A Novel Test Coverage Metric for Safety-Critical Software

Debashis Mukherjee
Indian Institute of Technology Kharagpur
Kharagpur,India
debashis_mukherjee@yahoo.com

***Abstract*—Software is a crucial component of many safety-critical applications, such as nuclear power plant, fly-by-wire application, medical instruments, space applications etc. They need to satisfy safety standards such as safety integrity level (SIL), IEEE series (IEEE 1012, 1074), IEC series (IEC 60880, IEC 62138). A requirement of all these safety standards is thorough testing. Thorough testing is measured using a test coverage metric. However, as we show every existing metric suffers from several shortcomings. In this context, we propose a novel test coverage metric based on dependency of data and control in code of the components for safety-critical software.**

***Index Terms*—Test coverage, White-box testing, System dependence graph (SDG), Java system dependence graph (JSysDG)**

I. Introduction

Test coverage analysis of programs based on control flow, and data flow are considered the main types of white-box testing strategies [9, 11–13]. The control flow based techniques are more popularly used in automated test coverage analysis tools. Criteria of code coverage using statement coverage, and branch coverage, in control flow coverage, are available with almost every tool, whereas only few tools include coverage using popular data flow criteria, namely all-defs, all-uses, or all-du-paths. Constructs of code, executed using statements, branches, conditions, and paths on control flow graph (CFG) are the concerns of control flow coverage. In contrast, the constructs of coverage of data flow, correspond to data accessed by program variables, in expressions of predicates and statements in the code.

Coverage of the dependences of the statements in a code on a graph-based representation is considered in criteria of white-box test coverage. The dependences of data and control are represented with edges in a program dependence graph (PDG) [8], corresponding to vertices representing predicates and statements in a code of a procedure. The control dependences and data dependences represented in a PDG, is defined using a control flow graph (CFG) of the procedure. The coverages of data flow could be related with coverages of the data dependences. The set of data dependences of a program can increase even without an increase in size of the set of corresponding statements and expressions. The data dependences can increase with accesses and storage types of the variables. It can grow with the support of a runtime system, in programs of object-oriented programming paradigm. A data flow coverage criterion, could require execution of elements of control flow constructs, in combination, and repetition. Similarly, data produced by variables of boolean expressions, conditions and predicates, are considered in combination in control flow coverage.

A main procedure of a program can include call statement, and procedure call to auxiliary procedures of the system. A system dependence graph (SDG) [15] extends the program dependence graph of the main procedure of a program, by representing a call statement with a callsite vertex. An SDG includes the interconnections of callsite vertex to procedure dependence graph of every auxiliary procedure corresponding to the procedure calls. An interconnection at a callsite vertex corresponding to a procedure call, includes representation of call dependence of an entry vertex of a procedure dependence graph on the callsite vertex. The dependences corresponding to data representing input parameters to be passed to the called procedure at a call statement is represented. The dependences corresponding to the output computed by the called procedure and the data passed and returned to the location of the call statement is also represented. The dependences of the data members, methods, and parameters in the method call of a class is represented in a class level dependence graph (ClDG) [5]. The dependences of the classes in a program in Java, is represented in a Java system dependence graph (JSysDG) [7]. A JSysDG includes the ClDGs corresponding to the classes, inter-class dependences of inheritance and inner classes, and dependence graphs corresponding to the constructs of Java programs, namely interfaces and abstract classes, and packages, as subgraphs.

Safety software used in monitoring and control of systems of power, transportation, avionics, or space, integrates several components and embedded applications. The applications include heterogeneous hardware driver, and software components provided by different manufacturers and vendors. They include fault-tolerant control, and stringent requirements of recovery of software components from a degraded performance. They include requirements of flexibility, and usability for thorough testing, in software maintenance, to adapt to changes in user requests and different operating conditions. The testing and development in these environment correspond to the assumptions of development in an object-oriented paradigm.

