ABSTRACT

With the advancement and increasing popularity of agile software development practices in large scale software development projects, frequent product releases are encouraged so that clients can actively participate in the software development life cycle (SDLC) by providing early feedback on developed features. This approach leads to iterative shorter cycles of development and continuous integration. So, the importance of regression testing and regression test suite is well emphasised in such methodologies. Regressions have become the most widely used approach in maintaining the quality of continuously changing software systems.

Even though the agile SDLC requires faster regression feedback given the shorter length of the release cycles, size and the complexity of the regression test suites increases over time; hence execution time keeps on growing. Therefore, it is not practical to run the regression test suite on every code change. In turn, it has become a significant dilemma in current regression testing. Therefore, it is essential to implement a regression testing strategy which is highly selective but accurate, to ensure the committed code changes does not inflict any ill behaviour on the current working software before it is merged and released for client feedback. To achieve this objective, it is critical to find out the distinct effects on behaviour that have impacted the software at the earliest during the continuous integration (CI) cycle. This research is focused on selecting and prioritizing the most suitable test cases from the regression test suite to detect any behaviour that is no longer intact due to the code change. Also, the capability of employing machine learning principles to learn and identify the most impactful characteristics of test cases is considered as another key objective of this study.

Keywords: Regression Test, Selection, Prioritization, Machine Learning, Clustering

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Dr Charith Chitraranjan for the continuous support of my MSc study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my MSc study.

I am grateful for the support and advice given by Dr Indika Perera, by encouraging continuing this research till the end. Further, I would like to thank all my colleagues for their help in finding relevant research material, sharing knowledge and experience and for their encouragement.

I am as ever, especially indebted to my parents for their love and support throughout my life. Finally, I wish to express my gratitude to all my colleagues at MillenniumIT Pvt. Ltd, for the support given me to manage my MSc research work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Description
BBD	Block Branch Diagram
CI	Continuous Integration
CR	Change Request
DB	Database
E2E	End to End
IR	Information Retrieval
ML	Machine Learning
OOP	Object Oriented Programming
ORD	Object Relational Diagram
OSD	Object State Diagram
PCA	Principle Component Analysis

QA	Quality Assurance
SDLC	Software Development Life Cycle
TF-IDF	Term Frequency – Inverse Document Frequency
WSS	Within-cluster Sum of Square

1 INTRODUCTION

1.1 Agile development practice

In the recent past, the software development methodologies have evolved from the traditional waterfall strategy to more adaptive agile-based strategies such as scrum. The primary intention is to absorb the requirement changes precisely without disturbing the on-going development process. Also, it helps to utilise both human and infrastructure resources of a company in the most optimum manner, promoting teamwork in an agile environment.

Another essential aspect of agile development is its rapid frequency in the delivery of working software. Traditionally it took months or years to deliver working software for client-side testing. However, with the incorporation of agile strategies into the software development life cycle, more frequent releases are being facilitated, which has been a significant reason for the improvements in customer satisfaction and engagements. Because of that continuous integration (CI) of the development changes and automated regression testing has become two crucial stages of the agile practice and these two stages maintain the quality of the output product [1].

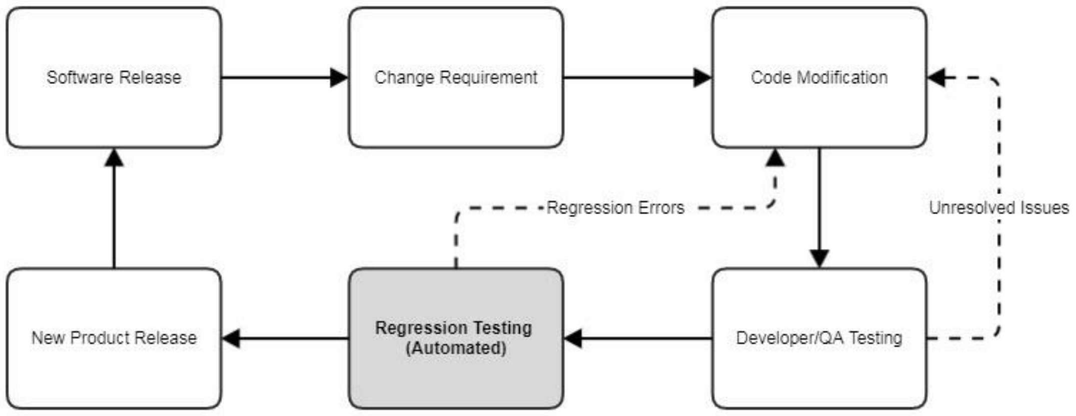


Figure 1.1: Agile software development life cycle

1.2 Continuous integration and gitflow

1.2.1 Continuous integration

Continuous integration (CI) is a cost-effective and efficient software development practice which is used to integrate the changes done by developers throughout the day in a shared repository, as early and often as possible. This includes software configuration management, version control, automated compilation, installation and regression testing of changed software [2]. Continuous integration has become increasingly popular among software industry due to its benefits such as

- Cheap, fast and easy integration
- Identify development bugs as early as possible
- Increasing visibility which enables greater communication
- Reduce integration issues and allow frequent releases
- Ability to invest more time on actual code development, hence increase the efficiency of the development process

Because of these reasons, CI helps to increase the project team efficiency and the quality of the product. There are several popular tools such as Buddy, TeamCity, Bamboo and Jenkins have been developed and used for the CI activities in software projects.

1.2.2 Gitflow

Git flow is a branching model used during the version control in software development. It is ideally suited for the projects that have frequently scheduled release cycles, and this model can be used from the existing version control system. In this approach, there are two git branches used to persist the history of the project, namely master branch and develop branch [3]. Master branch stores the history of client releases while the develop branch serves for the new development and integration.

When a code change is required to be done to fix a bug or develop a CR, a new bugfix branch or feature breach will be created on top of develop branch. So, all the code and system changes are committed to the newly created bugfix branch or feature branch, and after the developer testing is completed, this branch will be merged into the develop branch.

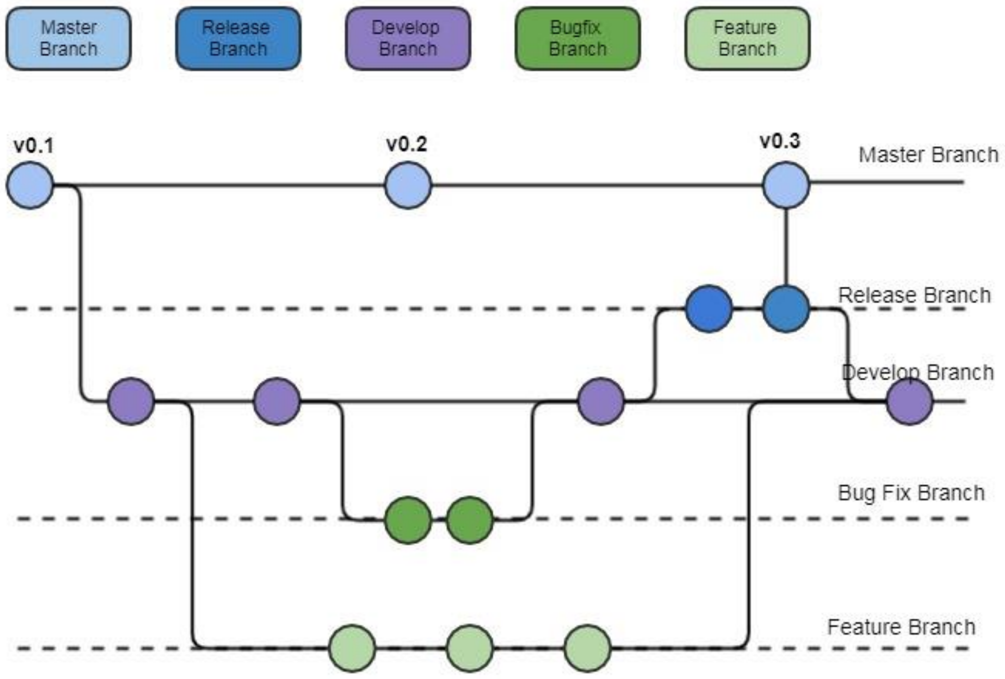


Figure 1.2: Git flow branching model

Typically, in mission-critical software development, the feature/bugfix branches will be merged into the develop branch after thorough unit level testing and multiple developer reviews. Once it is merged, the changes will be captured to the regular CI cycle which will build, deploy and perform the regression testing on the updated component. So, the git-flow approach supports parallel development, Collaboration and for emergency fixes.

1.3 Software maintenance and regression testing

1.3.1 Software maintenance

Maintenance of the software plays a vital role in the SDLC process, and it cost approximately 60% of the total software life cycle. Software maintenance is required when modifying a product after it has been delivered to the client. To fix bugs, improve the design, implementing enhancements, interfacing with other subsystems, and due to the improvements in the hardware platform, software maintenance is essential and unavoidable. Software maintenance can be divided into four main categories [5]

1. Corrective maintenance

   Corrective maintenance of a software system is required to rectify error/bugs in the system or enhance the performance of the system while it is in operation

2. Adaptive maintenance

   Adaptive maintenance is essential when the client requested changes to run the product on new hardware or software platform as well as when they need to interface the product with other software systems

3. Perfective maintenance

   Activities have to be done to support the new features requested by customer or change requests (CR) to an existing functionality upon client demand are categorised under Perfective maintenance

4. Preventive maintenance

   This type of maintenance is to prevent future problems of the software system. The goal is to attend the issues which could cause a severe impact in the future even though it is less significant at the moment.

Whatever the change done in the maintenance period, it has to be thoroughly tested before release to the production. So, the testing has become the most critical and time-consuming task of software maintenance activities. It evaluates the capability and quality of the program and reveals as much as errors to achieve desired results of the system. There are three types of testing can be identified in software SDLC process.

- Unit testing
- Component testing
- End to end regression testing

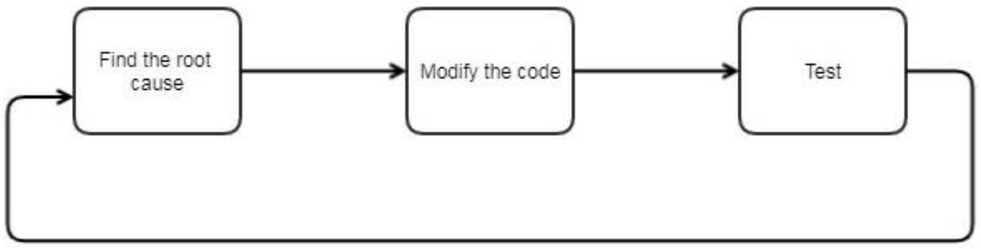


Figure 1.3: Software maintenance process

Unit testing and component testing are considered as developer testing, and it will verify the unit level (functional level) and component level functionality of the software. Usually, these two types of testing are done by the developers soon after the changes were committed to the bugfix/feature branch (before merging them to develop branch) and it will take fewer resources and time to complete, hence add less overhead to the development process.

1.3.2 Regression testing

On the other hand, regression testing is the most popular and most significant testing method out of the three options. It is costly as well as repetitive activity to perform after software update before to release it to the production as the final and vital quality check. Regression testing builds the confidence that modifications do not harm to the existing functionality and stakeholders often rely heavily on the capabilities of regression suite. It is comprising with a large number of end-to-end business test cases for the defined test scenarios which cover the complete functionality of the software system. For complex mission-critical software systems, this could be over thousands of test cases. All these test cases are formatted and included into an automated test suite and expected to be run within continuous integration process. Due to its end to end testing nature and functional dependencies, regression testing is usually considered as slow during the execution, and it is costly from resource consumption as well. Therefore, this research is also focused on utilising and managing regression test suite more optimum and efficient manner to achieve its goals.

The defect detection capability of the regression test suite is its leading indicator of quality. Further the characteristics such as

- Completeness – whether the test suite is complete in terms of functional coverage as well as the code coverage
- Redundancy – whether multiple test cases detect the same defects in the test suite
- Maintenance status – indicator to assess whether the regression test suite evolves together with the source code which means system code and test codes will be given the same priority and consider equality important during the development

are also an important measure of its overall quality and efficiency. Test cases are the base unit in the regression test suite and can be categories as follows considering their behaviour [6]

- Obsolete test case – test cases which are no longer applicable for the updated code are called obsolete test cases, and those must be removed/ not executed from the regression test suite
- Re-testable test case – test cases related to the affected code due to modifications done to the original program are called re-testable test cases and those must be rerun during regression testing
- Redundant test case – when compared to a particular regression test, redundant test cases are the ones which execute code segments that are not changed. These test cases also can be omitted from the regression testing.

Good test cases should be,

- Automated - be able to execute without any human interaction
- Repeatable - be able to run multiple times with the same result, preferably by a continuous integration server
- Relevant for tomorrow
- Easy to run without the need for complicated prerequisites
- Isolated - be able to run independently of other tests

When a bug fix or feature implementation is carried out the specific parts of the code will be changed and test cases which ignite those parts of the code considered necessary. From these set of test cases, there can be cases such as [1]

- Fault revealing test cases – they are the test cases which produce incorrect output and causes program to fail
- Modification revealing test cases – if the output of the test case is different from the original output, yet accurate are fallen into this category. So, these test cases output values have to be modified before regression testing
- Modification traversing test cases – if the execution trace is different from the original trace, yet the output is similar. No modifications required for these test cases.

In this research study, regression test case is considered as a script which evaluate the end to end business functionalities. It is consisting of multiple various business transactions and their expected results within a single test case, related to a business scenario under test. These test cases are independent and have all the prerequisites required for the business scenario within the test case itself. Test case should expect to evaluate and verify all the outputs and all the status changes in the system.

1.4 Problem addressed by the research

Agile software development methodologies promote faster product releases and feedback cycles while maintaining software quality. Also, it is encouraged to follow the git-flow approach, where developers commit their changes into the bugfix or feature branches rather than directly releasing the changes. For large scale software projects and open source projects, several parallel developments are expected, and hundreds of change commits will be submitted. Due to the lightweight, easy and fast execution nature, developer testing can be carried out for each and every code update. However, unfortunately, it is not feasible to execute regression test suite upon each code update since it takes a long time to complete (typically in the range of hours and in some cases even in days). For complex systems, even adequately developed and optimised regression suites can grow extremely large, making it infeasible to incorporate into each CI cycle.

Because of such limitations in regression testing, most of modern companies tend to execute their regression test suites offline, typically over the weekend, while investing more on costly hardware and operational activities for the execution. Therefore, the regression test cycle captures a large number of code updates which had been committed over the course of that period by multiple engineers, which then leads to below complexities, if any regression tests had failed.

- Developers will not be able to quickly identify and isolate the erroneous code change, which caused the tests to fail since multiple changes are captured into the regression test run.
- Since frequent regression testing is not practical, errors in the system could be undetected for a long time (a few days if the regression happens over the weekend only).
- Failures to detect bugs at the earliest delays the product releases and creates impedance for the agile practice.
- Also, there can be further changes carried out on top of erroneous untested code segments, all of which will have to be reversed and redone resulting in considerable rework, costing both time and money.
- Integrating erroneous code changes into the develop code path will cause to consume unexpected developer and QA effort to analyse and retest the issue

So if we can identify these problems by carrying out a regression test run, before merging each bugfix/feature branch to the develop branch, it will significantly improve the efficiency of the SDLC process, while eliminating the development overhead that could incur later on due to required rework. Unfortunately, as already pointed out it is impractical to run a complete regression before merging each bug fix/feature branch. Therefore, it is crucial to derive a method which can identify the minimum regression test suite that could detect any possible problems by looking at the particular code changes so that it can be executed efficiently, in parallel to developer testing.

1.5 Objectives and expected outcomes of the research

In an era, where software development practices are moving towards shorter iterations of delivery, leading to more frequent regression testing, it is unnecessarily expensive to perform a full regression test, especially when only a small segment of code has been changed. So there arises the requirement of identifying a selective set of tests to be incorporated into the continuous integration cycles targeting the specific changes done.

The primary objective of this research is to select and prioritise the minimum set of tests for from the full regression with respect to a given code change, in a way that any possible problems of the modified system can be identified as early as possible. The subset of existing test cases has to select based on criteria such as a change in the code, impacted business functionality and considering the duplicate test cases in terms of the function-level call graph. Once the most promising test cases are selected, those test cases need to prioritise so that the failure revealing test cases are executed before the others to detect errors. Also, this approach should be able to identify the distinct code level bugs early as possible rather than executing the multiple failing test cases due to the same issue in the code. So, it will help the developer to detect various code-level errors at the initial stage of the CI cycle and take necessary corrective actions quickly. Following are the advantages of regression test minimisation by code change based test selection and prioritization approach.

- Reduced test suite helps faster CI compared to full regression
- Early and easy error detection in modified code
- Increase the confidence of the developer and client
- Save QA’s time and effort of testing buggy solutions and retesting
- Improve the efficiency of the agile SDLC which helps to achieve rapid product releases

This will make sure that the developer’s output received for the QA testing is in high quality; hence, straightforward issues will not pass down to the QA testing level. The figure 1.4 shows the general test case reduction by regression test case selection and prioritisation methodology.

