Clustering Based Test Suite Selection for Ranking of Program Execution Sequence Using Improved Precision in Regression Testing

B. Bhaskar kumar Rao, R. Vasanth kumar Mehta, B. Narendra kumar Rao

Abstract: ***In Software development, Testing plays an important role. Testing becomes tedious with increase in no of test cases. Test Suite size increases and it becomes difficulty with redundant test cases and faulty test cases. Localization of fault is the process of isolating faulty location of program during its execution. Current approach uses the comparison of a successful run test case with a failed run test case spectrum for identification of faults. Along with above specified approach, criteria of program cohesion and program history are also used in effective localization of faulty components. The results suggest better performance when compared with other scenarios.***

Index Terms: Fault Localization, Regression testing, Test case Selection, Inclusiveness, Precision.

I. INTRODUCTION

Programmer's code and unit test software to cover many issues in them, but eventually during testing of the program so many issues arise, because of which a programmer has to localize and the problem and correct it. The current approach is useful in such a scenario where in fault localization [11] can minimize the number of test cases selected to detect similar faults and improve precision in fault detection. The approach relies on identifying similar test case traces and components which are fault prone, which further minimizethe number of non- defect revealing test cases and hence improve the precision during regression testing process.

II. LITERATURE

Program spectrum is a terminology which is unique for a successful and a failure run. This uniqueness can be considered for differentiation or identification of fault components. A programmer can contrast a failed run in current version with a passed run in previous version of a given test case. Vpoisotool have been proposed in literature which can compare a successful run program binaries with a failed run program in terms of phases and orders functions for fault localization. Delta debugging technique, a greedy approach which used a thread scheduling program for verification. DYNADIFF tool isolates faults from intra procedural calls and identify the paths that were not taken in prior version. A fault localizer program relies on a successful run and a failed run in literature as such. In current work similar case is followed, but there are two other criteria which are deemed to be important in fault localization like program cohesion and change history applied on program spectrum. Few authors in literature have proposed tools and other related work [1, 2, 3, 4, 8].

III. MOTIVATION

Program spectrum is obtained by capturing the stack trace of the given test case. These stack traces are then used to generate call trees. These call trees are then traversed to produce a call or program spectrum. The program spectrum recorded per test case is then mapped on to a mesh model for all requirement based test cases for a give version. The test case spectrum is recorded for different versions and a current version failed test case is compared with a previous version success test case. This provides an insight into variation of flow or program spectrum. The differentiation phase takes into account such failed and successful test case and localizes the part or segment of program spectra that is affected.

During regression testing phase there is a need to identify faults that affect other test cases, i.e., identify those affected test cases due to fault components and use them for effective fault detection or in other way, reduce the test case number that are necessary to identify faults around a given fault module. This approach is very much useful in cases where there is a need to improve fault detection with reduced number of test cases for testing. This work demonstrates the above intention with a conceptual system which is presented in section under Conceptual system and its representation.

IV. PROBLEM STATEMENT

In the process of regression testing based on clustered set of test cases, the safe regression testing is the most sought out criteria. The **major** parameters to be achieved are inclusiveness and precision. As suggested in [9], clustered set of test cases based on frequent segments, the inclusiveness is 100%, where both relevant and irrelevant modification revealing test cases are chosen for test suite reduction. But the parameters on Precision, which is measured in terms of eliminating the non-modification revealing test cases, are not quite effective, which means reduction of test cases is not effective. Hence, a mechanism which can reduce the number of test cases for locating similar faults is very much required for those test cases, which comprise of error prone modules. This can be achieved based on comparing a successful run of a test case in version (Vi-1) and version(Vi), having a failed run for the same test case.

