Int. J. of Computer and Communications
Vol. 1, No. 1, June 2011

www.ijcc.yolasite.com

Testing Anomalies in Multiple and Multilevel Inheritance in Object-Oriented Systems

Shubpreet Kaur[1] , Shivani Goel[2]

[1] MTech Student, Thapar University, Patiala, Punjab, India
[1]shubpreetkaur@gmail.com,
[2] Assistant Professor, Department of Computer Science, Thapar University, Patiala, Punjab, India
[2]shivani@thapar.edu

Abstract

Software testing is an important phase of software development process that can be easily missed by software developers because of their limited time to complete the project. One of the major challenges in software testing is the generation of test cases that satisfy the given competence criterion. Object oriented testing is a technique for testing software units that has great potential for improving the quality and reliability of software. In this paper, our focus is on classes, objects, inheritance, method overriding, and polymorphism because the testing of inheritance and polymorphism creates many binding anomalies during static and dynamic time i.e. objects and the values assigned to the objects vary. Consequently we have developed an algorithm to overcome the anomalies for multilevel inheritance.

Keywords: *Anomalies, Generation of test cases, Multiple and Multilevel inheritance, Object oriented testing, Software testing.*

1 Introduction

Testing means providing the system with inputs and letting it operate on them in order to prove that it works correctly. If it doesn't, the purpose of testing is to show the differences between the obtained results and the expected ones. Such a difference is caused by a fault (often referred to as bug) which must be exactly identified. For finding even the most subtle bugs, testing must be thorough. The test cases must be chosen very carefully, and their execution should cover most of the existing code. To test the code intensively, we need to generate many test cases. Importance of software testing is widely recognized and researchers invest a lot of efforts in order to make testing less tedious and more effective. Despite this, progresses in this field are not yet advanced enough to offer software developers satisfactory solutions for their testing needs. Testing of object oriented systems is challenging task with respect to inheritance, polymorphism, method overriding and binding of data etc. *Binding* refers to the linking of a procedure call to the code to

be executed in response to the call. *Dynamic binding* (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with *polymorphism* and *inheritance*. In *polymorphism* an operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in operation. *Inheritance* is the process by which objects of one class acquire the properties of objects of another class. Inheritance changes the behavior when object changing its classes and method overrides and creates ambiguities.

Testing of object-oriented systems presents a variety of new challenges due to features such as inheritance, polymorphism, dynamic binding, and object state. Programs contain complex interactions among sets of collaborating objects from different classes. These interactions are greatly complicated by object-oriented features such as polymorphism, which allows the binding of an object reference to objects of different classes.

2 Literature Survey

Object oriented approach for programming is the need of the hour. Many researchers are trying to make it error free. The work done in this area till date is summarized here. Andrew has used class diagrams, test adequacy criteria which are defined for use in dynamic testing. Test criteria is defined by using collaboration diagram [1]. Rountev has presented a general approach for adapting whole program class analyses to operate on program fragments in polymorphism [2].Various issues and problems that are associated with testing polymorphic behavior is discussed by Saini [13]. His approach is based on single inheritance and we are going to enhance his work to multiple and multilevel inheritance approaches. These approaches will be tested based on static and dynamic methods. No approach is discussed in [9] to find its anomalies by Alexander. Stroustrup, B., believed that multiple inheritance complicates a programming language significantly, is hard to implement, and is expensive to run [8]. Alexander describes the syntactic patterns for each OO fault type say the software contains an anomaly and possibly a fault [11] There have been some conflicts in ideas, concepts, and opinions among researchers regarding object oriented programming [14]. Robert V. Binder focused on test case design - heuristic and formal techniques to develop test cases from object-oriented representations and implementations, testability - factors in controllability and observability [10]. Gorden Frozer find problems in model checkers for test case generation.

3 Anomalies during Binding in Inheritance

Multilevel Inheritance is a method where a derived class is derived from another derived class. The relationship between classes during multilevel inheritance is shown in Fig 1.

