A Hybrid Approach for Regression Testing in Interprocedural Program

Yogesh Singh*, Arvinder Kaur* and Bharti Suri*

Abstract—Software maintenance is one of the major activities of the software development life cycle. Due to the time and cost constraint it is not possible to perform exhaustive regression testing. Thus, there is a need for a technique that selects and prioritizes the effective and important test cases so that the testing effort is reduced. In an analogous study we have proposed a new variable based algorithm that works on variables using the hybrid technique. However, in the real world the programs consist of multiple modules. Hence, in this work we propose a regression testing algorithm that works on interprocedural programs. In order to validate and analyze this technique we have used various programs. The result shows that the performance and accuracy of this technique is very high.

Keywords—Regression Testing, Test Prioritization, Test Selection, Interprocedural

1. INTRODUCTION

Software maintenance is defined as activities performed on a software product subsequent to its release for use. These activities manage the changes that are often indispensable during this phase of the software life cycle. It is important to retest in order to verify that these modifications or changes do not have unintended effects and, therefore, the system still complies with its specified requirements [1]. It is not possible to execute all the test cases due to time and resource constraints. The selective retesting of the system or component is called regression testing. Regression testing establishes the confidence in the modified program. It may account for as much as half of the cost of software maintenance [2]. The importance of regression testing can be understood from the fact that the single most costly bug in software history could have been revealed by regression testing [3, 4].

Regression testing techniques are categorized as: regression test selection techniques, regression test prioritization techniques, and hybrid techniques. The regression test selection technique chooses the tests from the old test suite to execute on the modified version of the software. Regression test prioritization techniques reorder test suite with a goal to increase the effectiveness of testing in terms of achieving code coverage earlier, checking frequently used features of software, and early fault detection. Hybrid techniques combine both selection and prioritization for regression testing.

Various techniques mentioned in literature are based on test selection criteria. Some criteria

Manuscript received December 16, 2009; revised March 2, 2010; accepted March 10, 2010.
Corresponding Author: Bharti Suri
* University School of Information Technology, GGS Indraprastha University, Delhi (ys66@gmail.com, arvinderkaurtakkar@yahoo.com, bhartisuri@gmail.com)

select test cases exercising functions in programs that have been changed or deleted in producing changed programs [5]; some seek a subset of test suites that is minimal in covering all functions of the program identified as changed as explained by Fischer, Raji, and Chruscicki [6] and Gupta, Harrold, and Soffa [7]; and some have been proposed in [8-12]. One of the techniques, proposed by Rothermel [8] and called the safe test selection technique, involves the construction of a control flow graph for different versions of the program and uses these graphs to make the test selection. The selection is carried out with an intention to execute the changed code. Elbaum, Malishevsky, and Rothermel [13] and Elbaum et al. [14] proposed 18 different test case prioritization techniques that are classified into statement-level and function-level techniques. Rothermel et al. [15] have presented many techniques for prioritizing test cases based on the coverage of statements or branches in the program. A prioritization technique based on historical execution data has also been presented [16]. The technique developed using modified condition/decision coverage for test suite prioritization is presented in [17]. Requirements based prioritization, which incorporates knowledge about requirements, complexity, and volatility, is proposed in [18]. Another prioritization technique, which takes into account the output influencing and branches executed by test cases, is proposed in [19]. The information about system model and its behavior is used to prioritize test suite in model-based test prioritization method in [20]. Krishnamurthy et. al. developed and validated requirement based prioritization scheme to reveal more severe faults at earlier phase of software development[21].

The hybrid approach combines both regression test selection and test case prioritization. A number of techniques and approaches have evolved in the past based on the following concepts: 1) the test selection algorithm proposed by Aggarwal, Singh, and Kaur [22]; 2) the hybrid technique proposed by Wong et al. [23], which combines minimization, modification, and prioritization based selection using test history; 3) the hybrid technique proposed by Gupta et al. [24] based on regression test selection using slicing and prioritizing the selected definition-use associations; and 4) the variable based hybrid approach by Singh et al. [25].

The remainder of the paper is organized as follows: Section 2 will give brief introduction of our proposed basic method and the terminology used. In Section 3, we present experimental setup and data collection. Section 4 explains the steps followed in the study, the results of which are presented in Section 5. In Section 6 application of the technique is given. Section 7 describes threats to validity for the proposed technique. Finally, conclusions are mentioned in Section 8.