CONCEPTUAL SYSTEM AND REPRESENTATION

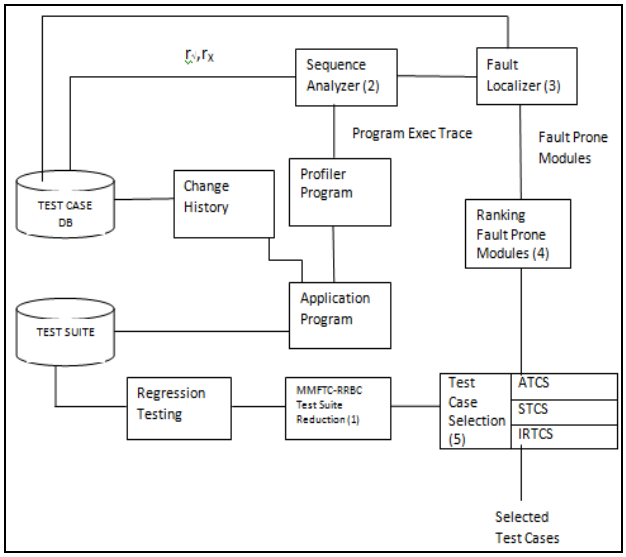


Figure 1: Conceptual System

Profiler program

Profiler program job is to record the all calls executed during its execution by a program. Specific tools like code Tune helps in recording the program traces and an excel format report will be produced. This format is further fed to trace program that can generate program traces in encoded identifiers form which suits for mapping test case-code coverage matrix. The output produced is in the form of strings of methods for each test case.

Application Program

The source for testing is standard SIR repository for testing purpose, and it can induces a bug per each revision. Hence program traces are sufficient for verification of test cases

Test Suite

Test suite contains complete amount of the test cases for the required system specification. They have been designed according to system requirements and the objective is to attain method coverage consisting minimum number of test cases.

Test Case Database

The aim of the test case database is to preserve the following and fetch them when it is necessary.

Revision history: Contains list of methods added, deleted, modified in each change required (can be selective) either through change in requirement or fixation of bugs. Source code will change history per file at functional level. Test case traceability matrix per standard code revision is stored.

Trace data of each program for which we need to run each test case needs to be stored. Test cases selected for every version, based on CVS (concurrent versioning systems) revision.

Change history

Change history codeDiffs between files which were revised in current version and the previous version and populates the lists like addM, delM, modM which represents the three types of changes that code is likely to undergo during its development process. The change history is capable of locating and classifying the methods based on the type of change like added, modified and deleted portions of the code. These are likely to be used further in the test selection process.

Two major tools are comprised in this module of one which codeDiffs two files of source code and generates HTML format reports which is further more analyzed by another program to produce or fill lists which are of the forms like addM, storing the list of newly added code, delM stores deleted methods and modM stores modified methods list in a given file. These lists further are useful for selecting test cases so that modified function traversing methods are chosen for testing of the system.

Test Case-Code Coverage matrix

Test cases which are executed generates set of sequences and by basing on the results achieved; the tester can conclude whether it leads to a success or failure run. All success runs are graphed to a matrix form where rows are test cases executed per revision and columns are methods or indices corresponding to methods used in chosen system requirements. These methods will form the code coverage with respect to requirements that test coverage is happened with test cases obtained.

All related columns are mapped to value i if the given method is encountered in ith position for a given test case execution. For each successful mapping of function or method, value of i increments 1 for the given test case. This will continue till all methods for a given test case are mapped as per requirements and all test cases are mapped in the matrix with its respective methods or functions.

Call sequence generation:

A call sequence is set of ordered sequence calls of methods or functions generated during execution of a test case or a program under study. These set of methods are recorded during execution of a program from call stack. Call stacks are recorded using profiling tools available in both open source and commercial formats.

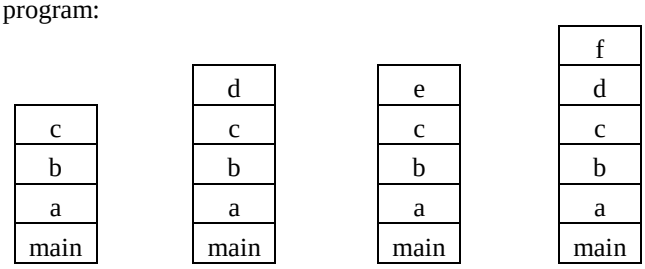
Profiling tools generate call stacks in form of HTML reports or .csv format reports which comprise of hierarchy of calling sequences of the stack at various instances of time for a given test case or program execution. Library call sequences are generally excluded from inclusion into the profiling reports generated by the profiling tools. The calling stack is sampled at regular intervals of time based on the time taken for the execution of simplest or smallest function in terms of time and complexity to ensure that all functions are observable in the profiling reports.