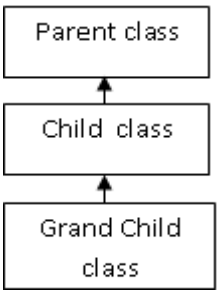

Fig 1: Multilevel inheritance

Multiple Inheritance is the ability of a class to have more than one base class(super class). The relationship between classes during multiple inheritance is shown in Fig 2.

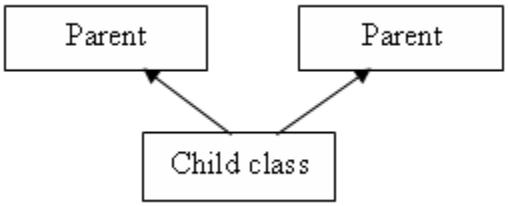

Fig 2: Multiple Inheritance

Many anomalies occur during compile time and run time in both multiple and multilevel inheritance. Fig 3 summarizes all these.

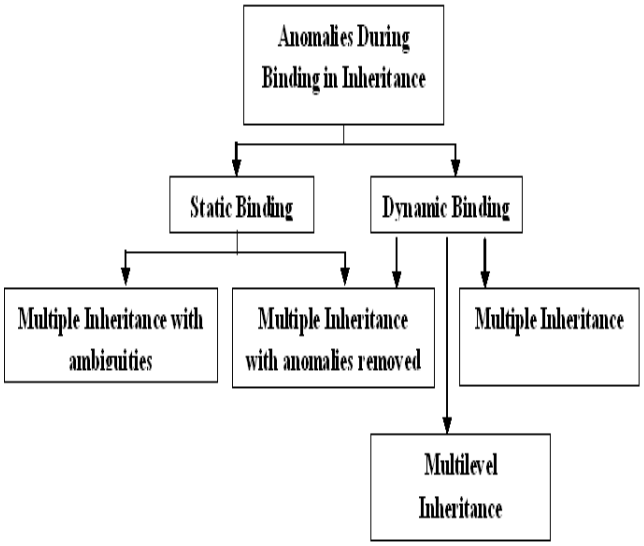

Fig 3: Anomalies during Binding in Inheritance

3.1 Multiple Inheritance with Ambiguities in Dynamic (i.e. during Run-time) Binding

Consider the following program using multiple inheritance where the anomalies are present at run time:

```
#include<iostream.h>
#include<conio.h>
1.  class A
2.  {
3.  public: int a;
4.  void show()
5.  {
6.  a=10;
7.  cout<<"\n a="<<a;
8.  }
9.  };
10. class B
11. {
12. public: int b;
13. void show()
14. {
15. b=20;
16. cout<<"\n b="<<b;
17. }
18. };
19. class C:public A,B
20. {
21. public:int c;
22. void show()
23. {
24. cout<<"\n c="<<a+b;
25. }
26. };
27. void main()
28. {
29. clrscr();
30. C ob;
31. ob.show();
32. getch();
```

```
33. }
```

In the above code, there is no ambiguity (i.e. no static error but a run-time error). There is a call to show() function which is present in all the three classes. Result will be garbage because we have taken class B as private by default in line no.19 and the value of variable b in class C will be taken garbage. Value of A will be correct as it is publicly derived. So, it will give run-time error.

3.2 Multiple Inheritance with ambiguities during Static (i.e. Compile time) Binding

Consider the following program using multiple inheritance where the anomalies are present at compile time:

```
#include<iostream.h>
#include<conio.h>
1.  class A
2.  {
3.  public:int a;
4.  void show()
5.  {
6.  a=10;
7.  cout<<"\n a="<<a;
8.  }
9.  };
10. class B
11. {
12. public:int b;
13. void show()
14. {
15. b=20;
16. cout<<"\n b="<<b;
17. }
18. };
19. class C:public A,public B
20. {
21. public:int c;
22. void show1()
23. {
24. cout<<"\n c="<<a+b;
25. }
```

```
26. };
27. void main()
28. {
29. clrscr();
30. C ob;
31. ob.show();
32. getch();
33. }
```

In the above code, there is a static error because show() is present in both the parent classes A,B and there is show1() present in the child class C. Now it become ambiguous to which show() of the parent classes should be executed because now we have show1() in the child instead of show(). So, there is a static ambiguity at line 31 and the code will not be executed.

