Object-Oriented Programming Testing Methodology

Chi-Ming Chung and Ming-Chi Lee

Department of Computer Science
Tamkang University
Taipei, Taiwan, R.O.C.

Abstract

Inheritance is an important attribute in object-oriented programming (OOP). This notion supports the class hierarchy design and captures the is-a relationship between a class and its subclass. It contributes to good properties of modularity, reusability and incremental design [11] [19]. However, misuse of multiple (repeated) inheritance will lead to improper class hierarchy which suffers from name-confliction and implicit errors. This type of errors is very difficult to be detected by conventional testing methodologies. This paper describes a graph-theoretical testing methodology for detecting this type of errors. An algorithm to support this testing methodology is also presented.

1 Introduction

Object-oriented design strategy is a new promising approach for developing software to reduce software cost and enhance software reusability. One of advantages of object-oriented programming over conventional procedure-oriented programming is supporting the notion of a *class hierarchy* and *inheritance* of properties (instance variables and methods) along the class hierarchy. A class hierarchy captures the *is-a* relationship between a class and its subclass, and a class inherits all properties defined for its superclasses. The notion of property inheritance and class hierarchy contributes to good properties of reusability and incremental design [2] [11] [19]. It has been the major tendency of software development in 1990's but it is still lack of testing methodologies based on the OOPLs to help to test software errors.

In the software life cylce, *software testing* is an important techniques to reduce the software errors and enhance correctness. Most conventional testing methodologies are derived from program factors such as control flow and data dependency. However, the program factors of OOPLs are not merely limited on these factors. Generally speaking, an object-oriented programming language must exhibit four program factors (features): *inheritance, data abstraction, dynamic binding, and information hiding* [23]. Most of these features do not exist in conventional procedure-oriented programming language. Especially inheritance does not exist in any procedure-oriented programming languages. It is conceivable that the development of testing techniques should consider the features of OOPLs. Inheritance is one of the most important features which will affect software reuse, and it supports the *class hierarchy design* and captures the *IS-A* relationship between a class and its subclass. This property has been widely applied in object-oriented software, object-oriented database, and graph system design. However, misuse of it would be prone to increase software errors and complexity [25]. Multiple inheritance and repeated inheritance allows a class to inherit more than one parent class. Although potential for code sharing is increased, the possibility of conflictions between parent classes not only increase the complexity of such systems but also leads to implicit software errors. This type of implicit errors called *name-confliction* [19] is difficult to be detected by procedure-oriented testing methodologies, but could be reflected by the graph-theoretical testing methodology presented in this paper. We propose two graph theorems which show that repeated (multiple) inheritance must consist of a set of of *unit repeated inheritance* (URIs) which not only help to test implicit errors, but also reflects the inheritance mechanism complexity. In Section 2, conventional procedure-oriented software testing and the relation between the object-oriented testing and OOPL's features are introduced. In section 3, the graph representation is applied to help to describe the inheritance mechanism. Two graph theorems and several properties are presented and proved; the testing technique based on these theorems is proposed. In Section 4, an algorithm to support this technique is presented, and illustrates how this testing technique could be applied to detect the object-oriented software errors. The final section discusses the future research.

2 Conventional testing v.s Object-oriented testing

Conventional software testing methods are divided into two categories: static testing and dynamic testing (see figure 1). In static testing the program is analyzed without executing it, while in dynamic testing the program is executed. Most testing methodologies fall into the category of dynamic testing, due to the

0-8186-2830-8/92 $03.00 © 1992 IEEE

fact that more information can be derived during programs execution. Functional testing and structural testing are two major approaches in dynamic testing. They are also called *black-box* testing and *white-box* testing. In functional testing, tests are constructed based upon the program's functional properties, ignoring its internal structure. In structural testing, the internal control flow strucuture or data dependencies are used to develop the testing methodology to conduct the testing. McCabe's structured testing [17], Chung's path complexity technique [8] and Chung's testing path methodology [9] are examples of the control structure testing, and Ntafos's required k-tuples criteria [21] and Rapps's testing criteria (all-defs, all-p-uses, all-uses, etc.) [24] , are examples of the data dependencies testing.