SymbolName is the attribute that represents the calling function name in the reports. The current mechanism involves generation of tree structure from the reports such that call stack is recorded in form of tree where in nodes represent the methods invoked during call stack invocation and edges represent the relation between the successive nodes in the call stack for the attribute SymbolName.

A call stack comprises of sequence of method calls which includes name of functions or methods invoked during execution of test case. The call stacks generated repeatedly are mapped on to the same tree. All these mappings of the call stack (in form of trees) are done on to the graph. The two major attributes representing the data structures are fan-in and fan-out. Fan-in represents the number of edges generating or invoking the

given node or method. Fan-out represents the number of edges invoked by a given method. The latest sequences of methods occupy the left most trees for a given node. At a given instant for a node, the number of children for a given method represent the number of methods invoked by it. The infix traversal of the final tree generates the call sequence for the corresponding test case.

Program Illustration is defined below in form of simple program:



C_1 = {main, a, b, c}
C_2 = {main, a, b, c, d}
C_3 = {main, a, b, c, e}
C_4 = {main, a, b, c, d, f}
C_s = {main, a, b, c, d, f, e}

Here, C1, C2, C4are subset sequences and Cs is superset sequence, such that all subset sequences (Cn) are units of the maximum superset sequences (Cs).

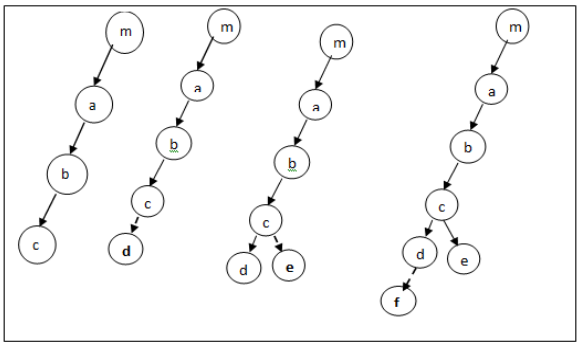


Figure 2: Test Case and Code Coverage matrix

Test cases executed in a certain time generates set of sequences and basing on the result the tester decides whether it has generated a success or failure run. A program run may generate different program traces for success run. All success runs are mapped to a matrix where the test cases executed per revision represent rows and methods or indices corresponding to methods used in selected system requirementsrepresents column. These methods will form the code coverage requirements that test cases required to ensure test coverage.

All related columns are mapped to value i if the given method is encountered in i^{th} position for a given test case execution. For each successful mapping of function or method, value of i increments 1 for the given test case. This will continuetill all methods for a given test case are mapped and all test cases are mapped in the matrix with its respective methods or functions.

Most Maximal Frequent Trace Clustering [9]

Test cases are repetitiousin many cases since they are designed keeping requirements in view. By using traceability matrix requirements are mapped to test cases, which helps in identifying the test cases required for testing.

Most frequent program trace clustering algorithmwill group most frequent traces of coverage items as a group among the most frequent traces of test cases in the given suite. The clustering process will be carried out until all the code coverage items are made as cluster in the form of test cases as described in the algorithm next.

The main intent of this algorithm is to cluster only those test cases that are redundant from coverage item viewpoint. The test cases are sequenced/aligned in the decreasing order of frequency of code coverage items based clusters, which comprises the test cases.

Residual Requirements based Test Suite Reduction [9]

Earlier in literature there are test suite reduction algorithms which take up greedy approach and another one takes up HGS approach which selects test cases for test coverage requirements. Algorithm is largely focusing on selecting residual code coverage requirements, in which the algorithm will select test cases which have high maximum code coverage. This involves generation of visited list of code coverage requirements for a given test case(s) selected from maximal clusters and then select test cases from next most frequent items from the clusters which were definitely not present or present minimally among the previously selected test cases for code coverage. It maximizes the code coverage ability and reduces the number of test cases selected for the testing step.

