2012 International Conference on Computer Communication and Informatics (*ICCCI* -2012), Jan. 10 – 12, 2012, Coimbatore, INDIA

Static and Dynamic Software Metrics Complexity Analysis in Regression Testing

Mrinal Kanti Debbarma, Nirmalya Kar and Ashim Saha
Department Computer Science & Engineering
National Institute of Technology, Agartala, India
E-mail : mkdb06@gmail.com, nirmalya.kar@gmail.com, ashim.nita@gmail.com

***Abstract*— In maintenance, assuring code quality and operation, software metrics is widely used by the various software organizations. Software metrics quantify different types of software complexity like size metrics, control flow metrics and data flow metrics. These software complexities must be continuously calculated, followed and controlled. One of the main objectives of software metrics is that measures static and dynamic metrics analysis. It is always considered that high degree of complexity in a fragment is bad in comparison to a low degree of complexity in a fragment. Software metrics can be used in different phases of the software lifecycle. In this paper we will discuss the different metrics and comparison between both static and dynamic metrics. We try to evaluate and analyze different aspects of software static and dynamic metrics in regression testing which offers of estimating the effort needed for testing.**

Keywords: Software metrics, Regression testing, object-oriented programming,

I. INTRODUCTION

The Software complexity is based on familiar software metrics, this would be likely to reduce the time spent and cost estimation in the testing phase of the software development life cycle. Improving quality of software is a quantitative measure of the quality of source code. This can be achieved through definition of metrics, values for which can be calculated by analyzing source code. A number of software metrics widely used in the software industry are still not well understood [1]. Although some software complexity measures were proposed over thirty years ago and some others proposed later. Sometimes software growth is usually considered in terms of complexity of source code. Various metrics are used, which unable to compare approaches and results. In addition, it is not possible or equally easy to evaluate for a given source code [3]. Software complexity, deals with how difficult a program is to understand. In software maintainability, the degrees to which characteristics that hamper software maintenance are present and determined by software complexity.

Software development experiences shows that it is difficult to set measurable targets when developing software products. Developed software has to be reliable and maintainable. On the other side, "You cannot control what you cannot measure"[16]. 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 [6].

Software systems are maintained by developers by doing regression test periodically 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. These initial test suites are generally saved by the developers in order to reuse these test suites in regression testing as their software evolves.

II. BACKGROUND AND LITERATURE REVIEW

Regression Testing

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.

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 execution paths through the software programming skill is essential. The tester select test case input to use paths

978-1-4577-1583-9/ 12/ $26.00 © 2012 IEEE

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 [8]. 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 [14]. 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 [14].

This paper presents analysis based on static and dynamic on selection of 'efficient paths', by using path selection technique. Following software complexity metrics are taken as program characteristics which can control the testing activity.

Software Metrics vs Software Complexity:

Software metrics is defined by measuring of some property of a portion of software or its specifications. Software metrics provide quantitative methods for assessing the software quality. Software metrics can be define as: "the continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information together with the use of those techniques to improve its products and that process" [12].

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 fragment (function, subroutine, object, class etc.) is bad in comparison to a low degree of complexity in a fragment 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 fragment 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 [14], NPATH [10], McCabe's cyclomatic numbe r[8], Data quality [14], Halstead's software science [13,15] etc.

Various software complexity metrics invented and can be categorized into two types.

Comparison between static and dynamic metrics as follows:

(a) Static metrics

Static metrics are obtainable at the early phases of software development life cycle and deals with structural features of software.

These metrics does not deal with object oriented features, real- time systems and easy to gather.

Static complexity metrics estimate the amount of effort needed to develop, and maintain the code.

(b) Dynamic metrics

Dynamic metrics are accessible at the late stage of the software development life cycle. These metrics capture the dynamic behavior of the system and very hard to obtain and obtained from traces of code. Dynamic metrics supports all object-oriented features and real time systems.

Low degree of complexity in a fragment is considered good as it affects the developer's productivity. If a path has high 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 [2].