2. BASIC METHOD AND TERMINOLOGY

The proposed hybrid approach is based on the selection and prioritization of the test cases for interprocedural programs. It is a version-specific technique that takes into account the variable usage in the old as well as the modified program, named as P_e and P_e' respectively. The technique requires that the test cases in the original test suite T_e not only contain test case identification, expected input and expected output (as per past practice) but also the variable(s) that is (are) being checked by this test case and the module to which the variable belongs. It selects all those variables that are in the changed statements and then selects only those test cases that either correspond to these variables or to the variables computed from them recursively. Multiple-level prioritization of the selected test cases is performed on the basis of variable usage. Variables are a vital source of changes in the program and this approach captures the effect of

change in terms of variable computation. The approach takes into account the changes in the variables and its ripple effect. Appendix 1 defines some related terminology.

A computed variable table (CVT_e) is prepared (maintained through development testing) in which the list of variables computed from other variables is maintained. An array with the information of the number of times the variable is used in computation is also maintained during development testing in VDC_e (Variable Dependency Count).

The algorithm is presented in Appendix 2 which demonstrates the technique. Initially, the resultant test suite is set to null. In step 2 of algorithm, a list of variables "V_e" is created from changed (inserted/modified/deleted) lines using array CLB which maintain changed line numbers. If any variable is deleted permanently from the program by modification or deletion of any line, it results in modified versions of V_e, VDC_e, and CVT_e (by deleting the row corresponding to those variables). The selection step and priority1 assignment step (step 3) selects all those test cases that correspond to variables contained in modified V_e. These test cases are assigned Priority1 as 1 (step 3(i), (ii)). Step 3(iv) of the algorithm gets the variable computed from variables found above from modified CVT and sets Priority1 of corresponding test case as 2 onwards. If the same test case already exists then Priority1 is kept as the minimum of the two.

After assigning Priority1, Priority2 are assigned, as stated in step 4 of the algorithm. The purpose of assigning Priority2 is to further prioritize the test cases that have the same value as Priority1. Priority2 is based on the dependency count as in the modified VDC_e. The variables which have highest dependency count are selected. The test cases corresponding to these selected variables are assigned Priority2 as 1. Then, the variables having next highest dependency count are selected. The test cases corresponding to them are assigned priority2 as 2 and so on. Step 4(i) to Step 4(iii) chooses all the test cases with same Priority1 and Step 4(v) further prioritize according to the dependency count.

The resultant test suite T' has test cases having Priority1 and Priority2 assigned.

3. EXPERIMENTAL SETUP AND DATA COLLECTION

To carry out the study, five programs written in C language were selected. The programs for the experiment include problems such as calendar, triangle, time-date, Kmap generation and tax calculation. The test suites were prepared for each of them. A group of four students from the Masters of Information Technology course at Guru Gobind Singh Indraprastha University was formed under the supervision of two Asstistant Professors. The students had prior knowledge of software testing. The test cases were prepared by the students and verified by the Assistant Professors. A prerequisite to this work is to have lines of source code numbered so as to perform the gray box analysis, which is the basis of our testing technique. Gray box testing is a combination of black box and white box testing approaches. Gray box testing includes testing from the outside of the product, as is done in the black box, but the test cases are designed incorporating the information about the code or the program operation [26]. The test suites in this approach are based on the gray box technique, as the input/output follows the black box strategy and at the same time keeping variable usage information, which is basically a white box approach.

The interprocedural technique proposed in the pioneering work of Rothermel [8] was referred to for comparison. The factors accounted for choosing this technique are the availability of the detailed explanation of all aspects of their work and the efficiency of using the control flow

graph (CFG) with the test case trace for test selection.

The overall experiment was carried out corresponding to each of the intermodule techniques. The experimental data was gathered for two techniques and, analysis was carried out on the collected data. The results are presented in a later section.

4. STEPS FOLLOWED IN THE STUDY

The following steps were followed in this study:

(1) There are two versions of each program: the old and the modified, and the original test suite T_e, CVT_e, VDC_e, CLB_e, and V_e (described in Appendix 1).

(2) We then updated/modified V_e, VDC_e, and CVT_e depending upon the changes made in the older version. Then we applied the proposed technique, and the result was the reduced test suite T_e' with priorities assigned. These steps were repeated for all the programs chosen for this study. With the resultant test suites, the objective was to measure the effectiveness of the technique in terms of statement coverage, branch coverage and average rate of fault detection.