3.3 Multiple Inheritance with Anomalies Removed

Consider the following program using multiple inheritance where the anomalies are removed:

```
#include<iostream.h>
#include<conio.h>
1.  class A
2.  {
3.  public: int a;
4.  void show()
5.  {
6.  a=10;
7.  cout<<"\n a="<<a;
8.  }
9.  };
10. class B
11. {
12. public: int b;
13. void show()
14. {
15. b=20;
16. cout<<"\n b="<<b;
17. }
18. };
19. class C:public A, public B
20. {
21. public: int c;
```

```
22. void show()
23. {
24. cout<<"\n c="<<a+b;
25. }
26. };
27. void main()
28. {
29. clrscr();
30. C ob;
31. ob.A::show(); // calling class A show
                  //  function
32. ob.B::show();   // calling class B
                    //  show function
33. ob.C::show(); // successful giving the
                  //  result
34. getch();
35. }
```

In the above code, there is no error. Both the run-time and static anomalies are removed because we are specifying there when and to which class show() function to be called. Line no.31,32,33 specifies the solution to the anomalies. Here we have show() present in all the classes and no show1() function in any class. Now we are calling the show() by specifying the object of specified class. So, it is the successful running of the code with no errors.

4 Dynamic binding (multilevel inheritance)

Fig 4 shows the class diagram of multilevel inheritance. It shows the various state various and methods present in respective classes.

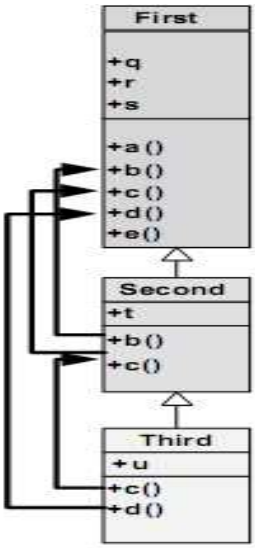

Fig 4: Class diagram of multiple inheritance

The table 1 shows the state variable definition and uses of the methods for each class hierarchy i.e. which state variables are defined in which methods and used in which methods. Here a(),b(),c(),d(),e() are methods. q,r,s,t,u are state variables. F,S and T are the objects of the First, Second and Third class.

Table 1: Anomalies

Method	Definitions	Uses
F::b	{F::q,F::s}	
F::c		F::q
F::d	{F::r}	{F::s}
S::b	{S::t}	
S::c		{S::t}
T::c	{T::u}	
T::d		{T::u}

Fig. 5 depicts the flow control of methods when c() is bound to be an instance of First. Suppose that a() calls b(), b() calls c(), c() calls d() and so on.

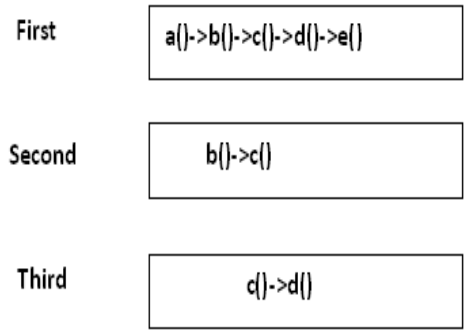

Fig 5: Sequence call of methods

Now suppose that calls to method F::b() precedes the call of F::c() and F::d(). Table 1 shows the definitions uses table that shows F::q and F::s are defined by F::b() and used by F::c() and F::d(). So here is no data flow anomaly because we are first defining the variables then we are using it.

