# Frequent Segment Clustering of Test cases for Test Suite Reduction

NARENDRA KUMAR RAO.B[1], Dr.A.RAMAMOHAN REDDY[2],
[1]Research Scholar, Dept. of CSE. [2]Professor,Dept. of CSE.
[2]JNTU Hyderabad [2]SV University College of Engineering.
[1]Hyderabad,India [2]Tirupati,India
[1]narendrakumarrao@yahoo.com. [2]ramamohansvu@yahoo.com

*Abstract:* - Execution profiles are indicators for code coverage of program; this has been demonstrated by researchers on a large scale through their contributions on the same. Test Suite reduction is a feature which achieves code coverage with minimum number of test cases ensuring that all code items have been tested. It is a Non-deterministic Polynomial-time Complete (NP-Complete) problem. Few approaches like Greedy approach, Harrold,Soffa and Gupta(HGS) approach have been used in literature which are good approaches. Current work achieves similar milestones with reduced test cases as well. This paper presents Maximal frequency item set clustering and sequencing of similar test cases, residue code requirements based test case reduction and modification based test case selection. In the current work, few interesting results were found, where in similar program trace test cases were greatly reduced ensuring high code coverage percentage during testing. Fault detection can be tuned by selecting test cases from similar groups.

*Key-Words:* - Clustering, Test Suite Reduction, Test case coverage, Program Profiles, Test case Selection, Regression Testing

## 1 Introduction

Test case is a pair of input data and corresponding program output. Its intent is to find issues in the code and to check whether the code meets its requirements or not. This is commonly known by two terms, verification and validation.

Observation based testing relies on the input of data for a test case and observing the output obtained along with the execution behavior of the program under study. These results are then analyzed through automation and a result test set is generated to ensure the conformance with product requirements. The purpose of automated analysis is to reduce the manual effort in reducing/filtering the number of test cases (subset) to be executed in the code. The reducing/filtering process needs to be inexpensive when compared to executing the entire set of test cases.

The goal of such an approach is to increase insight into modification based fault removal. Filtering process performed on the profiles of program are also known by the name Cluster analysis. Cluster analysis is a multivariate analysis for grouping of objects which have been categorized by attribute values. Cluster analysis group objects with similar attribute values in a cluster, while objects with dissimilar attribute values are placed in different clusters. Similarity or dissimilarity is measured in terms of metric called distance like Euclidean distance. After clustering phase, it moves ahead to test suite reduction.

Test suite is a collection of test cases. It comprises of a large number of test cases that can test various requirements. In this scenario it is possible that redundant test cases are generated, because test cases are generated on basis of requirements for system testing. Also an effective strategy is to assign minimal number of test cases to attain maximum code coverage at a point of time during testing without compromising maximum defect detection.

Test Suite reduction is the process of reducing test cases from the test suite. Two test cases are redundant if they are verifying same program structures with same intent producing same output. It is quite effective to remove such redundant test cases from test suite. This reduces the efforts for testing, all these are to be performed at the cost of maximal code coverage. If an effective code coverage criteria chosen for the test suite reduction process then testing process can reveal more bugs.

"The first formal definition of test suite reduction problem introduced in 1993 by Harrold et al. is as follows: Given. {t1, t2,…, tm} is test suite T from m test cases and {r1, r2,…, rn} is set of test requirements that must be satisfied in order to provide desirable coverage of the program entities and each subsets {T1, T2,…, Tn} from T are related to one of ris such that each test case tj belonging to Ti satisfies ri.

**Problem**. Find minimal test suite T' from T which satisfies all ri s covered by original suite T."

In current paper the program profiles are taken as criteria for test suite reduction, also in further sections it is applied for regression testing. This is not a new criterion as such from literature and many other authors have effectively used program profiles for the same.

Testing activities occur after subsequent code changes. Regression testing usually refers to testing activities during software maintenance phase. The major objective of regression testing is retest the changed components and check the affected parts. Regression testing can be done at different levels like unit level, function level and code level. Software change information (change notes), updated software requirement and design specifications, and user manuals form the basis for regression testing. Formal definition for regression testing is as follows: Regression Testing refers to a portion of test cycle in which program P' is tested to ensure that not only does newly added or modified code behaves correctly but also code carried over from version P continues to behave correctly.

In Current paper the discussion on background requirements is presented in section 2, Literature survey in brief is presented in section 3, proposed system is introduced with important procedures in section 4, Experimentation and results are presented in section 5.Conclusion and future work is presented in last section before references.

## 2 Background

### 2.1 Program Profiling

Program profiling refers to the observation of behaviour of program under consideration. Profiling is performed by the usage of tool called "Profiler". Generally, profilers are used during runtime of program to collect data relevant to program like events, function calls, values held by variables which reflects the behavior of the system. Profilers are classified as Flat, Call graph, Input sensitive profilers. Flat profilers compute the call times but do not depend on the context of program for the computation. Call chains, full profiles of the programs are stored in Call graph profiles. Performance measurement of programs based on varying inputs, workloads can also be performed using Input sensitive profilers.

Based on the data granularity in profilers, they can be classified as Event based profilers, Statistical profilers and Instrumentation based profilers. All event based languages directly support event-based profilers like JVMTI (JVM Tools Interface) API in Java. .NET supports direct attachment of profiling objects in COM terminology, hotshot profiling module is readily available in python. Statistical or Sampling profilers can directly interact with Program counters, Operating systems level for profiling programs under consideration without much burden or effect on the program under observation. CodeXL from AMD, Oprofile for Linux, Intel VTune are few profilers of this type.

### 2.2 Call Stacks

Call stack is the collection of active function calls during program execution. It is also generally defined as sequence of functions which are ordered represented by set C such that it comprises of elements $f_i$ where i corresponds to the various functions executed during execution. Every stack trace starts with a function $f_1$, from there on $f_1$ allows invocation of functions $f_2$ such that $f_i$ always invokes $f_{i+1}$. A call stack is recursive if it contains recursive calls to itself.

### 2.3 Call stacks and Test cases

Execution of a test case produces a sequence of calls , which represents behaviour of the program for a given test case. Call trace for a given test case convey more semantic sense compared to functions, because they represent the context of the program under execution.

Our work assumes the availability of program execution trace before start of the test suite reduction process, with fair degree of accuracy. A Programmer obtains this information by attaching call profilers to programs under test to capture the call stacks for every test case. Primarily this module comprises of two steps like recording the traces of programs under execution as a first step and second step involves pre-processing the obtained information offline.

### 2.4 Requirements Traceability

Requirements traceability maps system requirements to test cases to be used in the system testing phase. This feature can be used in risk analysis phase, requirements analysis and specification phase, design analysis and specification phase, source code analysis, unit testing & integration testing phase, validation – system testing, functional testing phase.

Multi dimensional memory format is widely used in industry for mapping system requirements to test case identifiers during system testing.

## 2.5 Clustering

Clustering is grouping of objects with similar properties or attributes without considering the outcome of it, but can sure be achieved something of the same. They are grouped only based on the attributes of the items under consideration in the universal set.

Clustering algorithms are classified among three types which are partition clustering, density based clustering and hierarchical clustering. The basic need for clustering in current work is for ease of testing, fault identification & isolation, grouping like test cases and grouping like coverage items for testing.

## 3 Literature Survey

Coverage is the extent that a structure has been exercised as a percentage of the items being covered. If coverage is not 100%, then more tests may be designed to test those items that were missed and therefore, increase coverage [5]. Test coverage helps in monitoring the quality of testing, and assists in directing the test generators to create test cases that cover areas of a program that have not been tested before. Coverage criteria survey is included in the following [7, 11, 4, 13, 8, 2, 10].