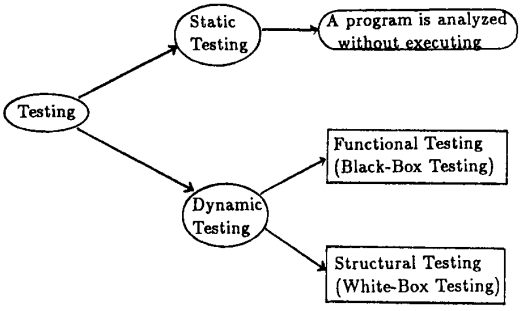


Fig.1 Procedure-oriented testing classification

A. Control Flow Based Testing

McCabe's Structured Testing

McCabe [17] proposed a structured testing methodology based on the control flow graph and the idea of cyclomatic complexity [16]. To accomplish a structured testing of a program P, the following criteria need to be met.

1. Every branch of each decision in P must be exercised at least once.

2. The least number of distinct paths needs be exercised is v. Where the value of v is the cyclomatic complexity p.

Chung's PCT Complexity and Testing

Chung's PCT is based on *path complexity* which is utilized to establish a testing order for path selection [3]. The idea is a path with higher path complexity is more error prone. Two testing criteria are proposed by Chung. They are: *intra level first criterion* and *inter level first criterion* [4] [5].

B. Data Flow Based Testing

Laski and Korel Testing

The data flow based testing strategy proposed by *Laski* and *Korel* [15] is based on two essential notations: *data environment* and *data context*. There are two testing strategies:

1. the liveness of each definition from the data environment of every statement must be tested at least once.

2. each elementary data context of every statement must be tested at least once.

Rapps and Weyuker's Testing Methodologies

Rapps and Weyuker proposed a family of testing criteria [24] based on the relationships between the definition of variables and how they are used in a program. Each occurence of a variable in a program is classified into three types: definition, computation use, or predicate use, denoted by *def, c-use, and p-use*, respectively. A *def* is the same as a variable definition; a *c-use* is defined as a variable defintion which is referenced to define a variable and a *p-use* is a variable which is referenced in a predicate in a program. These testing criteria form a hierarchy, including *all-nodes criterion, all edges criterion, all-r-uses criterion, all-uses criterion* and *all-paths criterion.*

Object-oriented Testing Methodology

Until now, there is no testing methodology developed based on the OOPLs. In the following, we develop a testing methodology based on the properties of OOPLs. Object-oriented programming languages (OOPLs) consists of four features: *information hiding, data abstraction, dynamic binding and inheritance. Information hiding* is important for ensuring reliability and modifiability of software system by reducing interdependencies between software components. The state of a software module is contained in a private variable, visible only from within the scope of the module. Only a localized set of procedures directly manipulates the data. In addition, since the internal state variables of a module are not directly accessed from without, a carefully designed module interface may permit the internal data structures and procedures to be changed without affecting the implementation of other software modules. *Data abstraction* could be considered a way of using information hiding. A programmer defines an abstract data type consisting of an internal representation plus a set of procedures used to access and manipulate the data. *Dynamic binding* allows a programmer to pursue a course of action by sending a message to an object without concern about how the software system is to implement the action. This capability becomes significant when the same general type of action can be accomplished in different types of objects. It increases flexibility by permitting the additon of new classes of objects (data types) without having to modify existing code. *Inheritance* is the center of object-oriented programming and will be discussed in the next section. The relationship between the object-oriented testing and OOPL's features is shown in figure 2.