In the following we give example of code corresponding to a simple program in Java consisting of few methods, and implementing the functions of lists. The size of the list can grow to an increased size, and can be lowered to a decreased

978-1-7281-1895-6/19/$31.00 ©2019 IEEE

size. We show that test coverage of the program using control flow coverage, saturates earlier of a test case to fail based on a fault related with certain uncovered dependences on data. Such faults could be possible when object-oriented programs are used in implementation of modules of safety critical applications. We present a set of criteria using the coverage of the data dependences of the objects in an object-oriented program. A test suite covering these dependences is considered to expose faults corresponding to definition and use of the data objects in the program.

Example

The program in Figure 1 includes code of $grow$ and low methods of Ac class, representing insert and remove operations respectively corresponding to an implementation of a list. The behavior represented by the methods of the class, is implemented with a data member object h of the A inner class, corresponding to the nodes and a head element of the list. The Sc sub class, of the Ac class, corresponds to an implementation of a derived type of list, where the low method is redefined. The Sc class, has S sub class of the A class as an inner class, corresponding to a type possible of a node of the list and of the data member h representing the head of the list. An object of Sc class, is constructed with data member object h of the type of S inner class. However, by call to the grow and low methods, invoked on the object, the data member object h can be either of A class, or S class. A data member object g in the base Ac class, corresponds to the reference of a new element inserted to a list by call to the grow method, and it is of a type related using polymorphism with the A class. An object of the type of A class is used, in reference of the data member object g, if it is not initialized with any object. The $init$ method, is new in Sc sub class of Ac class, and is used to initialize the reference g, with an object of the S class.

The $main$ method in statements at line 171 to line 194 in the code in Figure 1, corresponds to a method of the $Test$ class, for test coverage of code corresponding to the Ac class, and Sc class. It corresponds to the test program, for coverage of the interactions of the data using the code of the method of the objects of the respective classes. Every statement, branch, condition, and path corresponding to every method of Ac class, and Sc class is covered by the test cases represented by the test program. Statements at line 172 to 192 in the test program, exercises the coverage of corresponding elements of Ac class, and Sc class, more than once. The coverage of the methods of Ac class, and Sc class are computed based on the object references a, and s respectively. Even after an extent of coverage of the code by the test suite, test case corresponding to statement at line 193, fails at runtime. The failure could be traced to a fault in statement at line 94, in the code of low method of Sc class. It is caused by a use of data member m of A class, when an object of the Sc class has data dependency on a data member object of the A class, corresponding to a node of the list. An object of Sc class is data dependent on an object of A class, using data member object reference h, or data member object references possible of that of the reference h.

The statement at line 94, contains a method call $compareTo$ of the Java library, with value of an input parameter t, invoked on a variable $t2$, representing object reference using polymorphism. The code of the compareTo method of the library and not of the code of the program tested, involves conditional outcomes, on the value of the object reference variable invoked, and on the input parameter. The statement is not adequately tested, unless every possible type of data corresponding to every possible variable is considered, even though the code of the compareTo method called at the call statement of line 94 does not belong to the code of the program tested. In this case, the fault could be detected trivially, by using only every possible combination of the type possible of the object reference variable using polymorphism, based on every statement. The data dependences possible of the set of objects involved in an interaction, could be required in conjunction, to produce the actual context of the data corresponding to the fault. The fault is possible to be detected by test selection corresponding to statement at line C193, by isolating the data dependences corresponding to object reference variable s.

This paper is organized as follows. In Section 2 we discuss preliminary terminologies. We present our dependence based criteria for test coverage of Java programs in Section 3. We briefly review related work, and present discussions in relation to this work, in Sections 4 and 5, respectively. We conclude this paper in Section 6.

II. Preliminary Terminologies

In this section we summarize a few basic terminologies. We refer to these in section 3 corresponding to a definition of the metrics related with this work.