Ammons et al. first proposed the concept of calling tree during run time of the program [1]. Later on Bond and McKinley proposed a probabilistic approach for the same. Prior to this work McMaster at al.[12] has used this approach for coverage based test suite reduction and obtained good results on a small scale. Our current work focuses on this point and moves forward to understand program behaviour from clustered test cases. Supporting work can be found in [12, 18, 9, 16, 4, 6, 3, 19, 17,20].

## 4 Proposed System

The model in Fig.1 proposes a Conceptual system design for accomplishing the test suite reduction and selection. Modules work in coordinated fashion for effective performance of the system.

*Test Suite*

Test suite comprises complete suite of the test cases for the system. They can be designed from view point of system requirements, code coverage etc.. In current work, the test cases are developed from requirements.

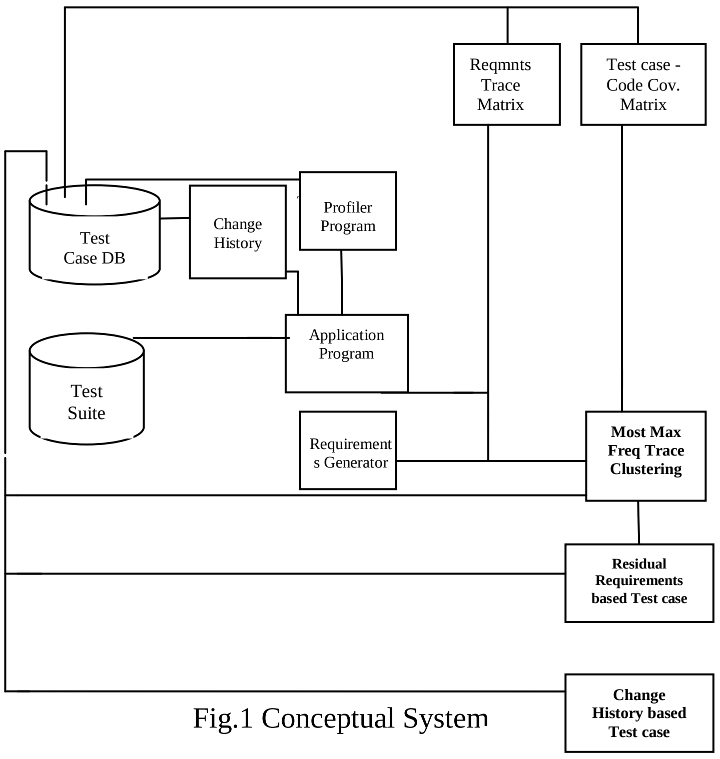
Reqmnts Trace Matrix
Test case - Code Cov. Matrix
Test Case DB
Change History
Profiler Program
Application Program
Test Suite
Requirements Generator
Most Max Freq Trace Clustering
Residual Requirements based Test case
Change History based Test case

Fig.1 Conceptual System

*Test case Database*

The intention of the test case database is to store the following and retrieve them when required. Revision history stores the list of methods added, deleted and modified in each revision, Source code changes history per file at level of functions, Test case traceability matrix, program trace data for each test case and indices of functions of code. Comment and description type changes are discarded manually.

*Change history*

This module comprises of two major programs one which codeDiffs two files of source code and produces a report in HTML format which is further analyzed by another program to generate or fill lists which are of the forms like addM, storing the list of newly added code, delM storing the list of deleted methods and modM storing the listed of modified methods in a give file.

*Test Case-Code Coverage matrix*

Test cases produce execution sequence and based on the results obtained, the tester decides whether it has produced a successful or failure run. All successful runs are mapped to a matrix where rows represent the test cases executed per revision and columns represent the methods or indices corresponding to methods used in selected system requirements. All corresponding columns are mapped to value i if the given method is encountered in $i_{th}$ position for a given test case execution. The given value of i increments in 1 for every successful mapping of function or method for the given test case. This proceeds until all methods for a given test case are mapped and all test cases are hence mapped.

*Requirements generator*

This is M *X* N format where in more than one test case may be used in a given requirement, same test case can be used across many requirements and likely repeated test cases also could have been induced through different requirements.

*Profiler program*

Profiler program records the sequence of calls executed by a program during its execution.

*Application Program*

The source for testing is taken from standard test suite SIR repository for purpose of testing, which induces a bug per each revision. Hence test cases on these versions can be used in the verification of programs.

*Most Maximal Frequent Trace Clustering*

Test cases are redundant in many cases since they are designed keeping requirements in view. Requirements are mapped to test cases through traceability matrix, which identify the test cases required for testing.

In this paper a clustering algorithm based on frequent program code coverage items and traces is proposed which groups most frequent traces of coverage items into a group among the test cases in the given suite. The clustering process continues until all the code coverage items are clustered in form of test cases as described in the algorithm next.

*Sequence Clustering of test cases based on frequent item segments*

In Most Maximal Frequent Trace Clustering algorithm (MMFTC), the coverage items are only clustered, but it does not, produce the desired effect in reduction and selection process. In such a case test case sequence are to be clustered such that they are sorted sequentially based on similarity scores of sequences of coverage items.

*Redundant test case Elimination*

The check for redundant test case is through identification of similar score sequences with in a cluster and check for length, if length and strings match in program traces, then the test cases are marked as redundant.

*Residual Requirements based Test Suite Reduction* (RRBC)

The current algorithm is more focused on selecting residual code coverage requirements, where in the algorithm selects those test cases which have high potential of code coverage ie., covering more unselected methods rather than selecting test cases which are of maximal length as in other approaches (Greedy and HGS).This algorithm fares better with HGS approach and similar to Greedy approach (better in few peculiar near similar test case sequences).

*Change history based test case selection*

Change history is made available to the test case selection process in the lists thereby ensuring that all relevant test cases are selected for testing. The change history comprise of details in form of lists like added methods, modified methods and deleted methods.

## 4.1 Data Structures

Following are essential data structures for implementation of Most Maximal Frequency trace based clustering -Residue requirements based test suite reduction (MMFTC-RRBC):

Requirements traceability matrix- $RTC_{ij.}$

Test Case-Code Coverage matrix- $RT_{ij.}$

Code index visited after selecting a test case- $V_{i.}$

Counting frequency of occurrence of a item in a given Test Case- Code Coverage matrix- $freqCount_{k.}$

coverage items of code base- $covItem_{k.}$

Storing the test case id for every cluster- $RTS_{ij.}$

Minimal test cases satisfying code coverage- $RTS_{min,i.}$

Representation for change history list for a given version.

```
struct ChHist
{
modM_i; - Modified Items
addM _i; - Added Items
delM_i; - Deleted Items
};
```

Safe Test cases after reduction- $R_{safe}(i)(j)$.