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

Control Flow Graph: NPATH Evaluation

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:

$NPATH=\prod_{i=1}^{i=N} NP(\text{statement}_i)$

NP(if)=NP(expr)+NP(if-range)+1

NP(if-else)=NP(expr)+NP(if-range)+NP(else-range)

NP(while)=NP(expr)+NP(while-range)+1

NP(do-while)=NP(expr)+NP(do-range)+1

NP(for)=NP(for-range)+NP(expr1)+NP(expr2)+

NP(expr3)+1
NP("?")=NP(expr1+NP(expr2)+NP(expr3)+2
NP(repeat)=NP(repeat-range)+1
NP(switch)=NP(expr)+$\sum_{i=1}^{i} NP(case - range)$+
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

Execution of Path Expressions (complexity expression) are expressed, 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. where *"(expr)"* represents expression which is derived from flow-graph representation of the statement. For example NPATH measure as follows:

```
Void func-if-else ( int c)
{
  int a=0;
  if(c)
   {
      a=1;
   }
        else
       {
       a=2;
       }
    }
```

The Value of NPATH = 2 as follows:
NP (if-else)=NP(expr)+NP(if-range)+NP(else-range)

In the above example, NP (exp)=0 for if statement.
NP (If-range)=1 for if statement and , NP(else-range)=1 for if-else statement. So, NP (if-else)=0+1+1=2.
NPATH, metric of software complexity overcomes the shortfalls of McCabe's metric which fail to differentiate between various kinds of control flow and nesting levels control structures.

***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 [9]. Identify infeasible paths is an undecidable question [11]. 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.

If A_1 are paths through a code module, suppose that A_1 consists of $Pred_1 > 0$ predicates and A_2 consists of $Pred_2 > Pred_1$ predicates respectively, then A_1 is more likely to be feasible than A_2 [9].

Halstead Metric
Another alternative software complexity measures have to be considered. M. Halstead's Software science measures [14] 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 by using following formulae:

n_1: number of unique operators
n_2: number of unique operands
N_1: total number of operators
N_2: total number of operands
Length (N) =N1+N2
Vocabulary (n) =n1+n2
*Volume of a Program (V) =N*log$_2$n*
Potential Volume (V) = (2+n_2) log2 (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) =$2n_2 / (n_1 N2)$
Estimated Difficulty (D) =1/L=$n_1 N_2 / 2n_2$
Effort (E) =V/L=V*D= (n_1 x N_2) / $2n_2$

Advantages and disadvantages of Halstead complexity: It does not require inside analysis of program's logic structure. Easy to compute, programming language independent. It defines the complexity from data stream only, while avoids the complexity from the control flow.

III. OUR APPROACH

Our approach deals with static and dynamic 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 after modification is given in Figure 1.

The code segment written in C (Program P and P') has been taken as an examples as following.

The structure of a program P and modified program P' can be represented by a control flow graph (CFG) in figure 2, such that 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, feasibility are evaluated and saved for each path. The comparison of static and dynamic software complexity metrics between old and new version of program is depend on LOC, NPATH, number of Predicates and Halstead's metrics (Difficulty and Effort).

```
    #include<stdio.h>                    #include< stdio.h>
    void main()                          void main()
1   {                                    1  {
1    int a,b,c,n;                        1  int a,b,c,n;
1    scanf("%d %d", & a,&b);             1   scanf ("%d%d", &a,&b);
2   if (a < b)                           2  if ( a < b)
2   {                                    2  {
3   c = a;                               3  c = a;
3   }                                    3  }
4    if(c= =b)                           4  if(c= =b)
4    {                                   4  {
5    c= =a+1;                            5    c= =a+1;
5    }                                   5  }
5   }                                    5   else
5   else                                 5  {
5   {                                    6  c= a - 1;
6   c=b                                  6  }
7   if  (c= =b)                          6  }
7   {                                    6  else
8   c=b+1;                               6  {
8   }                                    7  c=b;
8   }                                    8  if (c= = b)
9   n = c;                               8  {
10   while ( n < 8 )                     9  c = b + 1;
10   {                                   9  }
11    if ( b > c )                       9  else
11   {                                    9  {
12   c = 2;                              10  c = b – 1;
12    }                                  10  }
12   else                                11  n = c;
12   {                                   12  while (n<=8)
13    n = n + c +7;                      12  {
14    }                                  13  if (b>c)
14    n = n + 1;                         13  {
14   }                                   14   c = 2;
15   Printf("%d%d%d",a,b, n);            14  }
15   }                                   14  else
                                         14  {
                                         15  n = n+c+7;
                                         15  }
                                         16  n = n+1;
                                         16  }
                                     17  Printf("%d%d%d",a,b, n);
                                 17  }
```

