Efficient Path Selection Strategy Based on Static Analysis for Regression Testing

Mrinal Kanti Debbarma, Shailesh Tiwari, and A. K. Misra

***Abstract*—Regression testing is used to revalidate modified program and provide confidence that changes does not harm the behavior of the existing code. Test suites grows in size as software evolve, a simple approach of regression testing is re-test all approach in which all the pre-existing test suites are executed on the code but it is too expensive and increase the cost of testing activity. Different problems have been involved with regression testing, e.g. test suites minimization problem, test selection problem, coverage identification problem, test case execution problem, test case maintenance problem etc. Another problem may occur, when tester has to select the changed paths from the set of modified paths for test case execution. This paper presents a new path selection strategy based on static analysis for regression testing which enables the tester to execute the test cases in an order that increases their effectiveness to find faults taking minimum efforts. With the proposed approach, tester can select the paths among the set of paths in an order to achieve the testing objective. Infeasible paths are also identified by the proposed approach which can reduce the effort, time, and cost.**

***Index Terms*—Regression testing, software complexity metrics, path selection strategy.**

I. Introduction

Software development experiences shows that it is difficult to set measurable targets when developing software products. Produced/developed software has to be reliable and maintainable. On the other side, “You cannot control what you cannot measure” [1]. To avoid this, regression testing is performed during changes are made to existing software; the purpose of regression testing is to provide modified program without obstructing the existing, unchanged part of the software [2].

Software systems are maintained by developers by doing regression test periodically in hope to find errors caused by changes and provide confidence that modifications made in the software are correct. Developers/testers often create an initial test suite and then reuse it for regression testing [3]. These initial test suites are generally saved by the developers in order to reuse these test suites in regression testing as their software evolves. This reuse of test suites is pervasive in software industries [4]. A variety of selective regression testing strategies are proposed to select an appropriate subset of test cases from previously run test suites, based on the changes made to the software system for enhancement. The most common approach to this problem is simply execute the existing test cases in the test suite; i.e. retest all approach, a systematically selected subset is chosen to be reuse, then substantial resources will be saved, due to the limited size of the selected test data [5], [6].

There are distinctions between classes of techniques of regression testing: Test suites minimization techniques can be used to reduce test suite by eliminating redundant test suites. In safe approaches, every test case is selected that exercised any program element which could be affected by a program modification. In coverage approaches, test cases are selected which met some structural criteria. In this approach, select a single test case satisfying each coverage requirement introduced by the criterion, minimization techniques work like coverage approaches [7].

During maintenance phase of software lifecycle, regression testing tasks comprise a significant percentage of the costs of software testing as cost always increases when modifications are made in later stages of software development. A key difference between regression testing and development testing is that during regression testing a well-established test suite is available for reuse. One necessary strategy for regression testing is retest-all strategy i.e. re-tests all the test of test suite but it may consume excessive time and resources which may lead to increase in cost. On the other side, regression test selection strategy reduces the time required to retest a modified program by selecting some subset of the existing test suite. Therefore the methods that reduce the cost of regression testing tasks are always valuable. Most recent researches in regression testing concerns selective retest techniques. Various regression test selection strategies are described in [7].

One important approach is considered as coverage approach in which test cases are assure and met with structural coverage criterion. To satisfy this criterion, the execution of complete paths that cover the required test data component is necessary; we select those paths that cover a given required component. Our goal is to identify ‘best paths’ from the set of covered paths which is infinite or greater number [8]. The term ‘best’ is related to some program characteristics which can influence the testing activities: (i) complexity [9] (ii) NPATH [10] (iii) feasibility [11] and (iv) Halstead software science [12], [13]. These program characteristics plays vital role to achieve testing objective. Selection of paths is depends on tester’s perception whether the ease of data generation is required orincrease efficacy is required or in between these two. These perceptions of tester are in view of overall time given for testing activity.

Following assumption are made to simplify the target problem:

Let T be the previously run test suite, saved for reuse in

Manuscript received September 13, 2012; revised November 26, 2012.

M. K. Debbarma is with the Department of Computer Science and Engineering, National Institute of Technology, Agartala, Tripura, PIN 799055, India (e-mail: mrinal@nita.ac.in).