Safe Minimum test cases per version of code

$R_{safe}=RTS_{min}\ U\ RTSA_{min}\ U\ RTSD_{min}\ U\ RTSM_{min}$

## 4.2 Procedures

```
Algorithm-1
   MaxFreqTraceClustering($freqCount_i$, $RT_{i,j}$)
   Process:
1  For all ($x_i$ and $y_k$ ∧ $visited_l$!=1) or until all items
   are visited or or until there are coverage item
   clusters less than 2(TWO).
   // $x_i$ and $y_k$ pair of code items
   a  Assign $x_i$,$y_k$ with max($FreqCount_m$),
      max($FreqCount_{m-1}$) from RTCij
   b  Populate test cases in $RTS_{temp}$ such that
         $RTS_{temp}=\{(xi,yk)/xi,yk \in RT \wedge$
         $F(x_i \cap y_k)=\max\{\bigcap_{i=k=0}^{n-1,\,n-2} F(x_i,y_k)\}\}$
   Where $F(x_i)$represents set of test cases traversing code item $x_i$. Repeat above steps such that $F((x∩y)_i∩ y_k)=F(x∩y)_i ∩ F(y_k)$.
c  Repeat F(x∩y)i, yk)=F(x∩y)i ∩ F(yk).
d  $RTS_n = RTS_n \cup RTS_{temp}$
   $curFreq^{(n)}_j=\{k / k∈ F(x_i∩y_k)!= Ǿ ∧ n∈ RTS^{(n)}\}$
   $visited_l=C\{C/C=1, if F(x_i∩y_k)!= Ǿ;C=0\}$
e  $RT_i^{(j)} = RT_i^{(j)}-RTS_n$ // Eliminate the test case from further clustering process.
f  n++;repeart using step-1.
Output:
$RTS_n$ contains n cluster of test cases with each cluster containing test cases of a particular coverage item(s) common in all the given clustered test cases. Every cluster is grouped based on common coverage item with a given item count.
```

```
Algorithm-2
   RRTestReduce($RTS_i$, $covItem_k$)
   Process
1  In a cluster $RTS_i$, the unselected maximal length test case from RTS and mark all $covItem_k$ for the set of methods the particular test case has covered.
   $t_k=x \{x,t_k/\forall t_k \ x=max(length(t_k))∧ t_k ∈RTS_i)\}$
2  Mark all the $covItem_k$ as visited and select the next maximal length test case from next maximal count cluster and compute the residue of coverage requirements, the test case with maximal residue count is selected as the target test case.
   $t_j=\{x_j/\forall x_j \ max(covItem(x_j))∧ x_j ∈RTS^{(i)}_j)\}$
3  Repeat the above steps by considering one test case at a time from each cluster RTS to cover all the items of $covItem_k$.
   $R_{Safe} = R_{Safe} \cup t_j$ ; iff $\{t_j/\forall t_j∈RTS_i\}$
   Output:
   Selects the maximum requirements coverage test case into $R_{safe}$. There by ensuring that maximal coverage items are selected from minimal number of test cases from different clusters.
```

```
Algorithm-3
   SeqClust($RTS_{ij}$)
   Process:
1  Repeat for all n until all clusters are done
   a  Computing the Position Weight Matrix (PWM)
   b  Computing similarity scores of sequences based on scores in PWM(SimScore).
   c  Sequencing the test cases based on similarity Scores(SortSeq).
a  PWM($RT_{ij}$)
   i    For frequent items of given cluster, compute the Position weight matrix for all sequences of cluster.
   ii   This involves computing frequency of an item at a given position, represented by M(x, P), where function M represents the frequency of x at position P.
   iii  M(x, P)= n(x, p)/N, where n(x, p) denote the number of occurrences of x at position p in the set of sequences, N is the number of maximum items in sequence and update positional scores in $RTPs_{ij}$
b  SimScore($RT_{ij}$ ,$RTPs_{ij}$)
   I    For each item of the sequence, substitute the corresponding positional score from PWM for the item.
   ii   Compute the position weight score for all other non frequent items (elements of sequence which are not part of current frequent item set of cluster as ZERO.
   iii  Score (M,S) = $\Sigma^{w}_{p=1}M(p, S[p])$ where sequence S of length w, P is the position in sequence and update score in $tScore_k$.
c  SortSeq($RT_{ij}$, $tScore_k$,$RTS_n$)
   i    Align the test cases in the order of similarity of test cases based on Score (M,S) in decreasing order of weights stored already in $tScore_k$
   Output
      Sorted and sequenced Test cases in $RTS_i$.
```

```
Algorithm-4
   RedntEliminate($RTS_{ij}$)
   Process
1  For all the clusters do the following
   a  For all test cases in $RTS_i$ do
   b  If(SimScore($t_j$)==SimScore($t_k$))
   c  If (Length($t_j$)==Length($t_k$))
   d  If(CompareString($t_j$,$t_k$))=0 then test case $t_j$ is redundant and eliminate it from RT.
   e  If $RTS_i=\{t_k\}$, append it to $RTS_n$ where n∈|RTS|-1.//(Singleton set)
      Output:
      Redundant test cases elimination in $RTS_{ij}$
```

```
Algorithm-5
   CHTestSelect($RTS_{min}$, Struct ChHist)
   Process
1    1. Select the type of modification list as A-added, M-Modified and D-Deleted methods.
2  If the list type is 'M' then perform the following
   a  For all the elements from the list modM[],
    which contains the index of the column in
    the requirement test case matrix $RT_{ij}$ do the
    following:
  b Select the corresponding rows k(test cases)
    which are marked non zero and store in $R_{Safe.}$

    $C(k) = \{k /(k \in RT_i^{(j)}) \wedge (\forall_i RT_i^{(j)} != NULL) \wedge \{j,q / j \in \forall q\ modM[q] = 1 \wedge q \in [0..m-1]\}\} \wedge i \in [0..n-1]\}$
    $R_{Safe} = RTS_{min} \cup \{\forall_k C(k)\}$
  c Repeat steps a and b for all elements of
    modM[].
3 If the list type is 'D' then perform the following
  a For all the elements from the list delM[],
    which contains the index of the column in
    the requirement test case matrix $RT_{ij}$ do the
    following:
  b Select the corresponding rows $t_j$(test cases)
    which are marked non zero and store in $R_{Safe.}$

    $D(k) = \{k /(k \in RT_i^{(j)}) \wedge (\forall_i RT_i^{(j)} != NULL) \wedge \{j,q / j \in \forall q\ delM[q] = 1 \wedge q \in [0..m-1]\}\} \wedge i \in [0..n-1]\}$
    $R_{Safe} = RTS_{min} \cup \{\forall_k D(k)\}$
  c From the $RTS_{ij}$ eliminate the corresponding
    column selected from delM[].
    $RT^{(i)}_j = RT^{(i)}_j - q$; where $q \in$ delM[]
  d Repeat until all elements of delM[] are
    covered
4 If the list type is 'A' then perform the following.
  a For all the elements from the list
    addM[],which contains the index of the
    column in the requirement test case matrix
    $RT_{ij}$ do the following:
  b Select the corresponding rows k(test cases)
    which are marked non zero and store in $R_{Safe.}$
    $A(k) = \{k /(k \in RT_i^{(j)}) \wedge (\forall_i RT_i^{(j)} != NULL) \wedge \{j,q / j \in \forall q\ addM[q] = 1 \wedge q \in [0..m-1]\}\} \wedge i \in [0..n-1]\}$
    $R_{Safe} = RTS_{min} \cup \{\forall_k A(k)\}$
  c Add the new column to current version of
    RT.
    $RT^{(i)}_j = RT^{(i)}_j \cup q$ where $q \in$ addM[] $\wedge$
    j= n-1;n+=1
  d Repeat steps a and b for all elements of
    addM[].
    Output:
    Manipulated $RT_{ij}$ and selected test cases in
    $RTS_{min}$ based on change history of test cases.
```

Working model is demonstrated in Appendix-1.

## 5 Experiments and Analysis

Current work compares the effectiveness of call stack based reduction based on function level granularity. This also compares the fault detection ability of proposed work with random reduced test suites for its evaluation.