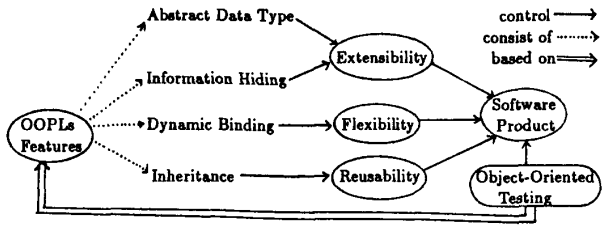


Figure 2. Object-oriented Testing and OOPLs features

3 Inheritance and Graph Theorems

Inheritance is the major mechanism of OOPLs for software reuse which is different from the module reuse, such as subroutine calls or package in Ada [19]. It allows the same code inherited from parent class without any function (subroutine) calls. It also supprots the *class hierarchy* design which captures the *is-a* relationship between a class and its subclass. The class hierarchy is usually represented by a directed graph, called *inheritance graph*.

An inheritance graph could be divided into three basic structures: *1. single inheritance*, *2. multiple inheritance*, and *3. repeated inheritance*; they are represented by a connected directed graph G=(V,E), where V is a set of classes, and E is a set of inheritance edges which are ordered relations such that E = { $x \rightarrow y \mid y$ inherits from x, where x and y $\in$ V }. Also, there are three types of inheritance edges *tree edges*, *forward edges*, and *backeges*: tree edges connect parents to children in the graph, and forward edges connect ancestors to decendants. However, *back edges* which connect decendants to ancestors should be avoided. The reason is that using back edge is prone to enter a *cyclic inheritance* [13]. Now, we discuss the three basic structures of inheritance graph and present three theorems to help develop the OOP software testing.

Defintion 1: A single inheritance is that each class inherits uniquely from one parent class.

For a single inheritance, it is a tree structure and no one class inherits from more than one parent class. For example, if there is a single inheritane, $\{A \rightarrow B, A \rightarrow C, B \rightarrow D, B \rightarrow E\}$, then class B and class C inherit uniquely from root class A, and class D and class E from class B (see Fig. 3).

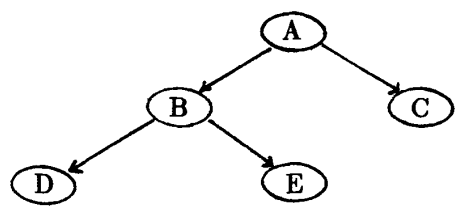


Figure 3. An example of single inheritance

Definition 2: If a class is permitted to inherit from more than one parent class, it is called a *multiple inheritance.*

Lemma 1 : Suppose inheritance graph $G_{mul} = (V, E)$ contains multiple inheritance, where V is a set of classes and E is a set of inheritances edges. Then, there is at least one vertex $v \in V$ whose in-degree is ≥ 2.

Proof: For a multiple inheritance graph $G_{mul} = (V, E)$, there exist a set of inheritance relationships, $\{ x_1 \rightarrow y, x_2 \rightarrow y, \cdots, x_n \rightarrow y \} \in G_{mul}$, where $\{ x_1, x_2, \cdots, x_n, y \} \in V$. By multiple inheritance definition, we know that *class* y inherits from more than one parent class. It implies that n is greater than 2. Therefore, in-edges of $y \geq 2$ is hold.

For example, If a class A inherits from two parent classes, class B and class C (see Fig. 4), then the in-edges of class A are 2. However, this case would lead to function name conflictions between the inherited classes. If, for example, both B and C contain a function *print*, it is an ambiguity for class A, because A cannot distinguish it [20]. This function clash is called *name-confliction*. The same problem also exists in repeated inheritance and we will discuss it in the following.

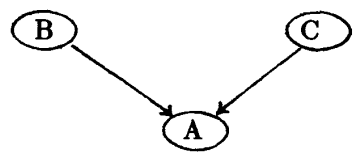


Figure 4. An example of multiple inheritance.