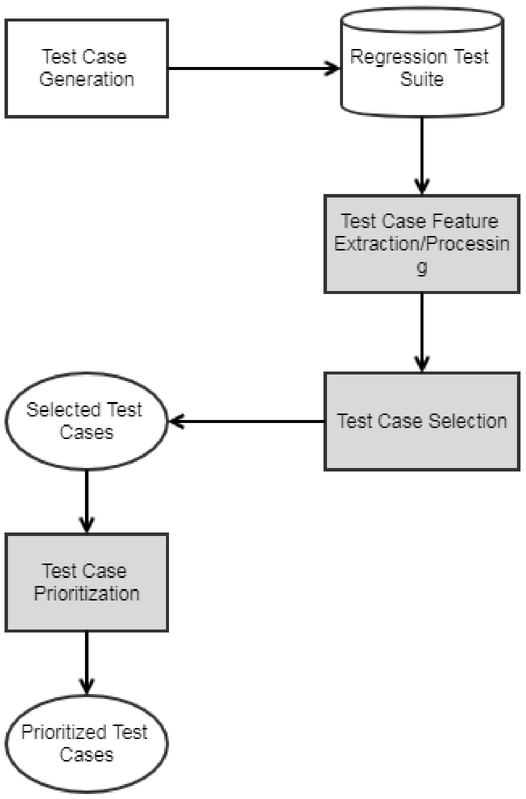


Figure 1.4: Regression test case selection and prioritization

To perform the test case minimization, it is necessary to recognise the features and characteristics of test cases and their behaviours. Further, it is essential to identify the relationship between the code and the test cases. It is already proven by the past researches done on this subject matter that the code change based test case selection delivers better outcome compared to other test case selection methods. So on that basis, this research further improves this approach focusing the following outcomes

1. It is required to maintain the code coverage details of each test case to perform the change-based test case selection. But the granularity of this test case to code coverage relationship is a research question since the higher granularity relationship (e.g.: logic level or statement level of the code to test case coverage mapping) causes high complexity of the system and high relationship maintenance cost while lower granularity relationship (eg: package level or

class level of the code to test case coverage mapping) will not provide usable input to the test case selection process. Hence it is important to identify the optimum level of the granularity of the test case to code coverage mapping, which is one of the key outcomes of the research.

2. Updating and maintaining the test case coverage details upon code changes is another research area which is going to be lightly touched during this study.
3. Removing duplicate test cases is an important aspect of regression test case minimisation. So, identifying those duplicates has to be done by analysing the changed code segment and characteristics of the test cases. During this study, the most significant features of the test case will be recognised and will provide a methodology to remove the duplicate, redundant test cases using the above-recognised features.
4. Another outcome of this research is to discover the most critical test case characteristics which can be used to predict possible failures during execution. Apart from these characteristics, it is equally valuable to find out the optimum order these features can be used for prioritisation which is also addressed during the study.
5. The possibility of finding the distinct bugs in the code using te least possible number of regression test cases will be analysed during this research as another aspect of test case selection and prioritisation.
6. Finally, the possibilities and approaches of integrating the proposed regression test case selection and prioritisation method into the current CI cycle will also be discussed as a part of this research study.

All these analyses are expected to achieve smooth CI cycle for each code change and provide better quality output for QA testing by identifying bugs introduced to the system due to a change as early as possible. Basically, this will improve the overall efficiency of the SDLC and facilitate timely delivery of product releases.

1.6 Scope of the research

The main intention of this research is to select the most suitable test cases from regression test suite for a given code change, so the developers can execute them before merging their changes into the integration and before releasing it for QA testing. The proposed approach should be able to apply without a prior knowledge on the test suite or without any initial expert training. Also, this research should deliver a fully automated regression testing selection and prioritization strategy and it should not be a burden to the current SDLC process.

This research study will only consider the code changes related to the business level functionalities which are mostly done on the solution layer of a product. Basically, it will include the validations, enrichments and processing of transactions and generating its outputs. So, the code changes done on platform level, library level or any other technical level implementations were not considered in the scope of this research. Further this approach is recommended for the projects in maintenance phase due to its limited code changes and requirement of extensive regression testing.

During this research, the capability of employing machine learning (ML) principles to select and prioritize most suitable test cases were discussed. When selecting appropriate ML algorithms, this study was limited for the unsupervised learning approach since it didn’t require any prior knowledge on the test suite and it’s features for the test case classification.

2 LITERATURE REVIEW

2.1 Regression test case selection

Previous studies of regression test case selection can be divided into three main categories based on their model of selecting and reducing test cases,

- Graph-based regression test case selection
- Code coverage-based regression test case selection
- Specification-based regression test case selection

2.1.1 Graph-based regression test case selection

2.1.1.1 Graph models

In literature, there are several different graph models can be identified which can be used to enhance the graph-based regression test case selection. Following are the most common models used in practice [8],

- Flow graph
  The flow graph is the simplest form of the graph model. It is a directed graph where the nodes represent the statement or the logic in the software code, while the edges connecting two nodes represent the relationship between the two statements of the code. Start and end nodes are default to any of the program. Hence there should be at least two nodes per graph.

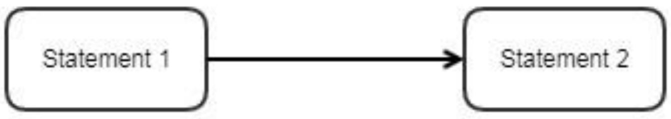


Figure 2.1: Statement level flow graph

- Control flow graph
  Control flow graph (CFG) used to represent the flow of control of the program which is very useful to understand the behaviour of the program for different test cases and identify the most significant test cases of the modified code. CFG

contains process blocks, junctions, decisions and case statements. Constructing and analysing the CFG is very complicated for more extensive programs, hence use various tools for analysis.

- Data dependency graph

  Data dependency graph (DDG) represents the dependency flow for each data (variable) in the program. Two adjacent nodes in the DDG graph exist if node 1 dependant on node 2, where the definition of variable v is at node 1 and its usage is at node 2. So DDG contain a single node for each statement which updates the data (variable) in the code.

- Control dependency graph

  Same as data dependency, control dependency graph (CDG) represents control dependencies of the program where each control statement corresponds to a unique node in the graph. Two adjacent nodes n1 and n2 in the graph happened to control dependent if at least single path from n1 to exit of the program including node n2 as well as one path from n1 to exit of the program that exclude n2.

- Program dependency graph

  Program dependency graph (PDG) makes both data and control dependencies for each operation in the program. Data dependences represent only the data flow relationship of the program, while control dependences represent meaningful control flow relationship. However, many analyses are efficient on PDG since it provides computationally related parts of the program in a single scan of the graph.

- System dependency graph

  The system dependency graph is the enhanced version of PDG, which can represent procedure calls of the system. It contains all the procedure level dependencies between them. Hence it eliminates the limitation of PDG, which can be modelled only for a single procedure.

When analysing the graph bases regression test case selection techniques, it can be divided into two categories based on the architecture of the program/system under test (SUT).

- Procedure level system
- Objected oriented system

2.1.1.2 Regression test case section models for procedural programs

Procedural programming is a programming paradigm based on the series of computational steps of procedures, routines or functions in sequence. So, the following regression test selection techniques are suitable for the procedural programs [8]

- Dataflow analysis based technique

  In software testing, definition-use pairs consist of definitions and uses of a variable, according to the sequence of their appearance in the source code. Uses of variables include computational such as multiplication (c-uses) or predicate such as route (p-uses) of a path. So, the c-uses directly effect on computations and indirectly effect on the control flow. Correspondingly P-uses have the opposite behaviour. The dataflow analysis based technique considers these definition use pairs which get affected due to the changes in the system under test (SUT) and the test cases validate these definition use pair are selected.

  This technique can analyse the changes introduced in multiple procedures and use CFG to represent the SUT. The steps of the test case selections of the approach as follows

  - Processed dataflow information incrementally and analyse the single change in the updated code.
  - Regression test cases validate these changes and select for execution
  - Update the information related to dataflow and test coverage is an update for these selected tests.

- Module-level firewall-based technique

In this technique, data dependencies and control dependencies are considered for the test case selection as follows

  - Modules which are affected and modified by the change are selected into a firewall
  - The flow of control is analysed using call graph, and in the firewall, direct ancestors and descendants modules of the call graph are selected.
  - Test cases which validate the selected modules in the firewall are selected for execution.

- Differencing based technique

  Different between the code before and after the change is considered during this technique. Two popular methods are discussed in the literature

  - Based on Modified code entity – Initially, all the test cases are run on the original code and identify the validated code entities (e.g. function). So once the modification is done check the updated code entities and select the test cases which validate updated entities.
  - Based on textual difference – This approach uses the textual difference of the code before and after the modification. Before analysis, both codes are converted into its canonical form to ensure both codes follow similar guidelines, so that blank lines, comments are excluded from comparison. First, test case coverage of the original programs is identified and then the syntax of both before and after modify code is analysed to capture the change. After that test cases which validate the changed code are selected for execution.

- Control flow analysis based technique

  This technique used the control flow graph to select the most relevant test cases for the modified code. Changes of the code are identified while traversing the control flow graph for the original code and modified code.

  - Execution traces are generated for all the test cases

    - Traverse on the control flow graphs for both original and modified programs in a depth-first manner accordance with the execution trace generated.
    - Compare each test case execution trace for both programs
      If the nodes of the graph are not the same, the edges linking these nodes consider as dangerous edges, and all these test cases which exercise dangerous edges are selected during this test selection technique.

2.1.1.3 Regression test case selection model for object-oriented program

Object-oriented programming paradigm (OOP) organise the data and behaviours of the system inside the objects. It is the most popular approach in extensive, complex system development. Below describe the regression test case selection techniques suitable for the OOP systems.

- Firewall based technique [1]
  As discussed during the firewall-based approach for procedural programming, this technique identifies all the classes which are affected by the code change. Those affected classes will be included in a firewall, and the test cases at least validate one class in the firewall are selected for execution. Two primary firewall-based selection techniques are discussed in the literature.
    - Kung’s class firewall technique – This approach uses object state diagram (OSD), Object relation diagram (ORD) and block branch diagram (BBD) to find out dependencies between program elements. ORD represents the inheritance, association and aggregation relationships as well as the dependencies between classes. Class method, interface and control structure can be represented via BBD. OSD represent the dynamic behaviour of the class.
      So, the mapping information between test cases and classes are captured and when the class is modified classes which are directly and indirectly affected to the modification are included in a firewall using

the above-mentioned graphs. So, the test cases validate the classes in the firewall will be selected as the output of this technique.

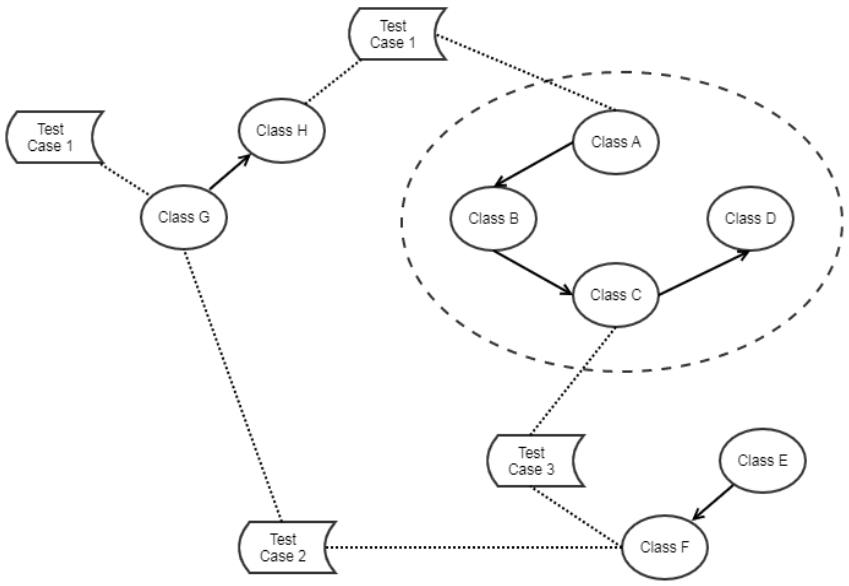


Figure 2.2: Firewall technique for D class

- Method level firewall technique – This is same as a class level firewall; only difference is to consider the methods instead of classes. So all the methods affected by the code modification are selected into the firewall, and test cases validate these methods are considered for execution.

- Design model-based technique [8]

  This technique is suitable for the systems that are developed using the paradigm model-driven software development (MDD). Its popularity has been increased, and the system model can be used to drive the code, hence the most significant test cases. For the object-oriented programs, the design model can be represented via UML.

  Advantages of MDD

  - The mapping between the design model and test cases can be easily maintained than the mapping between code and test cases

- This is a more efficient and cheap approach, especially for the complex systems with a large codebase.
- Modifications in the software can be easily identified via design model rather than the code.
- Solutions are language independent.

Using the design model-based approach; following 3 test case selection techniques are discussed based on analysing different diagrams.

- Based on class and sequence diagram

  This technique uses sequence diagrams and class for the test case selection. It analyses the sequence diagram of the system and generates the concurrent control flow graph (CCFG), which contains additional concurrency details of the program compared to the control flow graph (CFG). Concurrency behaviour is possible using parallel instructions and asynchronies messages. The using the class diagram of the programs, it generates the extended concurrent control flow graph (ECCFG) adding information of both diagrams. So once the modifications are made on the code, both ECCFGs (an original and modified version of the system) are analysed and select the test cases to validate the changes introduced into the system.

- Based on class and state diagrams

  When modifications are introduced, the class diagram of the program as well as the state diagram will be changed, and those two graphs can be used to detect the changes to the elements in the program. So the test cases validate those elements can be selected as the out of this technique.

- Based on UML architecture and design model

  This technique is based on the traceability between program code, design model and test case. Changes to the software can be easily identified from the design model. Hence the test cases exercise those affected areas. Also, this technique uses sequence, class and use case diagrams to distinguish test cases into three categories: reusable, re-testable and obsolete.

2.1.2 Code coverage based regression test case selection

Code coverage based regression test case selection is the most popular as well as most preferred test case selection methodology in literature. Also, as per the past studies, this approach has delivered promising and robust outcomes with greater accuracy irrespective of the complexity and size of the system under test. Since this method directly accesses the changes done to the program and select related test cases by analysing the code change, it can provide an effective set of test cases for prioritisation. The simplicity of this method and the relevance of the features or the process of selection are other factors for its popularity.

The main idea of this approach is to generate mapping between the software code and the test cases [18]. These mapping details are generally stored in the DB level, and it is called coverage database. Basically, it contains the test case and the code section covered from that particular test case. So that if a particular code segment was changed due to a change request or a fix done for a defect, using the coverage database subset of test cases can be identified. Those test cases are the ones which evaluate the changed code segment by executing different functional path s related to various business functionalities.

2.1.3 Specification-based regression test case selection

Software specification is a description of the software system which contains both functional and non-functional requirements. Specification documents are frequently updated to capture the changes of the software due to bug fixes as well as change requests by the client. So, the specification-based test selection technique has been introduced to overcome the drawbacks and limitations of the design-based models discussed under the graph-based technique section. For some occasions, the design model, as well as the source code, is not available for the testers to analyse to recognise

the change and select relevant test cases. For such situations, the specification-based technique for regression test selection is more suitable since the software specification is generally available for the testers throughout the SDLC.
In this technique, an activity diagram will be created to model the affected requirement and system behaviour due to the modifications of the system. Before selecting test cases, they are categorized into two depending on their coverage [1]

- Target test cases – target test cases validate the code elements which are modified during the change implementation.
- Safety test cases – test cases which are selected to achieve the redefined coverage target is called as safety test cases.

Specification-based test case selection follow the below steps to identify the target test cases for execution.

- Create traceability matrix to map the requirements in the specification to the test cases of the regression test. It maps the requirements with its validating test cases.
- When the program is modified, its specification can change. So, by traversing the activity diagram, all the modes and edges affected to the modification can be identified.
- In the final step, all the test cases which validate the identified edges are selected using the traceability matrix as the target test cases.

Identification of the safety test cases of this approach is as follows [8],

- Calculate the cost of each test case
- Calculate the severity probability of the test cases by multiplying total defects and average severity of defects of each test case.
- Calculate the risk exposure of the test case by multiplying the cost and the severity probability of each test case.
- Select the test cases with higher risk value as a safety test.