### 5.1 Subject Application & Metrics

SIR repository based *space* program was used in this work as program data. It has a test pool of nearly 1400 test cases and 38 versions of the same program containing faults. Code Tune was used in instrumentation of call stacks and call coverage tree for the work. It generates reports in excel, which requires programs further to analyze it and populate the program profile. Current work traces based on only the function name but not based on its complete signature. Out of 136 odd functions of space program, test case trace generated to selective test cases covering set of functions. The code is well analyzed before testing and test cases set to target given functions. Nearly 38 versions are maintained for the same program, such that each program differs from the other with a single failure. The experiment was conducted by seeding multiple failures between two versions and then clusters the test cases by taking into consideration its previous program trace information.

During course of trace recording, recursive programs are not taken repetitively, but considered once for its multiple runs. Many iterative calls were skipped to single calls following the modified sequitur algorithm [14] for program profile representation. Library functions invoked during the test case runs were eliminated, as we felt they may not influence on our work.

Fault detection effectiveness is one corresponding measure that is being used in current work. Percentage of code requirements coverage, percentage of size reduction and percentage fault detection reduction has been calculated over the entire experiment to measure the effectiveness of the proposed method over random experiments.

$$R_{COV(\%)} = \frac{|R_{cov}|}{|R_{tot}|} \times 100 \qquad (1)$$

$R_{tot}$-represents the total number of test consideration requirements and $R_{cov}$ is the total number of requirements satisfied by test cases selected during reduction for equation-1.

$$TS_{(\%\ Red)} = 1 - \frac{|T_{rs}|}{|T|} X100 \qquad (2)$$

|T|- Total number of test cases in the original suite and $|T_{rs}|$ represents test cases in representative set for equation-2.

$$FD_{(\%\ Red)} = 1 - \frac{|FD_{Reduced}|}{|FD_{Full}|} X\ 100 \qquad (3)$$

Percentage reduction in fault detection $FD_{(\%\ Red)}$, and other formulae in (fig-3)$FD_{Reduced}$ represents Defect detection from reduced suite, $FD_{Full}$ is complete suite for equation-3.

There were few hundred and above functions involved in program run. The possible call stacks were recorded for given suite. It is possible to generate a test case from suite and record its call trace. In different versions of program only those test cases that failed due to the induced defect in the version and relevant test cases alone failed, all other succeeded. Hence it is not necessary to test entire set of test cases for every version. The result of clustering algorithm was sufficient for identifying whether defective test case was included or not.

## 5.2 Method of Experimentation

The defects can be detected from the suite in two ways:

1. Select a given number of test cases by applying the test case selection approach and detect the fault detection effectiveness.
2. Select a given 'K' number of cases where it depends on algorithm which selects the number of test cases and identify the number of faults detected by each k test cases. Repeat this until all faults are detected.

No Change history information is taken into consideration during this process, so that comparison is with similar set up i.e. random approach. In our case, second approach was used where test cases were selected in units of K, where test cases were obtained from random MMFTC.

*Random based Fault detection*

1. Test suite of specified number of test cases are selected.
2. Test the application with given 'K' tests and record faults.
3. Repeat until all the faults are detected.

*MMFTC-RRBC based Fault detection*

1. Form the reduced set of Test coverage satisfying test cases by using MMFTC. Repeat the process by selecting 'K' unselected test cases from the clusters such that minimal of K/n cases are selected from each cluster, where n- is the number of clusters. In clusters where there are less than k/n test cases select the residue test cases from other clusters.
2. Form the set of faults detected by the reduced suite, such that faults are recorded after every iteration until all faults are identified.

The number of test cases used in each run of proposed algorithm was incremented in value of 50, starting from minimal value of 50 to 500.This was performed to detect the fault detection capability of suite. All the procedures were implemented in C language on Windows operating system with a Pentium dual core processor 2.0 GHz. It was observed that as number of test cases increased, the program execution time increased as shown in graph (Fig-2).

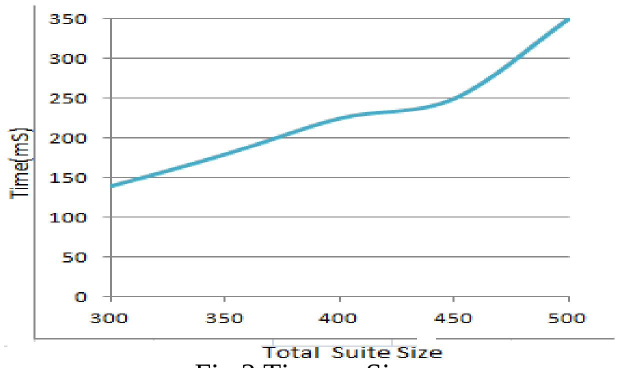

Fig.2 Time vs Size

The experiment procedure specified above section is performed with 'K' number of test cases selected by random approach and by means of MMFTC and following results were obtained. The proposed approach is definitely scaling better in terms of identifying faults by selecting test cases from clusters that had relatively more number of faults (Fig-3). Current approach is compared with other approaches like HGS and Greedy approach in Appendix-2. Appendix-3 depicts performance compared with Greedy approach in graph.

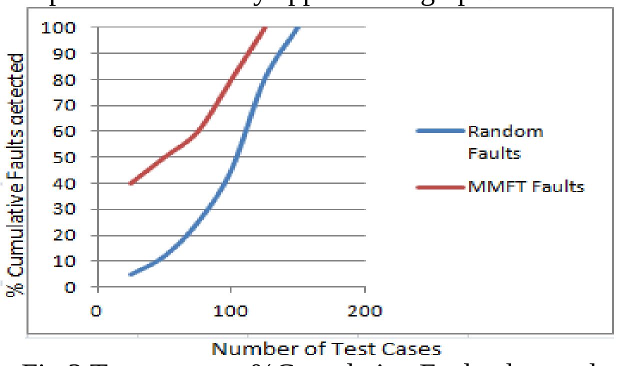

Fig.3 Test cases vs %Cumulative Faults detected

In sample runs it was also observed that as the number of test cases increased the percentage of fault detection effectiveness decreased gradually as large test case chunks (size K) were induced into testing in both the experiments of random and MMFTC. This was for the reason that similarity sequenced test cases were selected during iterations from clusters for detecting faults.

Sufficient benefit is acquired when this technique uses the sequencing of similar test cases, which enhances detection of co-associated or change impact faults which span similar test cases having similar call sequence.

Table 1. Coverage Criteria Clustering

| **Means over Test Suites** | | | | | |
|---|---|---|---|---|---|
| **Original Suite** | | **Random reduced** | | **MMFTC-RRBC reduced** | |
| **CASE-I** | | | | | |
| **Size** | **Faults Detected** | **Size** | **Faults Detected** | **Size** | **Faults Detected** |
| 500 | 34 | 60 | 24 | 60 | **28** |
| **% Reduction from Original** | | **88** | **30** | **88** | **17** |
| **CASE-II** | | | | | |
| 500 | 34 | **80** | **34** | **72** | **34** |
| **% Reduction from Original** | | **84** | **0** | **85.6** | **0** |

Above results (table-1) were obtained with total suite, random suite and a MMFTC reduced suite. We can infer that this work can considerably decrease test suite size and improve faults detected, compared to random suites. Experimentation can be classified as two cases (Case-I & II).

*Case-I* represents a scenario where in K test cases are chosen randomly from the suite based on random experimentation and MMFTC-RRBC. The result under case-I depicts scenario where in there is a considerable decrease in suite size and effective defect detection.