Definition 3: Given a multiple inheritance $G_{mul} = (V, E)$, if there exists a common ancestor class such that the parent classes of V inherit from it, the repeated inheritance is defined as $G_{mul} \cup$ the common ancestor $\cup$ the inhertiances edges between the parent classes and the common ancestor.

For example, given a multiple inheritance $G_{mul} = (V, E)$, where $V = \{ D,B,C \}$ and $E = \{ B \rightarrow D , C \rightarrow D \}$, if there exists a common ancestor class A from which class B and C inherit (i.e., $A \rightarrow B$, $A \rightarrow C$), then the repeated inheritance is constructed by $G_{mul} \cup \{A\} \cup \{A \rightarrow B, A \rightarrow C\}$ (see Fig. 5).

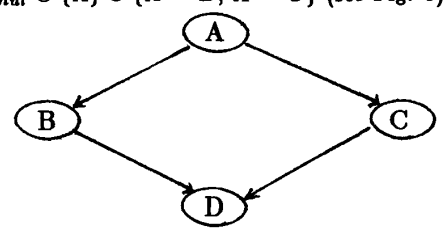


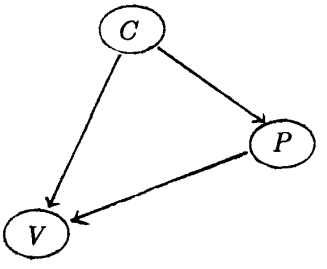
Figure 5. An example of Repeated Inheritance

Lemma 2 : Given a repeated inheritance graph $G_{rep} = (S, F)$, there must exist a multiple inheritance graph $G_{mul} = (V, E)$ which is a subgraph of $G_{rep} = (S, F)$ such that $V \subseteq S$ and $E \subseteq F$ is hold.

Proof: Let $G_{mul} = (V, E)$ be a multiple inheritance, where $E = \{x_1 \rightarrow y, x_2 \rightarrow y, \cdots, x_n \rightarrow y\}$ and $V = \{y, x_1, x_2, \cdots, x_n\}$. By the definition of repeated inheritance [19], there must exist a common ancestor class connected to the parent classes of V. If z is the common ancestor class, then there must exist a set of inheritance relationships (edges), $\{ z \rightarrow x_1, z \rightarrow x_2, \cdots, z \rightarrow x_n \}$ such that z is connected to parent classes $\{ x_1, x2, \cdots, x_n \} \in V$. It leads to a repeated inheritance graph $G_{rep} = (S, F)$ such that G_{mul} is a subgraph of G_{rep}, where $S = V \cup \{z\}$ and $F = E \cup \{ z \rightarrow x_1, z \rightarrow x_2, \cdots, z \rightarrow x_n \}$.

Theorem 1: Let $G = (V, E)$ be a repeated inheritance graph, then the vertex numbers of V ≥ 3 is hold and G contains closed regions.

Proof: By Lemma 1, there must exist at least a vertex v whose in-degree is ≥ 2, that is, v inherits from more than two parent classes. Therefore, the vertex numbers are ≥ 3. By Lemma 2, we know that there is a common ancestor class connected to the parent classes of v. This will lead to a closed region. For example, let the in-degree of v be 2 and vertex c and vertex p are its parent classes. We can choose either one of the two parent classes to be the common ancestor vertex. If class c is chosen, $c \rightarrow p$, $c \rightarrow v$ and $p \rightarrow v$ are hold. It will lead to a closed region shown as belows:



Lemma 3: If $G = (V, E)$ is an inheritance graph containing repeated inheritance, then the *euler's region number* of $G \geq 2$ is hold (i.e, G contains at least one closed region).

Proof: By theorem 1, we have shown that a repeated inheritance graph must contain closed regions. By *euler's formula* [1], if there exists closed regions in a plannar graph, the euler's region number ≥ 2 is hold.