Figure 1. Source Program P and modified program P'

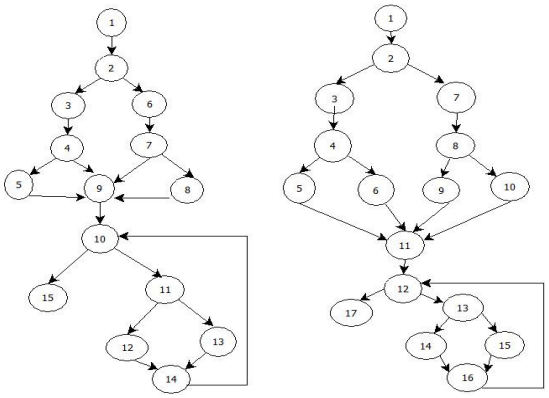

Figure 2. CFG for program P and P'

TABLE I. Computed weights Software metrics complexity for program P

Path #	Path Contents	LOC	NC	Pred.	Halstead metrics	
					D	E
P^1	{1,2,3,4,5,9, 10,11,12, 14,10,15}	30	12	4	31	10726
P^2	{1,2,3,5,9,10, 11,13,14, 10,15 }	27	12	4	35	12810
P^3	{1,2,4,9,10, 11,12,14, 10,15}	25	6	4	29	9309
P^4	{1,2,4,5,9, 10,15}	19	8	3	27	7118
P^5	{1,2,3,4,9, 10,15}	14	4	3	23	4692
P^6	{1,2,6,7,8,9,10, 11,12,14,10,15}	27	6	5	33	11781
P^7	{1,2,6,7,9,10, 11,12,14,10,15}	24	3	5	31	11470
P^8	{1,2,6,7,8, 9,10,15}	16	4	3	24	5904
P^9	{1,2,6,7,9, 10,15}	13	2	3	23	4508

TABLE II. Computed weights Software metrics complexity for program P'

Path #	Path Contents	LOC	NC	Pred.	Halstead metrics	
					D	E
P^1	{1,2,3,4,5, 11,12,13, 14,16,12,17}	28	12	4	33	11626
P^2	{1,2,3,4,5, 11,12,,13 15,16,12,17}	26	8	4	35	12764
P_3	{1,2,3,4,6,11,1 2,1314,16,12,1	29	12	4	33	11625

	7}					
P^4	{1,2,3,4,6,11,12,1315,16,12,17}	28	12	4	38	14250
P^5	{1,2,3,4,5,11,12,17}	18	8	3	25	6250
P^6	{1,2,3,4,6,11,12,17}	19	8	3	23	5267
P^7	{1,2,7,8,9,11,12,13,14,16,12,17}	26	6	5	30	10170
P^8	{1,2,7,8,9,11,12,13,15,16,12,17}	25	6	5	28	9240
P^9	{1,2,7,8,10,11,12,13,14,16,12,17}	25	4	5	33	10296
P^{10}	{1,2,7,8,10,,11,12,13,15,16,12,17}	23	4	5	35	12635
P^{11}	{1,2,7,8,9,11,12,17}	17	4	3	26	5798
P^{12}	{1,2,7,8,10,11,12,17}	15	2	3	24	5616

Complexity of each path can be calculated by using LOC, NPATH found in each node or path. Various approaches may be taken in measuring complexity characteristics given in literature, e.g. NPATH [10], McCabe's cyclomatic number [8], LOC [14], Data quality [14], Halstead's software science [15] etc.