(3) Control Flow Graphs were constructed for all the programs. A CFG for a program is a directed graph with vertices and edges where vertices are the statements and edges represent flow of control. Statement and Branch coverage metric was analyzed using CFG. Statement coverage measures the number of statements traced by a particular test case. A branch is an edge in a CFG from a decision node. A test case covers a branch of a program if the flow of control passes through it [26].

(4) The third coverage criterion is related to the Average Percentage of Faults Detected (APFD)[12, 14, 27]. We created faulty versions of the programs and then analyzed the effectiveness of T_e' in exercising the contained faults. For creating the faulty versions of the programs, simple errors in the operator/operand were manually seeded on the model of the competent programmer hypothesis and coupling effect [28, 29]. The competent programmer hypothesis states that competent programmers tend to write programs that are close to being correct, that is, a program written by a competent programmer may be incorrect by relatively simple faults in comparison to a correct program. The coupling effect states that a test data set that detects all simple faults in a program will also detect more complex faults. The faults severity and test case cost are assumed to be uniform. [15, 30] incorporates varying test case and fault cost. The results presented here may be different if the varying cost is considered.

Rothermel's technique [8] was applied to the same programs and all the results were computed.

5. RESULTS AND DISCUSSIONS

Table 1 summarizes the "min," "max," "mean," and "standard deviation" of parameters computed from both the techniques. The values prefixed by "R" are computed by Rothermel's technique and the others are computed by the suggested technique. The mean value of the tests se-

Table 1. Descriptive statistics of computed measures from the two techniques

	Min	Max	Mean	Std Deviation	Variance
Rothermal Technique % of statement covered	35.35	85.50	65.13	24.09	580.38
Hybrid Technique % of statement covered	33	83.35	61.06	22.67	514.08
Rothermal Technique % of branches covered	29	79.16	56.49	22.43	503.22
Hybrid Technique % of branches covered	27.33	79.14	56.17	23.09	533.30
Rothermal Technique % of modified statements covered	94.44	100	98.88	2.48	6.18
Hybrid Technique % of modified statements covered	94.44	100	98.88	2.48	6.18
Rothermal Technique % of test case selected	95.34	100	98.36	2.27	5.17
Hybrid Technique % of test case selected	49.41	92.98	71.40	16.85	284.01
Rothermal Technique % of prioritized test cases for modified coverage	1.17	38.59	14.77	16.02	256.70
Hybrid Technique % of prioritized test cases for modified coverage	1.1700	38.59	17.56	15.00	225.02

lected by the proposed technique is 71.40 and by the compared technique 98.36. However, the mean of statement coverage achieved by our technique is 61.06 and with the compared technique is 65.13.This shows that our technique selects few test cases with comparable statement coverage. The branch coverage, modified coverage and percentage of test cases selected for modified coverage are almost the same.

Table 2 shows the summarized results for the two techniques. Fig. 1 to Fig. 5 represents results shown in this table.

Figure 1 compares statement coverage for the two techniques. Figure 2 gives the comparison in terms of branch coverage results. The graph shows comparable results from the two techniques for statement and branch coverage. Figure 3 shows modified statement coverage. It shows that both the techniques cover 94-100% of the modified statements. The percentage of test cases selected after implementing the techniques is shown in Figure 4. The selected test cases using our technique are less than those selected with the compared technique. Figure 5

Table 2. Results of the compared techniques

	Calender	Time- Date	Triangle	Kmap	Tax for Employee
Rothermal Technique % of statement covered	78.75%	35.35%	42.71%	85.50%	83.34%
Hybrid Technique % of statement covered	80.30%	33.00%	41.85%	66.80%	83.35%
Rothermal Technique % of branches covered	56.83%	29.00%	39.59%	77.88%	79.16%
Hybrid Technique % of branches covered	57.82%	27.33%	38.78%	77.82%	79.14%
Rothermal Technique % of modified statements covered	94.44%	100%	100%	100%	100%
Hybrid Technique % of modified statements covered	94.44%	100%	100%	100.00%	100.00%
Rothermal Technique % of test case selected	100%	95.34%	96.49%	100%	100%
Hybrid Technique % of test case selected	49.41%	60.46%	92.98%	78.18%	76.00%
Rothermal Technique % of prioritized test cases for modified coverage	1.17%	4.65%	38.59%	5.45%	24.00%
Hybrid Technique % of prioritized test cases for modified coverage	1.17%	18.60%	38.59%	5.45%	24.00%