*Case-II* depicts scenario where in defect revealing test cases are chosen by reducing test cases from more defective clusters, i.e. selecting test cases from similar clusters which relatively reveal more defects during testing. This is in-line with software myth, where more defects tend to concentrate in relative locations of defects.

## 5.3 Change history based test case selection

Change history inclusion in test case selection is a better approach, since the code which underwent code changes were populated in separate lists and test cases corresponding to those changes are included into testing process as introduced in section-4.

In above experimentation section (4), change history module computes the codeDiff between the code modules and generates a program report on code differences between the two versions. A program analyzer module (as discussed above) reduces them to a change history list. The change history list is capable of inducing test cases which are affected by the corresponding code changes.

In current experimentation set up for defect detection, code version ($V_i$) was made diff with earlier version ($V_{i-1}$) and recorded the changes in change lists accordingly, it recorded the changes and incorporated the test case for the corresponding change.

This method was effective in incorporating not only the defect rendering changes, but also many test cases, which were introduced as part of code changes, which did not introduce any defect.

The Change history module comprises of two sub functionalities such as codeDiff and records changes at coarse level like added function, modified function, deleted function from the diff engine and populates them into lists corresponding to addM, delM, modM. These lists are further used in analysis for test case selection.

The results were suggestive of the fact that all test cases for changes were introduced, which is capable of introducing defects (irrespective of type of changes taken into consideration). Comment type changes and code formatting changes are not at all included, but had to be removed manually. Future study can incorporate such changes as well. The above approach was used for test suite reduction and apart from them, test cases which are not redundant, but are change introduced test cases are selected for testing.

Following observations were made on percentage of reduction in size of suite and reduction in percentage of defect detection capability.

Table 2. Percentage Reduced for random vs. MMFTC-RRBC based TEST SUITE

| **Means over Test Suites** | | | |
|---|---|---|---|
| **Original Suite** | | **MMFTC-RRBC + change Induced** | |
| **Size** | **Faults Detected** | **Size** | **Faults Detected** |
| 500 | 34+4* | 70 | 28+4* |
| **% Reduction from Original** | | 86 | 15.7 |

The above table-2 demonstrates the fact that all change inducing test cases are incorporated during test case selection, there by capable of detecting defects for all scenarios (given fact that all test cases are present in test suite and incorporated in requirements traceability during test plan).The actual defect detection was based on MMFTC and change induced test case selection. In above table-2, * indicates the faults due to program induced failures. The change induced test cases were able to select and hence detect the change induced failures. This experiment also produces similar results on comparison with Table-I case-II, when more test cases are chosen from fault revealing clusters to achieve 100% defect detection. Future work is in these directions to select context sensitive test cases revealing faults and improve precision.

During change based test case selection, analysis of regression test selection is performed, in which we consider three important categories as described by Rothermal, which are Inclusiveness, Precision and Efficiency.

Inclusiveness refers to the extent to which the current test case selection technique is capable of capturing the modification revealing test cases from original suite T into new suite. It is defined in [15] as “Suppose T contains n tests that are modification revealing for P and P’ and suppose M(test selection mechanism) selects m of these tests. Then inclusiveness of M relative to P, P’ and T is 100 * (m/n).”Accordingly this method is able to select all the modification revealing test cases, given a modification entity.

Hence, this achieves m/n ratio as 1, which ensures that technique is safe. If a given approach of test selection is more inclusive, then it has the potential to expose faults, which is hypothesized in current results as in [15].

Precision refers to the extent to which M omits tests that are non-modification revealing. It is defined again in [15] as “Suppose T contains n tests that are non-modification revealing for P and P’ and suppose M omits m of these tests. The precision relative to P, P’ and T is 100 * (m/n).”

The precision indicates the omission of non-modification revealing test cases, but our technique selects all possible test cases traversing the method, irrespective of whether the test case is traversing the modified portion of the code, this ensures that more test cases are selected and none of the test cases are omitted, which is an indication of Safe regression technique.

Efficiency is measured in terms of space and time requirements. As per time constraint, it should be economical than retest-all approach, such that cost selection should be less than cost of running tests in T-T’. Space efficiency represents the test history and program analysis information the technique must store and access.

During regression testing, the test case selection approach should perform well during both preliminary phase, where initial version is released and there is sufficient duration for locating and fixing issues and Critical phase where regression testing is crucial and needs to reduce the time and cost for testing [15].But this is likely to perform better in former case rather than later case, but from quality perspective is a good choice at all times.

Table 3. Reduced results for Random vs. MMFTC-RRBC based TEST SUITE

| **Original Suite Size** | **Induced Faults** | **Inclusiveness** | **Precision** | **Efficiency** |
|---|---|---|---|---|
| **CASE-I** | | | | |
| 500 | 4 | 100% | 70% | *O*(\|T\|*\|P\|) |
| **CASE-II** | | | | |
| 500 | 4 | 100% | 55% | O(\|T\|*\|P\|) |

|T|-represents size of suite selected using the current technique and |P|-represents size of program in terms of functions. These results in the above table-3 are validated with work by Rothermal et al., proposed in the modified entity form to be a safe technique.

The experimentation follows section 5.2 in which similar approach as case-I, case-II and the given precision was observed. There was considerable improvement in reduction of non-modification revealing test cases (related to precision).

Few benefits of approach to practitioners include the following:

- Faults were detected early compared to random suites.
- Early fault detection leads to early fixing of issues.
- Relatively less testing effort in terms of number of test cases and testing cost reduction.

## 5.4 Threats to validity

A program run may produce different program traces for successful run. A given test case producing distinct program traces at different times based on environment and variation in input but produce similar outputs are not included, which is a potential threat for the current work and its validity.

The granularity adopted in this work is a method, hence statement level defects cannot be detected until unless all test cases are introduced during test selection process.

Test suites used in current work are designed complete with respect to requirements, any other criteria based test suites have not been tested with this approach.

## 6 Conclusion

Even though much required inclusiveness and precision is achieved in this work, this can only be achieved at higher cost, this can be further improved by prioritizing the test cases before selection.

Change history based test case selection is effective process in revealing faults in the program, except that all modification traversing test cases are selected. Hence they can be prioritized for selection step, such that higher priority is assigned to test cases with more number of methods/functions and precedence of priority for various changes like deletion, addition and modification of methods.

The current work is more effective in terms of identifying defects in modification traversing and revealing test cases rather than change effect impacted methods.

Intention further is to take up this work in GUI development and testing for mobile environment, where traces are unique sequences and some of methods are common or used as libraries, basic user interfaces, event handlers and other reusable forms. This will be a perfect match for this work.

Fine grained coverage items can be used in this study. The effectiveness in terms of clustering, defect detection effectiveness are broad areas of further study.

Improvement in intelligent clustering methods can attribute to better results. Program behavior and context sensitive information are two important criteria that affect test suite reduction.

*References:*

[1] Ammons G and James R, “Improving data-flow analysis with path profiles.”,SIGPLAN Not. 39, 4 (April 2004), 568-582.

[2] [Angeletti, D., E. Giunchiglia,“Automatic Test Generation for Coverage Analysis of ERTMS Software”, In the International Conference on Software Testing Verification and Validation, 2009. ICST '09.

[3] Arafeen, M.J.; Hyunsook Do, "Test Case Prioritization Using Requirements-Based Clustering", Software Testing, Verification and Validation, 2013 IEEE Sixth International Conference on , vol., no., pp.312,321, 18-22 March 2013.

[4] Arvind K.," Pair-Wise Time-Aware Test Case Prioritization for Regression Testing ", Information Systems, Technology and Management Communications in Computer and Information Science, vol., no.285, pp.176,186, 2012.

