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 Strohmaier [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.

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.

Fig 1: Java code depicting the inheritance and polymorphic interactions

START

Byte code of java program

Generate System Dependence Graph (SDG)

Set to first node of SDG

Compute definition set of each method binding

Is definition set same?

Y

N

Is poly call-site?

Y

N

Move to next node

Is call-site?

Y

N

Identify all possible method bindings

Compute Slice w.r.t. stored method bindings on the basis of control and data dependencies of SDG

Store only one of those method bindings having same definition set

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
3.2 Results

Table 1 shows all the possible method bindings of various polymorphic call sites of the program shown in Fig 3. Slice of control dependence tree (c) data dependency tree and (d) graphical representation of control dependence tree shown in 4(b). Table 2 depicts the type of reference variable to be declared and the object to be referenced to test the first polymorphic call site since both method bindings of the first polymorphic call site have different definition sets. 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. And the object to be referenced to test the second polymorphic call site have same definition set. The possible method bindings of various polymorphic call sites of the program, shown in Fig 3, are listed in Table 2.

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

Table 1: All the possible method bindings of various polymorphic call sites of the program, shown in Fig 3

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Reference Variable</th>
<th>Type of Polymorphic Call Site</th>
<th>Possible method bindings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pl</td>
<td>otherPkg.Sum1.add</td>
<td>otherPkg.Sum.add</td>
</tr>
<tr>
<td>2</td>
<td>pl</td>
<td>otherPkg.Sum1.add</td>
<td>otherPkg.Sum.add</td>
</tr>
<tr>
<td>3</td>
<td>otherPkg.Sum</td>
<td>otherPkg.Derived.assign</td>
<td>otherPkg.Derived.assign</td>
</tr>
<tr>
<td>4</td>
<td>otherPkg.Sum</td>
<td>otherPkg.Derived.assign</td>
<td>otherPkg.Derived.assign</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Reference Variable</th>
<th>Line No.</th>
<th>Polymorphic Call Site</th>
<th>Tested Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pl</td>
<td>57</td>
<td>otherPkg.Sum1.add</td>
<td>otherPkg.Sum1.add</td>
</tr>
<tr>
<td>2</td>
<td>pl</td>
<td>1</td>
<td>otherPkg.Sum1.add</td>
<td>otherPkg.Sum1.add</td>
</tr>
</tbody>
</table>

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


International Journal of Computer Applications (0975 – 8887) Volume 103 – No.4, October 2014