Java System Dependence Graph (JSysDG)

A JSysDG represents the dependences among the program elements of a Java program [2, 7]. The elements of the program are represented by the classes and relations, the data members, and the methods. The dependences corresponding to the elements are structurally represented in the graph. A partially constructed JSysDG of the sample program in Figure 1, is shown in Figure 2.

Def/Use table (DUT)

A tabular relationship among the methods of the classes in a program, based on the data members of every classes in the program, correspondingly defined and used either directly, or indirectly by method call, by the methods in pairs, is represented in a Def/Use table (DUT) [4]. A DUT d is associated with a data member variable a of a class, and a set of DEF methods and USE methods of classes representing the columns and row of the table respectively. A cell $d(i,j)$ in a DUT, represents a possible data flow, where data member a, is possible to be defined by DEF[i] method, and used by USE[j] method. A level l of DUT d, indicates a specific indirection of the direct, and indirect definition and use of the data member a

```
CE01:  class Ac  {
S01:       protected A g;
S02:       protected A h;
S03:       public Ac() {
S04:           h=new A();
S05:           h.d=h; }

CE08:      class A  {
S09:           protected A d;
S10:           protected Boolean m;
E11:           public A() {
S12:               d=this; }
               }

E15:       public void m1() {
S16:           if(h.d==null) {
S17:               System.out.println(h.
                       toString());
S18:           } else if(h.d==h) {
S19:               System.out.println(h.d.
                       toString());
               } else {
S20:               A t = h;
S21:               System.out.println(t.
                       toString());
S22:               while(t.d!=null) {
S23:                   t=t.d;
S24:                   System.out.println(t.
                           toString());
                   } }
           }
E29:       public void grow() {
S30:           if(g==null) {
S31:               g=new A();
S32:               g.d=null; }
S33:           if(h.d==h) {
S34:               h.d=g;
               } else {
S35:               A t = h.d;
S36:               while(t.d.!=null) {
S37:                   t=t.d; }
S38:               t.d = g;}
S39:           g=null;
           }

E50:       protected void low() {
S51:           if(h.d==h) {
S52:               h.d=null;
S53:               return; }
S54:           A t=h;
S55:           A t2=null;
S56:           while(t.d!=null) {
S57:               t2=t;
S58:               t=t.d; }
S59:           t2.d=null;
           }
       }

CE65:  class Sc  {
E67:       public Sc() {
S68:           h=new S();
S05:           h.d=h; }

CE70:      class S extends A {
S71:           public S() {
C72:               super();
S73:               m=true; }
           }

E75:       public void m2() {
C76:           this.m1(); }
E80:       public void init() {
S81:           g=new S();
S82:           g.d = null; }
E85:       protected void low() {
S86:           if(h.d == h) {
S87:               h.d=null;
S88:               return; }
S89:           A t = h;
S90:           A t2=null;
S91:           while(t.d!=null) {
S92:               t2=t;
S93:               t=t.d;
S94:               int n = t2.m.compareTo(
                       t.m); //This call can give error
               }
S95:           t2.d=null;    }
       }

CE170: class Test  {
E171:      public static void main(String[] args) {
S172:          Ac a = new Ac();
S173:          Sc  s  = new Sc();
C174:          a.m1();
C175:          a.grow();
C176:          a.m1();
C177:          s.m2();
C178:          s.init();
C179:          s.grow();
C180:          s.m2();
C181:          s.init();
C182:          s.grow();
C183:          s.low();
C184:          s.m2();
C186:          s.grow();  //Grows with a data of A
C187:          s.m2();
C188:          s.init();
C189:          s.grow();
C190:          s.m2();
C191:          a.low();
C192:          a.m1();
C193:          s.low(); //This call will give error
C194:          s.m2();
       }
   }
```

Fig. 1. An example of a Java program and test case for coverage of data and control flow of the program by thorough coverage of the classes and the corresponding objects.