[5] Beizer, B. Software Testing Techniques, Second Edition, International Thompson Computer Press, Boston, 1990.

[6] Carlson, R.; Hyunsook Do; Denton, A., "A clustering approach to improving test case prioritization: An industrial case study", Software Maintenance (ICSM), 2011 27th IEEE International Conference on , vol., no., pp.382,391, 25-30 Sept. 2011.

[7] Eugenia Díaz, J. T., Raquel B (2004). "A Modular Tool for Automated Coverage in Software Testing.", Software Engineering Notes. Vol. 26, no.5, pp. 256-267. Sept. 2001.

[8] Kapfhammer G.M., Soffa, M.L.(2008) “Database-Aware Test Coverage Monitoring”, Proceedings of the Ist India Software engineering conference ISEC’08, Hyderabad, India, ACM.77-86."

[9] Khalilian A and Saeed P , “Bi-criteria test suite reduction by cluster analysis of execution profiles.”, In Proceedings of the 4th IFIP TC 2 Central and East European conference on Advances in Software Engineering Techniques (CEE-SET'09), Springer-Verlag, Berlin, Heidelberg, 243-256.

[10] Koochakzadeh V,"An Empirical Evaluation to Study Benefits of Visual versus Textual Test Coverage Information.", in the fifth conference on The Testing: Academic and Industrial Conference, Practice and Research Techniques (TAIC PART), 2010.

[11] Lormans M D. "Reconstructing Requirements Coverage Views from Design and Test using Traceability Recovery via LSI”, TEFSE, USA, ACM 2005.

[12] McMaster S, Memon, A.M., "Call-Stack Coverage for GUI Test Suite reduction", Software Engineering, IEEE Transactions on , vol.34, no.1, pp.99,115, Jan.-Feb. 2008.

[13] Lingampally R, Gupta A,Jalote P "A Multipurpose Code Coverage Tool for Java", In Proceedings of the 40th Annual Hawaii International Conference on System Sciences, IEEE Computer Society, 261b, 2007.

[14] Reiss, Steven P., and Manos R. "Encoding program executions. "proceedings of the 23rd International Conference on Software Engineering. IEEE Computer Society, 2001.

[15] Rothermel, G.; Harrold, M.J., "Analyzing regression test selection techniques," Software Engineering, IEEE Transactions on , vol.22, no.8, pp.529,551, Aug 1996.

[16] Shin Yoo, Mark Harman, Paolo T, and Angelo Susi,”Clustering test cases to achieve effective and scalable prioritisation incorporating expert

knowledge.", XVIII international symposium on Software testing and analysis (ISSTA '09). ACM, New York, NY, USA, 201-212.

[17] Vipindeep V, Jacek Czerwonka, and Phani Talluri. "Test case comparison and clustering using program profiles and static execution", In Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering ,ACM, New York, NY, USA, 293-294,2009.

[18] William Dickinson, David Leon, "Finding Failures by Cluster Analysis of Execution Profiles", Intl. Conf. on Software Engineering, Toronto, Candada, May 2001, pp. 339-348. Alireza Khalilian and Saeed Parsa. 2009.

[19] Yi Miao, Zhenyu Chen, Sihan Li, Zhihong Zhao,and Yuming Zhou. "Identifying Coincidental Correctness for Fault Localization by Clustering Test Cases", Software Engineering and Knowledge Engineering, 267-272, 2012.

[20] T Lertphumpanya, T Senivongse, "Basis Path Test Suite and Testing Process for WS-BPEL", WSEAS TRANSACTIONS on COMPUTERS, Vol 7, Issue 5, pp.483-496, May 2008.

## Appendix-1
## Working model

R={$r_1,r_2,r_3,\ldots.r_m$} ➔ {a,b,c,d,e,f,g,h,i,j}
Ts = {$t_1,t_2,t_3,\ldots t_{11}$}

Table 4.Requirements vs Test case Table

$t_1$→abfg
$t_2$→aegi
$t_3$→abcdj
$t_4$→afbcd
$t_5$→afghi
$t_6$→abfg
$t_7$→acdhi
$t_8$→bcij
t9→cdgh
$t_{10}$→ijab
$t_{11}$→abcd

| | **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** |
|---|---|---|---|---|---|---|---|---|---|---|
| **t1** | 1 | 2 | | | | 3 | 4 | | | |
| **t2** | 1 | | | | 2 | | 3 | | 4 | |
| **t3** | 1 | 2 | 3 | 4 | | | | | | 5 |
| **t4** | 1 | 3 | 4 | 5 | | 2 | | | | |
| **t5** | 1 | | | | | 2 | 3 | 4 | 5 | |
| **t6** | 1 | 2 | | | | 3 | 4 | | | |
| **t7** | 1 | | 2 | 3 | | | | 4 | 5 | |
| **t8** | | 1 | 2 | | | | | | 3 | 4 |
| **t9** | | | 1 | 2 | | | 3 | 4 | | |
| **t10** | 3 | 4 | | | | | | | 1 | 2 |
| **t11** | 1 | 2 | 3 | 4 | | | | | | |
| F | **9** | **7** | 6 | 5 | 1 | 4 | 5 | 3 | 5 | 3 |

***1. Clustering Most Maximal Frequent Items***
Maximal Frequent items for R is a and b.
F(a) ∩ F(b) ➔ {$t_1,t_2,t_3,t_4,t_5,t_6,t_7,t_{10},t_{11}$} ∩ {$t_1,t_3,t_4,t_6,t_8,t_{10},t_{11}$}
⇨ {$t_1,t_3,t_4,t_6,t_{10},t_{11}$}
F(a ∩ b) ∩ F(c) ➔ {$t_1,t_3,t_4,t_6,t_{10},t_{11}$} ∩ {$t_3,t_4,t_7,t_8,t_9,t_{11}$}
$RTS_{i\ min}$ ➔ {$t_3,t_4,t_{11}$}
F(a ∩ b ∩ c)∩ F(g) ➔ Ǿ; F(a ∩ b ∩ c ) ∩ F(i) ➔ Ǿ
F(a ∩ b ∩ c ) ∩ F(d) ➔ {$t_3,t_4,t_{11}$} ∩ {$t_3,t_4,t_7,t_9,t_{11}$}

**$RTS_i$ ➔ F(a ∩ b ∩ c ∩ d) ➔ {$t_3,t_4,t_{11}$}**

Insert {a,b,c,d} into visited[],select unvisited maximal elements and again repeat clustering next most maximal frequent items
F(a ∩ b ∩ c ∩ d) ∩ F(f) ➔ Ǿ;
F(a ∩ b ∩ c ∩ d) ∩ F(h) ➔ Ǿ
F(a ∩ b ∩ c ∩ d) ∩ F(j) ➔ Ǿ
F(a ∩ b ∩ c ∩ d) ∩ F(e) ➔ Ǿ
Repeat above with other high frequent functions until no more elements to spare for clustering by selecting unvisited maximal elements and proceed.
F(g) ∩ F(i) ➔ {t2,t5}
F(g ∩ i) ∩ F(h) ➔ Ǿ;F(g ∩ i) ∩ F(j) ➔ Ǿ
F(g ∩ i) ∩ F(e) ➔ Ǿ ;$RTS_{i+1\ min}$ ➔{$t_2,t_5$}
Cluster1 = {$t_3,t_4$};Cluster2 ={$t_2,t_5$}

Clustering process can eliminate test cases which were previously selected on basis of frequent items, until all the test cases are grouped under clusters. After all the iterations clusters are classified and arranged as follows:

*2. Sequence Clustering of test cases based on frequent Coverage items (SeqClust)*

Table 5.Requirements vs Test case Table

$t_1$→abfg
$t_2$→aegi
$t_3$→adcbj
$t_4$→afbcd
$t_5$→afghi
$t_6$→abfg
$t_7$→acdhi
$t_8$→bcij
$t_9$→cdgh
$t_{10}$→ijab
$t_{11}$→abcd

| | **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** |
|---|---|---|---|---|---|---|---|---|---|---|
| ~~**t1**~~ | ~~1~~ | ~~2~~ | - | - | - | ~~3~~ | ~~4~~ | - | - | - |
| **t2** | 1 | | | | 2 | | 3 | | 4 | |
| **t3** | 1 | 2 | 3 | 4 | | | | | | 5 |
| **$t_4$** | 1 | 3 | 4 | 5 | | 2 | | | | |
| **$t_5$** | 1 | | | | | 2 | 3 | 4 | 5 | |
| **$t_6$** | 1 | 2 | | | | 3 | 4 | | | |
| **$t_7$** | 1 | | 2 | 3 | | | | 4 | 5 | |
| **$t_8$** | | 1 | 2 | | | | | | 3 | 4 |
| **$t_9$** | | | 1 | 2 | | | 3 | 4 | | |
| **$t_{10}$** | 3 | 4 | | | | | | | 1 | 2 |
| **$t_{11}$** | 1 | 2 | 3 | 4 | | | | | | |
| F | 9 | 7 | 6 | 5 | 1 | 4 | 5 | 3 | 5 | 3 |

**$RTS_{i\ min}$ ➔ F(a ∩ b ∩ c ∩ d) ➔ {$t_3,t_4,t_{11}$}**

*Step-1 (PWM)*
$t_3$→adcbj
$t_4$→afbcd
$t_{11}$→abcd

| | I | II | III | IV | V |
|---|---|---|---|---|---|
| a | 1 | 0 | 0 | 0 | 0 |
| b | 0 | 1/5 | 1/5 | 1/5 | 0 |
| c | 0 | 0 | 2/5 | 1/5 | 0 |
| d | 0 | 1/5 | 0 | 1/5 | 1/5 |

*Step-2 (Score(M,S))*
Score($t_3$) = 1+1/5+2/5+1/5+0 = 9/5
Score($t_4$) = 1+0+1/5+1/5+1/5 = 8/5
Score($t_{11}$) = 1+1/5+2/5+1/5+0 = 9/5
*Step-3 (Aligning the sequences)*
$RTS_{min}$ = {$t_3$, $t_{11}$, $t_4$}

Test cases of all clusters are sequenced and sorted as in above case based on sequence similarity. In case of cluster F(a∩b∩f∩g) with frequency count 4 falls under a group and their similarity score is same for (t1,t6}.It can be observed that $t_1$ is redundant and hence eliminated by the algorithm.

Table 6.clustered test cases

| #Cluster | Cluster-Id | Test case(s) | Count of frequent Items |
|---|---|---|---|
| 1 | $RTS_0$ | $\{t_3, t_{11}, t_4\}$ | 4 |
| 2 | $RTS_1$ | $\{t_2,t_5\}$ | 2 |
| 3 | $RTS_2$ | $\{t_7,t_9\}$ | 3 |
| 4 | $RTS_3$ | $\{t_6,t_8,t_{10}\}$ | - |

It can be observed that t1 is redundant and hence eliminated by the algorithm. Still clusters with count item frequency 1 and 0 are available respectively they are discarded based on criteria max cluster count frequency > 2, which produces clusters but not sufficient enough to understand program behavior. *Adaptive Sampling* (residual code requirement based algorithm) is used for reducing test cases from clusters.

$$\mathbf{RTS_{min} = \{t_3, t_2, t_5\}}$$

Insert {a, b, c, d, e, f, g, h, i, j} into visited [].

3. ***Modifications and test case Selection***

**a.** *Delete function* ,after removal of function ”b” during code change.

| | | |
|---|---|---|
| $t_1\rightarrow$abfg | $\rightarrow$ | $t_1\rightarrow$afg |
| $t_2\rightarrow$aegi | $\rightarrow$ | $t_2\rightarrow$aegi |
| $t_3\rightarrow$abcdj | $\rightarrow$ | **$t_3\rightarrow$acdj** |
| $t_4\rightarrow$afbcd | $\rightarrow$ | **$t_4\rightarrow$afcd** |
| $t_5\rightarrow$afghi | $\rightarrow$ | $t_5\rightarrow$afghi |
| $t_6\rightarrow$abfg | $\rightarrow$ | **$t_6\rightarrow$afg** |
| $t_7\rightarrow$acdhi | $\rightarrow$ | $t_7\rightarrow$acdhi |
| $t_8\rightarrow$bcij | $\rightarrow$ | **$t_8\rightarrow$cij** |
| $t_9\rightarrow$cdgh | $\rightarrow$ | $t_9\rightarrow$cdgh |
| $t_{10}\rightarrow$ijab | $\rightarrow$ | **$t_{10}\rightarrow$ija** |
| $t_{11}\rightarrow$abcd | $\rightarrow$ | **$t_{11}\rightarrow$acd** |

Table 7. Requirements vs Test case

| | **a** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** |
|---|---|---|---|---|---|---|---|---|---|
| ~~**$t_1$**~~ | ~~1~~ | - | - | - | ~~3~~ | ~~4~~ | - | - | - |
| **$t_2$** | 1 | | | 2 | | 3 | | 4 | |
| **$t_3$** | 1 | 3 | 4 | | | | | | 5 |
| **$t_4$** | 1 | 4 | 5 | | 2 | | | | |
| **$t_5$** | 1 | | | | 2 | 3 | 4 | 5 | |
| **$t_6$** | 1 | | | | 3 | 4 | | | |
| **$t_7$** | 1 | 2 | 3 | | | | 4 | 5 | |
| **$t_8$** | | 2 | | | | | | 3 | 4 |
| **$t_9$** | | 1 | 2 | | | 3 | 4 | | |
| **$t_{10}$** | 3 | | | | | | | 1 | 2 |
| **$t_{11}$** | 1 | 3 | 4 | | | | | | |
| Freq | 9 | 6 | 5 | 1 | 4 | 5 | 3 | 5 | 3 |

From table- 7, affected test cases D(b)

$\rightarrow \{t_3,t_4,t_6,t_8,t_{10}, t_{11}\}$

$R_{safe} = RTS_{min} \cup R(b)$

$= \{t_3, t_2, t_5\} \cup \{t_3,t_4,t_6,t_8,t_{10}\}$

$R_{safe} = \{t_3, t_2, t_5,t_4,t_6,t_8,t_{10}\}$

***b**. Change in code of function identified*

From table-8 change in code identified in method c and hence, the * mark represents the change or modification. Hence test cases traversing **c** are selected.