Now suppose here is an innocuous call to S.b() instead of F.b() then data flow anomaly would exists because according to definitions uses table S.b() has called to S::t here is no problem but after S.b() we are calling F::c() and F::d() then it creates data flow anomaly shown in Fig 6 because F::c() uses F::q and F::d() uses F::s. They are used before they are defined.

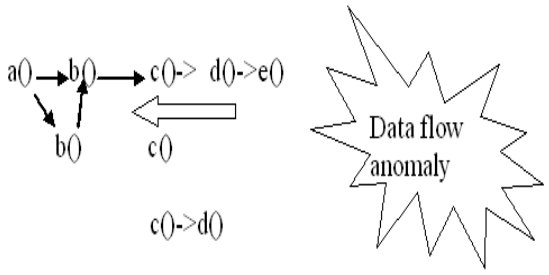

Fig 6: Data flow anomaly

Now suppose F.a() is called then S.b() and then T.c() called S.c(), S.c() called F.c() and F.c() called T.d(). It creates data flow anomaly because according to definitions uses table S.b() has called to S::t and T::c() is defining T::u then we are calling S::c here is no problem but after S.c() we are calling F::c() it creates data flow anomaly because we F::c() uses F::q. This anomaly is shown in Fig 7.

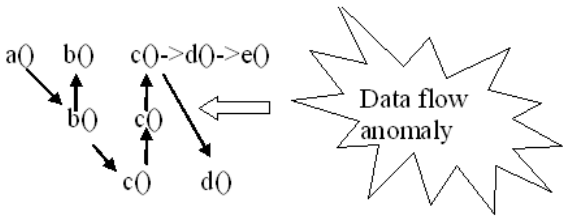

Fig 7: Data flow anomaly

5 Algorithm for Anomaly Detection

An algorithm is designed for detecting the anomalies in multilevel inheritance. For this various definitions used in the algorithm are discussed :

5.1 Some Definitions

Type Family: It is a set of classes that share a common behavior with respect to a base class A (we call it family (A)). Each descendent of A is a member B of A's family. If B is in A's family, polymorphism means that any instance of B may be freely used wherever an instance of A is expected. Every class A defines a type family, and that type family includes at least A[13].

Let A is a class in some type family.

Used by: A state variable v ε A is used by some method A if v is used in some expression in m.

Defined by: A state variable v ε A is defined by some method m ε A if m assigns first legal value to v.

Dependency of Methods: Two methods m,n ε A, we say that m is dependent on n if m uses a state variable v ε A which is defined by n.

Algorithm

1. ***initialize*** *result=false*
2. ***for*** *every class ε TF* ***do***
3. ***for*** *all methods* m_i *ε A* ***do***
4. ***for*** *each state variable* $\boldsymbol{v_j}$ *ε parent used by* m_i ***do***
5. ***if*** v_j *is defined by* m_k *ε parent for some k between 1 to n*
6. ***and*** *if* m_k *is overridden by some class child ε TF and child is descendent of parent* ***then***
7. ***if*** *child class contains state variable of parent class and there exists a call to* m_i *from* m_k ***then***
8. *result=true*
9. ***if*** *child class contains sub class* ***then***
10. ***set*** *child=subclass*
11. ***end*** *for*
12. ***end*** *for*
13. ***set*** *parent=parent +1*
14. ***end*** *for*
15. ***return*** *result*

Let's suppose initially there is no bug in the program. So, for that we initialized result is false. Then we are checking for every class starting from parent class that belongs to a type family. In that class we are checking all the methods defined the parent class. Then we check for each state variable that is defined in the different methods of parent class. After that we check whether that method is overridden by its child if it exists. Then in those methods we check for the state variable. If that state variable is used then we can say exists some data flow anomaly. And the result becomes true. If there exists sub class of that child class i.e. the grand child then we will swap grand child with a child and we will have a parent child relationship there. And will check for the anomalies again and this continues till we reached at the last level. After completing all the levels we increment parent i.e. we come one level down and make level 2 a parent and then the process continues from step2 for finding anomalies in the parent child relationship.