When the class numbers of an inheritance graph grows linearly, the number of repeated inheritances would increase in an exponential rate. It is difficult to find the repeated inheritances, much less test those implicit software errors. Furthermore, the problem of finding out all the repeated inheritances is a *NP-complete* problem which will be discussed in the the next section. Since it is impratical to exercise all repeated inheritances in an inheritance graph, techniques to guide the testing units become important. The idea is that repeated inheritances of an inheritance graph is composed of a set of *unit repeated inheritances (URIs)*, and *name-confliction errors* could be found and solved easily from them. To formally describe the testing methodology, a theorem is presented as below:

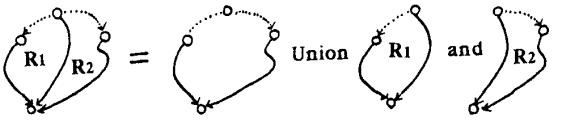
Theorem 2: Let G = (V,E) is an inheritance graph. If it contains repeated inheritances, then the graph G could be decomposed into a set of unit repeated inheritanes (URIs).

Proof: The proof proceeds reversed induction on the *euler region number*. Let G_r = (V,E) be an inheritance graph whose euler region number is r. Suppose we remove a common edge, an edge between any two closed regions, from G_r, then its euler region number will be decreased by one, and two unit repeated inheritances (unit closed regions) will bel lost. Let $G_{r-1}(V, E')$ be the remainder graph after removing a common edge, where its euler region number is *r-1*, E'=E - *an common edge* between two closed edges and the decreased regions must contain the removed common edge.

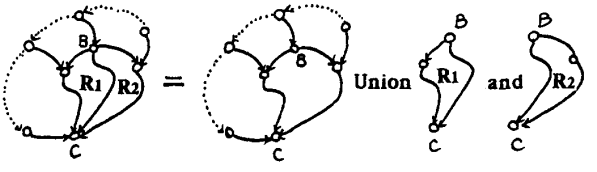
Induction hypothesis:

$G_r(V, E) = G_{r-1}(V, E')$ Union 2 decreased URIs ...(1)

1. The base case is trivial. We observe that for a graph with r = 2 shown as below, (1) is satisfied.



2. Assume that the induction hypothesis is true for abitrary r, and now consider *r-1*. Let us consider figure 6, where $G_{r-2}(V, E'')$ is the remainder graph after removing a common edge from $G_{r-1}(V, E')$. After removing the common edge $\overline{BC}$, the region number is reduced by one and the remainder edges $E'' = E' - \overline{BC}$. There are 2 decreased URIs, region R_1 and region R_2. Therefore, $G_{r-1}(V, E')$ $= G_{r-2}(V, E'')$ Union R_1 and R_2 . $G_{r-1}(V, E')$ statisfies the induction hypothesis and the proof is completed.



$G_{r-1}(V, E')$ $G_{r-2}(V, E'')$

Fig. 6 the induction on the $G_{r-1}(V, E')$

4 Algorithm for Finding URIs

In this section, the algorithm for finding unit repeated inheritances is proposed (see Fig. 7). The data structure of Inheritance relation structure is represented by a directed graph G=(V,E), where V is the set of all classes and E is the set of all inheritance edges. To illustrate this algorithm, many definitions are needed.

Definition:
1. root class: it is a class node with no in-edges.
2. terminal class: it is a class node with no out-edges.
3. Ancestor(v): it is a set which records all the ancestor class numbers of v and itself, where v is a class of V.

Algorithm: Finding Unit Repeated Inheritances (URIs)

Input: A directed graph G(V,E)
Output: Unit repeated inheritances

Step 1. Build a directed graph consists of classes and inheritance edges and initialize that Ancestor(v) = $\{v\}$.

Step 2. Using *bread-first* traverse algorithm to traverses all root classes; in the process of traverse, parent ancestor set is added to its children ancestor sets.