In this paper, we deals 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 [2].

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 [11].

All the complexity weights for all paths of old and new version of program P and 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 nine identified paths from Table 1. P^4,P^5,P^8 and P^9 have greater probability of being feasible among all the paths and from table II. P^5,P^6,P^{11} and P^{12} have greater probability of being feasible among all the paths.

IV. CONCLUSION AND FUTURE WORK

Software characteristics play vital role in path selection strategy. Characteristics used here for path selection was complexity, testability and feasibility. If tester's objective is to achieve ease of test case generations then those paths are selected in which LOC, NPATH, Halstead metrics 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.

It is important to take into the consideration static and dynamic complexity metrics to find out the deviation in a program. Finally, the compared evaluated values from both version of program P and P', the changed performance of code is identified and tester can use this approach to execute test case and improve software productivity and software quality as well as execution point of view. If the complexity is higher, then tester has to modify the code. In the future study, more complicated program has to be measured with other complexity attributes. Empirical investigation is required to ascertain software cost, effort and time factors such as mainatanability.

REFERENCES

[1]. Letica 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).

[2]. Anjaneyulu 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.

[3]. David S. Rosenblum and Elaine 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.

[4]. T. L Graves et al. "An empirical study of regression test selection techniques." ACM Transaction on Software Engg. & Methodology, Vol. 10, No. 2, April 2001.

[5]. G. Rothermel and M.J. Harrold, "Analyzing Regression Test Selection Techniques", IEEE Trans. Software Eng., vol. 22, no. 8, pp. 529-551, Aug. 1996.

[6]. Shin Yoo & Mark Harman, "Regression Testing Minimisation, Selection and Prioritisation - A Survey", Technical Report TR-09-09

[7]. W. Eric Wong et.al, "A Study of Effective Regression Testing in Practice", IEEE Eighth International Symposium on Software Reliability Engg (ISSRE '97).

[8]. T.A. McCabe. A Complexity Measure. IEEE Transactions on Software Engineering, 2(4):308-320, December 1976.

[9]. D.F. Yates and N. Malevris. Reducing The Effects Of Infeasible Paths In Branch Testing. ACM SIGSOFT Software Engineering Notes, 14(8):48-54, December 1989.

[10]. B.A. Nejmeh. NPATH: A Measure of Execution Path

Complexity and Its Applications. Comm. of the ACM, 31(2):188-210, February 1988.

[11]. S. Rapps and E.J. Weyuker. Data Flow Analysis Techniques for Test Data Selection. Proc. Int. Conf. Software Engineering, Tokyo, Japan, September 1982.

[12]. T.DeMarco; Controlling Software Projects; Prentice Hall, NY, 1982.

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

[14]. S.D. Conte, H.E Dunsmore, and V.Y. Shen. "Software Engineering Metrics and Models". Benjamin/Cummings Publishing Company, Inc., 1986.

[15]. Ann Fitzsimmon and Tom Love" A Review and Evaluation of Software Science",Computing Survey,Vol. 10.No1, March 1978

[16]. E. Miller, "Coverage Measure Definitions Reviewed, "Testing Techniques Newsletter, Vol.3, No.4, Nov 1980, p.6

[17]. Shailesh Tiwari, K.K Mishra, Anoj Kumar and A.K Misra, " Path Selection Strategy for Regression Testing", SERP 2010 in World Comp 2010, Las Vegar, July 12-15, 2010.

[18]. M. K. Debbarma, Shailesh Tiwari and A. K Misra, " Efficient Path Selection Strategy based on Static Analysis for Regression Testing", 4th IEEE International Conference on Computer Sc. & Information Technology (ICCSIT), 2011, Chengdu, China, 10-12, June, 2011.

[19]. M. K. Debbarma, N. P. Singh, A. K. Shrivastava and Rishi Misra, " Analysis of Software Complexity Measures for Regression Testing", ACEEE International Journal on Information Technology (IJIT), Volume 1, Issue 2, September, 2011, Page: 14-18.