by the methods of the corresponding table. A DUT is possible to be constructed using a JSysDG. DU-pairs represent an ordered pair of statements, respectively corresponding to the DEF and USE methods, involving the definition and use of the data member a. A cell in a DUT represents du-pairs. A DUT corresponding to level=0, of a data member h of class Sc of the source in Figure 1, is shown in Figure 3.a. A def-use pair represented by the cell with DEF[2]=$E85 : low$, and USE[1]=$E29 : grow$, is (87,33). It corresponds to a test adequacy of definition of the data member h, by statement at line 87, of the method low, and use of the data member, by statement at line 33, of the method $grow$. Another def-use pair corresponding to the cell is (87,35). The element (93,33), is also a def-use pair, considering statement at line 93, represents a transitive redefinition of data member object reference h defined of statement at line 89. A cell of a DUT can correspond to multiple du-pairs. A DUT that includes every du-pair, and only one du-pair at a cell is called a statement-level DUT (SDUT).

Strong and Weak Coverage Metric

A coverage computed of a class using a test coverage criterion is considered to be a strong measure, if the coverage is maximum of the coverage achieved of an object from a set of objects of the class. The coverage measured is considered a weak measure, if the coverage is a union of the coverage achieved by a set of objects of the class [1, 2].

SDUTcontain relation

$SDUTcontain$ is a relation defined on a partially ordered set of classes. A class $c1$ has an $SDUTcontain$ relation with

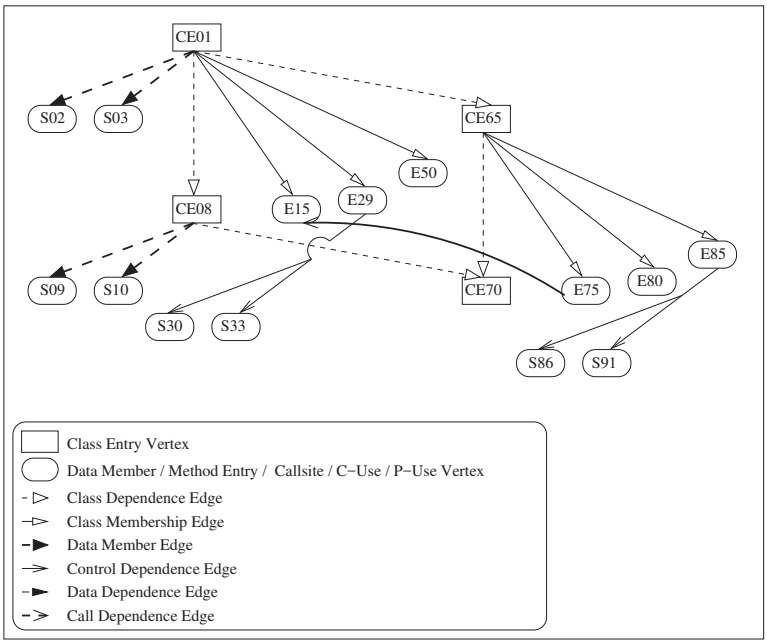

Fig. 2. Partially constructed Java system dependence graph (JSysDG) representation of the source of example.