Sequence Analyzer:

Sequence ana1yzer is sequence of steps which can compare a success (passed test case-r√) one in version Vi-1 with next test case which provides a failed run (passed test case-rx) in version Vi. The sequence analyzer which is a sequence of steps compares the program call sequences of these test cases and stores the variations in patterns`fs from the actual passed one. The o/p is in form of program methods which differs in their failed runs.

This sequence analyzer is composed of sequitur [7] algorithm Sequence modeler, sequence diff generator and assign weight module.

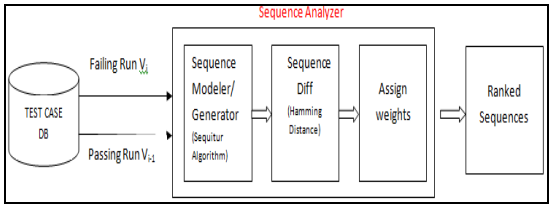


Figure 3: Sequence Analyzer

Sequence Modeler: The sub module inputs the corresponding runs(r√, rx)and splits the sequence based on k-sequencer, this determines the number of segments into which the given call sequence is split.

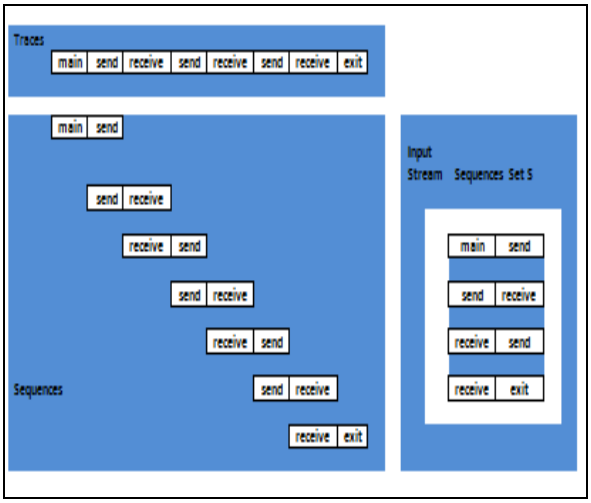


Figure 4: Sequence Modeler

Sequence Diff Generator: The corresponding module performs hamming distance computing and it will compare the sequence of both passed and failed case, such that it can identify the unique and non unique traces to form two lists called sim() and dissim().

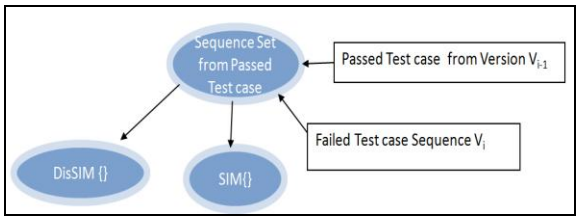


Figure 5: Sequence Diff Generator

Assign Weights: It assigns ranks to faulty modules basing on program cohesion, spectral difference, and change history based selection to dissim set of modules.

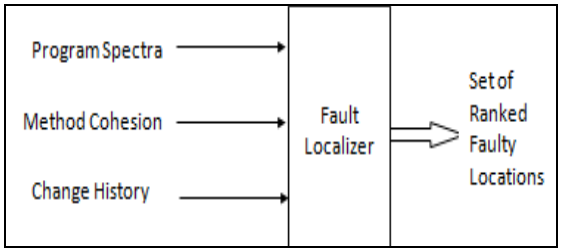


Figure 6: Ranking Fault Sequences

Fault Localizer [5, 6]:

Fault localizer ranks the individual test case sequences from groups of clustered test cases, those which are similar to failed ranked sequences identified in sequence analyzer. The i/p consists of all regression testing chosen cases and o/p consists of ranked test cases in priority of identifying fault so that most likely test cases which fail or mostly failure test cases are in top (max ranking) and less probable cases in bottom (low rank). The objective is to remove duplicte modification revealing test cases from diffeent passed runs or test cases. This is the intention of precision in regression testing.