S. Tiwari and A. K. Misra are with the Computer Science and Engineering Department of Motilal Nehru Institute of Technology Allahabad, PIN 211-004, India, (e-mail: shail.tiwari@hotmail.com, akm@mnnit.ac.in).

regression testing and $\{T_1, T_2, \ldots\ldots T_n\}$ are the test cases belongs to T. Let P be the set of paths in modified program which may be finite or greater in number i.e. $\{P_1, P_2, P_3, \ldots. P_n\}$. Our main objective is to run the test cases on different paths in order to achieve various parameters given in literature: branch, statement, paths and dataflow coverage etc. It is not possible to exhaustive testing by rerunning the test cases on every path if the paths are infinite or greater in number. Tester has its own objectives to perform testing activity; in this paper two of these objectives are considered: making test data generation easier and enhanced efficacy is required. To perform testing activity totally depends on tester. Problem may arise how a tester selects the path that meets with the defined testing objective defined by [14].

This paper presents an approach by which the tester selects the modified path. Path selection strategy is used to achieve the tester's objective.

II. Background Details

A. Regression Testing

Regression testing is a process to uncover errors by partially retesting a modified program. Regression testing is done after modification is made in the implemented program. This can be done by rerunning the existing test suites against the modified code to determine whether the changes affects anything that worked properly prior to the change or writing new test cases where necessary. Adequate coverage should be primary consideration when conducting regression tests.

For simplification: Let P be a program and P' be a modified version of program P; let *T* be a set of test cases for P then *T'* is selected from *T* that is subset of T for executing *P'*, establishing *T'* correctness with respect to *P'*, if necessary, create *T''* and execute *T''* on *P'*, establishing *T''* correctness with respect to P', if necessary, create *T'''* and execute *T''* on *P'*, establishing *T'''* correctness with respect to *P'*. Each of these steps is involved with some problems of selective retest technique: Regression test selection problem, Coverage identification problem, Test suite execution problem and Test suite maintenance problem.

B. Software Complexity Metrics: Program Characteristics

Structural testing criteria consider on the knowledge of the internal structure of the program implementation to derive the testing criteria. To identify all possible executionpaths through the software programming skill is essential. The tester select test case input to use paths through the code and determines the coverage gained. Test cases are generated for actual implementation, if there is some change in implementation then it leads to change in test cases. They can be classified as Control flow, complexity and data flow based criteria. For the control flow based criteria, testing requirements are based on the Control Flow Graph (CFG). It requires the execution of components (blocks) of the program under test in condition of subsequent elements of the CFG i.e. nodes, edges and paths. The complexity based criterion requires the execution of all independent paths of the program; it is based on McCabe's complexity concept [12]. Another method is number of unit tests needed to test every combination of paths in a method. In Data Flow based criteria, both data flow and control flow information are used to perform testing requirements [15]. These coverage criteria are based on code coverage. Code coverage/Test coverage is the degree to which source code of a program has been tested. Test coverage is measured during test execution. Once such a criterion has been selected, test data must be selected to fulfill the criterion.

It is usually impossible to test all the paths in a program because it may be possible that program contains an infinite or greater number of paths. Path selection criteria given in the literature has some weaknesses that these criteria cannot assure that set of test data are capable of uncovering all errors will be chosen. Therefore, a practical path selection criterion which specifies a finite subset of paths and adequacy is needed to bring closer establishing correctness [9].

This paper presents analysis of selection of 'best paths', by using path selection technique which is our main objective. Following software complexity metrics are taken as program characteristics which can control the testing activity.

1) Complexity

Complexity of software is measuring of code quality; it requires a model to convert internal quality attributes to code reliability. High degree of complexity in a component (function, subroutine, object, class etc.) is bad in comparison to a low degree of complexity in a component is considered good. Various internal code attributes that are used to indirectly assess code quality. Software complexity measures which enables the tester to counts the acyclic execution paths through a component and improve software code quality. In a program characteristic that is one of the responsible factors that affect the developer's productivity [9] in program comprehension, maintenance, and testing. There are several methods to calculate complexity measures were investigated, e.g. different version of LOC[9], NPATH [10], McCabe's cyclomatic number [12], Data quality [12], Halstead's software science [12], [13] etc.

Low degree of complexity in a component is considered good as it affects the developer's productivity. If a path hashigh degree of complexity then there may be a greater probability of containing errors in that [2]. Tester can select the path with the greatest weight or with the least weight of LOC which depend on tester's observation whether the ease of test data generation is required or improved efficacy is required [8].