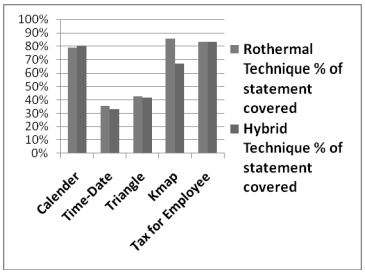


Fig. 1. Statement Coverage for the Compared Techniques

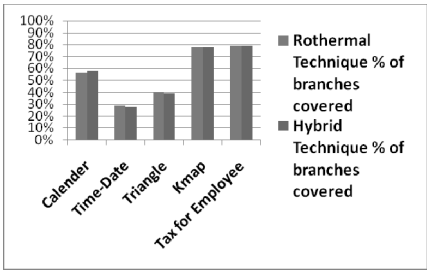


Fig. 2. Branch Coverage for the Compared Technique

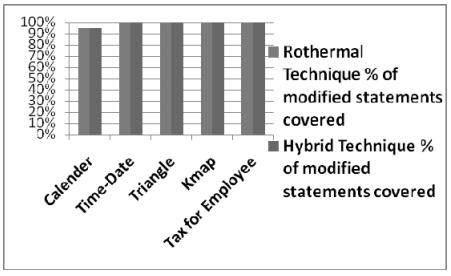


Fig. 3. Percentage of Modified Statement Coverage

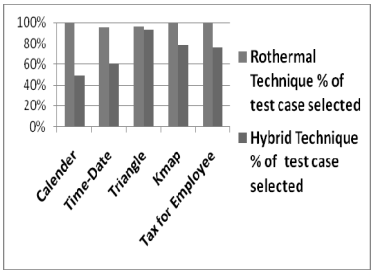


Fig. 4. Percentage of Test Case Selected

displays the percentage of test cases needed for the modified coverage achieved. It is clear from Fig. 1 to Fig. 5 that though the number of test cases selected from our approach is less than the compared technique, the coverage achieved with respect to statement, branch and modified statements is equivalent. Very few test cases are required to cover the modified portion of the programs. The coverage criterion does not mean that the technique is better. But it depends on whether it selects those test cases that have the potential to catch the faults. We are selecting those test cases that check faults corresponding to the variables used in the changed lines or variables computed from them. These test cases have a high potential of catching faults as either they check those variables that are in the modified lines or are computed from them. The only way a fault can travel from one part to another part of the program is through the variables. Also, those variables that have a high dependency count are given higher priority so as to check faults corresponding to those variables and hence the most affected part of the program. Figure 6 gives

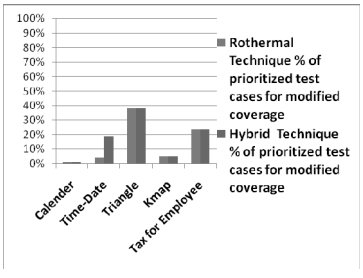
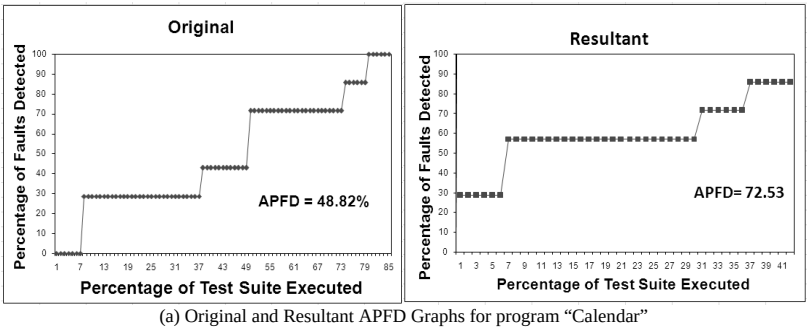
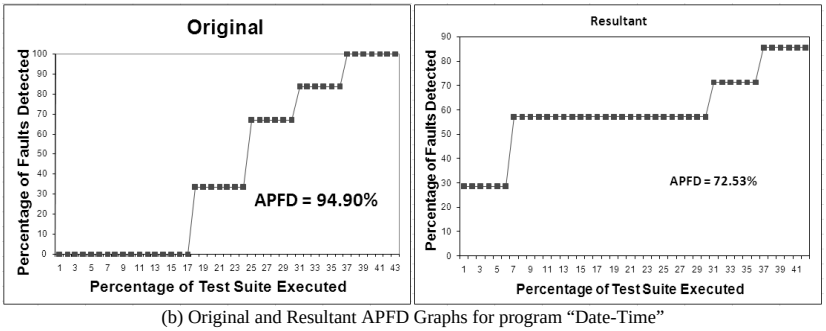