6 Test Results

In order to test the proposed algorithm, various test cases were generated and executed and are summarized in table 2.

Table 2: Test cases of multilevel inheritance

Test Case ID	Sequence call	Expected() value	Actual value	Status
1.	F.a()->F.b()->F.c()->F.d()->F.e()	q=10,s=20,r=30	q=10,s=20,r=30	pass
2.	S.b()->S.c()	t=50	t=50	pass
3.	T.c()->T.d()	u=60	u=60	pass
4.	F.a()->F.b()->F.c()->F.d()->F.e()->S.b()->S.c()	q=10,s=20,r=30,t=50	q=10,s=20,r=30,t=50	pass
5.	F.a()->F.b()->F.c()->F.d()->F.d()->F.e()->S.b()->S.c()->T.c()->T.d()	q=10,s=20,r=30,t=50,u=60	q=10,s=20,r=30,t=50,u=60	pass
6.	F.a()->S.b()->T.c()->T.d()	u=60	u=60	pass
7.	F.a()->F.b()->S.c()->T.c()->T.d()	t=50,u=60	T=50,u=60	pass
8.	F.a()->F.b()->S.b()->T.c()->T.d()	u=60	u=60	pass
9.	F.a()->F.b()->S.b()->S.c()->T.c()->T.d()	t=50,u=60	T=50,u=60	pass
10.	F.a()->F.b()->S.b()->F.c()->F.d()->F.e()	S=20,r=30	S=20,r=30	pass
11.	F.a()->F.b()->S.b()->S.c()->F.c()->F.d()->F.e()	T=50,q=10,r=30,s=20	T=50,q=10,r=30,s=20	pass
12.	F.a()->S.b()->F.b()->F.c()->F.d()->F.e()	q=10,s=20,r=30	q=10,s=20,r=30	pass
13.	F.a()->F.b()->S.c()	t=50	T=gv	fail
14.	F.a()->F.b()->S.c()->T.d()	t=50,u=60	T=gv,u=gv	fail
15.	F.a()->F.b()->S.c()->T.c()->T.d()	T=50,u=60	T=gv,u=60	fail
16.	F.a()->F.b()->S.b()->S.c()->T.c()->T.d()	T=50,u=60	T=50,u=60	pass
17.	F.a()->F.b()->S.c()->T.d()	T=50,u=60	T=gv,u=60	fail
18.	F.a()->F.b()->F.c()->S.c()->T.d()	q=10,t=50,u=60	q=10,t=gv,u=gv	fail
19.	F.a()->F.b()->F.c()->S.c()->T.c()->T.d()	q=10,t=50,u=60	q=10,t=gv,u=60	fail
20.	F.a()->S.b()->S.c()->F.c()->F.d()->F.e()	q=10,t=50,s=20,r=30	q=gv,t=50,r=30,s=gv	fail
21.	F.a()->S.b()->S.c()->T.c()->T.d()->F.e()	r=30,t=50,u=60	r=gv,t=50,u=60	fail
22.	F.a()->F.b()->S.b()->S.c()->T.c()->T.d()->F.e()	r=30,t=50,u=60	r=gv,t=50,u=60	fail
23.	F.a()->S.b()->S.c()->T.c()->T.d()->F.d()->F.e()	r=30,t=50,u=60,s=20	r=30,t=50,u=60,s=gv	fail
24.	F.a()->F.b()->S.b()->S.c()->T.c()->T.d()->F.d()>F.e()	r=30,t=50,u=60,s=20	r=30,t=50,u=60,s=20	pass
25.	F.a()>F.b()->F.c()->S.b()->T.c()->S.c()->F.d()->F.e()	r=30,t=50,s=20,q=10	r=30,t=50,s=20,q=10	pass
26.	F.a()>F.b()->F.c()->S.b()→T.c()>T.d()->F.d()->F.e()	r=30,u=60,s=20,q=10	r=30,u=60,s=20,q=10	pass
27.	F.a()->S.b()->S.c() ->F.d()->F.e()	r=30,t=50,s=20	r=30,t=50, s=gv	fail
28.	F.a()->F.b()->T.c()->T.d()->F.d()->F.e()	u=60,s=20,r=30	U=60,s=20,r=30	pass
29.	F.a()->F.b()->T.c()->T.d()->S.c()->S.d()	u=60,s=20,r=30	u=60,s=gv,r=30	fail
30.	F.a()->S.b()->F.c()-F.d()	q=10=,s=20	Q=gv,s=gv	fail
31.	F.a()->S.b()->T.c()->S.c()->F.c()->T.d()	t=50,q=10,u=60	T=50,q=gv,u=10	fail