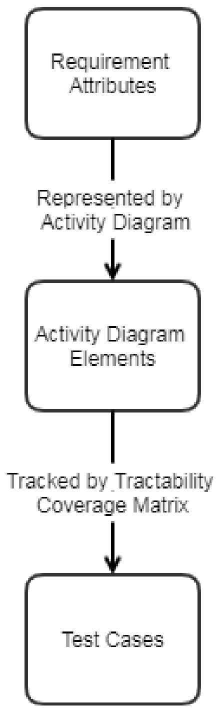

Requirement Attributes
Represented by Activity Diagram
Activity Diagram Elements
Tracked by Tractability Coverage Matrix
Test Cases

Figure 2.3: Specification based regression test case selection

2.2 Regression test case prioritization

Once the most suitable test cases are selected and minimised, regression test case prioritisation (RTP) step can be carried out to decide the priority of test case execution. Test case prioritisation is the process of sequencing the test cases which required to execute in a particular order, so that test cases with a higher priority are executed earlier in the sequence. This is an extension of software testing to increase the test suite rate of fault detection, i.e. how fast a test suite detects errors in the changed program to increase reliability. For a time-constrained condition such as shorter product delivery cycles, test case prioritisation is beneficial to perform during regression testing. Also, it helps to minimise the time and cost consume during software testing phase and make sure that the delivered software product is of excellent quality.

Early fault detection of test case prioritisation allows faster feedback of the system under test so that software engineers can identify the issue and correct it as earlier as possible which leads to smooth software releases process. Following factors has to be considered while executing test case prioritisation. These will validate all discrepancies and ensure that the proper testing is executed in appropriate order during the process of testing.

- Function/Procedure usage frequency or the probability of test failure in software should be considered for test prioritisation.
- Visibility or detectability of an issue to the end client is another aspect of test case prioritisation
- Test case failure risks should be measured to calculate the priority.
- Test cases can be selected as per the priority of all the stakeholder’s requirements.
- Should consider the different importance of quality characteristics for the customer or client.
- History of failures and areas of complex coding/logics should be considered.
- Prioritisation can also be one from the perspective of system architecture or design.

There are several literature statistics on the comparison between random test selections vs prioritised test selection. All the research output confirms that more faults can be identified if test cases are prioritised rather than random selection. For these testing, researches have taken a set of attributes of test cases for prioritisation such as

- Size of test case
- Time taken by the test case
- The effort taken by the test case
- Cost taken by the test case
- Efficiency
- Number of defects found by the test case
- Ability to use in other projects

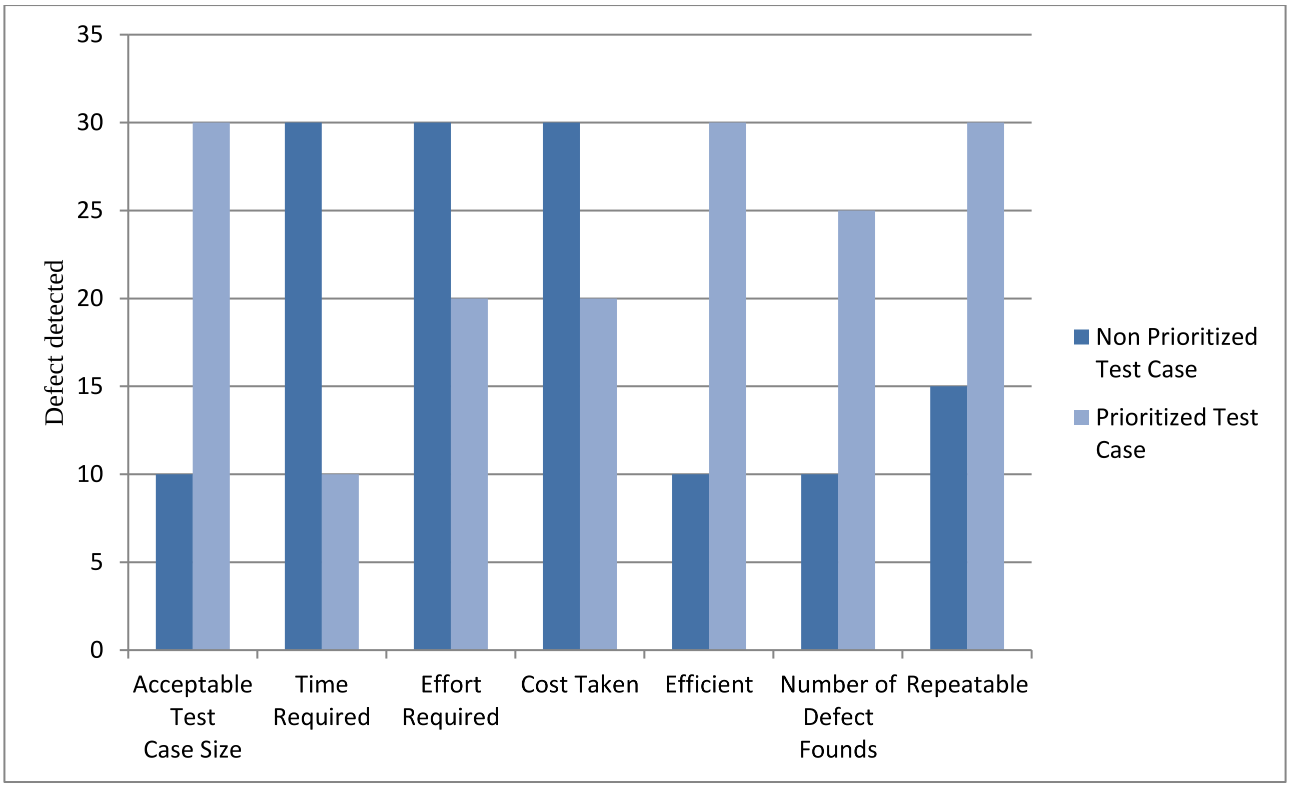


Figure 2.4: Comparison between random and prioritized test cases on different attributes [21]

2.2.1 Test case prioritisation techniques

Test case prioritisation is a complex process which requires the domain knowledge, system architecture awareness and experience of testing of the system. Selecting suitable test case prioritisation technique is an equally important decision that has to be taken from experienced testers considering all the factors mentioned above.

Following are the popular techniques used to prioritise test cases in software testing phase.

- Average percentage fault detected -
  Average percentage fault is the rate where the fault is detected in the code under test. In this prioritisation technique, two different criteria are used to decide a factor based on which priorities are assigned to the test cases.

- Prioritisation using faulty severity –

  Prioritisation of this technique is based on the priority of the requirement to be tested. Test cases satisfy the changed business requirements are assigned high priority while rests of the test cases are considered as low priority. Further, in this approach, the considered requirements are based on the number of times a fault can occur in the code i.e. fault severity. Weight of the requirements are decided considering the following factors [21],

  - Development complexity
  - Measure of business value
  - Volatility of project change
  - Fault proneness of requirement

- Prioritisation in case of regression testing

  In software testing literature, there are nine techniques to prioritise the regression test cases, and each of the technique uses various features of test case in the regression test suite to decide the weight of the test case [22]

  - Random prioritisation

    This is where the test cases are randomly ordered in the test suite, and this is to have additional control of the study of test case prioritisation.

  - Optimal prioritisation

    Results of the known faults are used to identify the effects of other prioritisation methods that will be used.

  - Total statement coverage prioritisation

    This technique instrument the program with test cases to build the coverage details of each test case and prioritised based on the number of statements covered.

- Additional statement coverage prioritisation

  Additional statement coverage prioritisation covers the limitations of total statement coverage prioritisation technique by iteratively select highest statement covered test cases and adjusting the coverage information of the rest of the test cases to find out the test cases stratified the statements which are not yet covered.

- Total branch coverage prioritisation

  Total branch coverage prioritisation is the same as the statement coverage techniques but uses the program branches to measure the test coverage.

- Additional branch coverage prioritisation

  Additional branch coverage prioritisation is similar to the additional statement coverage techniques but uses the program branches to measure the test coverage and not statement.

- Total fault-exposing potential prioritisation

  Prioritise based on fault exposing capability of test cases with higher weight compared to other test cases

- Additional fault-exposing potential Prioritization

  This technique is a combination of total coverage and branch coverage prioritisation which is an extension of total fault exposing potential.

2.2.2 Benefits and challenges of test prioritisation

Regression test case prioritisation is carried out on top of the reduced test suite to identify most critical program bugs/issues at the early stage of regression testing. So that the developers will be able to resolve these issues promptly, which leads to efficient software development life cycle and timely product releases. Also, early detecting of product issues will save both QA and developer’s time since no unplanned efforts will waste in the product testing/error detection. Following are the key benefits of test case prioritization,

- Improves the regression test suit performance and efficiency SDLC process
- Able to detect maximum available faults in a shorter period in the early stage of the SDLC process
- Allows testers to detect defects in the system/process as early as possible.
- Able to integrate the prioritization technique into the continuous integration tool leads to quality product output.

Even though the test case prioritisation has significant benefits on regression testing, there are challenges that makes it time consuming and problematic.

- Identifying the most impacting and relevant features from test cases is one of the challenging activities in test prioritisation. These features have to be unique and should improve the defect detecting capability of the test suite. Features such as code coverage, test case failure history and test case execution time are some of the standard features used in test prioritisation.
- Detecting the application change and asses the impacted areas of the program is another challenge in test prioritisation. Depending on the prioritisation strategy selected test cases for execution may follow the complex program paths. Hence finding and analysing the erroneous code segment may be a complicated task for the developer.
- Selecting scalable prioritisation technique for a complex software system is another challenge in test case prioritisation.

2.3 Usage of machine learning principles for test case selection and prioritization

In the previous literatures, there are few studies related to the usage of machine learning principles for select and prioritize or minimize the regression test suite. During these studies they have used machine learning approaches such as supervised learning, unsupervised learning and reinforcement learning to group and prioritize the test cases. This section described the details of these studies related to each ML approach.

2.3.1 Unsupervised learning based test case minimization

In this approach, researches [14] have used k mean clustering and hierarchical clustering algorithms to cluster the test cases into the 2 groups, effective test cases and non-effective test cases based on their coverage details. Following are the steps carried out to select the test cases for execution.

- Statement level code coverage details of each test case were gathered by instrumenting the source code while executing each test case. So, each code line was marked as 1 or 0 representing whether the line is covered from the test case or not.
- Convert the statement/line coverage details into a binary vector and calculate this vector for all the test cases.
- Cluster the test cases into 2 groups which is considered as effective and non-effective test case group. This clustering has been done using the vector value calculated based on the coverage details. For clustering, the researches have used k mean clustering and hierarchical clustering algorithms. This is done by calculating the Euclidian distance between 2 vectors which gives the similarity of the test cases.
- Select the test group with most previous failures considering the code change, as the effective test group for execution.

As per the results of this studies, it can be concluded that the k mean clustering provides higher performance compared to hierarchical clustering, where 73.18% accuracy, 19.32% precision and 100% recall rate. Also, this test was done for the 3 different levels of test case to source code granularity, which are statement, block and method level. Here the statement level coverage provides higher performance and method level coverage provides lower performance.

2.3.2 Supervised learning based test case minimization

Supervised learning requires labelled data as a pre-requisite to train the classification model which is the major drawback in this approach. For the training data, importance and non-importance of a test case can be considered based on the functionality evaluated and previous defects. In the related literature [12], following test case details were considered to extract from test cases for feed into the ML model.

- Test case description (natural language)
- Test case age
- Number of linked requirements
- Number of linked defects (history)
- Severity of linked defects
- Test case execution cost (time)
- Project-specific features (e.g., market)

Also, this study was done using multiple supervised learning algorithms such as

- Ranked Support Vector Machines (Ranked SVM) [12]
- K Nearest Neighbour [12]
- Logistic Regression [13]
- Neural Network [15]

2.3.3 Reinforcement learning based test case minimization

Reinforcement learning approach used the feedback of the classification model output as an input. In the related studies [17], result of a test case is used as the reward function which provides the feedback. So, the failures of a test case can be considered as the positive rewards and success execution of a test case can be considered as the negative reward to the learning model. Further, this approach needs to keep track of the past test execution results and methodology to invalidate the older test data which could be irrelevant over time.

2.4 Evaluation criteria

Evaluation and benchmarking of the regression test case selection and prioritisation approaches must be done comparing a standard general methodology. In software testing literature, most popular benchmarking approaches can be categorised into below three approaches.

- Retest all test case selection
- Random/Ad-Hoc test case selection
- Manual test selection – Smoke test

Retest all test case selection

Retest all test case selection can be considered as the most straightforward and oldest regression test case selection technique. This approach simply selects all the existing test cases in the regression test suite. It is most appropriate when the size of the source code and the test suite is manageable. However, the problem starts when the test suite size is getting bigger with the improvements of the source code which leads to increase the running time of the regression test suite. The main advantage of this method is since all the test cases are selected for execution; all the faults/bugs can be detected compared to the reduced test suite.

Random/Ad-Hoc test case selection

Random/Ad-hoc selection approach chooses a subset of test cases randomly from the regression test suite and the random algorithm as well as the number of test cases can be varied by the judgement of the human. Hence this technique provides faster test case selection, but due to its random nature, the defect detection capability (performance) could not be guaranteed.

Smoke test selection

This is a manual test case selection method where QA engineer select the fixed subset of test cases from the regression test suite which covers the main functionalities of the software system. This approach is more suitable for large scale software systems where the full regression is not feasible to execute in daily CI cycle. Since this is a fixed set of test cases, no guarantee on the defect detection capability and reduced test suite performance.

2.4.1 Evaluation parameters

To measure the evaluation parameters of the test case selection and prioritisation methodologies, it is required to define and calculate the baseline for each test case. The baseline can be obtained by executing the full regression (complete test suite) and identifying the actual test failures as well as successful execution. For the purpose of statistic gathering and classification following key terms are computed comparing the actual test results of the full regression and results of the proposed regression test case selection and prioritisation method.

- True Positive (TP)
- True Negative (TN)
- False Positive (FP)
- False Negative (FN)

The term positive and negative refers to the prediction from the proposed test selection approach also known as the expectation, and the term true and false refers to the actual results from the baseline approach also known as the observation [19].

Table 2.1: Test case classification

	Truly effective	**Truly non-effective**
Predicted effective	TP	FP
Predicted non-effective	TN	FN

Considering the above mentioned classification metrics, the below evaluation parameters can be defined to evaluate the performance of the proposed test case selection approaches [7].

Accuracy

Accuracy is the measurement of the closeness of the proposed approach compared to the actual (true) values. It can be represented as the percentage of the sum of all true positives and false negatives out of all the true positives, true negatives, false positives and false negatives.

$$accuracy = \frac{TP + FN}{TP + TN + FP + FN}$$

Precision

The precision of a system can be defined as the degree of the repeated measurements under unchanged condition shows the same results. Precision is also interpreted as the fraction of selected test cases that are relevant to a particular change. It is calculated as the rate of true positives versus the number of test cases selected from the reduces test suite and range between the 0 and 1. A precision of 1 means all the selected test cases are relevant.

$$precesion = \frac{TP}{TP + FP}$$

$$precision = \frac{(relevant\ tests) \cap (selected\ tests)}{(selected\ tests)}$$

Recall

Recall measurement which is also called as sensitivity shows how many of the relevant tests were selected from the proposed methodology. It is calculated as the rate of true positives versus the sum of true positives and true negatives and it ranges between 0 and 1 where 1 mean all the relevant test are included in the set of selected test cases [19].

$$recall = \frac{TP}{TP + TN}$$

$$recall = \frac{(relevant\ tests) \cap (selected\ tests)}{(relevant\ tests)}$$

F-measure

F – Measure is defined to calculate the trade-off between precision and recall value of an information retrieval system [7].

$$F - measure = 2 * \frac{precision * recall}{precision + recall}$$

Test case reduction rate and efficiency

Number of test cases selected from the proposed test case selection and prioritisation method compared to the test cases selected in the base methods such as full regression, random ad-hoc selection can be considered as the test case reduction rate of the proposed approach. Also, the same figure can be measured by calculating the ratio of the average execution time of a test case into the number of selected test cases and the time taken to the base methods i.e. for the full regression.

Mutants killed

A mutant is a changed to the source code which introduces a defect to the system. A test case which reveals the corresponding defect is said to kill the mutant [16]. If the number of mutants killed from the reduced test suite compared to the retest all approach could be considered as another evaluation parameter rank the test case selections.

3 METHODOLOGY AND CONCLUSION

3.1 Methodology

3.1.1 Introduction

The proposed regression test selection and prioritisation methodology is designed to detect defects in the system as soon as possible without executing redundant test cases which evaluate the same functional paths in the source code. To achieve this objective, the machine learning approach has been taken to group the business-wise similar test cases based on their identified features. Once the most relevant test cases are selected, it will be prioritised based on the same features so that the test cases which have more potential to detect issues will be executed early in the reduced test suite. So with this approach most relevant test cases will be dynamically selected for the code change and will be executed in a lesser amount of time; hence it is practically possible to detect defects in the changed code before the changes are merged into the release code path and QA testing.

3.1.2 Machine learning

The machine learning (ML) approach is becoming more popular in software solutions which can be used to automate and improve the computer-based learning process using their experience without being programmed or any human intervention. The idea is to feed quality training data into the machine learning algorithm which build the ML model based on the sample data. There are different types of ML algorithms available in the literature and has to select considering the nature of data and task required to achieve.