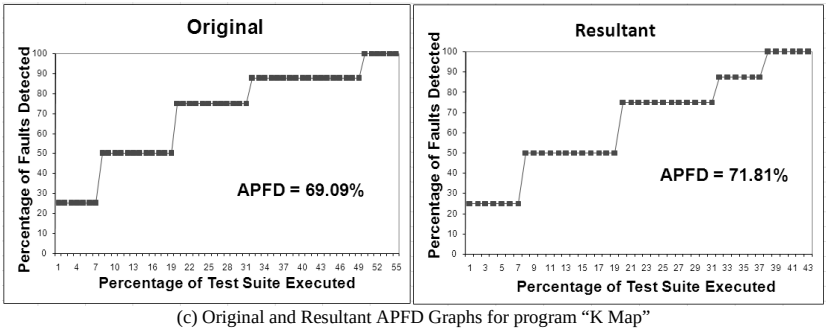
Fig. 5. Percentage of Test Cases for Modified Coverage



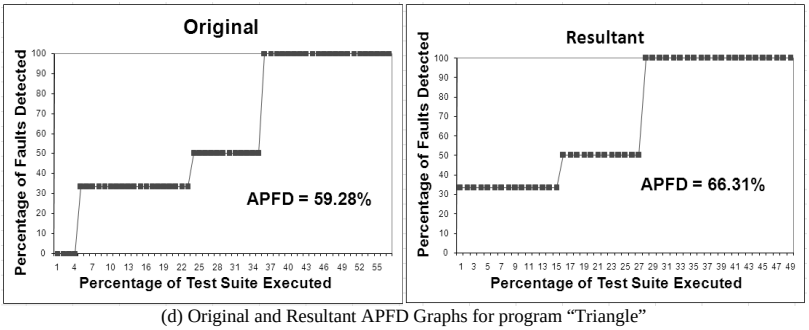
(a) Original and Resultant APFD Graphs for program "Calendar"



(b) Original and Resultant APFD Graphs for program "Date-Time"

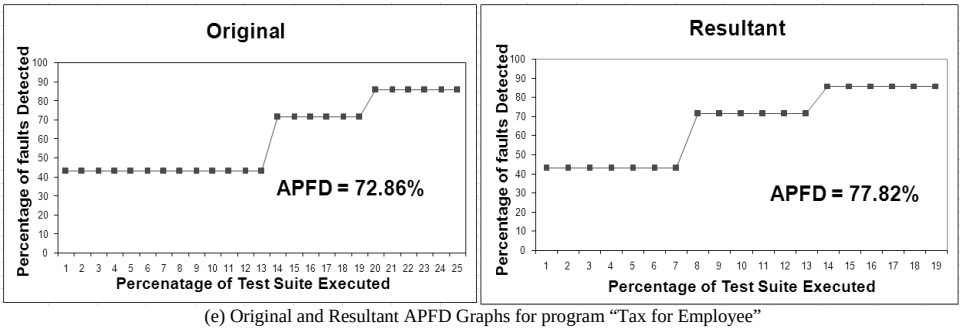


(c) Original and Resultant APFD Graphs for program "K Map"



(d) Original and Resultant APFD Graphs for program "Triangle"

Fig. 6. APFD Graphs for Original and Resultant Test Suites for respective programs



(e) Original and Resultant APFD Graphs for program "Tax for Employee"

Fig. 6. Continued

APFD graphs for the original and resultant test suite for the programs. It shows the effectiveness of the technique in terms of the percentage of faults detected in the least possible time.

6. APPLICATION OF TECHNIQUE

Software practitioners may use the technique developed to reduce the time and effort required for regression testing. This technique may lead to greater savings when applied to large and complex programs as compared to small and simple programs. The application of this work may improve the quality, reliability, and effectiveness of the code, which may, in turn, increase the level of customer satisfaction.

