Test Path Selection of Polymorphic Call-sites

Anshu Bansal
SRI-Noida
Samsung India Electronics Pvt Ltd
Noida-201301, India

Rajesh K. Bhatia
Department of Computer Science & Engineering
Punjab Engineering College
Chandigarh-160 012, India

ABSTRACT

The paper proposes System Dependence Graph (SDG) based algorithm to select different test paths for testing polymorphic call-sites. SDG, includes control and data dependencies, helps both the testers and developers of object-oriented programs to better understand the polymorphic interactions within the software. In addition, the algorithm considers only the method bindings of a polymorphic call-site having different definition sets. As a result, the number of test paths for testing polymorphism gets reduced. Also, the algorithm has been implemented in a prototype Graphical User Interface (GUI) based tool. The results are obtained by using the tool, which demonstrate the proposed technique.

General Terms

Algorithm, Testing, Polymorphic call-sites.

Keywords

System Dependence Graph, Polymorphic Method, Test Paths.

1. INTRODUCTION

Object-oriented features such as inheritance, polymorphism and dynamic binding etc. introduced the new types of faults like Yo-Yo graph problem, Inconsistent Type Use (ITU), State Definition Anomaly (SDA) and State Definition Incorrectly (SDI) etc [1]. Traditional procedural language approaches are not able to detect the faults. It is required to develop or modify the existing techniques to detect the faults.

In this paper, the proposed technique tests the polymorphic call-sites by considering only the method bindings having different definition sets. It eliminates the need to test all the possible method bindings to test the polymorphic call-sites. As a result, the efficiency of testing polymorphism in terms of time needed to test the number of selected paths gets reduced. The technique is based upon System Dependence Graph (SDG). SDG is a connection of various Procedure Dependence Graphs (PDGs) at call-sites, which represents control and data dependencies among the statements. Statement B is *control dependent* on statement A, if the execution of statement B depends upon the execution of statement A. Statement B is *data dependent* on statement A, if statement B is using a variable that has been defined by statement A.

2. RELATED WORK

The field of object-oriented testing was first established in the late 1980s. In 1987, Harrold and McGregor [4] proposed an approach to test object-oriented software at system level.

The first publication in 1990 that addressed the challenges in object-oriented software testing was by Perry and Kaiser [5]. In 1994, Barbey and Strohmeier [6] identified the faults that occur due to three object-oriented features: encapsulation, inheritance and polymorphism. Many researchers tried to modify existing techniques and some of them proposed new techniques for fault detection in object-oriented software. Polymorphism also poses difficulty in testing. The dynamic binding allows the target of a call to be decided at runtime, not at compile-time. So, testing is required to exercise each possibility of method binding of polymorphic call-site.

Frankl and Weyuker [7] proposed data flow testing technique that was further modified by Orso and Pezze [8]. They defined new set of *polymorphic definitions* and *polymorphic uses* by considering dynamic binding. In addition, they constructed Inter Class Control Flow Graph (ICCFG) to find *du* paths.

Alexander and Offutt discussed about quasi inter-procedural analysis technique to analyze the problem that can occur due to inheritance and polymorphism in 2000 [2, 3] and proposed the graphical model named “Yo-Yo graph” to analyze the problem in 2001 [9]. They also proposed coupling-based analysis technique to test polymorphic interactions in paper [10], which used two concepts of 1970s: data abstraction of Parnas in 1972 [11] and coupling concept of Constantine and Yourdon [12] to understand the module interactions.

Supavita and Suwannasart [13] presented different forms of polymorphic interactions. They showed that its behavior was affected by distinct factors. Using sequence diagram, they identified which form of polymorphic interaction present and then designed test cases to test its polymorphic behavior depending upon the affecting factors and by considering all the possible subclasses in the inheritance hierarchy.

Saini [14] proposed an algorithm for testing polymorphism to find the sequence of method calls in which data anomaly can occur. The algorithm made use of sequence diagram, which depicts the dynamic behavior of the system, to identify the problems.

The researchers proposed various graph models to represent program features such as Control Flow Graph (CFG) [15], Data Flow Graph (DFG) [17], Program Dependence Graph (PDG) [16], System Dependence Graph (SDG) [18], and Call Graph (CG) [19]. Najumudheen, Mall, and Samanta [20] proposed the dependence-based graphical representation called Call-based Object-Oriented System Dependence Graph (COSDG) to represent various object-oriented features such as class, inheritance and polymorphism. It was based on Extended System Dependence Graph (ESDG), which was proposed by Larsen and Harrold [21].