$C(c)\rightarrow\{t_3,t_4,t_7,t_8,t_9,t_{11}\}$

$R_{safe} = RTS_{min} \cup C(c) = \{t_3, t_2, t_5\} \cup \{t_3,t_4,t_7,t_8,t_9, t_{11}\}$

$R_{safe} = \{t_3, t_2, t_5,t_4,t_7,t_8,t_9\}$

Table 8. Requirements vs Test case Table

| | a | b | **c*** | d | e | f | g | h | i | j |
|---|---|---|---|---|---|---|---|---|---|---|
| ~~**t1**~~ | ~~1~~ | ~~2~~ | - | - | - | ~~3~~ | ~~4~~ | - | - | - |
| **t2** | 1 | | | | 2 | | 3 | | 4 | |
| **t3** | 1 | 2 | 3 | 4 | | | | | | 5 |
| **t4** | 1 | 3 | 4 | 5 | | 2 | | | | |
| **t5** | 1 | | | | | 2 | 3 | 4 | 5 | |
| **t6** | 1 | 2 | | | | 3 | 4 | | | |
| **t7** | 1 | | 2 | 3 | | | | 4 | 5 | |
| **t8** | | 1 | 2 | | | | | | 3 | 4 |
| **t9** | | | 1 | 2 | | | 3 | 4 | | |
| **t10** | 3 | 4 | | | | | | | 1 | 2 |
| **t11** | 1 | 2 | 3 | 4 | | | | | | |
| Freq | 9 | 7 | 6 | 5 | 1 | 4 | 5 | 3 | 5 | 3 |

*c. Adding new function in code of function*

Table 9. Requirements vs Test case Table

$t_1\rightarrow$abfgk
$t_2\rightarrow$aegi
$t_3\rightarrow$abcdjk
$t_4\rightarrow$afbcdk
$t_5\rightarrow$afghi
$t_6\rightarrow$abfg
$t_7\rightarrow$acdhi
$t_8\rightarrow$bcijk
$t_9\rightarrow$cdgh
$t_{10}\rightarrow$ijab
$t_{11}\rightarrow$abcd

| | **a** | **b** | **c** | **d*** | **e** | **f** | **g*** | **h** | **i** | **j*** | **K** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ~~**t1**~~ | ~~1~~ | ~~2~~ | - | - | - | ~~3~~ | **4** | - | - | - | ~~**5**~~ |
| **t2** | 1 | | | | 2 | | 3 | | 4 | | |
| **t3** | 1 | 2 | 3 | 4 | | | | | | **5** | **6** |
| **t4** | 1 | 3 | 4 | **5** | | 2 | | | | | **6** |
| **t5** | 1 | | | | | 2 | 3 | 4 | 5 | | |
| **t6** | 1 | 2 | | | | 3 | 4 | | | | |
| **t7** | 1 | | 2 | 3 | | | | 4 | 5 | | |
| **t8** | | 1 | 2 | | | | | | 3 | 4 | |
| **t9** | | | 1 | 2 | | | 3 | 4 | | | |
| **t10** | 3 | 4 | | | | | | | 1 | 2 | |
| **t11** | 1 | 2 | 3 | 4 | | | | | | | **5** |
| | 8 | 6 | 5 | 4 | 1 | 4 | 5 | 3 | 5 | 3 | 3 |

From table-9, code or method being added is notified.

A(k) = M(g) ,M(d), M(j)

$\rightarrow\{t_2,t_5,t_6\},\{t_3,t_4,t_7,t_9\ t_{11}\},\{t_3,t_8\}$

$\rightarrow\{t_2,t_3,t_4,t_5,t_6,t_7,t_8,t_9,t_{11}\}$

$R_{safe} = RTS_{min} \cup A(k)$

$RTS_{min} = \{t_3, t_2, t_5\} \cup \{ t_2,t_3,t_4,t_5,t_6,t_7,t_8,t_9\ t_{11}\}$

$R_{safe} = \{ t_2,t_3,t_4,t_5,t_6,t_7,t_8,t_9,t_{11}\}$

## Appendix-2
## Comparison Scenarios

*Scenario-1:*

Comparison of Greedy heuristic vs Most Maximal Frequent Clustering based Test Suite reduction as in table-10.

Table 10. Sample Table-I

| | r1 | r2 | r3 | r4 | r5 | r6 |
|---|---|---|---|---|---|---|
| t1 | X | X | X | | | |
| t2 | X | | | X | | |
| t3 | | X | | | X | |
| t4 | | | X | | | X |
| t5 | | | | | X | |

Applying Greed heuristic results in selection of test cases as follows : test cases {t1, t2, t3 ,t4} - # Test cases(TC)-**04.** Most Maximal frequent Clustering and Residual requirements based clustering - test cases { t1, t2, t3 ,t4} - # TC-**04**

*Scenario-2*

Comparison of Greed heuristic vs Most Maximal Frequent Clustering based Test Suite reduction as in table-11:

Table 11. Sample Table-II

| | r1 | r2 | r3 | r4 | r5 | r6 | r7 |
|---|---|---|---|---|---|---|---|
| t1 | X | X | X | X | | | |
| t2 | X | X | X | X | | | X |
| t3 | X | X | X | X | | X | |
| t4 | X | | | | X | X | |
| t5 | | | | | X | X | X |
| t6 | X | | | | | | X |

Greedy heuristic test cases {t2, t3, t5} -#TC-03
Most Maximal frequent – { t2, t5}- #TC-02

*Scenario-3*

Comparison between HGS(Heuristic General to specific) approach and Current approach as in table-12:

Table 12. Sample Table-III

| | r1 | r2 | r3 | r4 | r5 | r6 |
|---|---|---|---|---|---|---|
| t1 | X | X | | | | |
| t2 | X | | | X | | X |
| t3 | | X | X | X | | |
| t4 | | | X | X | | X |
| t5 | | | | | X | |
| t6 | | | | | | X |
| t7 | | | | | | X |

HGS heuristic for this example is{t1, t2, t3}- #TC-03
Most Maximal Frequent clustering based approach- {t2, t3} - #TC-02.

## Appendix-3
## Cumulative Percentage of Code coverage

Cumulative Percentage of Code coverage expected (Fig.4) on test case basis will be higher when compared with heuristic approach since, more clusters are created and similar test cases are in same cluster. This ensures that in short runs this approach can ensure more coverage compared to other approaches.

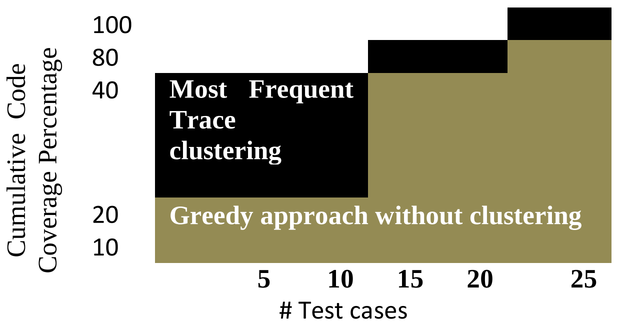

Fig.4. Cumulative Code Coverage Percentage

Graph in Fig.4 illustrates fact that with Most frequent coverage clustering provides more coverage with less number of test cases as compared with Greedy approach. The above inference may not hold when there are less redundant test cases or all test cases focusing dissimilar behavior, otherwise the performance is likely to be same as the Greedy approach(Fig.5).

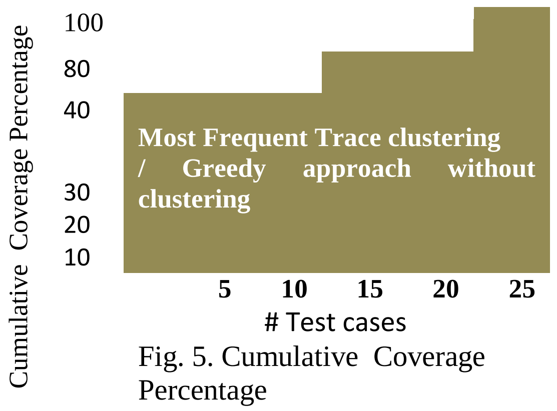

Fig. 5. Cumulative Coverage Percentage