7. THREATS TO VALIDITY

On carefully analyzing the behavior of the two techniques, we observed that the proposed technique gives better results for the programs containing intensive variable computations. Further, the technique does not build the new test cases required for the code added due to modification. Moreover, the types of decision statements may affect the percentage of coverage achieved. Coverage depends on the type of decision statements: Some decisions are taken after the execution such as in "do…while," and "for," and some before execution such as "while." There are other options available in the programming language such as "switch statement," "multiple condition decision statement," "if…else," and so on, which give different coverage for the same test case.

8. CONCLUSIONS

In this paper, we have proposed and validated a technique, which is an extension of an existing technique proposed by us in an analogous study. The technique proposed in this work is compared with a technique given in literature by Rothermal et. al.[8]. The main results of this work are:

- Numbers of test cases selected are less for the proposed technique than the compared one.
- The technique selected less number of test cases as compared to other technique.
- The rate of fault detection using the technique is higher for the resultant test suite

REFERENCE

[1] K.K. Aggarwal, Y. Singh, Software engineering programs documentation, operating procedures, third edition, New Age International Publishers, New Delhi, 2008.

[2] B.Beizer, Software testing techniques, Van Nostround Reinhold, New York, 1990.

[3] R. V. Binder, Testing object- oriented systems Reading, Mass.: Addison Wesley, 2000.

[4] S. Elbaum, P. Kallakuri, A. G. Malishevsky, G. Rothermel, S. Kanduri, Understanding the effects of changes on the cost-effectiveness of regression testing techniques. Journal of Software Testing, Verification, and Reliability 13(2) (June 2003) 65-83.

[5] Y.Chen, D. Rosenblum, K. Vo., Test tube: A system for selective regression testing, in: Proceedings of the 16th International Conference on Software Engineering, Los Alamitos, Calif., 1994, pp.211-220.

[6] K. Fischer, F. Raji, A. Chrusciki, A methodology for retesting modified software, in: Proceedings of the National Telecommunications Conference B-6-3 (November 1981) 1-6.

[7] R. Gupta, M. J. Harrold, M. Soffa, An approach to regression testing using slicing, in: Proceedings of the Conference on Software Maintenance, 1992, pp. 299-308.

[8] G. Rothermel, Efficient effective regression testing using safe test selection techniques, PhD thesis, Clemson University, 1996.

[9] J.Bible, G. Rothermel, D. Rosenblum, Coarse- and fine-grained safe regression test selection. ACM Transactions on Software Engineering and Methodology 10 (2), (2001) 149-183.

[10] T. Graves, M. J. Harrold, J. M. Kim, A. Porter, G. Rothermel, An empirical study of regression test selection techniques, in: Proceedings of the 20th International Conference on Software Engineering, IEEE Computer Society Press, Kyoto, Japan, 1998, pp.188-197.

[11] G. Rothermel, M. J. Harrold, Empirical studies of a safe regression test selection technique, IEEE Transactions on Software Engineering 24(6) (1998) 401-419.

[12] G. Rothermel, M. J. Harrold, J. Dedhia, Regression test selection for C++ programs, Software Testing, Verification and Reliability 10(2) (2000) 77-109.

[13] S. Elbaum, A. G. Malishevsky, G. Rothermel, Test case prioritization: A family of empirical studies, IEEE Transactions on Software Engineering 28(2), (February 2002), pp.159-182.

[14] S. Elbaum, G. Rothermel, S. Kanduri, A. G. Malishevsky, Selecting a cost-effective test case prioritization technique. Software Quality Journal 12(3) (2004) , pp.185-210.

[15] G. Rothermel, R. H. Untch, C. Chu, M. J. Harrold, Prioritizing test cases for regression testing, IEEE Transactions on Software Engineering 27(10) (October 2001) 929-948.

[16] J. M. Kim, A. Porter, A history-based test prioritization technique for regression testing in resource constrained environments, in: Proceedings of the 24th International Conference on Software Engineering, Orlando, Fla., (2002) 119-129.

[17] J. A. Jones, M. J. Harrold, Test-suite reduction and prioritization for modified condition/decision coverage, in: Proceedings of the International Conference on Software Maintenance, Florence, Italy, (2001) 92-101.

[18] H. Srikanth, Requirements-based test case prioritization, Student Research Forum in 12th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, Newport Beach, Calif, 2004.