Survey of the related work discussed above shows that the selection of test paths for polymorphic call-sites in object-oriented software using system dependence graph is still an area open for research.

3. PROPOSED TECHNIQUE

The main problem in testing the polymorphic call-site is the methods, possible to be called at the call-site, have different definition sets. Consider the java code shown in Figure 1.

```
1. package myPackage;
2. class Base
3. {
4.    Base() {}
5.    public Base b1;
6.    public int i;
7.    public void assign()
8.    {
9.        b1=new Base();
10.       b1.i=1;
11.       System.out.println("base");
12.   }
13.   public void use()
14.   {
15.
   System.out.println("use:i="+b1.i);
16.   }
17. }
18. class Derived extends Base
19. {
20.   Derived() {}
21.   public void assign()
22.   {
23.       System.out.println("derived");
24.   }
25. }
26. public class Test
27. {
28.   public static void main(String
   args[])
29.   {
30.       Base p1;
31.       p1=new Derived();
32.       p1.assign();
33.       p1.use();
34.   }
35. }
```

Fig 1: Java code depicting the inheritance and polymorphic interactions

At line no. 32, polymorphic call-site: - *p1.assign()* has two possible method bindings :- assign() method of base class (line no. 7) and assign() method of derived class (line no. 21). Both method bindings are defining different variables. The method *p1.use()*, called at line no. 33, uses (line no. 15) the variable "b1.i" defined (line no. 10) by assign() method of base class. Method *assign()* of derived class (line no. 21) is not defining it. Method *assign()* of derived class is called at the polymorphic call-site (line no. 32) then it will cause error at line no. 33. This problem is known as State Definition Anomaly (SDA).

Another problem is State Definition Inconsistency due to state variable Hiding (SDIH). Suppose derived class declares "b1" as a local reference variable and *assign()* method of derived class (line no. 21) defines the variable "b1.i". The problem will still occur at line no. 33 when method *p1.use()* tries to use (line no. 15) the variable "b1.i". Since *assign()* method of derived class (line no. 21) has defined the variable "b1.i" without using *super* keyword. It means that it has defined its local variable named "b1.i".

The researchers targeted these problems by proposing various techniques. They considered all the possible method bindings of the polymorphic call-site. The proposed technique has selected only the method bindings that have different definition sets. If they have same definition set, then we assume that it will not cause any problem. In this way, it reduces the number of test paths. The proposed technique is described in the form of flowchart, shown in Figure 2.

3.1 Experimental Setup

Let us consider the java program shown in Figure 3 (on next page).

A prototype tool, shown in Figure 4 (on next page), has been developed to implement the given proposed technique.