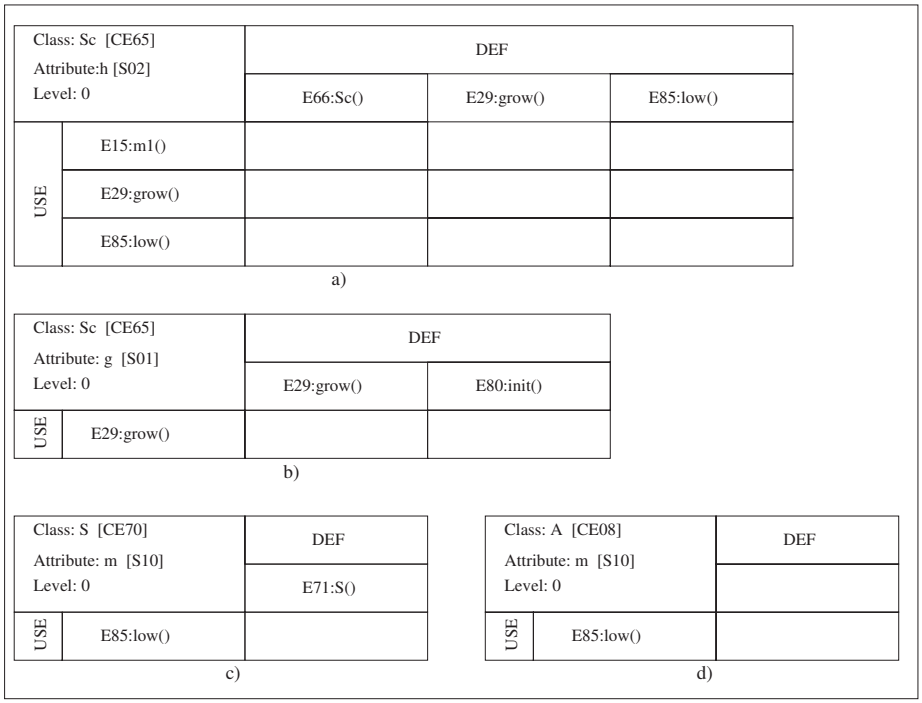

Fig. 3. Def/use table (DUT) representation of few data members of the classes used in the example.

class $c2$, if an object of class $c2$ is a data member of a class c, and class $c1$ has an $SDUTcontain$ relation with class c. A class L representing the greatest lower bound of the partially ordered set is called the $SDUTenclose$ class.

SDUTContSet

An $SDUTContSet(C)$ is a set of SDUTs defined on a class C corresponding to an SDUT. If $sd(c, a, l)$, is an SDUT corresponding to class c, data member a, and level l, and $sd(c, a, l) \in SDUTContSet(C)$, then class C has $SDUTcontain$ relation with class c, and is the greatest lower bound of the relation.

SDUTConcSet

An $SDUTConcSet(SC)$ is a set of SDUTs defined on a set SC of classes. An SDUTcontain relation does not exist in class $c1$ to class $c2$, and class $c2$ to class $c1$ on any two classes $c1$,$c2 \in SC$, An SDUT $sd(c,a,l) \in SDUTConcSet(SC)$ is considered corresponding to a class $c \in SC$, of every level l, and every data member a of class c.

III. Data Dependence Coverage of Classes (DDCC)

In this section, we present a few test coverage criteria based on the dependences in relation with definition and use of data in interacting objects. Adequacy based on def-use of inter-object data in the system of runtime of object-oriented programs is considered in data dependence coverage of classes.

All-SDUT-du-pairs coverage (ASDPC)

An SDUT-du-pair coverage of a cell $x[i][j]$ of an SDUT $sd(C,a,l)$, of class C, data member a, and level l, requires a test execution. The test execution requires the data member a is defined by a statement corresponding to the def method $DEF[i]$, and is used by a statement corresponding to the use method $USE[j]$ of the SDUT $sd(C,a,l)$, with no intervening definition. The all-SDUT-du-pairs coverage, is defined for SDUT-du-pair coverage of all cells of set $allSDUTSet$ of all SDUTs. It is expressed as follows:

$$ASDPC = \frac{\sum_{sd(C,a,l) \in allSDUTSet} exec(sd(C,a,l))}{\sum_{sd(C,a,l) \in allSDUTSet} NCells(sd(C,a,l))} \quad (1)$$

where,

- $NCells(sd(C,a,l))$: represents the cardinality of the set of all cells in SDUT $sd(C,a,l)$.
- $exec(sd(C,a,l))$: represents the cardinality of the subset of cells in $sd(C,a,l)$ exercised by the test suite.