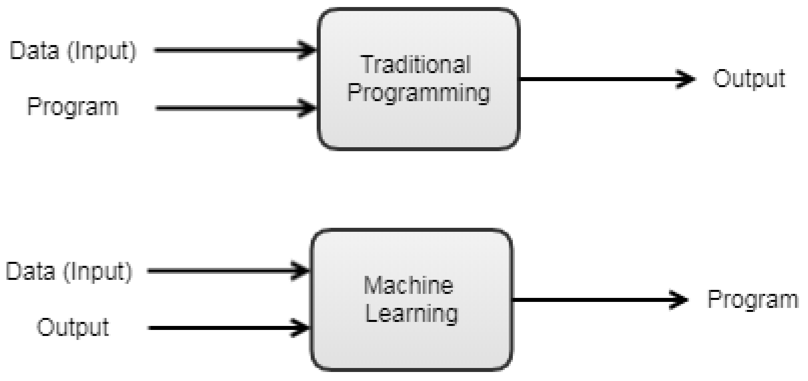


Figure 3.1: Traditional programming vs Machine learning

The fundamental difference in machine learning and traditional programming is, in traditional programming, the data and program logic is feed into the machine as input and expect the output as a result. In contrast, machine learning required data and expected output results as input to the machine and provide program logic as the output in the time of learning which is also called as training. This conceptualised model can be tested using data which was not fed to the model during the training phase and evaluating its performance using metrics such as precision, recall and F1 score. Once the ML system was trained, it can be used to evaluate the input and gets the output as results.

There are different ways of classifying machine learning problems to identify the most suitable machine learning algorithm. It is primarily based on the nature of the input data for learning and the feedback availability for the learning process.

- Supervised learning

  In supervised learning, the system is presented with the input data and their expected output generated from the known source. So the goal of the ML system is to learn a general rule to map between inputs and outputs/ this training will continue until the model reach specified level of accuracy on the training data set. So, this approach is possible if the labelled input data are available for training. E.g., Image classification, market prediction

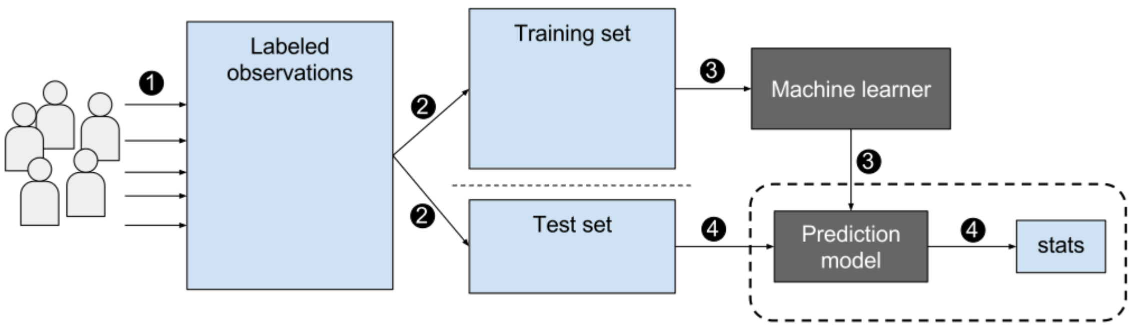


Figure 3.2: Supervised learning process [23]

- Unsupervised learning

  In this approach, no labelled data is given to the system. Instead, the ML system has to identify the features of the data and find its own structure for classification/clustering. Unsupervised learning is used to discover the hidden patterns of data and cluster the given data population so it can be labelled accordingly. Most common unsupervised learning mechanisms are clustering, high dimension visualisation and generative models.

- Semi-supervised learning

  This type of learning sits between both supervised learning and unsupervised learning, where a large amount of input data exists, and only some of them are labelled. So the techniques used in both supervised learning and unsupervised learning has to be used in this type of ML modelling.

- Reinforcement learning

  This type of modelling is based on the feedback mechanism where the system is interacting with the dynamic environment to achieve a specific goal and receives positive or negative feedback as rewards or punishments. According to the feedback, the ML system adjusts the processing logic to achieve the best possible mapping between input and output.

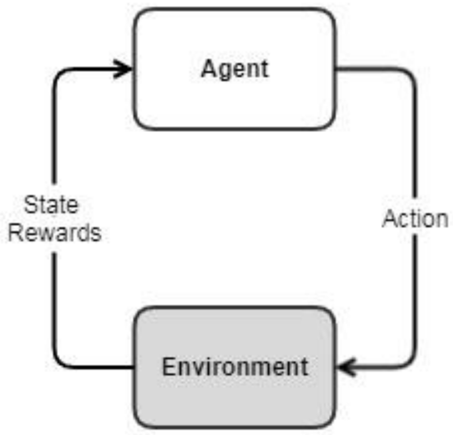


Figure 3.3: Reinforcement learning process

3.1.3 TF-IDF scoring

Tf-idf is an information retrieval and text mining methodology based on the statistical weight on the words in the document. It stands for term frequency-inverse document frequency and evaluates the importance of a word to a document in a collection or corpus. According to the tf-idf calculation, the importance of a word will increase proportionally to the number of times a word appears in the document, but it will offset by the frequency of the word in the corpus [24]. This is considered as the most frequently used document scoring and ranking scheme in search engines and text extraction modules. There are different varieties of tf-idf weighting scheme in information retrieval literature. The most straightforward ranking function is computed by summing the term frequency and inverse documents frequency component for each query term.

Typically, the tf-idf weight consists of two terms: normalised term frequency and inverse document frequency.

- TF: Term frequency

  This component calculates the number of times a word appears in a document, and it measures how frequently a term occurs in a document. Since the different documents are in different lengths, the term may exist multiple times in long documents than shorter documents. Therefore, to normalise the term frequency value it is divided by the total number of terms in the document (document length), so it can be used to compare the importance of the word across the document.

  TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document) [24]

$$tf(t,d) = \frac{n_t}{\sum_k n_k}$$

- IDF: Inverse document frequency

Inverse document frequency is computed as the logarithm of the number of all documents in the corpus divided by the number of documents [24] where the interested word appears, and it will measure the importance of the word. So, the most commonly used words in the entire corpus such as "is", "of", "that" weighted as not significant compared to other words which are not commonly available and rare words can participate for the document classification.

IDF(t) = log_e(Total number of documents / Number of documents with term t in it) [24].

$$idf(t, D) = \log \frac{|D|}{|\{d_i \in D \mid t \in d_i\}|}$$

Once above mentioned two components are calculated, TF-IDF weight can get from the product of TF and IDF scores. Therefore, the TF-IDF score provides higher weights for rare terms.

$$tf - idf(t, d, D) = tf(t, d).idf(t, D)$$

3.1.4 Data clustering

Clustering is a task of dividing a set of data into several groups where the data points in the same group share the same behaviours/features while data points between the groups are dissimilar each other. It is an unsupervised machine learning method which uses unlabelled data set as input. Generally, clustering is used to

- Find meaningful structures of the population
- Grouping inherent
- Explanatory underlying processes
- Generative features

Clustering methods [20]:

- Density-based method

  This type of clustering is based on the density of the data population. Data in higher dense region are expected to have similar behaviour compared to data in the lower dense region. Density-based clustering is known to have good accuracy and the ability to merge two clusters.

- Hierarchical based method

  Clusters created in this method are based on tree type hierarchical structure. There are two main categories

  - Agglomerative – bottom-up approach
  - Divisive – top-down approach

- Partitioning method

  Partitioning method groups data points into k clusters and each group create one cluster. Grouping is done based on the similarity functions such as distance between data points etc. most common example for partitioning method is K mean clustering.

- Grid-based method

  In grid-based method, data points are mapped into a finite number of cells which form a grid like structure. So the clustering operations are based on the defined grids, hence it is fast and independent of the underline data set.

3.1.4.1 K-Mean clustering

K-mean is one of the famous and simple unsupervised clustering algorithms used in data analysing/mining solutions. This algorithm follows a simple and straightforward approach to classify a given set of data into a predefined number (assume K) of clusters. The main idea is to define the K centres of the data set, and each data point will be assigned to a particular centre for clustering. Following are the steps carried out in the k-mean clustering algorithm

1. First, the number of groups/clusters (k) has to be selected as a precondition for the algorithm. To figure out the optimum number of groups, it is required to check the data as a whole and identify the distinct grouping.
2. Then randomly initialise all k centre point.
3. Compute the distance from each data point to each group centre and classify the data points to the respective centre point group whose centre is closer to it.
4. Based on the classified points, the group centre is recomputed by taking the mean of all the vectors of the group
5. Repeat the steps 3 and 4 for a predefined number of iterations or until the group centre is fixed between iterations.

Advantages:

- K-Mean clustering is fast and robust
- Due to the simplicity it’s easier to understand and implement.
- Relatively efficient with O(tknd), where
    - n – Number of objects
    - k – Number of clusters
    - d – Number of dimensions of each object
    - t – Number of iterations. Generally, k, t, d << n.
- Gives best results when the dataset is dissimilar or well separated.

Disadvantages:

- Number of cluster centres required to be identified and fed into the algorithm prior to the execution.
- This algorithm will not be able to identify highly overlapping data as separate clusters due to its exclusive assignment.
- Local optima of the squared error function are used for the learning algorithm.

- Random selection of the cluster centre may not lead to the productive result and less consistency.
- This approach is applicable for the data where the mean can be defined. It is not suitable for categorical data.
- This technique is not able to handle outliers and noisy data.

3.1.4.2 Optimal number of clusters

Most of the clustering algorithms are required to specify the number of clusters k to partition the data. So, it is required to identify the optimal number for clustering to apply the clustering methods, which is another fundamental problem. Number of clusters for a particular data population depends on the methods used to measure the similarity of the data points as well as the parameters used for partitioning.

There are different methods to determine the optimal number k for clustering. However, none of them is provided absolute value yet provide a good estimation. These methods can be categorized as

- Direct method

  The direct method uses the within-cluster sum of square or the average silhouette to optimise the criterion of clustering. Known examples are the elbow method and the silhouette method.

- Statistical testing method

  This method comprises of associating evidence against the null hypothesis. Gap statistics method is a famous example.

3.1.4.2.1 Elbow method

Elbow method calculates the within-cluster sum of square (WSS) of each data point as a measure of compactness of the cluster. Smaller the total WSS, higher the cluster compactness. In this method, the total WSS is calculated for different k (cluster) values and should select a cluster number therefore adding another cluster does not improve/impact on the total WSS value. Usually, this is determined by plotting the total WSS value against the number of clusters. Following are the steps used for the calculation [20]

1. Compute the relevant clustering algorithm (e.g., k-means) for the given data set for different k values.
2. Compute the total within-cluster sum of square (WSS) for each k value.
3. Plot the calculated WSS value against the k value.
4. The optimal number of clusters (k) for the given data set can be identified from the sudden bend (knee) position of the curve.

3.1.4.2.2 Average silhouette method

This method calculates the average silhouette of observations for various k values. The k value, which provides the maximum average silhouette, is considered as the optimal number of clusters for clustering. This measure the quality of clustering based on the data point arrangement between clusters. Following are the steps to compute the average silhouette method [20]

1. Perform the clustering algorithm (i.e. k-mean clustering) for range of k values (k from 1 to k)
2. For every k value, compute the average silhouette of observations
3. Plot the graph of average silhouette value vs. cluster count k
4. The k value of the maximum average silhouette is reflected as the optimal number of clusters

3.1.4.2.3 Gap statistic method

The gap statistic method evaluates the total within intra cluster variation for k values with the same value of the data distribution with no obvious clustering (null reference). The k values which maximise the gap statistic can be identified as the optimal clusters for the data set. And it will guarantee that the clustering is isolated from the random uniform distribution of points.

1. Cluster the data and compute the total within intra cluster variation (Wk) for the range of k values 1 to k.
2. Perform the above step for generated data sets (B) with random uniform distribution and calculate the corresponding total within intra-cluster variation Wkb.
3. Calculate the gap statistic using computed Wk and Wkb values as follows. Also compute the standard deviation of the statistics.

$$Gap(k) = \frac{1}{B}\sum_{b=1}^{B} \log(Wkb) - \log(Wk)$$

4. Select the number of clusters as the smallest value of k where the gap statistic is within one standard deviation of the gap at k+1 [20]:

$$Gap(k) \geq Gap(k + 1) - sk + 1$$

3.1.5 System under evaluation

To implement and evaluate the proposed regression test case selection and prioritization method, a post-trade system developed by LSEG technology has been selected with the available regression test suite. This system is designed as a mission-critical distributed system which has both software and hardware fault tolerance scheme. The communication between components in the system happens via in-house develop message passing platform. The system consists of 20 business processes, including five critical processes (Engines) and 15 transaction gateways. The system has five primary logical partitions based on business functionality.

- Clearing partition
- Settlement partition
- Payment partition
- Depository partition
- Corporate action partition

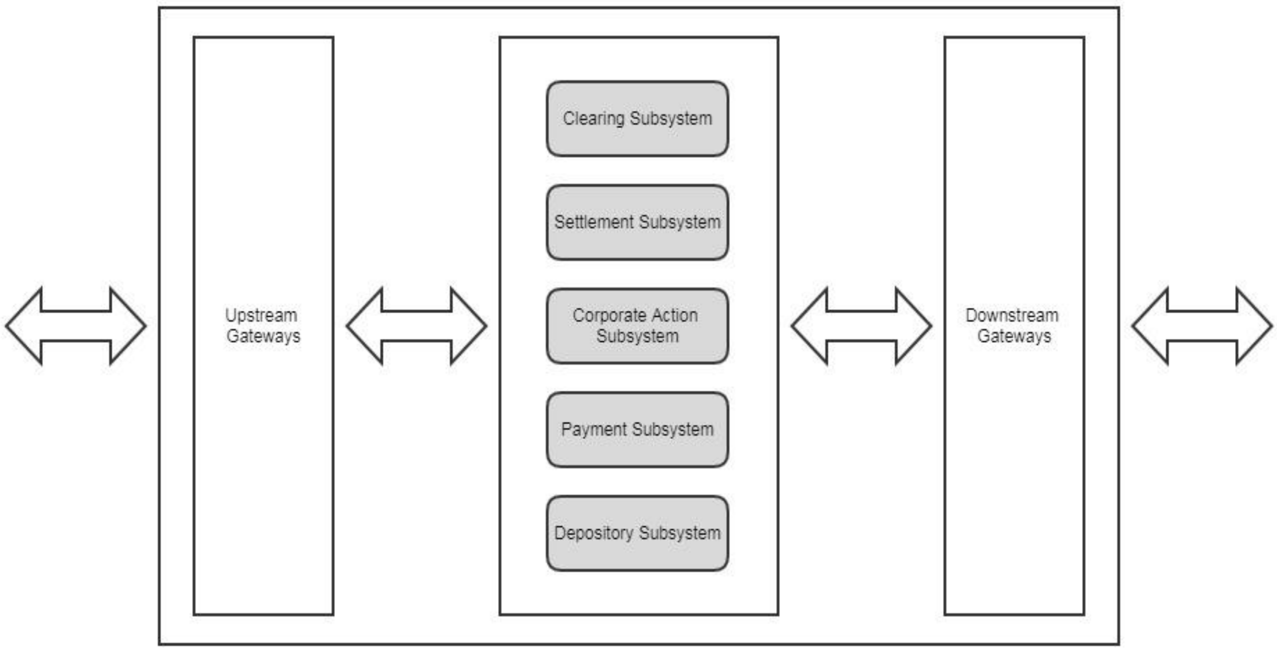


Figure 3.4: Post trade system overview

Each partition has its own engine to perform the business functionalities and input and output transaction gateways are connected to each engine process to receive the incoming messages from external systems and send processed output messages to downstream systems. Post-trade systems are complex mission-critical software system which expected to have high availability and fault tolerance. To cater these requirements, each process has replicas which can be configured to operate as hot standby or cold standby mode. Also, these systems process millions of real-time and batch transactions using complex business logics and should have high real-time performance.

The main functionality of this system is to perform the activities which need to be carried out on the trades generated by exchange systems (post-trading activities). For example, most common tasks are, distributing securities from seller to buyer and transferring money from buyer to seller. Following system and practical features are considered when selecting the system to implement and assess the propose regression test selection and prioritisation method.

- Should have an in-depth understanding of the system, its architecture, business functionalities and operations so that the results of the research can be analysed and understand easily
- The system should have a comprehensive well-maintained automated regression test suite with high functional and code coverage
- The system should have the capability to dump the code coverage information for a particular regression
- If the system/project is already included in a continuous integration plan, it will be added advantage for the research testing
- Project should follow git-flow approach for versioning which include both source code and test suite.

The framework layer of the system is implemented using C++, and most of the business functionalities are implemented using in-house developed preparatory scripting language. These business scripts are consisting of procedures, and there are 510 such procedures exist in Corporate action subsystem. The complete script contains 45000+ lines which is compiled and run on the in house developed rule engine. Since the business logics are based on this procedure language and end to end regression test suite covers mainly the business implementation of the system, it is decided to apply the proposed test case selection and prioritisation methodology on it. On the other hand, platform level implementation is hardly changed and has less impact to the regression suite compared to the business level implementation.