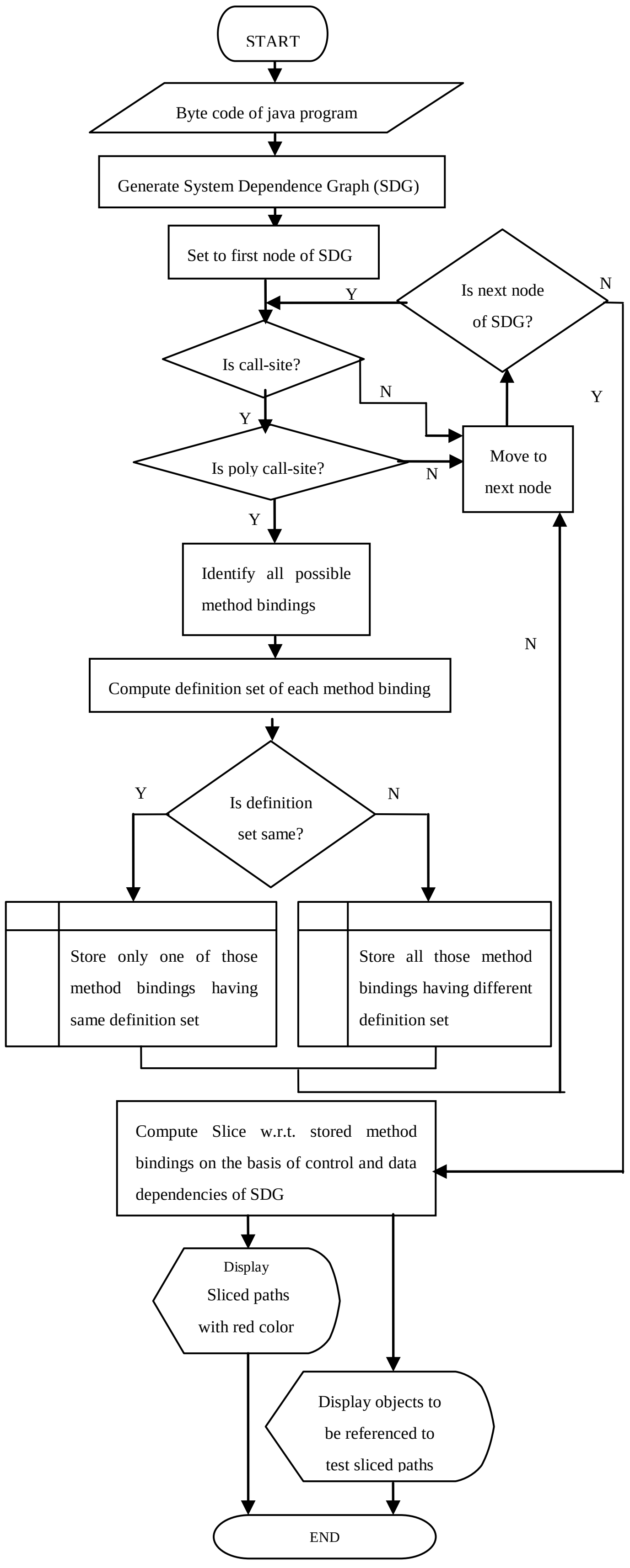

START
Byte code of java program
Generate System Dependence Graph (SDG)
Set to first node of SDG
Is next node of SDG?
Y
N
Is call-site?
N
Y
Is poly call-site?
N
Move to next node
Y
Identify all possible method bindings
N
Compute definition set of each method binding
Is definition set same?
Y
N
Store only one of those method bindings having same definition set
Store all those method bindings having different definition set
Compute Slice w.r.t. stored method bindings on the basis of control and data dependencies of SDG
Display Sliced paths with red color
Display objects to be referenced to test sliced paths
END

Fig 2: Flowchart giving the overview of the proposed technique

```
1. package otherPkg;
2. class Sum
3. {
4.   public int add(int x,int y)
5.   {
6.     return(x+y);
7.   }
8. }
9. class Sum1 extends Sum
10. {
11.   public int add(int x,int y)
12.   {
13.     y=y+2;
14.     return(x+y);
15.   }
16. }
17. class Base
18. {
19.   Base() {}
20.   public Base b1;
21.   public int i;
22.   public void assign()
23.   {
24.     b1=new Base();
25.     b1.i=1;
26.     System.out.println("base");
27.   }
28.   public void use()
29.   {
30.
      System.out.println("use:i="+b1.i);
31.   }
32. }
33. class Derived extends Base
34. {
35.   Derived() {}
36.   public void assign()
37.   {
38.     System.out.println("derived");
39.   }
40. }
41. public class Test
42. {
43.   public static void main(String args[])
44.   {
45.     Base p1;
46.     p1=new Derived();
47.     Sum b=new Sum();
48.     int sum, i;
49.     sum = 0;
50.     i = 10;
51.     while (i>0)
52.     {
53.       if(i==5)
54.       {
55.         p1.assign();

56.       }
57.       sum = b.add(sum, i);
58.       i = i-1;
59.     }
60.     System.out.println("\nsum ="+sum);
61.     System.out.println("\ni = "+i);
62.     p1.use();
63.   }
64. }
```

Fig 3: An example java program depicting the inheritance and polymorphic interactions

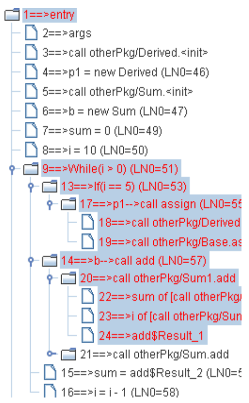
3.2 Results

Table 1 shows all the possible method bindings of various polymorphic call-sites of the program, shown in Figure 3. At the first polymorphic call-site shown in Table 1, the possible method bindings have different definition sets. But at the second polymorphic call-site, the possible method bindings have same definition set.

Table 2 depicts the type of reference variable to be declared and the object to be referenced to test the method bindings of polymorphic call-sites having different definition sets. Table 2 has first two rows to test the first polymorphic call-site since both method bindings of the first polymorphic call-site have different definition set. To test the second polymorphic call-site, only one last row is needed since both method bindings of the second polymorphic call-site have same definition set.