All the complexity weights for all paths of are saved that can be used to select the paths if software evolves.

2) NPATH evaluation of control flow graph

The control flow measure by NPATH, invented by Nejmeh [10], it measures the acyclic execution paths, NPATH is a metric which counts the number of execution path through a functions. One of the popular software complexity measures NPATH complexities (NC), is determined as:

$NPATH == \sum_{i=1}^{i=n} NP(\,statement_i)$

$NP\ (if) = NP\ (expr) + NP\ (if\text{-}range) + 1$

$NP\ (if\text{-}else) = NP\ (expr) + NP\ (if\text{-}range) + NP\ (else\text{-}range)$

$NP\ (while) = NP\ (expr) + NP\ (while\text{-}range) + 1$

$NP\ (do\text{-}while) = NP\ (expr) + NP\ (do\text{-}range) + 1$

$NP\ (for) = NP\ (for\text{-}range) + NP\ (expr1) + NP\ (expr2) +$
$NP\ (expr3) + 1$

$NP\ (\text{"?"}) = NP\ (expr1 + NP\ (expr2) + NP\ (expr3) + 2$

$NP\ (repeat) = NP\ (repeat\text{-}range) + 1$

NP (*switch*)= *NP* (*expr*)+$\sum_{i=1}^{i=n} NP(case - range)$+ *NP* (*default-range*)
NP (*function call*)=1
NP (*sequential*)=1
NP (*return*)=1
NP (*continue*)=1
NP (*break*)=1
NP (*goto label*)=1
NP (*expressions*)=*Number of* && and || *operators in expression*

Fig. 1. NPATH complexity measure.

where '*N*' represents the number of statements in the body of component (function) and '*NP*' (Statement) represents the acyclic execution path complexity of statement *i*. Here '(*expr*)' represents expression which is derived from flow-graph representation of the statement. NPATH measure follows:

if (*a*>*b*)
z=*a*;
else
z=*b*;
The NPATH value is 2 as follows:
NP(<*if-else*>)=*NP*(<*expr*>)+*NP*(<*if-range*>)+*NP*(<*else-range*>)

3) Feasibility

A path is feasible if there is an input datum for these paths to be executed. In contrast, a path is said to be infeasible if there is no set of values for the input test data that cause path to be executed [11]. Identify infeasible paths is an undecidable question [4]. If a path contains lower number of predicates then it has greater probability of being feasible. On the other side, if a path consists of greater number of predicates then it may have greater probability of finding out errors in the program. Predicate is considered as simple Boolean form in condition. So, when the paths having few predicates are selected then the numbers of infeasible paths are reduced.

4) Halstead software science

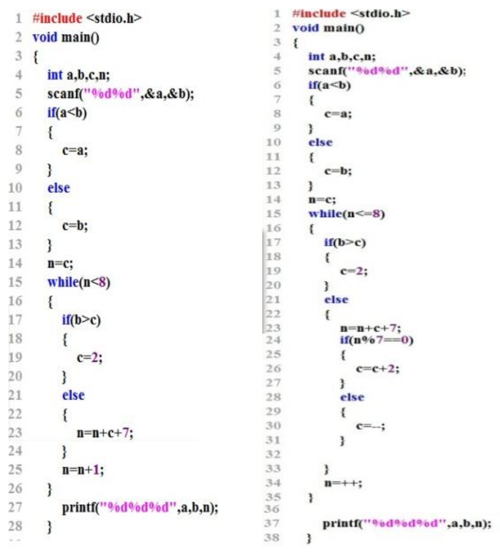
Another alternative software complexity measures have to be considered. M. Halstead's Software science measures [6] are very useful. Halseatd's software science is based on an enhancement of measuring program size by counting lines of code. Halstead's metrics measure the number of number of operators and the number of operands and their respective occurrence in the program (code). These operators and operands are to be considered during calculation of Program Length, Vocabulary, Volume, Potential Volume, Estimated Program Length, Difficulty, and Effort and time by using following formulae:

n_1: number of unique operators,
n_2: number of unique operands,
N_1: total number of operators, and
N_2: total number of operands,
Program Length (N) =N_1+N_2
Program Vocabulary (n) =n_1+n_2
Volume of a Program (V) =$N \times log_2 n$
Potential Volume of Program,
(V^*) = (2+n_2) log_2 (2+n_2)
Program Level (L) =L=V^*/V
Program Difficulty (D) =1/L
Estimated Program Length (N) =$n_1 log_2 n_1$+$n_2 log_2 n_2$
Estimated Program Level (L) =2n_2/ ($n_1 N_2$)
Estimated Difficulty (D) =1/L=$n_1 N_2$/2n_2
Effort (E) =V/L=$V \times D$= (n_1 x N_2) / 2n_2