3.1.6 Regression test suite

The selected system has well maintained and automated regression test suite, which can be run multiple times using the continuous integration platform. LSEG uses bit bucket as its version tool following the git-flow approach for all the developments, including test framework. Also, they integrate their products changes into the release code path using the automated continuous integration tool called bamboo, which is a product of Atlassian. The system has five different subsystems as mentioned earlier and each subsystem as its own automated regression test suite. All these tests are end to end (E2E) functional tests implemented using ClearTH, test case development platform by Exactpro Systems. This platform has the capability to implement different test cases by providing the input data as a CSV file format and execute them and record the results into a file.

Table 3.1: Post trade system regression test results

Subsystem	**Total Test Cases**	**Passed Test Cases**	**Failed Test Cases**	**Success Rate %**	**Duration (HH: MM)**
Corporate Action	912	911	1	99.89	4:17
Settlement	433	426	7	98.38	9:26
Clearing	234	226	8	96.58	2:06
Depository	57	50	7	87.72	1:12
Payment	65	63	2	96.92	2:13
Total	1701	1676	25	98.53	19:16

For this research exercise, the regression test suite of the CA (corporate action) subsystem is considered due to the below reasons

- CA regression test suite has the highest success rate of over 99% for the last ten full regression cycles
- It has lesser number of fluky test cases
- Consists of 912 individual test cases which are considered as E2E tests
- CA test cases are independent of each other so that test cases can be executed in any given order and all the preconditions for each test case will be generated within the test case itself
- CA regression test suite has over 95% of source code coverage
- CA functionality have widely spread business scope, and it has complex computational logics which leads to be the perfect candidate for the test selection and prioritisation model
- Version control of the test suite updates are done following the git-flow approach and have each test update can be mapped to the particular source code change easily.

3.1.7 Proposed methodology

The key stages of the proposed regression test case selection and prioritisation methodology can be listed down as below.

- Gather statistics for identified features for each test case
  - Code functions invoked during the test
  - Function/procedure call sequence
  - Code coverage of each function/procedure
  - Overall code coverage
  - Statement coverage map for each function/procedure
  - Execution time
- Preparation of each function/procedure call graph for clustering
- Cluster similar test cases using function/procedure call graph
- Select and prioritise test cases upon code change based on
  - Affected code function/procedure
  - Similarity of the test cases
  - Code coverage of the affected function/procedure
  - Code coverage of the overall test E2E case
  - Test case execution time

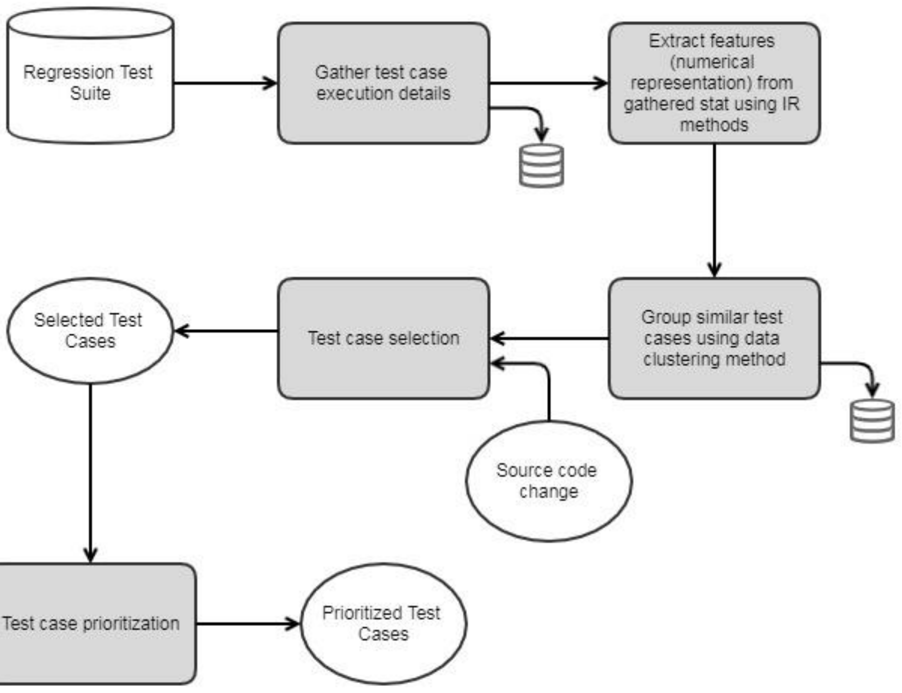

Regression Test Suite
Gather test case execution details
Extract features (numerical representation) from gathered stat using IR methods
Group similar test cases using data clustering method
Test case selection
Source code change
Selected Test Cases
Test case prioritization
Prioritized Test Cases

Figure 3.5: Proposed test case selection and prioritization process

3.1.7.1 Gather test case statistics

Test case statistics information plays a vital role in implementing the proposed regression test case selection and prioritisation method as it is the key driver of this methodology. Test case statistics have to be collected for all the test cases in the full regression for the selected corporate action subsystem. Before collecting the statistics, the CA full regression test suite has been run on the verified bug-free source code five times to identify the fluky/unstable test cases. As per the test, 38 such test cases have been recognised and removed from the regression test suite.

The granularity of the coverage statistic information is an important research question that arises during this implementation. Basically, there were three options to select considering the selected system architecture design and technology stack used

- Package/class level

  This is the most abstract level of coverage statistics information for the test cases. Hence the complexity and the load of the statistic information will be minimal for this type. So, the capturing and processing of this type of statistic data will be less complicated, and test case selection will be fast. Further, the low granular test information will not invalidate the mapping between code and test cases for every code change. So, the collection of statistical information is not required to be done very frequently, which is another advantage in this method.

  However, the major drawback of this approach is that the gathered test case statistic information will not be sufficient to decide on the most impacting test cases upon code change due to its abstract nature. Therefore, more granular level test case to source code mapping and statistic information is required to feed into the test case selection algorithm to get more accurate selection output.

- Statement/logic level

  In this approach, the statistic information is collected on the statement or the logic level of the code. So, the mapping between the code and the test case contains more granular details than the previous method and hence the gathering, persisting, maintaining and processing this statistical information is a costly activity. Also, due to its high granular nature, this information could get invalidated even by a simple code change which the statistical data to be rebuilt more frequently. Nevertheless, the test selection and prioritisation performed on this approach will be more accurate due to its fine grain input data.

- Function/procedure level
  Test case statistics gathered up to the function, or procedure level sits between the above two methods and delivers average test selection accuracy and performance while providing maintainable and detailed statistic information. For the system chosen system last 100 commits had only 8 call graph changes

Considering the above mentioned advantages and disadvantages of different granular level test case coverage statistic extraction, function/procedure level feature extraction is chosen for the proposed methodology.

Once the stable test cases are selected from the regression suite, the next important step is to identify the features/statistic information of the test cases and process to extract those features. Considering the proposed test case selection and prioritisation mechanism, following test case-related information is extracted while performing full regression for the corporate action subsystem.

Table 3.2: Test case execution details extraction process and usage

Feature	Extraction process	Usage of the feature
System functions or Procedures	To identify the procedures called form a test case, change has been done to the script execution engine so that the procedure	This information is used to construct the mapping between the source code (procedure) and the test

invoked for a test case	name is dumped into the log file during test case execution. So this log file is collected for each test case and process using a python script to extract the triggered functions.	case. So if procedure is changed, impacted test cases can be identified easily.
System function or procedure invoked sequence during test case execution	Same as above, the procedure call sequence per each test case is dumped into a file and extracted using a python script.	Procedure call sequence is intended to use to identify the similar test cases in terms of the procedure call sequence. This is to avoid business functionality-wise duplicate test cases been selected multiple times.
Coverage map (Heat map) of each procedure for a test case	Script execution engine has the capability to identify the excited code statements during test execution and mark those statements in the full code file.	Code coverage heat map which has the statement level mapping between test case and source code will be used to check the possibility of identify the impacted test cases with finer granularity.
Code coverage of each function or procedure for a test case	A python script has been implemented to capture the invoked statements and persisted above to calculate the lines covered in each procedure for a particular test case.	Code coverage information within the procedure can be used to prioritise test cases. Test cases with high code coverage will be

		prioritised over the other test cases when the particular procedure was updated.
Overall code coverage of a test case	Same as above, the persisted information is used to deduce the total code coverage of a test case	Overall code coverage of the test case will be used to prioritise the selected test cases. Test cases with high overall code coverage will be prioritised over the other test cases when the particular procedure was updated.
The execution time of each test case	The test execution tool (ClearTH) provides the test case execution time in milliseconds and execution tool was updated to write the execution time into the same log file discussed above.	This information is useful to prioritised test cases considering the time limitations of the execution plan.

During this test case feature gathering process, the above details were extracted for each test case in the corporate action full regression and persisted into two DB tables. These data will be processed and used in the subsequent steps in the proposed test case selecting and prioritising methodology. Details of the two DB tables are described in table 3.3 and table 3.4.

TESTCASE_COVERAGE_STAT – This table persists the details of the procedures and there call sequence for each test case execution

Table 3.3: Test case coverage statistic persisted data

Column Name	Description
TESTCASE_NAME	Name of the test cases. In this exercise, each test case has a separate input data file. This filename was considered as the test case name
PROCEDURE_NAME	Code procedures of the corporate action subsystem triggered from each test case
PROCEDURE_SEQ	Code procedure sequence called from each test case. This will be maintained as the incremental number sequence per test case.
PROCEDURE_LINE_COUNT	Code line count of each procedure was recorded based on the source code
COVERED_LINE_COUNT	Triggered line count of each procedure during test case execution
COVERAGE_PERCENTAGE	(Covered line count/procedure line count) *100% as the coverage percentage per procedure per test case
IS_REQUIRED	This is an enable/disable flag to control each record to participate into the proposed data processing.

TESTCASE_EXECUTION_STAT – This table persists the test case execution details for each test case

Table 3.4: Test case execution statistics persisted data

Column Name	Description
TESTCASE_NAME	Test case name (input data file name)
EXECUTION_TIME	Test case execution time
TOTAL_COVERED_LINE_COUNT	Total covered line count of the source code during test case execution
TOTAL_LINE_COUNT	Total code line count of the corporate action subsystem
COVERAGE_PERCENTAGE	(total covered line count / total line count) * 100% as calculated as the code coverage percentage of the test case
TEST_GROUP_ID	Group id of the test case is calculated based on its procedure call sequence similarity which is explained in the subsequent steps

3.1.7.2 Extract test case features based on the procedure call graph

One of the most important concepts of the proposed regression test selection method is that the duplicate test cases based on the similar function call graph, i.e. test cases which evaluate the same business flow, are eliminated so that the selected test cases are functionally independent and will cover different business functionalities. This will expand the distinct issue detecting capability of the reduced regression which is intended to run on every code change.

To achieve the above requirement, function call graph or the procedure call sequence of each test case has been formatted as a sentence where function names are listed as words and separated using spaces. Before formatting, the technical and utility functions were removed, so that the sentence will only contains the business functions.

So, the test cases which evaluate similar business flow can be identified by the analysing the similarity of the above formatted sentences. To extract the features of each sentence and converted it into a numerical format, following text information retrieval (IR) methods were considered.

- Bag of wards
  This method only checks the term frequency of each sentence.
- Word2vec
  Word2vec represent a word in a sentence as a vector. This approach required complex processing and it consider the placement of word in the document to some extent.
- TF-IDF
  This method calculates the importance of a word in a sentence where it will increase proportionally to the number of times word appear in the document but offset by the frequency of word been in the group of documents.

TF-IDF text analysing method has been selected to extract the information from the function/procedure call sequence of each test case so that the test cases test the similar business functionalities will have similar TF-IDF score. When selecting text classification and scoring method, the following factors has been considered,

- Tf-idf is a popular information retrieval and text mining methodology based on statistical weight on the words in the document
- Since the tf-idf evaluate and score based on the importance of the word to a sentence, the key procedures were prioritised over other procedures and will have a higher impact on the tf-idf score.
- Implementation and computation of this methodology is simple

Before applying the tf-idf on the extracted procedure call sequence of the test case following data formatting and filtering steps were carried out in order to obtain better output. This was done using a script created from sklearn python library

- Procedures which are not participating in the business functionality, yet called from the test case, i.e. platform/framework level procedures were removed from the analysis manually by disabling the IS_REQUIRED flag in the TESTCASE_COVERAGE_STAT table
- The rest of the procedures were formatted as a sentence where the procedure order (word order of the sentence) reflect the call sequence of the test case during execution
- The created procedure sentences were cleaned to remove any stop words, punctuation marks and digits
- Finally, these cleaned sentences were fed into the tf-idf vectorizer for scoring.
- Tf-idf scores for each test case were plotted for the visualization of the input data/features

As the output of this step, sparse matrix of test case vs function/procedure name was calculated where the tf-idf score of each function are the values in this matrix. So, test case can be represented as a vector of tf-idf scores and each position in this vector represent a function in the call graph. Figure 3.6 and 3.7 shows the output of these vectors after formatting them on 2D and 3D space using dimension reduction technique in principle component analysis (PCA).

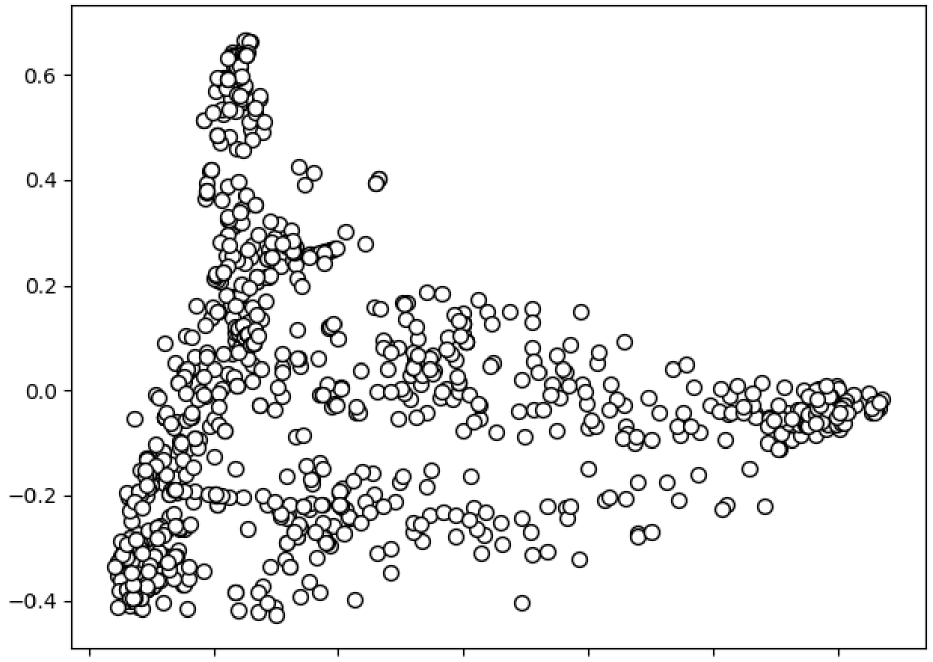


Figure 3.6: TF-IDF score of each test case - 2D graph

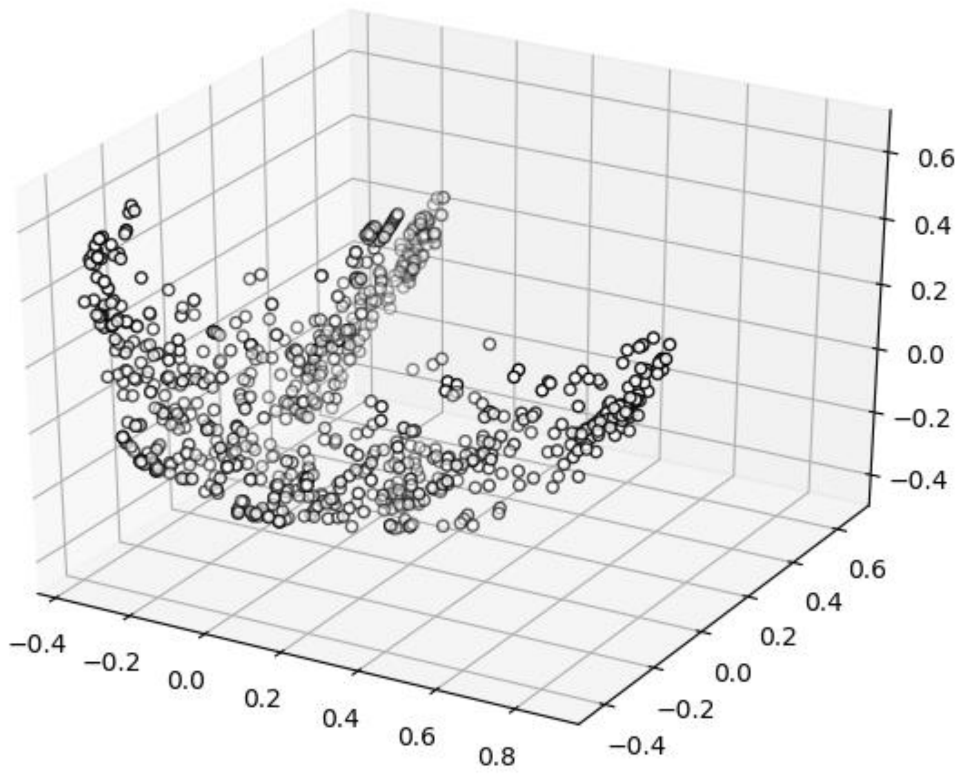


Figure 3.7: TF-IDF score of each test case - 3D graph

3.1.7.3 Group similar test cases

Objective of this step is to cluster the similar procedure call graphs so that the test cases related to those procedure call graphs can be assigned to the same cluster. TF-IDF output of the previous step, i.e. the vector representation of function call graph based on the tf-idf score, is considered as the input for clustering the regression test cases. So, the functions invoked in each test case is the key factor of test case clustering. By identifying similar test cases in terms of business functionality, most distinct test case selection is possible in the subsequent steps.