🟪 Polymorphic Call-Site ⬛ Simple Statement 🟦 Method Call-Site

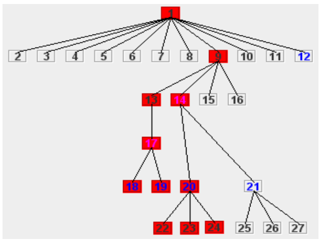
🟥 Test Paths: Statements that are control/data dependent on method bindings of polymorphic call-sites to be tested and on which they are control/data dependent.



```
1==>entry
  2==>args
  3==>call otherPkg/Derived.<init>
  4==>p1 = new Derived (LN0=46)
  5==>call otherPkg/Sum.<init>
  6==>b = new Sum (LN0=47)
  7==>sum = 0 (LN0=49)
  8==>i = 10 (LN0=50)
  9==>While(i > 0) (LN0=51)
    13==>If(i == 5) (LN0=53)
      17==>p1-->call assign (LN0=55
        18==>call otherPkg/Derived
        19==>call otherPkg/Base.as
    14==>b-->call add (LN0=57)
      20==>call otherPkg/Sum1.add
        22==>sum of [call otherPkg/
        23==>i of [call otherPkg/Sun
        24==>add$Result_1
      21==>call otherPkg/Sum.add
    15==>sum = add$Result_2 (LN0=5
    16==>i = i - 1 (LN0=58)
```

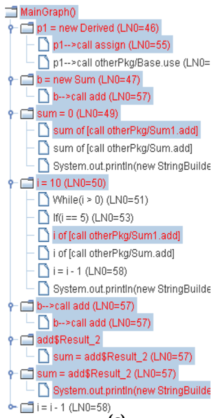
(a)

Slice of Control Dependence Tree



(b)

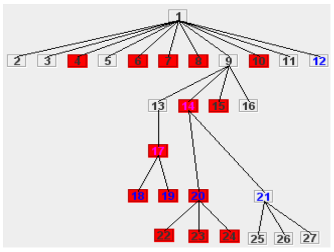
Slice of Control Dependence Graph



```
MainGraph()
  p1 = new Derived (LN0=46)
    p1-->call assign (LN0=55)
    p1-->call otherPkg/Base.use (LN0=
  b = new Sum (LN0=47)
    b-->call add (LN0=57)
  sum = 0 (LN0=49)
    sum of [call otherPkg/Sum1.add]
    sum of [call otherPkg/Sum.add]
    System.out.println(new StringBuilde
  i = 10 (LN0=50)
    While(i > 0) (LN0=51)
    If(i == 5) (LN0=53)
    i of [call otherPkg/Sum1.add]
    i of [call otherPkg/Sum.add]
    i = i - 1 (LN0=58)
    System.out.println(new StringBuilde
  b-->call add (LN0=57)
    b-->call add (LN0=57)
  add$Result_2
    sum = add$Result_2 (LN0=57)
  sum = add$Result_2 (LN0=57)
    System.out.println(new StringBuilde
  i = i - 1 (LN0=58)
```

(c)

Slice of Data Dependence Tree



(d)

Slice of Data Dependence Graph

Fig 4: Slice of (a) control dependence tree; (b) graphical representation of control dependency tree shown in 4(a); (c) data dependence tree and (d) graphical representation of data dependence tree shown in 4(c)

Table 1. All the possible method bindings of various polymorphic call-sites of the program, shown in Figure 3

S. No.	Polymorphic Call-site		Possible method bindings
	Line No.	Call-site	
1	55 in otherPkg\Test.class	p1→call assign	otherPkg.Derived.assign
			otherPkg.Base.assign
2	57 in otherPkg\Test.class	p1→call add	otherPkg.Sum1.add
			otherPkg.Sum.add

Table 2. Reference Variables to test selected paths of the program, shown in Figure 3

S. No.	Reference Variable		Polymorphic Call-site		Binded Methods to be Tested
	Type of reference variable	Object to be referenced	Line No.	Call-site	
1	otherPkg.Base	otherPkg.Derived	55	p1→call assign	otherPkg.Derived.assign
2	otherPkg.Base	otherPkg.Base	55	p1→call assign	otherPkg.Base.assign
3	otherPkg.Sum	otherPkg.Sum1	57	p1→call add	otherPkg.Sum1.add

4. CONCLUSION

The paper has proposed a new technique, based on system dependence graph, to address the problems: State Definition Anomaly (SDA) and State Definition Inconsistency due to state variable Hiding (SDIH) that can occur due to inheritance, dynamic binding and polymorphism in object-oriented software.

As the technique is based on System Dependence Graph (SDG), it considers both control and data dependencies. In addition, with SDG representation, both the testers and developers of object-oriented programs can better understand the polymorphic interactions within the software.

Slicing of system dependence graph has helped in reducing the number of paths to be tested. Moreover, the technique is considering only those method bindings whose definition sets are different, which further reduces the test paths. As a result, the efficiency of testing polymorphism has been improved in terms of time required to test the number of selected paths.