SDUT-contain coverage (SDCTC)

An SDUT-contain coverage is defined of a class, and represents the test requirements corresponding to a subset of the du-pairs of the set required in an ASDPC. An SDUT-contain coverage of a class C, corresponds to the set $SDUTContSet(C)$ of SDUTs. A cell $x[i][j]$ of an SDUT $sd(c, a, l)$ is ignored if the class C does not have $SDUTcontain$ relation with class corresponding to the method $DEF[i]$ and method $USE[j]$. Let, set $SDT(c,a,l)$ represents the ignored cells. $SDCTC(C)$ is expressed as follows:

$$SDCTC(C) = \frac{\sum_{sd(c,a,l) \in SDUTContSet(C)} execCt(sd(c,a,l))}{\sum_{sd(c,a,l) \in SDUTContSet(C)} NCellsCt(sd(c,a,l))} \quad (2)$$

where,

- $NCellsCt(sd(c,a,l))$: represents the cardinality of the set of all cells in SDUT $sd(c,a,l)$, where du-pairs corresponding to $SDT(c,a,l)$ are ignored.
- $execCt(sd(c,a,l))$: represents the cardinality of the subset of cells exercised by the test suite.

SDUT-concur coverage (SDCCC)

An SDUT-concur coverage of a set SC of classes, corresponds to the set $SDUTConcSet(SC)$ of SDUTs. A cell $x[i][j]$ of an SDUT $sd(c, a, l)$ is ignored if a class $c' \in SC$ has $SDUTcontain$ relation with class corresponding to the method $DEF[i]$ and method $USE[j]$. Let, set $SDC(c,a,l)$ represents the ignored cells. $SDCCC(C)$ is expressed as follows:

$$SDCCC(SC) = \frac{\sum_{sd(c,a,l) \in SDUTConcSet(SC)} execCc(sd(c,a,l))}{\sum_{sd(c,a,l) \in SDUTConcSet(SC)} NCellsCc(sd(c,a,l))} \quad (3)$$

where,

- $NCellsCc(sd(c,a,l))$: represents the cardinality of the set of all cells in SDUT $sd(c,a,l)$, where du-pairs corresponding to $SDC(c,a,l)$ are ignored.
- $execCc(sd(c,a,l))$: represents the cardinality of the subset of cells exercised by the test suite.

IV. Review of Related Work

Chen et. al. [6] proposed all-bindings and all-du-pairs criteria, in object-flow metrics for test coverage of object-oriented programs. The program model of OCFG is constructed based on the object flows represented of the test program. The adequacy of coverage of the objects, are considered on binding of the possible classes with the program variables, and definition and use of an object across a DU-pair of vertices. Baldini et. al. [4] proposed metrics based on the program dependences of methods, of classes, in all-uses criteria of DUTs. The criteria considered coverage of the inter-class data dependences, introduced by the interaction of the members, namely the methods of the classes, having definition and use of the data members. Najumudheen et. al. [3], proposed criteria for inheritance and polymorphic coverage of object-oriented programs, defined on program model of COSDG, based on relations of method calls. Alexander et. al. [14] proposed metrics of object-oriented coupling-based testing (OOCBT), including coverage of coupling paths, during definition and use of state variables, of objects having dynamic binding.

V. Discussions

The importance of coverage of data dependences of objects has been considered in this work. Test coverage of an object-oriented program, depends on thorough coverage of the program paths, at required states of the objects in combination [10]. In this context, coverage of a class as a union of the coverage of the objects is considered to be a weak measure of coverage for test coverage criteria of classes. The present work considers that criteria defined for coverage of classes, based on a maximum of the coverage of the data dependences, of objects of corresponding classes, can increase thoroughness of the testing, by requiring more test cases for an equal measure. As named as strong measure, it is suitable for use-based integration, of safety-critical software. A higher extent of the coverage measured of the software as a composite, can scale the coverage of a set of the components of the software into a measure of its coverage using respective criteria, by increments in the test execution.