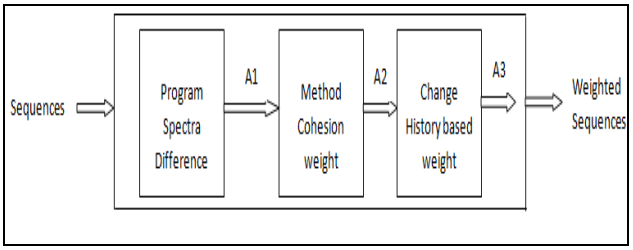


Figure 7: Fault Localizer

Method Cohesion:

Methods invoked during test case generation are represented in matrix for as discussed in Test Case-Code Coverage matrix. The three types of cohesion as specified in [8] are discussed as follows:

- Common modules those execute under all test cases as per requirements based test cases. Eg:{m1, m20}
- Potentially involved modules are those which need not be executed with all test cases. Eg,{m3}
- Indispensably involved modules are those which are particular only to a particular test case but not linked to all other test cases. Eg.{m7}

The methods correspond to entries T in Fig-8.

$t \times M$	m_1	m_3	m_7	m_{15}	m_{16}	m_{20}	m_{17}
t_1	T	T				T	
t_2	T		T			T	
t_3	T	T			T	T	T
t_4	T	T		T		T	

Figure 8: Test case vs Methods relationship

Ranking Fault Prone modules:

The test case which produced a failing result on comparison with a success version of the same generates a set of methods which are likely to contain the defect. The faulty components are ranked based on change history and method cohesion using a ranking algorithm as explained in next section.

Test case Selection approaches:

Selection of Test case during regression testing consists of selection of test cases that are very much likely to the changes made and should reveal defects in modified code. This is measured in terms of inclusiveness and precision. This refers to earlier work by Rothermal. The present research aims at identifying effective test case selection with more precision based test cases. The approach planned is Item Ranking based-Test case selection (IRTCS). For improving Precision based test cases, this current approach is being compared with two available approaches are known as Selective-Test case selection (STCS) and All-Test case selection (ATCS) which will be discussed during experimentation(discussed in section-VII).

V. PROCEDURES

MostMaximalFrequentTraceClustering:
Input:
Requirement based test cases in Test Case-Code coverage matrix format.
Output:
Clustered test cases.
Procedure:
Perform Most Maximal frequent trace clustering on requirement based test cases.

ResidueRequirementBasedReduction:
Input:
Clusters of test cases, code coverage requirements in terms of functions.
Output:
Reduced test cases
Procedure:
Select test cases from clusters until all code coverage requirements are satisfied

ChangeHistoryTestselection:
Input:
Requirement based test cases in Test Case-Code coverage matrix format, change history list
Output:
Change history based test cases.
Procedure:
Based on change history category such as added, modified and deleted select test cases from Test Case-Code coverage matrix.

TestCaseSelection:
Input:
Test cases selected from MMFTC-RRBC approach.
Output:
Modification revealing test cases.
Procedure:
Test case selection will depend on the three approaches like:
a. All-Test case selection (ATCS)
b. Selective-Test case selection (STCS)
c. Item Ranking based-Test case selection (IRTCS).

Sequence Modeler:
Input:
Failing version V_i trace-T_i-$r_{\surd}$
Passing version V_{i-1} trace-T_i'- r_X
Output:
Generate pSeq[],fSeq[] for given two sequences.
Procedure:
a. Inputs two sequences, $r_X$from Vi and $r_{\surd}$ from $V_{i-1.}$
b. Split them to corresponding different segments.

SequenceDiffGenerator:
Input:
Segments from Sequence Modeler
Output:
Sim{},DisSIM{}
// similar and dissimilar segments
Procedure:
Separates the input segments into two sets known as similar and dissimilar segments.

FaultLocalizer:
Input:
Faulty components from disSim{}
Output:
Ranking of segment components
Procedure:
Perform ranking of segment components based on change history and method cohesion.

AssignWeights:
Input:
disSim{}and selected test cases for regression test case selection
Output:
Perform ranking of segment components
Procedure:
Ranking for fault components and prioritizing test cases based on change history and method cohesion.