The paper has focused on selecting test paths for polymorphic interaction but still there are some points that can be further explored. The proposed technique can be further extended to handle other problems such as State Definition Incorrectly (SDI), Indirect Inconsistent State Definition (IISD) etc. and the faults that can occur due to exception handling, friend function and aggregation relationship.

5. REFERENCES

[1] R. T. Alexander and J. Offutt, Coupling-based Testing of O-O Programs, *Journal of Universal Computer Science (J.UCS)*, 2004, vol. 10, no.4, pp. 391-427.

[2] R. T. Alexander and J. Offutt, Criteria for Testing Polymorphic Relationships, in *Proceedings of the 11th International Symposium on Software Reliability and Engineering (ISSRE’00)*, San Jose, California, 2000, pp. 15–23.

[3] R. T. Alexander and J. Offutt, Analysis Techniques for Testing Polymorphic Relationships, in *Proceedings of the Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00)*, Tokyo, Japan, 2000, pp. 172-178.

[4] M. J. Harrold and J. D. McGregor, Incremental testing of object-oriented class structures, Department of Computer Science, Clemson University, Clemson, SC, USA, 1987.

[5] D. E. Perry and G. E. Kaiser, Adequacy Testing and Object-Oriented Programming, *Journal of Object Oriented Programming*, 1990, vol. 2, no. 5, pp. 13-19.

[6] S. Barbey and A. Strohmeier, The Problematics of Testing Object-Oriented Software, in *Proceedings of SQM’94 Second Conference on Software Quality Management*, Edinburgh, Scotland, UK, 1994, vol. 2, pp. 411-426.

[7] P. G. Frankl and E. J. Weyuker, An Applicable Family of Data Flow Testing Criteria, *IEEE Transactions on Software Engineering*, October 1988, vol. 14, no. 10, pp. 1483-1498.

[8] A. Orso and M. Pezze, Integration Testing of Procedural Object-Oriented Programs with Polymorphism, in *Proceedings of the 16th International Conference on Testing Computer Software: Future Trends in Testing (TCS’99)*, Washington D.C., June 1999.

[9] R. T. Alexander, Testing the Polymorphic Relationships of Object-oriented Programs, Dissertation, George Mason University, 2001.

[10] R. T. Alexander, J. Offutt, and A. Stefik, Testing Coupling Relationships in Object-Oriented Programs, George Mason University, June 2009, vol. 20, no. 4, pp. 291-327, http://onlinelibrary.wiley.com/doi/10.1002/stvr.417/abstract [13 September, 2010].

[11] D. Parnas, On the criteria to be used in decomposing a system into modules, *Communications of the ACM*, 1972, vol. 15, no. 12, pp. 1053–1058.

[12] L. L. Constantine and E. Yourdon, *Structured Design*, Prentice-Hall, Englewood Cliffs NJ, 1979.

[13] S. Supavita and T. Suwannasart, Testing Polymorphic Interactions in UML Sequence Diagrams, in *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC’05)*, 2005.

[14] D. K. Saini, Testing Polymorphism in Object Oriented Systems for Improving software Quality, in *Proceedings of SIGSOFT*, March 2009, vol. 34, no. 2, pp. 1-5.

[15] F. E. Allen, Control flow analysis, in *Proceedings of a Symposium on Compiler Optimization*, July 1970, vol. 5, no. 7, pp. 1–19.

[16] J. Ferrante, K. J. Ottenstein, and J.D. Warren, The program dependence graph and its use in optimization, *ACM Transactions on Programming Languages and Systems*, July 1987, vol. 9, no. 3, pp. 319-349.

[17] K. J. Ottenstein, Data-Flow Graphs as an Intermediate Program Form, *PhD thesis*, Computer Sciences Department., Purdue University, Lafayette, IN, August 1978.

[18] S. Horwitz, T. Reps, and D. Binkley, Interprocedural slicing using dependence graphs, *ACM Transactions on Programming Languages and Systems*, January 1990, vol. 12, no. 1, pp. 26-60.

[19] B. G. Ryder, Constructing the call graph of a program, *IEEE Transactions on Software Engineering*, May 1979, vol. 5, no.3, pp. 216–226.

[20] E. S. F. Najumudheen, R. Mall, and D. Samanta, A Dependence Representation for Coverage Testing of Object-Oriented Programs, *Journal of Object Technology (JOT)*, 2010, vol. 9, no. 4, pp. 1-23.

[21] L. Larsen and M. J. Harrold, Slicing object-oriented software, in *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, March 1996, pp. 495–505.