For this proposed test case selection and prioritisation methodology, K mean clustering algorithm is selected as the test case grouping approach using their tf-idf score of each test case as the input. To apply the K mean algorithm, the number of clusters (K) has to be predefined. To identify the optimal K value for the given input data, the Elbow method was used, and a separate python script was implemented for this calculation.

During this method, the within-cluster sum of squared distance was calculated for a range of K values starting from 1 to 40. This output was plotted into a graph where x-axis represents the cluster number, and the y-axis represents the within-cluster sum of squared distance value for each K.

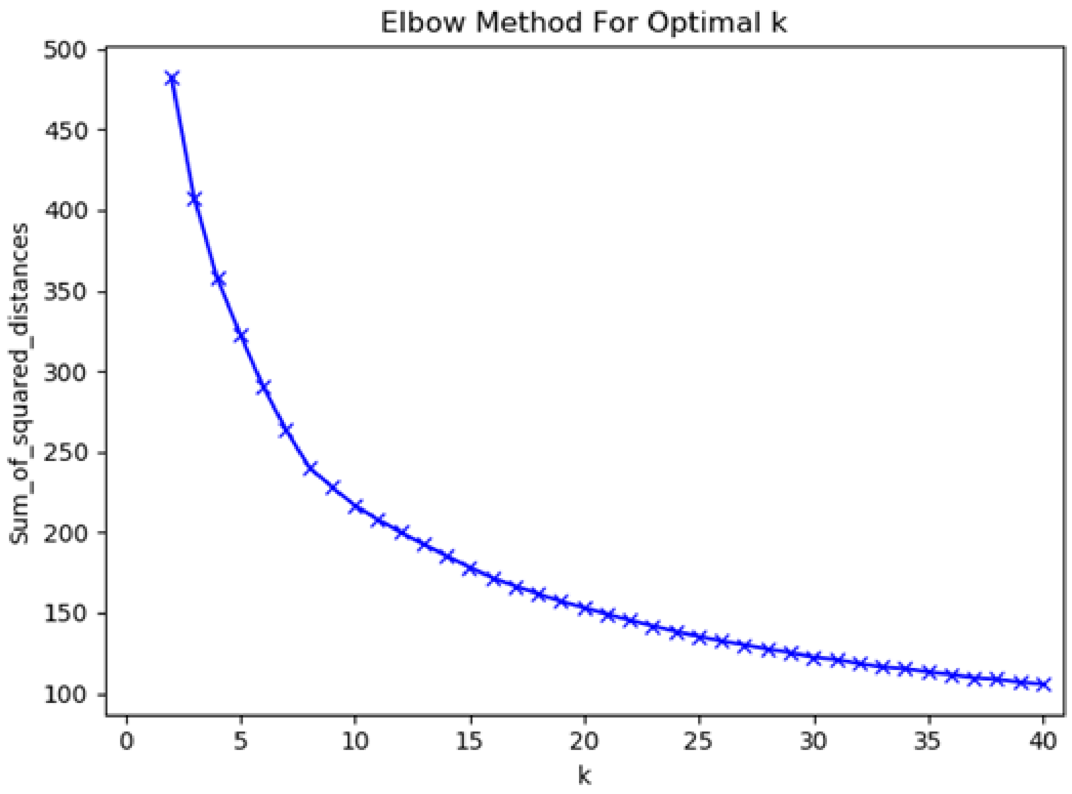


Figure 3.8: Elbow method output for different K

As per the figure 3.8, the bend (knee) of the graph can be recognized when the k value reaches 12. So the optimal cluster number (groups) for the given input test cases is chosen as 12, which is approximately equal to the number of distinct business functionalities of the corporate action subsystem.

The next step is to apply the k mean clustering on the tf-idf output of the test case procedure call graphs. Script has been updated to use the sklearn k mean python package to cluster the tf-idf output of each test case and persist it in the TEST_GROUP_ID column in the TESTCASE_EXECUTION_STAT table.

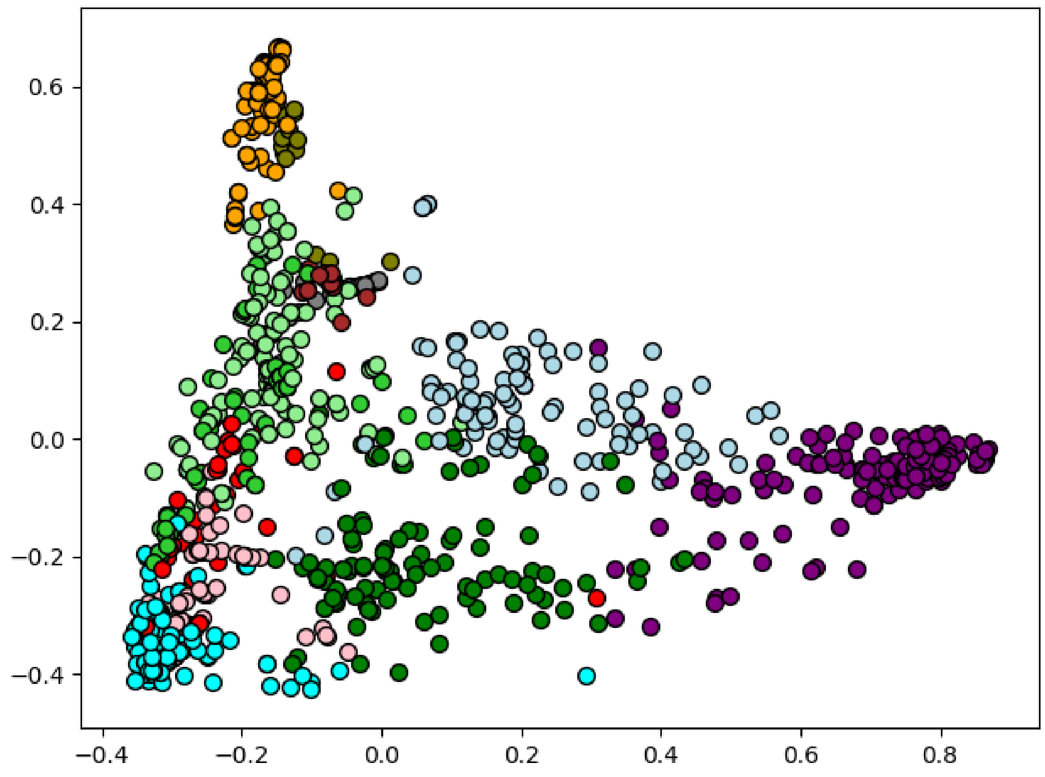


Figure 3.9: K-mean clustering on top of test cases - 2D graph

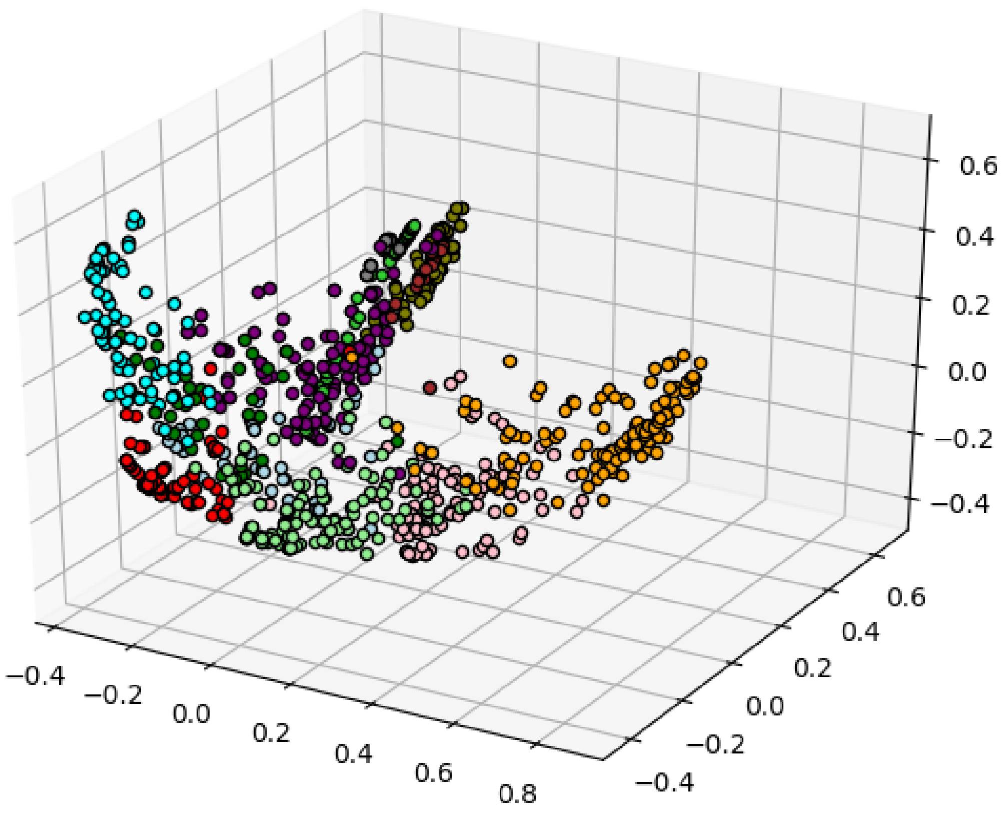
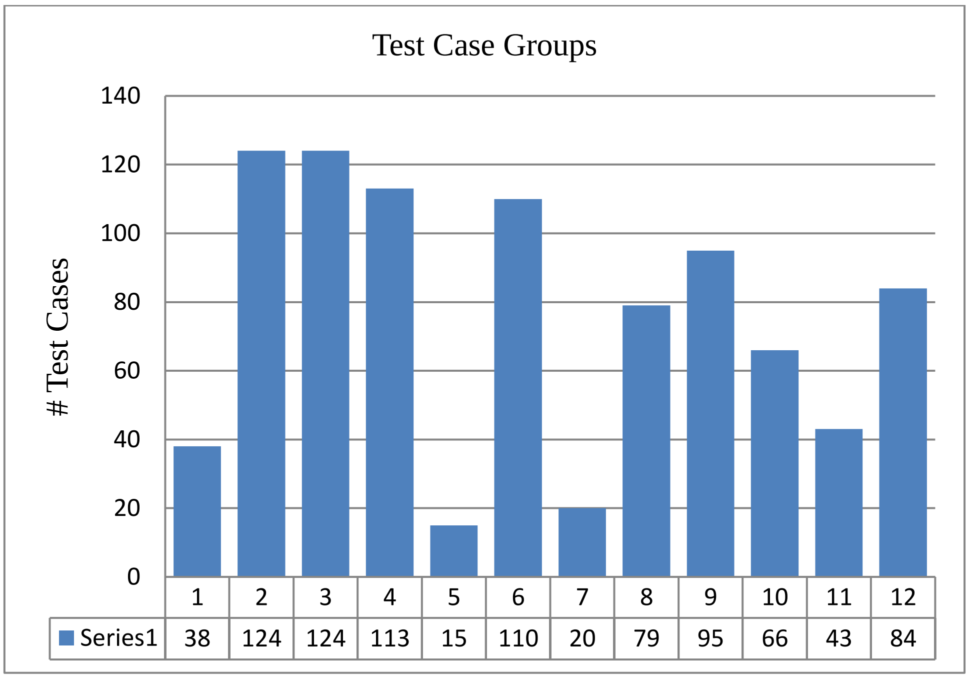


Figure 3.10: K-mean clustering on top of test cases - 3D graph

Total of 911 test cases is considered for clustering into 12 clusters (test groups). Following k mean cluster parameters were used during the implementation.

- K: Cluster count = 12
- n_init: Number of times the k-means algorithm will be run with different centroid seeds = 10000
- max_iter: Maximum number of iterations of the k-means algorithm for a single run = 100000



	1	2	3	4	5	6	7	8	9	10	11	12
■ Series1	38	124	124	113	15	110	20	79	95	66	43	84

Figure 3.11: Test cases selected for each test group

3.1.7.4 Test case selection and prioritization

This step defines the criteria for the regression test case selection and prioritisation based on the information gathered and calculated from previous steps. Initially, testing is carried outperforming the test case selection and prioritisation once and evaluate the results. While testing it is identified that the single level of test case selection does not provide a promising outcome. Hence it is decided to introduce a second level of test case selection and prioritisation for a more accurate output. Also, the second level of test selection is only performed for the test groups where the selected test cases did not fail from the first level of selection.

Following are the steps carried out during the first level of processing

- Identify and extract the changed code procedures

  In this step, procedures/functions which were updated during the bug-fix or feature development will be captured using the git commit change log. Once the developer creates the pull request of the change from bug-fix/feature branch to develop branch, the commits inside the pull request are considered for this activity.

  For the testing of the proposed method, a python script has been implemented to checkout to each commit and extracts the changed procedures by processing the change log while traversing upwards through the git tree of the existing code repository of the corporate action subsystem. Along with the source code git commit, the regression test repo git commit is also switched to get the corresponding test cases for the same code version.

- Select the test cases which evaluate the changed procedures

  Once the updated procedures were identified, the script will search for the impacted test cases which evaluate those procedures. The mapping between the test cases and the code procedures is available in the TESTCASE_COVERAGE_STAT DB table. So, this table is used to find the test cases which evaluate the changed procedures.

- Group the selected test cases into the pre-calculated groups

  This step will group the selected test cases based on the test group id generated based on procedure call graph using k mean clustering. The first level of test case grouping details is available in the table TESTCASE_ EXECUTION_STAT under TEST_GROUP_ID column.

  Per each group select the test case which has the highest coverage within the changed procedures and if there is a tie choose the test with highest total code coverage.

    - Coverage percentage of the changed procedure from the selected test case is considered as the primary selection criteria within the test group. This information is gathered and persisted under COVERAGE PERCENTAGE column of the TESTCASE_COVERAGE_STAT table. This will make sure that the selected test case will cover most of the functionality of the updated procedure in the event of code change.
    - Total test case coverage, i.e. total lines covered from the test case over the total line count of the corporate action source code, is considered as the secondary selection criteria for the test case selection within the same test group. Total test coverage per test case is available in the COVERAGE_PERCENTAGE of the TESTCASE_ EXECUTION_ STAT table. This is to select the test cases which evaluate the most complex business scenario with higher code coverage.
- Prioritised the selected reduced test suite based on the below features using weighted average score methods
    - Procedure coverage
    - Total line coverage of the system
    - Execution time
- Run the reduced and prioritised test cases and extract the test case result

After executing the test cases selected and prioritised from the above criteria, the test groups of the passed/completed test cases will be considered for the secondary level of processing. This will reattempt the test case selection and prioritisation task for the passed test groups individually to recognise another two test cases which could potentially fail during execution.

- Find the groups where the selected test case is passed in the first level processing

  Once the selected test cases were executed from the first level of processing, the passed test cases will be selected to identify their test group id. Those test groups will be considered as the passed test groups for the next step.

- Identify the test cases for secondary processing

  For each passed test group, select all the test cases which evaluate the changed procedure and remove the test case which was already executed in first level processing. So, the secondary level processing will only consider the newly selected test cases for each test group for the next step.

- Extract features for secondary level test cases

  As described in section 3.1.7.2, apply tf-idf information retrieval method to the selected test cases for the test group based on the procedure call graph of each test case. Since the selected test case sample for the secondary processing is very low compared to the primary processing, dynamically performing this clustering during the actual secondary test case selection is feasible.

- Cluster similar test cases within the primary test group

  Cluster the test cases within the group selected for the secondary processing using k-mean for the predefined cluster count. Considering the number of test cases selected from each primary test group, cluster k number for the secondary clustering is configured as 2 (k = 2) for testing. Rest of the steps are similar to section 3.1.7.3. So additional 2 test cases which are distinct from each other compared to the procedure call graph, will be selected from each passed test group from the secondary test case selection approach.

- Select a test case from each cluster within the group where the procedure coverage and total code coverage is higher.