RankingAlgorithm:
Input:
disSim{}and selected test cases for regression test case selection
Output:
Perform ranking of segment components
Procedure:
Identification of methods which are causes of Failures from program spectral difference and assign the following priorities(priority values specified for each case in parenthesis):

Priority 1: If a method is from codeDiff changes and also cohesive thenassign priority as follows (TOP):
1.1: If modules are common and part of codeDiff list then assign highest priority (7).
1.2: If modules are potentially involved and part of codeDiff then assign moderate priority (6).
1.3:If modules are indispensably involved and part of codeDiff then assign low priority (5).
Priority 2: Program spectrum difference modules and CodeDiff generated modules (4).
Priority 3: Methods with cohesion relationship are assigned the following:
3.1: If modules are common then assign highest priority (3).
3.2:If modules are potentially involved then moderate priority(2).
3.3: If modules are indispensably involved then low priority (1).

VI. EXPERIMENTATION

Current work is based on improvement of precision during test case selection of regression testing. This work also compares three approaches for the same such as All-Test case selection (ATCS), Selective-Test case selection (STCS), Item Ranking based-Test case selection (IRTCS).

Subject Application & Metrics

S*pace* program in SIR repository is used in this work as program data. SIR repository has 1400 test cases of test pool and faults which consists of 38 versions of the same program. For instrumentation of call stacks and call coverage tree, Code Tune is considered for the work. To

populate the Program profile, forward analyzing of program is needed, such that reports will be create in excel. Function name denotes the Current traces Analysis of the codeis being done perfectly before testing is being performed and test cases are set to target given functions. Many versions will be maintained for each progam, for eg: Nearly 38 different versions are maintained for a program, but each program has the difference from others with at least a single failure. The main reason for experiment to be initiated between two versions and then clusters the test cases by considering many failures between considering its previous program trace information.

Iterative programs are not treated when recording of trace is done, but multiple traces can be studied once. Skipping of loop calls will be done to single calls following the modified algorithm [14]. Library functions which are conjure while test case runs were removed, as we feel that their presence may not be much appreciated in the work..

Method of Experimentation

The identification of defects can be done from the suite in two ways:

1. By using the approach of test case selection and identifying the efficiency of fault detection, test cases are considered.
2. For considering number of test cases which depends on algorithm and identify the number of defects
   detected by each k test cases. Continue these steps till all faults are identified.

Inclusiveness:

For considering C considerations, n of these tests Let's consider that A has n tests changes revealing for B and B',. Safest testing technique is the Regression testing if it is fully inclusive.

Precision:

If A contains test cases, that are static for B and B' then, c suppose. The Precision of M relative to P and, P', and T is the percentage given by expression (100 * (A/B)) if n!=0 or 100% , if n = 0.eliminates A of tests.

Change history based test case selection [10]

Including Change history [5] in test case is an appropriate approach, as there will be change in code with respect to requirements.

Here in experimentation section (4), code difference in code modules can be found with change history module and creates a report on difference between versions. New code changes can be reflected in change history with respect to changes in code.

In experimentation if changes done to the test cases will be reflected in the change history once the test case is run successfully. Changes will be reflected immediately.

Not only to find the defect in the test cases, has it helped to change the code when some changes in the code are required. The approach is presented as follows:

1. Clusters will be formed based on most frequent tracing of test cases.
2. Item set frequency Group can be formed basing on item set frequency and same item frequency with different items with can be clustered into sub clusters.
3. Repeat this process till all test cases are clustered and maximum no of frequency cluster are found.
4. Formation of Clusters includes sub-clusters combine those test cases that are similar by:
   4.1 When Test cases pocess like program profiles.
   4.2 Test case is appropriate if all elements belong to given program trace of other test case.
5. Select next test case based on Residue requirements based test case reduction.
6. Perform test case selection based on changes history components such as method addition, modification and deletion.
7. For test case given, we need to look for test case of fault finding type and do the below steps:
   7.1 Look whether test case is thru in version Vi-1.
   7.2 When the test case is thru then perform the following:
      7.2.1 Contrast the stack trace vs previous version Vi-1.
      7.2.2 Must apply sequence investigation on the same.
      7.2.2.1 Create different k-Sequence from thru and fail run test cases.
      7.2.2.2 Find Hamming distance based sequence differencing on k-sequences.
      7.2.2.3 Sim and disSim sets are generated.
   7.2.3 Ranking for thedisSim set generated elements basing on
      7.2.3.1 Method cohesion which is of any three forms like common, potentially involved and indispensably involved.
      7.2.3.2 Methods basing on Change history are added, and ranks will be allocated to faulty components.
8. From regression selected safe test cases computed in Step-7, compute the nearest similar test case from clusters formed in step-1.
10. During regression testing minimize the no of test cases by giving rank to test case basing on the following criteria:
    a. Program spectra difference
    b. Difference in history change
    c. Cohesion in method.
11. Checkout the results are basing on the three approaches
    a. ATCS
    b. STCS
    c. IRTCS

There are two sub functionalities in Change history module like codeDiff and changes in records at course level like function which is added, function which is modified and function which is deleted from the diff engine and adds them into addM, delM, modM. These lists can be used in investigation for selection of test case.

It is evident that results were symbolic that all test cases for history changes were introduced, helps in introducing defects. Change types like new Comment and formatting of existing changes [8] are not at all induced. Future study can incorporate such changes as well.

Selection of Test case in change based, regression test selection is taken-up with investigation, in which weconsider Inclusiveness, Precision and Efficiency which are important factors.

Precision with respect to test case can be found using this formula.

$$|Pr(x)| = 1 - \frac{|Tn(x)|}{|Tn\ (ATCS)|} \times 100$$

Tn (x) – Number of test cases available in a precise precision improvement approach.
x – represents the precision improvement approach (ATCS), (STCS), (IRTCS).

ATCS: Selection of all relevant and irrelevant test cases is a criteria for inclusion of all test cases.

STCS: unplanned test cases are picked from inclusive test case for precision improvement.

IRTCS: Test cases selected by following entire approach as specified and ranks entire faulty modules before choosing the confirmed modification declaring cases in form of ranking the test cases from forming clusters to diagnosing likely faults.

The evaluation (shown in TABLE-I) of this work is targeted towards observation of precision of the approaches specified.

COMPARISON OF APPROACHES

S.No	Total Test cases	Total Defects	Number of Clusters	Average Test cases per cluster	All-Test case selection (ATCS) (1)	Selective-Test case selection (STCS) (2)	Percentage Reduction (STCS compared to ATCS)	Item Ranking based-Test case selection (IRTCS) (3)	Percentage Reduction (IRTCS compared to IRTCS)
1	50	8	10	5	20	14	**30**	13	**35**
2	100	23	14	7	38	33	**13.1**	29	**23.6**
3	150	42	21	6.5	65	57	**18.4**	52	**20**
4	200	50	24	7.5	78	67	**14.1**	60	**23**

The above results depicted in table form are as shown in Fig-9.

Figure 9: Comparison of approaches

Threats to validity:

When a program is executed, it generates varying program traces for each correct run. Context sensitive test cases are not part of current work.Multiple defects can be suggested not identified.

By following requirements, designing of test suites are carried out in this work and other criteria based test suites will never be tested with this approach.

VII. CONCLUSION

Program traces can be represented as sequence of ordered strings. Program traces can be represented as sub sequences of length –K without losing generality. Comparing a test case for Passed run and failed run is effective means for identifying similar faults. Similar program traces reveal more defects, when test cases are chosen from a similar group of sequences/Clusters revealing defect(s).

The evaluation of this work reveals that clusters of test cases were chosen to be tested on SUT using the approaches ATCS, STCS and IRTCS, it is observed that better defect detection was possible using the proposed IRTCS approach.

Current work is to be carried out where unique traces are possible in GUI development and some methods must be used as libraries event handlers. This will be a good challenge for this current work.