[19] D.Jeffrey, N. Gupta, Test case prioritization using relevant slices, in: Proceedings of Computer Software and Applications (COMPSAC'06), Chicago, (2006) 411-420.

[20] B. Korel, G. Koutsogiannakis, L.H. Tahat, Model –based test prioritization heuristic methods and their evaluation, in: the proceedings of the 3rd International Workshop on Advances in Model-based Testing, London, UK, (2007) 34-43.

[21] R. Krishnamoorthi, S.A. Sahaaya, Factor oriented requirement coverage based system test case prioritization of new and regression test cases, Information and Software Technology 51, (2009) 799-808.

[22] K.K. Aggarwal, Y. Singh, A. Kaur, Code coverage based technique for prioritizing test cases for regression testing, ACM SIGSOFT Software Engineering Notes 29 (5) (September 2004).

[23] W. E. Wong, J. R. Horgan, S. London, H. Agrawal, A study of effective regression testing in practice, in: Proceedings of the 8th IEEE International Symposium on Software Reliability Engineering, (1997) 264-274.

[24] R. Gupta, M. L. Soffa, Priority based data flow testing, IEEE ICSM, 1995, pp.348-357.

[25] Y. Singh, A. Kaur, B. Suri, Regression Test Selection and Prioritization Using Variables - Analysis and Experimentation. Software Quality Professional, Vol.11, No.2, (March, 2009), pp.38-51.

[26] C. Kaner , J. Bach, and B. Pettichord, Lessons learned in software testing, John Wiley and Sons, New York, 2002.

[26] Hierons, R.M., M. Harman, C.J. Fox, Branch coverage testability transformation for unstructured

programs, The Computer Journal, Oxford university press, 48(4), (2005) 421-436.
[27] Gregg Rothermel, Roland H. Untch, Chengyun Chu, Mary Jean Harrold, Test Case Prioritization: An Empirical Study, in: Proceedings of the International Conference on Software Maintenance, Oxford, UK, (September, 1999) 179-188.
[28] R. A. DeMillo, R. J. Lipton, and G. Sayward, Hints on test data selection: Help for the practicing programmer. Computer 11(4) , (1978) 34-41.
[29] A.F. Offutt, Investigations of the software testing coupling effect, ACM Transactions on software engineering and methodology 1(1) (January, 1992) 5-20.
[30] A.G. Malishevsky, J.R. Ruthru, G. Rothermel S.Elbaum, Cost-cognizant Test Case Prioritization, Technical Report TR-UNL-CSE-2006-0004, Department of Computer Science and Engineering, University of Nebraska, Lincoln, Nebraska, U.S.A., March, 2006.

Yogesh Singh

He is a professor with the University School of Information Technology (USIT), Guru Gobind Singh Indraprastha University, India. He is also Controller of Examinations with the Guru Gobind Singh Indraprastha University, India. He was founder Head and dean of the University School of Information Technology, Guru Gobind Singh Indraprastha University. He received his master's degree and doctorate from the National Institute of Technology, Kurukshetra. His research interests include software engineering focusing on planning, testing, metrics, and neural networks. He is co-author of a book on software engineering, and is a Fellow of IETE and member of IEEE. He has more than 200 publications in international and national journals and conferences.

Arvinder Kaur

She is an Associate Professor with the University School of Information Technology, Guru Gobind Singh Indraprastha University, India. She obtained her doctorate from Guru Gobind Singh Indraprastha University and her master's degree in computer science from Thapar Institute of Enggineering and Technology. Prior to joining the school, she worked with B.R. Ambedkar Regional Engineering College, Jalandhar and Thapar Institute of Enggineering and Technology. She is a recipient of the Career Award for Young Teachers from the All India Council of Technical Education, India. Her research interests include software engineering, object-oriented software engineering, software metrics, software quality, software project management, and software testing. She also is a lifetime member of ISTE and CSI. Kaur has published 40 research papers in national and international journals and conferences.

Bharti Suri

She is an Assistant Professor at the University School of Information Technology, Guru Gobind Singh Indraprastha University, Kashmere Gate, India. She holds masters degrees in computer science) and information technology. Her areas of interest are software engineering, software testing, software project management, software quality, and software metrics. Suri is a lifetime member of CSI. She was co-investigator of University Grants Commission (UGC) sponsored Major Research Project (MRP) in the area of software testing. She has many publications in national and international journals and conferences to her credit.