- Prioritised the selected test cases based on below features using the weighted average score

  - Procedure coverage
  - Total line coverage of the system
  - Execution time

3.2 Evaluation results

The evaluation of the proposed regression test case selection and prioritisation methodology is carried out comparing the evaluation parameters defined in the section 2.3 with the following test approaches.

- Full regression test

  Complete test suite including 874 E2E test cases

- Smoke test

  Manually selected 15 E2E test cases from the CA test suite which cover all the primary functional paths of the CA subsystem

- Random test

  Randomly selected 25 E2E test cases from the CA test suite

The proposed methodology introduces a new test case categorisation method (test groups) based on the functional workflows (business functionalities) of the system under test. Hence the assessment of the evaluation parameters is more accurate and relevant when it is done on the test group-level results instead of the results of the individual test cases. In this section, the results of the evaluation parameters are presented in both approaches, i.e. using individual test cases as well as the results of the test groups, for more clarity.

3.2.1 Evaluation methodology

Following steps were carried out to gather the results of the test selection and prioritisation approach for the comparison.

1. Introduces a bug/defect into the system

   The first step is to select a procedure from the source code and plant a bug by changing the code of the procedure so that the expected result will not be met. This step was done manually by analysing the git commits for past procedure changes. When selecting procedures and introducing defect, following aspects were considered and tried to adhere as much as possible so that the actual performance of the proposed approach could be evaluated.

- Select the procedures which are not triggered by the main functional paths

  By avoiding the procedures/code segments used in the core business functional paths, failed test cases will be limited, and it will help to capture the true performance of the proposed test selection method, by evaluating the edge conditions and uncommon test scenarios. If the procedures in the core functional path were selected, most of the test cases will fail and will be hard to compare the performance of the proposed approach with the conventional methods. Because when only a small number of test cases fail, it is not easy to identify those test cases from a large pool of test cases in the regression test suite and capture the bug using the available test case selection methods.

- The planted defect should change the output of the system

  The introduced defect should change the output of the system, and it should fail at least one test case in the regression test suite.

2. Integrate the defect code into the system and start the system.
3. Run the full regression test suite and persist the results

   Execute the full regression test suite consists of 874 test cases and persist the results into a DB table named TESTCASE_RESULT_STAT. This table contains the output of each test case captured during the full regression test for each defect introduced.
4. Select the test cases to execute using the proposed method

   Execute the proposed regression test case selection and prioritisation tool by providing the changed procedures and obtain the test cases to execute. Since the proposed approach has two steps of selection, the second-based on the output of the initial selection, the status of each test case from the the full regression test were used as the results of the initial selection, instead of actually executing the test cases. Then evaluate the performance of all the selected test cases against the code change using the results in the full regression test.

5. Select the test cases used for the smoke test and compare the output of each test case using the full regression results for each defect.
6. Select test cases from random test selection techniques and compare the output of each test case using the full regression results for each defect.
7. Calculate the evaluation parameters for each test approach using the test case results persisted in the TESTCASE_RESULT_STAT table and evaluate the performance of each test case selection/prioritisation approach.

For the evaluation parameter calculation, selected test cases are considered based on the subset of test cases selected from each test selection approach, and relevant test cases are considered based on the failed test cases during the test case execution for each approach.

3.2.2 Test case reduction rate and efficiency

Test case reduction is one of the main reasons for promoting the regression test case selection so that the reduces test suite can be used to detect the defects of a code change as soon as possible. With higher reduction rate, test cases can be executed during code change integration which helps to identify the defect as early as possible.

As per the test results in table 3.5, the proposed approach has a 99% average reduction rate compared to the full regression test suite and average execution time is under 350s (~6mins) per code change.

Table 3.5: Test case reduction for different selection methods

Test #	Full Regression Test Count	Smoke Test Count	Proposed Approach Selected Test Count	Execution Time (s)	Reduction Rate (%)
1	874	15	15	699	98.28
2	874	15	4	80	99.54
3	874	15	2	93	99.77
4	874	15	5	97	99.43

5	874	15	4	217	99.54
6	874	15	21	1327	97.6
7	874	15	3	99	99.66
8	874	15	3	160	99.66
9	874	15	16	653	98.17
10	874	15	13	962	98.51

3.2.3 Precision

Precision is one of the leading performance parameters in the test case selection methodologies which can be calculated as the ratio of failed test cases of the selected test set (relevant test case) vs. selected test cases. Table 3.6 shows the failed test case count, total selected test case count and precision in percentage for the proposed test selection method as well as the three other traditional test selection methods, i.e. full regression, smoke test and random test selection, for a sample 10 code changes which have planted defects.

Table 3.6 presents the precision comparison based on the individual test case counts, and table 3.7 presents the precision comparison based on the test case groups which is calculated as per the proposed test case selection method.

The proposed approach is not intended to select all or the majority of failed test cases from the regression test suite since it is not efficient to execute multiple test cases which evaluate the same code path even though those test cases are failure cases for the particular code change. The rationale behind this approach is to cover the maximum possible distinct functional paths to identify more defects in the system. Even though this approach is not projected to have higher test selection precision, it is still better than the other three approaches due to its reduced test suite.

Table 3.6: Precision comparison based on the individual test cases

Test#	Full regression			Smoke Test			Random Test			Proposed Method		
	Selected Test Count	Failed Test Count	P%	Selected Test Count	Failed Test Count	P%	Selected Test Count	Failed Test Count	P%	Selected Test Count	Failed Test Count	P%
1	874	14	1.6	15	0	0	25	1	4	15	2	13.33
2	874	4	0.46	15	0	0	25	0	0	4	2	50
3	874	7	0.8	15	0	0	25	0	0	2	1	50
4	874	3	0.34	15	0	0	25	0	0	5	1	20
5	874	13	1.49	15	1	6.67	25	0	0	4	4	100
6	874	16	1.83	15	0	0	25	1	4	21	3	14.29
7	874	15	1.72	15	1	6.67	25	1	4	3	3	100
8	874	2	0.23	15	0	0	25	0	0	3	1	33.33
9	874	33	3.78	15	0	0	25	1	4	16	3	18.75
10	874	13	1.49	15	0	0	25	0	0	13	3	23.08

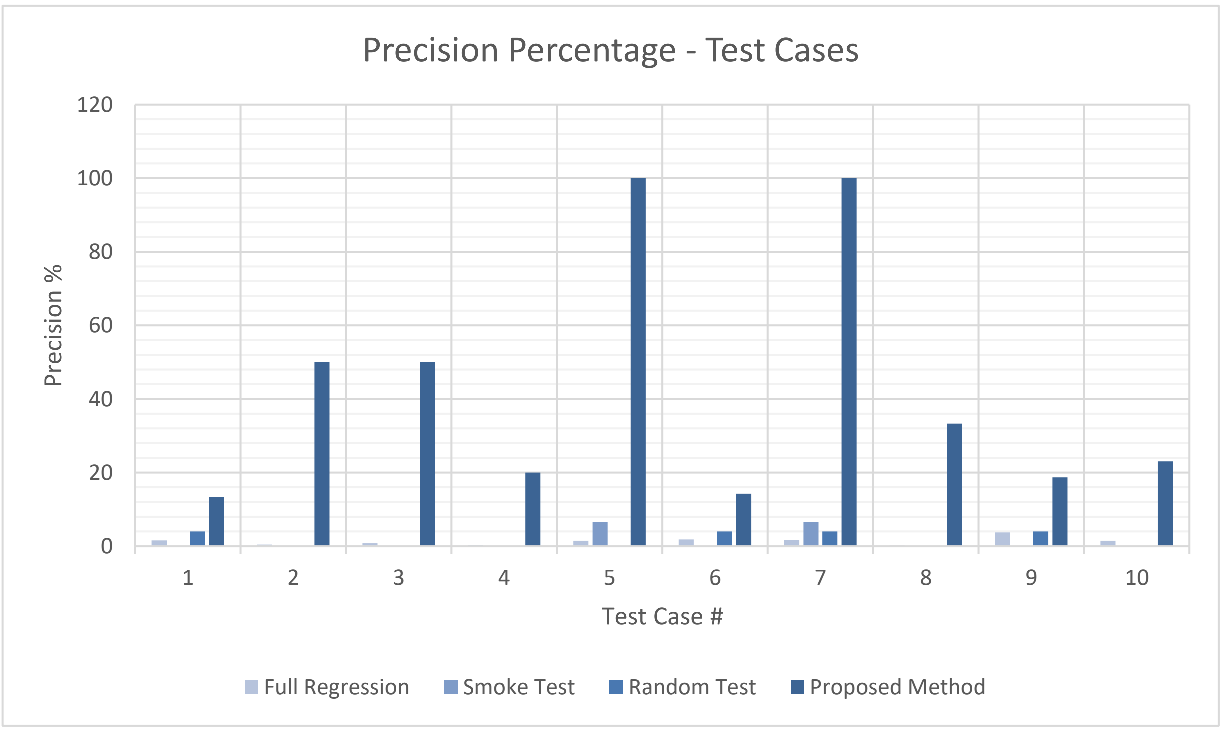


Figure 3.12: Test case selection precision

The precision of the test case selection can be calculated considering the selected test groups for execution and failed test groups from the selected groups for each test selection methods. As per the results, the test group precision of the proposed approach is higher than the traditional test selection methods. Even though all the test cases are not selected from the proposed method, all the failed test groups are selected for execution. Hence it is guaranteed to cover all the functional code paths which could have potential defects in the system.

Table 3.7: Precision comparison based on the test case groups

Test#	Full regression			Smoke Test		Random Test		Proposed Method	
	Selected Test Groups	Failed Test Groups	P%	Selected Test Groups	P%	Selected Test Groups	P%	Selected Test Groups	P%
1	ALL	4,7	16.66	0,1,2,3,5,6,10,11	0			1,2,3,4,5,7,9	28.57
2	ALL	2,5	16.66	0,1,2,3,5,6,10,11	25			2,3,5,9	50
3	ALL	10	8.33	0,1,2,3,5,6,10,11	12.5			6,10	50
4	ALL	2	8.33	0,1,2,3,5,6,10,11	12.5			2,3,5,9	25
5	ALL	0,2,3,8	33.33	0,1,2,3,5,6,10,11	37.5			0,2,3,8	100
6	ALL	1,3,5	25	0,1,2,3,5,6,10,11	37.5			0,1,2,3,4,5,8,9,11	33.33
7	ALL	0,3,11	25	0,1,2,3,5,6,10,11	37.5			0,3,11	100
8	ALL	10	8.33	0,1,2,3,5,6,10,11	12.5			10	100
9	ALL	1,3,5	25	0,1,2,3,5,6,10,11	37.5			0,1,3,4,5,8,9,11	37.5
10	ALL	1,2,3	25	0,1,2,3,5,6,10,11	37.5			1,2,3,5,6,7,11	42.86

3.2.4 Recall

The ratio between selected failed test cases and the total failed test cases in the regression test suite is calculated as the recall of the test selection method. As per the test results in table 3.8, full regression test provides 100% recall since it executes all the test cases in the test suite at the cost of ample execution time. Also, the recall rate of the proposed method has comparatively better performance than the smoke test, and random test approach regardless of the proposed method has no intention to capture multiple failed test cases within the same test groups.

Table 3.8: Recall comparison based on the individual test cases

Test#	Full regression			Smoke Test		Random Test		Proposed Method	
	Total Failed Test Count	Executed Failed Test Count	R%	Executed Failed Test Count	R%	Executed Failed Test Count	R%	Executed Failed Test Count	R%
1	14	14	100	0	0	1	7.14	2	14.29
2	4	4	100	0	0	0	0	2	50
3	7	7	100	0	0	0	0	1	14.29
4	3	3	100	0	0	0	0	1	33.33
5	13	13	100	1	7.69	0	0	4	30.77
6	16	16	100	0	0	1	6.25	3	18.75
7	15	15	100	1	6.67	1	6.67	3	20
8	2	2	100	0	0	0	0	1	50
9	33	33	100	0	0	1	3.03	3	9.09
10	13	13	100	0	0	0	0	3	23.08

According to the proposed test selection approach, more relevant recall values could be calculated considering the failed test group ids for the different test selection methods, as shown in table 3.9. As per the test results, the recall rate calculated based on the selected test groups, the proposed test selection approach has the 100% recall rate which is same as the performance of the full regression test execution.

Table 3.9: Recall comparison based on the test case groups

Test#	Full regression			Smoke Test		Random Test		Proposed Method	
	Total Failed Groups	Executed Failed Groups	R%	Executed Failed Groups	R%	Executed Failed Groups	R%	Executed Failed Groups	R%
1	4,7	4,7	100	-	0	7	50	4,7	100
2	2,5	2,5	100	-	0	-	0	2,5	100
3	10	10	100	-	0	-	0	10	100
4	2	2	100	-	0	-	0	2	100
5	0,2,3,8	0,2,3,8	100	0	25	-	0	0,2,3,8	100
6	1,3,5	1,3,5	100	-	0	5	33.33	1,3,5	100

7	0,3,11	0,3,11	100	0	33.33	3	33.33	0,3,11	100
8	10	10	100	-	0	-	0	10	100
9	1,3,5	1,3,5	100	-	0	1	33.33	1,3,5	100
10	1,2,3	1,2,3	100	-	0	-	0	1,2,3	100

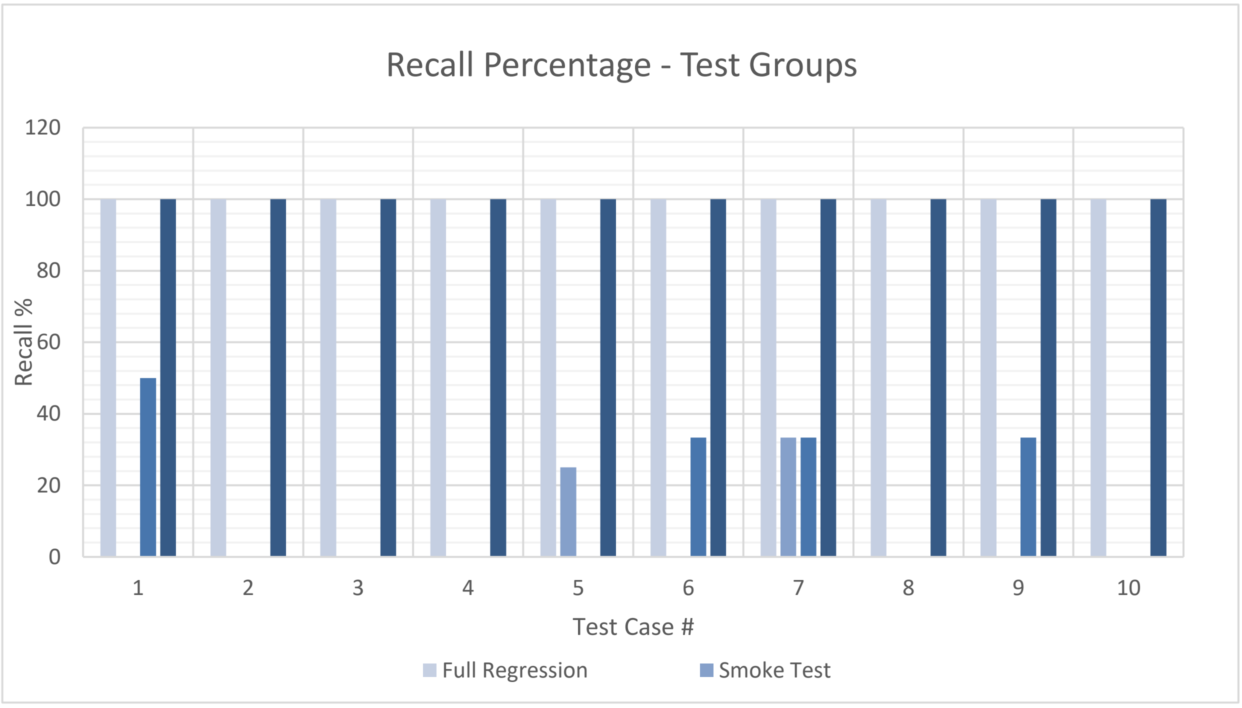


Figure 3.13: Test case selection recall rate

3.2.5 F – Measure

The trade-off between precision and recall value are harmonised by calculating the F-measure for the precision and recall values for each test scenario for different test selection methodologies. According to the results in table 3.10, the proposed test selection method secured the highest percentage value of F-measure compared to the other 3 traditional test selection methods.

Table 3.10: F-measures for different test selection methods

Test #	Full Regression (F%)	Smoke Test (F%)	Random Test (F%)	Proposed Method (F%)
1	3.15	-	7.41	23.52
2	0.92	-	-	66.67
3	1.59	-	-	66.67
4	0.68	-	-	33.33
5	2.94	10.53	-	100
6	3.59	-	7.14	25.01
7	3.38	11.12	7.14	100
8	0.46	-	-	50
9	7.28	-	7.14	31.58
10	2.94	-	-	37.50
Avg	2.69	2.16	2.88	53.43

3.2.6 Mutants killed

As per the test carried out on test selection methods, the defect detection capability of the proposed test selection approach is comparatively higher than the other test selection methods, obviously except for the full regression. In fact, for all the test scenarios considered during this evaluation, all the planted defect was identified from the proposed method, hence had 100% mutant killed rate.