REFERENCES

1. Sri vidhya J," Modified Genetic approach for Regression Testing Cost Reduction", International Journal of Infinite Innovations in Engineering and Technology, Volume 1, Issue 1, May 2014.
2. Alireza Khalilian and Saeed Parsa ,Bi-criteria Test Suite Reduction by Cluster Analysis of Execution Profiles ,International Federation for Information Processing ,CEE-SET 2009, LNCS 7054, pp. 243–256, 2012.
3. Mala, D.J., Mohan, V., "Quality Improvement and Optimization of Test cases– A Hybrid Genetic Algorithm Based Approach", ACM SIGSOFT Software Engineering notes, Vol. 35(3), pp: 1-14, ACM Press, 2010.
4. Osamu Mizuno, and Hideaki Hata,"Prediction of Fault-prone Modules Using A Text Filtering Based Metric", International Journal of Software Engineering and Its Applications, Vol. 4, No. 1, January 2010.
5. Ying, A.T.T.; Murphy, G.C.; Ng, R.; Chu-Carroll, M.C., "Predicting source code changes by mining change history," Software Engineering, IEEE Transactions on , vol.30, no.9, pp.574,586, Sept. 2004.
6. Xia Cai, Michael R. Lyu,"The Effect of Code Coverage on Fault Detection under Different Testing Profiles",A-MOST '05 Proceedings of the 1st international workshop on Advances in model-based testing, ACM SIGSOFT Software Engineering Notes, Volume 30 Issue 4, July 2005.
7. Munson, J.C.; Elbaum, S., "Software reliability as a function of user execution patterns," Systems Sciences, 1999. HICSS-32. Proceedings of the 32nd Annual Hawaii International Conference on , vol.Track8, no., pp.12 pp.., 5-8 Jan. 1999.
8. Hassan, A.E.; Holt, R.C., "Predicting change propagation in software systems," Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on , vol., no., pp.284,293, 11-14 Sept. 2004.
9. Frank Eichinger, KlemensBöhm, Matthias Huber,"Improved Software Fault Detection with Graph Mining ",Proceedings of the 6th International Workshop on Mining and Learning with Graphs (MLG), Helsinki, Finland, 2008.
10. Reiss, S.P.; Renieris, M., "Encoding program executions," Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on , vol., no., pp.221,230, 12-19 May 2001.
11. Gang Shu, Boya Sun, Andy Podgurski, Feng Cao, "MFL: Method-Level Fault Localization with Causal Inference", ICST, 2013, 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation ICST 2013.
12. Narendra Kumar, A. RamamohanReddy,"Frequent Segment Clustering of Test cases for Test Suite Reduction",WSEAS Transactions on Computers, Vol-13,July 2014.
13. Narendra Kumar, A. RamamohanReddy,"Frequent Item Test Case Clustering Based Test SuiteReduction", The Mediterranean Journal of Computers and Networks,Vol-10, issue-2,249-260.

AUTHORS PROFILE

Mr. B. Bhaskar Kumar Rao is an Assistant Professor from Department of IT, Sree Vidyanikethan Engineering College, A. Rangampet. He has done his M. Tech in Department of CSE from SRM University, Chennai and B.E from Sree Vidyanikethan Engineering College, A. Rangampet. His area of research includes Software Engineering, computer vision related approaches.

Dr. R. Vasanth Kumar Mehta is currently working as an Associate Professor, Department of CSE at SCSVMV. He obtained his Ph.D. in CSE from SCSVMV and M.Sc. (Tech) and B.Sc. in CSE from BITS Pilani His research interests are data mining, machine intelligence and image processing.

Dr. B. Narendra Kumar Rao, obtained Bachelor Degree in Computer Science and Engineering from University of Madras, M.Tech in Computer Science from JNTU, Anantapur, Ph.D. from JNTUH, Hyderabad. He has more than 18 years of experience in Area of Computer Science and Engineering which includes four years of Industrial Experience and Twelve years of Teaching Experience. Research interests include Software Engineering, Deep Learning and Embedded Systems. Currently he is working as Professor, Chairman Board of Studies in Department of Computer Science and Coordinator, IQAC at Sree Vidanikethan Engineering College