VI. Conclusions

Coverage of data flow and dependences are also considered with coverage of control flow and dependences in programs, to ensure thoroughness of white-box test coverage of programs developed in heterogeneous platforms in paradigm of object-oriented programming. We presented an example as a motivation on an adequacy of test coverage based on data dependences of a simple Java program. Based on the representation of a Java program on a JSysDG [7], and references on representation of DUTs [4], we have presented a set of metric and criteria possible of data dependence coverage of classes (DDCC). We have discussed on strong coverage metric of classes based on coverage of objects from our earlier work [1], and have defined sets of data dependences from the relationship of the classes considered for test coverage. A strong measure of any class-based coverage computed of the set of classes, using objects from data member relations and from the sets SDUTContSet and SDUTConcSet, can detect faults at inter-object data dependences, by ensuring thorougher coverage of the program dependences in object-oriented programs.

We have only discussed on metrics of DDCC in Section III in this work. We could not discuss on the possible control dependence coverage of classes (CDCC) based on coverage of the control dependences of the MDG corresponding to every individual method of the classes represented on JSysDG, due to constraints of space. We have presented criterion of integration control dependence coverage (ICC), of methods of related classes, based on terminologies of control dependence sequence (CDS), in our earlier work [1, 2]. The data and control dependence coverages of the classes could be usually combined using the elements of the dependences in the equations in Section III in metrics of dependence coverage of a set of classes (DCC).

References

[1] Debashis Mukherjee, and Rajib Mall. An integration test coverage metric for Java programs. *International Journal of System Assurance Engineering and Management (IJSA), Springer*, pages 1–26, June 2019.

[2] Debashis Mukherjee, Dibyanshu Shekhar, Rajib Mall. Proposal for A Structural Integration Test Coverage Metric for Object-Oriented Programs. *ACM SIGSOFT Software Engineering Notes*, 43(1):1–4, 2018.

[3] R. Mall E. S. F. Najumudheen and D. Samanta. Test coverage analysis based on an object-oriented program model. *Journal of Software Maintenance and Evolution: Research and Practice*, 23:465–493, 2011.

[4] Leonardo Grassi Fabrizio Baldini, Giacomo Bucci and Enrico Vicario. Test coverage analysis for object oriented programs, structural testing through aspect oriented instrumentation. *ICSOFT'07*, 2007.

[5] L. Larsen and M. Harrold. Slicing object oriented software. *in 18th International Conference of Software Engineering*, pages 495–505, Mar 1996.

[6] M.H. Chen, and H.M. Kao. Testing object-oriented programs - an integrated approach. *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE'99), Boca Raton, FL, USA*, pages 73–82, 1999.

[7] Marc Roper Neil Walkinshaw and Murray Wood. The Java system dependence graph. *Proceedings of 3rd IEEE International Workshop on Source Code Analysis and Manipulation*, pages 859–879, 2003.

[8] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, 1984.

[9] P. Ammann and J. Offut. *Introduction to Software Testing*. Cambridge University Press, 2008.

[10] P. Samuel, R. Mall, and A.K. Bothra. Automatic test case generation using unified modeling language (UML) state diagrams. *IET Software*, 2(2):79–93, 2008.

[11] Rajib Mall. *Fundamentals of Software Engineering, 4th ed.* PHI Learning Private Limited, New Delhi, 2014.

[12] S. Rapps and E. J. Weyuker. Data flow analysis techniques for test data selection. *Proceedings of the Sixth International Conference of Software Engineering, Tokyo, Japan*, pages 272–277, Sep. 1982.

[13] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Object Technology, 1999.

[14] Jeff Offutt Roger T. Alexander and Andreas Stefik. Testing coupling relationships in object-oriented programs. *Software Testing Verification and Reliability*, 20:291–327, 2010.

[15] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program.Lang. Syst.*, 12(1):26–60, 1990.