Table 3.11: Mutants killed from each test selection methods

Test Selection Method	Total defects	Identified Defects	Mutants Killed %
Full regression Test	12	12	100
Smoke Test	12	2	16.67

Random Test	12	4	33.33
Proposed Test	12	12	100

3.2.7 Performance of the test selection levels

The proposed regression test selection and prioritisation method consist of two levels of test selection, where the second level of selection is done based on the output of the primary selection. As per the test results in table 3.12, over 82% of the identified defects are detected from the primary selection, whereas the rest was detected from the secondary selection. Hence it is vital to have both processing levels for the test selection in the proposed approach.

Table 3.12: Defect detection capability of proposed test selection levels

Test#	1	2	3	4	5	6	7	8	9	10	Total
Failed Test Groups - Level 1 Processing	2	2	1	1	4	2	3	0	2	2	19
Failed Test Groups - Level 2 Processing	0	0	0	0	0	1	0	1	1	1	4
Total Defects	2	2	1	1	4	3	3	1	3	3	23

3.2.8 Summary

When considering the evaluation parameters as mentioned in table 3.13, it is evident that the proposed regression test selection and prioritisation approach standout the other traditional test selection methods practised in the industry.

Table 3.13: Evaluation results summary for different test selections

Test Methods	Test Reduction %	Precision %	Recall %	F %	Mutants Killed %
Full regression Test	0	1.37	100	2.69	100

Smoke Test	98.28	1.33	5.83	2.16	16.67
Random Test	97.13	1.6	15	2.88	33.33
Proposed Test	99.01	42.28	100	53.43	100

All the test selection approaches expect regression test has excellent test case reduction capability where the proposed method has a slightly higher reduction of 99.01 % compared to smoke test and random test which have fixed number of test cases. When it comes to precision, the proposed method clearly defeats the other three approaches having over 42% precision rate. The recall rate of the proposed test selection method is calculated based on the selected test groups. As per the results, the proposed method also achieved the 100% recall rate as the full regression test, whereas the other two methods obtain lower recall performance. Same as recall the detected defects (mutants killed) percentage is 100% in both full regression and proposed method. Therefore, the proposed test selection method has the full regression performance in the aspect of recall and mutants killed and much higher performance in test reduction and precision compared to full regression testing.

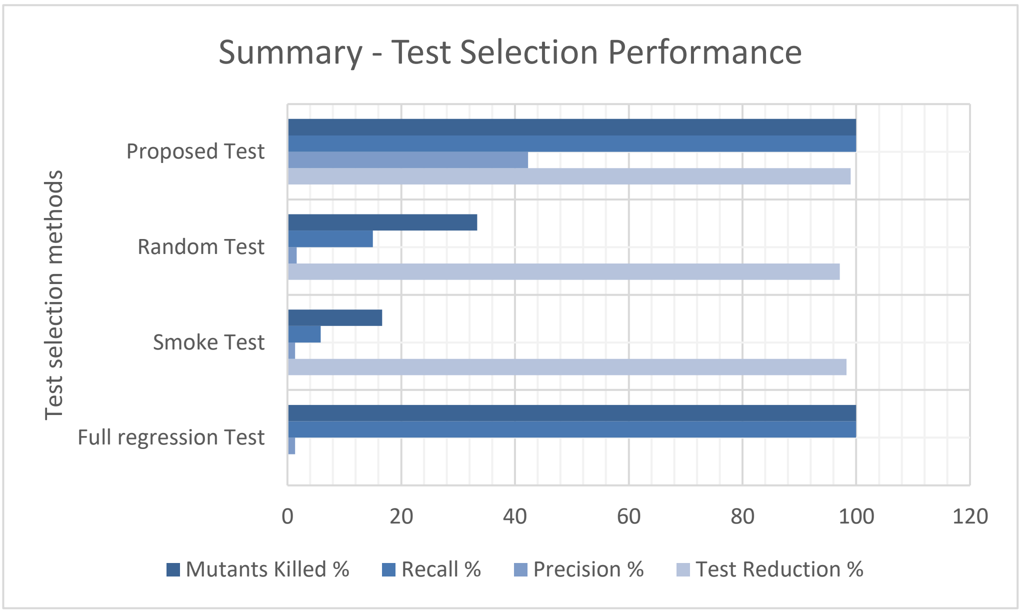


Figure 3.14: Evaluation results summary for different test selections

3.3 Conclusion

3.3.1 Outcomes of the research

As described in section 3.2, the proposed regression test case selection and prioritisation approach provide better overall performance compared to other traditional test cases selection methods. So, from this research, the following outcomes can be identified on efficient regression testing.

1. To achieve the code changes based test case selection, it is required to maintain a mapping between test cases and source code and granularity of this mapping is one of the research question addressed during testing. Considering the performance of the proposed test selection model, the procedure level test case to source code mapping provides better results with lesser processing complexity and higher statistic data maintainability.
2. Along with the development activities (CRs and bug fixes), gathered test cases statistic details get deviated from the actual values and it requires constant statistic gathering and maintenance. As per the evaluation carried out on the proposed method, once a week statistic update is sufficient, which can be scheduled to perform during the weekly full regression cycle as described in section 3.1.7. Hence the overhead of managing and updating test case statistic/coverage details is minimal.
3. Identifying and excluding the redundant test cases which evaluate the same functional implementation is one of the main objectives of this research. This is to achieve most effective reduced test suite for early defect detection. The procedure level call graph-based test case grouping and selection technique which described in this research can be recognized as a provable and feasible solution for this requirement. Also, the performance gain from this technique is comparably acceptable.
4. As per the test results and analysis carried out during the research implementation, the coverage of the changed procedure for the test case and the overall code coverage of the test case have been identified as the most

impactful features for change based test case selection and prioritisation. Also, the procedure level coverage has a higher impact than the overall code coverage for the defect detection capability of the test case. Statement level heat map of the test case can be considered as a very accurate measurement for the change based test selection. However, due to its complexity and high maintenance cost, it was decided not to use this feature for this research study. Further, the execution time of the test case has no significant impact on the proposed test selection and prioritisation approach since it is capable of achieving over 98% test case reduction.

5. According to the test results, the proposed method has 100% defect detection capability (given that the regression suite covers all the business functionalities) for test case reduction of over 98%. Hence the proposed approach which selects test cases based on the code change and the test case grouping and prioritises test cases based on its coverage details would be capable of capturing all the defects with minimum test case execution.
6. The proposed test selection and prioritisation method can be easily automated and integrated into the daily and weekly CI cycles, so that there will be no overhead for the developers. Test case statistic gathering, extract useful features and grouping functionality similar test cases can be integrated into the weekly CI cycle which can be executed during full regression testing. Test case selection, prioritisation and execution can be done during daily integration or when merging the code change into the release path.

3.3.2 Challenges and limitations

Regression test case selection and prioritisation using machine learning principles is a new area of research; hence there are limited literature on this topic. In this research study, several challenges and limitations were encountered as described below.

3.3.2.1 Challenges

1. Selecting a suitable system for implement and test the proposed test selection method

One of the main challenges of this research is to select a system to test the proposed regression selection method. As mentioned in section 3.1.6, a system with a well-maintained regression test suite which has higher source code and business functionality coverage is required to obtain the true performance of this research.

2. Identifying and excluding the flaky test cases

Identifying and excluding the flaky test cases exists in the regression test suite is another challenging task faced during research. Thirty-eight of such test cases has been identified after performing multiple rounds of testing by executing the full regression test suite. Hence rest of the test cases (874) were considered to be stable test cases which were used for the research study.

3. Selecting defects to evaluate the proposed method

To test the performance of the proposed test selection and prioritisation model, it is required to identify suitable code changes to the system which introduce defects into the system. Those defects should not fail the most of test cases in the regression suite; otherwise, the true performance of the proposed test selection approach will not be evaluated. So, selecting such defects is a challenging task which requires a thorough understanding of the source code as well as the business functionality

4. Running and analysing the full regression results

This is the most time-consuming activity during the evaluation. As per the regression test implementation when there is a failure in the test scrip it takes long time to complete because the script is waiting the system to provide expected results. Also, limitations in the test environment cause additional overhead to the research evaluation.

3.3.2.2 Research limitations

1. Limitations of available test case characteristics

During this study, we have considered only the test case characteristics which can be readily generated by executing the test case, such as functions invoked by test case, function coverage, total test case coverage and execution time. So, the characteristics which requires history data or export knowledge of the test scenarios were not considered for this evaluation. For example, past test case results, number of defects identified from the test case, severity of these defects, importance of the business scenarios evaluated by the test case and number of linked functional requirement for the test case.

2. Evaluation was limited to unsupervised ML techniques

Due to the unavailability of expert knowledge on test case scenarios and their relevant business requirements it is difficult to generate a labelled test data to train supervised learning model. Also, it will create an additional overhead to the current development practice, as it requires manual intervention for training. Hence this research study is focused on fully automated regression testing selection strategy and evaluate only the unsupervised ML methods which do not require labelled data for model training.

3. Function call sequence of a test case is not used for feature extraction

The test case feature extraction is done based on the importance of the functions invoked by a test case and it is not considering the function invoked sequence. The tf-idf text IR technique is employed for test case feature extraction and it gave us dependable results to proceed with clustering. Hence requirement of employing function invoked sequence didn’t arise for the selected test suite.

4. Changes to the code could invalidate the test case details

When the changes are carried out on the existing code due to the bug fixes or CRs, the procedure call graph could change for the impacted test cases. This will invalidate the gathered test case statistic as well as the test case grouping. Hence the test case selection could not be accurate as expected. This is a limitation of the proposed approach which can be minimised by frequent statistic gathering. Also, it is not expected to have drastic changes to the code, which could change the procedure call graphs of a test case for the projects in the maintenance phase.

3.3.3 Research assumptions

Following are the key research assumptions considered during the analysis.

1. Regression test case should evaluate the end to end functionality of the test scenarios including any ripple effects to the system.
2. Test cases should be independent on each other and execution of one test case should not cause any impact on the subsequent test case execution.
3. It is assumed that by selecting test cases which have highest line coverage of the modified function and highest total code coverage per each test group i.e. each business function, will evaluate updated code change in the function by following the most complex business execution path.
4. It is expected not to have frequent changes which invalidate the gathered test case details, specially the functions invoked by the test case, during the maintenance periods of a product.
5. Also, it is assumed that the regression test suite covers the maximum possible code coverage considering all the positive and negative test scenarios.
6. If changes to the test suite is required, those should be done before the code changes are implemented and committed into the system.

3.3.4 Future work

Employing machine learning principles to enhance the code change based regression test case selection and prioritisation method is an emerging research area which has the potential to improve the efficiency of agile development as well as the quality of the output product. Hence, this research is of much importance and this can be enhanced further as suggested below.

1. The past test results of the test cases can be gathered and incorporated into the selection algorithm because test cases with a higher failure rate have a high chance of detecting defects of the updated code.
2. Employ various information retrieval algorithms to extract features of the procedure call graphs, especially the order of the called procedures, and compare the performance of these IR methods to find most suitable IR algorithm.

3. Study on various clustering methods to group the test cases which evaluate the similar functional area and compare the performance of each clustering method to identify the most suitable clustering algorithm for this proposed regression test case selection approach.
4. In this research, the test cases were selected based on the procedure level coverage of the test cases. However, the statement level test case to source code mapping provides more accurate information for the test selection with the cost of complex and more volatile mapping logic. As future work, it is vital to analyse the feasible method of gathering and maintaining the statement level test case mapping against the code.
5. It is proposed to implement this test case selection method as a pluggable module to integrate into the system's version control system or CI system.

REFERENCE LIST

[1] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran, “Regression Test Selection Techniques: A Survey,” 2010.

[2] Atlassian, “What is Continuous Integration,” Atlassian. [Online]. Available: https://www.atlassian.com/continuous-delivery/continuous-integration. [Accessed: 17-Oct-2018].

[3] Atlassian, “Gitflow Workflow: Atlassian Git Tutorial,” Atlassian. [Online]. Available: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow. [Accessed: 17-Oct-2018].

[4] R. Kazmi, D. Jawawi, R. Mohamad, I. Ghani, and M. Younas, “A Test Case Selection Framework and Technique: Weighted Average Scoring Method,” Nov. 2017

[5] Sanjoy_62Check out this Author's contributed articles., sanjoy_62, and Check out this Author's contributed articles., “Software Engineering: Software Maintenance,” GeeksforGeeks, 11-Oct-2018. [Online]. Available: https://www.geeksforgeeks.org/software-engineering-software-maintenance/. [Accessed: 27-Dec-2019].

[6] S. Yoo and M. Harman, “Regression testing minimization, selection and prioritization: a survey,” Software Testing, Verification and Reliability, 2010.

[7] E. D. Ekelund and E. Engstrom, “Efficient regression testing based on test history: An industrial evaluation,” 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015.

[8] S. Puri, A. Singhal, and A. Bansal, “Study and Analysis of Regression Test Case Selection Techniques,” International Journal of Computer Applications, vol. 101, no. 3, pp. 45–50, 2014.

[9] J. Kasurinen, O. Taipale, and K. Smolander, “Test Case Selection and Prioritization,” Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM 10, 2010.

[10] A. Lawanna, “A Model for Test Case Selection in the Software- Development Life Cycle,” 2013.

[11] M. K. Suppriya and A. K. Ilavarasi, “Test Case Selection and Prioritization Using Multiple Criteria,” International Journal of Advanced Research in Computer Science and Software Engineering, 2015.

[12] Lachmann, “Machine Learning-Driven Test Case Prioritization Approaches for Black-Box Software Testing,” The European Test and Telemetry Conference, 2018.

[13] K. Jammalamadaka, and V. Ramakrishna, “Test Case Selection Using Logistic Regression Prediction Model,” International Journal of Mechanical Engineering and Technology, 2017.

[14] Y. Pang, X. Xue, and A. S. Namin, “A Clustering-Based Test Case Classification Technique for Enhancing Regression Testing,” Journal of Software, vol. 12, no. 4, pp. 153–164, 2017.

[15] K. Dhanadevan, J. Nallasamy, and S. Murugavel, “Neural Network Based Regression Test Selection,” Sep. 2017.

[16] A. A., M. Akour, I. Alazzam, and F. Hanandeh, “Regression Test-Selection Technique Using Component Model Based Modification: Code to Test Traceability,” International Journal of Advanced Computer Science and Applications, vol. 7, no. 4, 2016.

[17] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, “Reinforcement learning for automatic test case prioritization and selection in continuous integration,” Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2017, 2017.

[18] A. Beszedes, T. Gergely, L. Schrettner, J. Jasz, L. Lango, and T. Gyimothy, “Code coverage-based regression test selection and prioritization in WebKit,” 2012 28th IEEE International Conference on Software Maintenance (ICSM), 2012.

[19] Q. D. Soetens, S. Demeyer, and A. Zaidman, “Change-Based Test Selection in the Presence of Developer Tests,” 2013 17th European Conference on Software Maintenance and Reengineering, 2013.

[20] K. Godfrey, Kassambara, J. Romero, Kassambara, V. Kumar, Kassambara, and M. Cassiano, “Determining The Optimal Number Of Clusters: 3 Must Know Methods,” Datanovia. [Online]. Available: https://www.datanovia.com/en/lessons/determining-the-optimal-number-of-clusters-3-must-know-methods/. [Accessed: 27-Dec-2019].

[21] Z. Sultan, R. Abbas, S. Nazir, and S. Asim, “Analytical Review on Test Cases Prioritization Techniques: An Empirical Study,” International Journal of Advanced Computer Science and Applications, vol. 8, no. 2, 2017.

[22] “Test Prioritization or Test Case Prioritization,” ProfessionalQA.com. [Online]. Available: http://www.professionalqa.com/test-prioritization. [Accessed: 27-Dec-2019].

[23] I. Salian, “NVIDIA Blog: Supervised Vs. Unsupervised Learning,” The Official NVIDIA Blog, 20-Aug-2019. [Online]. Available: https://blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/. [Accessed: 23-Feb-2019].

[24] “idf :: A Single-Page Tutorial - Information Retrieval and Text Mining,” Tf. [Online]. Available: http://www.tfidf.com/. [Accessed: 27-Sep-2019].

[25] Kasurinen, J., Taipale, O. and Smolander, K., 2010. Test Case Selection and Prioritization: Risk-Based or Design-Based?

[26] Tibshirani, R., Walther, G. and Hastie, T., 2001. Estimating the number of clusters in a data set via the gap statistic. Journal of the Royal Statistical Society: Series B (Statistical Methodology), 63(2), pp.411-423.

[27] Pravin, A. and Srinivasan, S., 2013. Effective test case selection and prioritization in regression testing. Journal of Computer Science, 9(5), pp.654-659.