III. Our Approach

Our approach deals with analysis of path selection problem. Four program characteristics are considered from the literatures that are responsible for software complexity measures to analyze the paths. Tester can select paths from given program characteristics. Weights of each program characteristics are evaluated for each path. Selection of paths is depending on interest of tester, so tester can select paths according to testing objective.

Let *P* be the old version of program and *P*' be the new version of program in '*C*' given in Fig. 2.



```
#include <stdio.h>
void main()
{
   int a,b,c,n;
   scanf("%d%d",&a,&b);
   if(a<b)
   {
      c=a;
   }
   else
   {
      c=b;
   }
   n=c;
   while(n<8)
   {
      if(b>c)
      {
         c=2;
      }
      else
      {
         n=n+c+7;
      }
      n=n+1;
   }
      printf("%d%d%d",a,b,n);
   }
```

```
#include <stdio.h>
void main()
{
   int a,b,c,n;
   scanf("%d%d",&a,&b);
   if(a<b)
   {
      c=a;
   }
   else
   {
      c=b;
   }
   n=c;
   while(n<=8)
   {
      if(b>c)
      {
         c=2;
      }
      else
      {
         n=n+c+7;
         if(n%7==0)
         {
            c=c+2;
         }
         else
         {
            c=--;
         }

      }
      n=++;
   }

      printf("%d%d%d",a,b,n);
   }
```

Fig. 2. Program *P* and *P*'

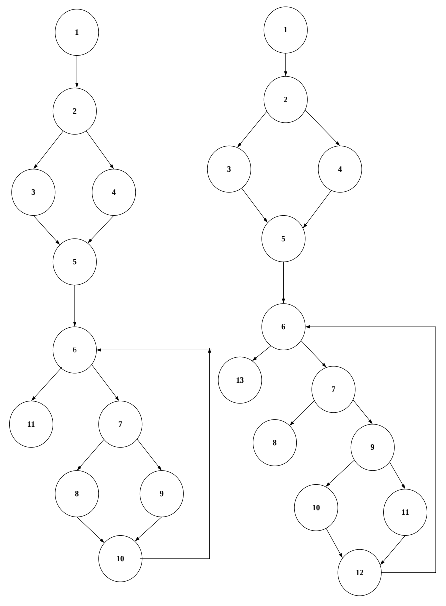


Fig. 3. CFG for program *P* and *P'*

TABLE I: COMPUTED WEIGHTS SOFTWARE COMPLEXITY METRICS FOR PROGRAM *P*.

Path #	Contents of Path	LOC	NPATH	Predi-cate	Software Science	
					D	*E*
P^1	{1,2,3,5,6,7,9,10, 6,11}	22	6	4	24	6663
P^2	{1,2,3,5, 6,7,8,10,6,11}	24	6	4	21	7707
P^3	{1,2,4,5,6,7,9,10, 6,11}	20	4	4	21	5691
P^4	{1,2,4,5,6,7,8,10, 6,11}	22	3	4	19	5491
P^5	{1,2,3,5,6,11}	14	4	2	17	3553
P^6	{1,2,4,5,6,11}	12	4	2	15	2505

The structure of a program P ad P' can be represented by a control flow graph in Fig. 3, $G(P) = \{N, E, s, e\}$, where *N* is a set of nodes representing basic blocks of code or branch points in the function; *E* is a set of edges representing flow of control in the function; s is the unique entry node and e is the unique exit node. At first, all paths are identified from the graph then weights for complexity, sensibility are evaluated and saved for each path.