7 Conclusion

Inheritance may take many from and each form of inheritance has some anomalies or the other. In this paper, various anomalies of multilevel and multiple inheritance are discussed. An algorithm is presented with which we can fetch anomalies that occur at dynamic binding. Testing system in an object oriented manner tests the system from whole perspective so that it can become error free. This algorithm traps all the dynamic errors. Its test cases are shown which tells in which cases the code will fail and pass. Above are the test cases shown.

References

[1] A. Andrews, R. France, S. Ghosh, and G. Craig, "Test Adequacy Criteria for UML Design Models", Software Testing, Verification, and Reliability Journal, Volume 13, Number 2, June 2003.

[2] Atanas Rountev, Ana Milanova, Barbara G. Ryder, Fragment Class Analysis for Testing of Polymorphism in Java Software, IEEE Transaction, June 2004, vol. 30, no.6, pp.372-387.

[3] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen, "In Black and White: An Integrated Approach to Class-Level Testing of Object-Oriented Programs", ACM Transactions on Software Engineering and Methodology, Vol. 7, No. 3, July 1998, pp. 250–295.

[4] Lv Ge-Feng, Zou Bei-Ji, Zhou Hao-Yu, Sun Jia-Guang, "Research on Model of Automated Test Case Generation for Complicated Interactive Software," Mini-Micro System (in Chinese) , 2006, Vol.1, No. 27, pp.131-135.

[5] L. Zhao, A new approach for software testability analysis, In Proceeding of the 28th international Conference on Software Engineering, Shanghai, China, May 20 - 28, 2006, ICSE '06. ACM Press, New York, NY, pp.985-988.

[6] McGregor D., John, A practical guide to testing object oriented systems.

[7] Meyer, B., Object-Oriented Software Construction. Prentice Hall, second ed., Apr. 1997..

[8] Mutant minimization for model-checker based test-case generation. In: Testing: Academic and Industrial Conference Practice and Research Techniques – MUTATION, 2007. TAICPART-MUTATION 2007. IEEE Computer Society, pp. 161–168.

[9] R. Alexander and J. Offutt. Criteria for Testing Polymorphic Relationships. In Proceedings of the 11th international Symposium on Software Reliability Engineering (Issre'00) (October 08 - 11, 2000). ISSRE. IEEE Computer Society, Washington, DC, pp.15-23.

[10] R. Binder, Testing object-oriented software: a survey, Journal of Software Testing, verification and Reliability, 1996, vol 6, pp.125–252.

[11] R. T. Alexander, J. Offutt, and J. M. Bieman, “Syntactic Fault Patterns in OO Programs”, Proceedings of the 8th International Conference on Engineering of Complex Computer Software (ICECCS '02), Greenbelt, MD, November 2002.

[12] R. V. Binder, Testing Object-Oriented Systems Models, Patterns, and Tools, Addison-Wesley, NY USA, 1999.

[13] Saini D.K, Testing Polymorphism in Object Oriented Systems for Improving software Quality, - ACM SIGSOFT Software Engineering Notes, 2009.

[14] S Supavita , Object-Oriented Software and UML-Based Testing: A Survey Report, 2009.

[15] Stroustrup, B., Multiple Inheritance for C++, Published in the May 1999 issue of "The C/C++ Users Journal"