APPENDIX 1:
Related Terminology

P_e - Original program (before modification)

P_e' - New program (after modification)

T_e - Original test suite having six columns as testcaseID, variable name, function name, input and output

$T_e°$ - Temporary set of test cases

T_e' - Resultant test suite having both priority1 and priority2 assigned

S_e, S_e', X_e - The intermediate sets with entries like (v,f) where v is variable name and f is function name, computed at different steps of algorithm

V_e - A set consisting of variables from the changed lines with their function name

CVT_e - A two dimensional array with the elements of type (v,f) in first column where 'v' is variable name and 'f' is the function name. The second column is the list of variables computed from v.

CLB_e - A three dimensional array with the following fields: Changed line number, function name of older version and a bit. The bit is 0 if the line is an inserted line and the bit is 1 if the line is deleted or modified.

VDC_e - Variable dependency count is a two dimensional array with the first column consisting of the element (v,f) and the second column is the dependency count (count of the number of times it is used as an operand).

Priority - If the priority values for two different test cases are i and j, respectively, and i<j, then the test case with priority i will be executed earlier than the test case with priority j (i and j are positive integers). Thus, priority i is higher than priority j.

Priority1 - The priority assigned to test cases corresponding to the variable in the CVT_e.

Priority2 - The priority assigned to test cases (within Priority1) on the basis of VDC_e.

APPENDIX 2:
Algorithm (Interprocedural)

1. T_e'= ϕ
   //Create V_e; update VDC_e, CVT_e
2. For i = 1 to r // r is number of entries in CLB
   i) For all variables v in line CLB (i, 1)
   $V_e = V_e \cup \{v, CLB_e(i,2)\}$
   ii) If $CLB_e(i,3) = 1$
   Then
   If some variables are deleted that are not used anywhere in new version
   Then
   Delete the row from test case table corresponding to this variable and change array V_e, VDC_e and CVT_e accordingly
   Endif
   Else
   Insert corresponding rows in V_e,VDC_e and CVT_e End If
   End for

// Test case selection and priority1 assignment

3. For l = 1 to total number of entries inV_e:
   i) i =1
   ii) For j = 1 to t // t is number of entries in T_e
   //here we find test case corresponding to each entry (v,f) € V_e, where v correspond to variable name and f to function name
   If $T_e(j,2) == V_e(l,1)$ && $T_e(j,3) == V_e(l,2)$
   Then
   T_e' = T_e' U test case corresponding to entry j
   $T_e(j,6) = 1$ //priority1 assignment
   End if
   End for
   iii) S_e={(v , function name of v)}
   iv) Repeat while flag = false
   // this loop gets the variables computed from variables in S_i and assign priority1 from 2 onwards
   begin
   a) From CVT_e find elements that are computed using variables in S.
   b) If no such entry is found
   Then
   Set flag = true
   Else
   Set $S_e = S_e \cup$ {all new elements found}
   $i = i + 1$
   For each variable (w, f) € S_e
   Select test cases T° corresponding to element (w, f) and assign priority1 to i.
   m = number of test cases in T_e°
   For n = 1 to m
   If T_e°(n) € T_e'
   then
   Set T_e'(n,6) = min (i,T_e'(n,6))
   Else
   T_e'= T_e' U T_e°(n).
   End if
   End for
   End for
   End if
   End
   End for

// Priority2 assignment

4. For k = 1 to max (priority1) //k is for priority1
   i) S_e'= ϕ
   ii) For i=1 to t' // t' = number of test cases in T'
   iii) If T_e'(i, 6) = k
   Then
   $S_e' = S_e' \cup \{(T_e'(i,2), T_e'(i,3))\}$ //S' contain entries such that the corresponding test case has priority1=k
   End if
   End for
   iv) j =1, $X_e = \phi$ // j is for assigning priority2
   v) Repeat while $S_e' \neq \phi$
   begin
   a) Select the elements belonging to S' having maximum dependency count (from VDC) and insert inX_e.
   b) For i=1 to t'
   If (T_e'(i,2), T_e'(i,3)) € X_e
   Then
   T_e'(i,7) = j
   End if
   End for
   //Remove those entries from S' for which priority2 has been assigned
   c) $S_e' = S_e' - X$, $j = j + 1$
   end
   vi) $k = k + 1$
   End for
5. Exit