TABLE II: COMPUTED WEIGHTS SOFTWARE COMPLEXITY METRICS FOR PROGRAM *P'*

Path #	Contents of Path	LOC	NPATH	Predi-cate	Software Science	
					D	*E*
P^1	{1,2,3,5,6,7,9, 11,12,6,13}	26	8	5	36	11844
P^2	{1,2,3,5, 6,7,8,12,6,13}	24	6	4	22	5214
P^3	{1,2,4,5,6,7,9, 11,12,6,13}	24	4	5	36	11232
P^4	{1,2,4,5,6,7,9, 10,12,6,13}	21	8	5	31	10943
P^5	{1,2,4,5,6,7,8, 12,6,13}	22	8	4	25	6250
P^6	{1,2,3,5,6,13}	14	4	2	19	3496
P^7	{1,2,4,5,6,7,8, 12,6,13}	14	4	2	19	3420

Our main objective for path selection criteria is to select the 'best' paths and this selection is made on the following program characteristics which can influence the testing activity:

Complexity of each path can be calculated by using LOC, NPATH found in each node or path. The degree to which characteristics that hamper software maintenance are present is called software maintainability. Software complexity measures how difficult the program is to work with. It includes understandability, modifiability, and testability of software. Various approaches may be taken in measuring complexity characteristics given in literature, e.g. NPATH[10], McCabe's cyclomatic number[12], LOC[9], Data quality[9], Halstead's software science[15] etc.

In this paper, we work with two software science measures; they are the difficulty and effort measure.

One major weakness of this complexity is that they do not measure control flow complexity and difficult to compute during fast and easy computation. As it affects the developer's productivity so if a path has low complexity the ease of test data generation is achieved. In contrast, if a path has high complexity then there may be a greater probability of containing errors in that [8].

In this regard, the tester can select the path with the greatest weight or with the least weight of LOC which depend on tester's perception whether the ease of test data generation is required or enhanced efficacy is required [4].

All the complexity weights for all paths of new version of program *P*' are evaluated.

Number of predicates in each path are identified and saved which help tester to distinguish between feasible paths and infeasible paths. If a tester selects feasible paths then the ease of test data generation is achieved and if complex paths are selected by tester then efficacy is increased.

From the above Table I and Table II, it is clear that the path can be feasible if it contain lower number of predicates. It means that out of these six identified paths from Table I. P^5, P^6 have greater probability of being feasible among all the paths and from Table II. P^6 and P^7 have greater probability of being feasible among all the paths.

IV. CONCLUSION

Path selection strategy to be used from the set of paths to achieve the testing objective and tester can reduce cost, time and effort to this activity. Software characteristics play vital role in path selection strategy. Characteristics used here for path selection was complexity, testability and feasibility. In addition , it was observed that If tester's objective is to achieve ease of test case generations then those paths are selected in which LOC, NPATH, Halstead's difficulty, effort and predicates found in path are lesser. If tester's objective is to increase the efficacy then those paths are selected in which LOC, NPATH, Halstead's difficulty, effort and predicates found in a path are greater.

Tester can use this approach to execute the test case on the selected paths. By the use of path selection strategy, infeasible paths are also identified which can reduce the time, effort, and cost. The strategy also eases the process of regression testing without affecting the quality of software testing. The proposed work is the extension of path selection strategy for regression testing [3].

REFERENCES

[1] D. Marco, *Controlling Software Projects*, Prentice Hall, New York, 1982

[2] S. Yoo and M. Harman, "Regression testing minimisation, selection and prioritisation - a survey," *Technical Report TR-09-09*. October 26, 2009, pp. 1-80

[3] S. Tiwari, K. K. Mishra, A. Kumar, and A. K. Misra, "Path selection strategy for regression testing," *Presented at the SERP 2010 in World Comp 2010*, Las Vegas, July 12-15, 2010.

[4] P. G. Frankl and E. J Weyuker, "An application family of data flow testing criteria," *IEEE Transactions on software Engg*, vol. 14, no. 10, pp. 1483-1498, October, 1988.

[5] D. S. Rosenblum and E. J. Weyuker, "Using coverage information to predict the cost effectiveness of regression testing strategies," *IEEE Transaction on Software Engg.*, vol. 23, no. 3, March 1997.

[6] R. Subramanyan and C. J. Budnik, "Test selection prioritization strategy," *33rd Annual IEEE International Computer Software and Applications Conference - Workshops*, 2009.

[7] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Trans. Software Eng.*, vol. 22, no. 8, pp. 529-551, Aug. 1996.

[8] L. M. Press *et al.*, "Path selection in the structural testing: proposition, implementation, and application of strategies," *IEEE XXI International Conference of Chilean Computer Science Society(SCCC'01)*

[9] S. D. Conte, H. E Dunsmore, and V. Y. Shen, "Software engineering metrics and models," *Benjamin/Cummings Publishing Company, Inc*., 1986.

[10] B. A. Nejmeh, "NPATH: A measure of execution path complexity and its applications," *Comm. of the ACM*, vol. 31, no. 2, pp. 188-210, February 1988.

[11] D. F. Yates and N. Malevris, "Reducing the effects of infeasible paths in branch testing," *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 48-54, December 1989.

[12] T. A. M. Cabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308-320, December 1976.

[13] A. Fitzsimmon and T. Love, "A review and evaluation of software science," *Computing Survey*, vol. 10, no. 1, March 1978.

[14] W. E. Howden, "Methodology for the generation of test data," *IEEE Trans. Computers*, vol. TC-24, May 1975.

[15] M. Halstead. *Elements of Software Science*. North-Holland, 1977.

[16] W. E. Wong *et al*., "A study of effective regression testing in practice," *IEEE Eighth International Symposium on Software Reliability Engg (ISSRE '97).*

[17] D. B. Wey, "Semantic differencing to reduce the cost of regression testing," *Proc. Conf. Software Maintenance-1992*, pp. 41-50, Nov. 1992.

[18] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, April 1997, pp. 173-210..

[19] N. Malevris, D. F. Yates, and A. Veevers, "Predictive metrics for likely feasibility of program paths," *Information and Software Technology*, vol. 32, no. 2, pp. 115-119, March 1990.

[20] S. Rapps and E. J. Weyuker, "Data flow analysis techniques for test data selection," *Proc. Int. Conf. Software Engineering*, Tokyo, Japan, September 1982.

[21] M. R. Woodward, D. Hedley, and M. A. Hennell, "Experience with path analysis and testing of programs," *IEEE Trans. Software Eng*., vol. SE-6, pp. 278-286, May 1980.

[22] T. L Graves *et al.*, "An empirical study of regression test selection techniques," *ACM Transaction on Software Engg. and Methodology*, vol. 10, no. 2, April 2001.

[23] E. Miller, "Coverage measure definitions reviewed," *Testing Techniques Newsletter*, vol. 3, no. 4, pp. 6, Nov. 1980.

[24] J. Voas, L. Morell, and K. Liller, "Predicting where faults can hide from testing," *IEEE Software*, pp. 41-47, March 1991.

[25] P. G. Frankl and E. J Weyuker, "An application family of data flow testing criteria," *IEEE Transactions on Software Engg*, vol. 14, no. 10, pp. 1483-1498, October, 1988.

[26] A. Pasala *et al.*, "Selection of regression test suite to validate software applications upon deployment of upgrades," *IEEE 19th Australian Conference on Software Engineering*, November 2008.

Mrinal Kanti Debbarma presently working as Assistant Professor at Computer Science & Engineering Department of National Institute of Technology Agartala, India, he received the B.Tech (IET, Lucknow), M.Tech (MNNIT, Allahabad), currently pursuing Ph.D.(Information Technology) at Assam University. His research interest include in Software Engineering with special interest in Regression testing, MANET Routing Protocols, Wireless Sensor Networks.He has published technical papers in various International Journals and Conferences.Mr. Debbarma is a member of IAENG, IACSIT. email: mrinal@nita.ac.in

Shailesh Tiwari is pursuing his Ph.D at Computer Science & Engineering Department of Motilal Nehru Institute of Technology Allahabad, PIN 211-004, India,(e-mail: shail.tiwari@hotmail.com)

Dr. A. K. Misra working as Professor, at Computer Science & Engineering Department of Motilal Nehru Institute of Technology Allahabad, India, He obtained Ph.D. (1990) from M.N.R.E.C., Allahabad, M.E. Hon's (1976) from M.N.R.E.C., Allahbad B.E. (1971) University of Roorkee, India. His current research interest includes Software Engineering, Programming Methodology, Artificial Intelligence. Dr. Misra earned Life Member of Computer Society of India, Indian Society of Technical Education., Institution of Engineers, India. (Fellow), Member of ACM. He has published more than 100 technical papers in various International Journals and Conferences. (e-mail: akm@mnnit.ac.in).