Step 3. For all terminal classes do
if the number of the ancestor set ≥ 2 then
begin
Union (set_i, set_j) { $2 \leq i,j \leq n$ and $set_i \neq set_j$ }
if common parent is found then
record the uniton set.
(i.e., An URI is found)
else if the number of ancestor set = 1
discard the union set (no URI exists)
end
else
no repeated inheritances exist
endif

Figure 7. Algorithm for finding URIs

In figure 8, the root classes are class 1 and 10; the terminal classes are class 5, 8, and 9. To get the ancestor sets for each class, bread-first traversal algorithm is utilized to traverse all the classes individually; in the process of traverse, parent class number is added into the *Ancestor(v)* set where v is the child class of parent class. First, we use the algorithm to traverse the root class 1 to its children class 2, 3, 4, and 5; add class 1 into the sets of Ancestor(2), Ancestor(3), Ancestor(4) and Ancestor(5) (i.e. the set of Ancestor(2) is (1,2), Ancestor(3) is (1,3), Ancestor(4) is (1,4), and Ancestor(5) is (1,5)). Next, we continue traversing class 2 whose children are class 6 and class 7. We get the set of Ancestor(6) is (1,2,6) and Ancestor(7) is (1,2,7). Laterly the ancestor sets of class 3, 4, and 5 can be found by the same way. Similarly, root class 10 can be traversed to find its ancestor sets after the completion of the traverse of root class 1. For each terminal class, union any two sets which has common parent number, then a repeated inheritance(closed region) is found. If in these sets there is no common parent number, then there is no repeated inheritance. For example, consider terminal class 8, there are four ancestor sets (10,6,8), (1,2,6,8), (1,3,8), and (1,2,7,8). When union (1,2,6,8) and (1,3,8), there exists a common parent class 1 between them; a union set (1,2,3,6,8) containing a URI is found. When union (10,6,8) and (1,2,6,8), there is no common parent; we can derive that there is no URI. After all the union operations of the sets of the ancestor set of terminal class 8, there are three URIs (see Fig.8). There is no URI found in the terminal class 5, because the set number of Ancestor(5) is less than 2. There is a URI, (1,3,4,9), found in the terminal class 9, since there are two sets of Ancestor(9) with common parent class 1.

Let us reconsider repeated inheritance graph. If the sharing ancestor or common parent class is removed, then multiple inheritance is get. In other words, a repeated inheritance must contain a multiple inheritance.

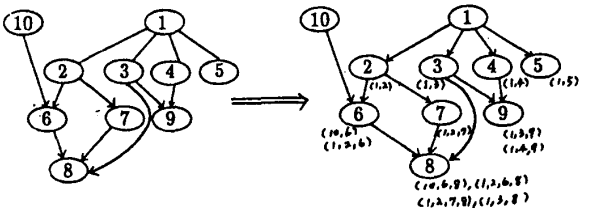
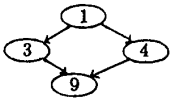


Fig. 8 Illustration of URIs Algorithm

After the *bread-first* traverse, we find that there are three terminal classes, 8, 9, and 5.

1. Class 5 has only one element in the ancestor(5), so there is no repeated inheritance.
2. Class 9 has two elements,(1,3,9) and (1,4,9), in the ancestor(9).
Union((1,3,9),(1,4,9))
=(1,3,4,9)

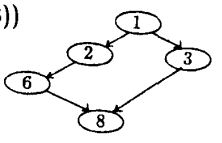


3. Class 8 has four elements in the ancestor(8); they are (10,6,8), (1,3,8), (1,2,6,8) and (1,2,7,8)

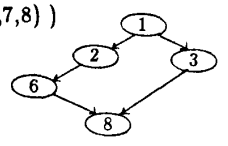
   (a) Union ((10,6,8), (1,2,6,8)) = ϕ
   Union ((10,6,8), (1,2,7,8)) = ϕ
   Union ((10,6,8), (1,3,8)) = ϕ

   The three *union* operations get empty set, because there are no common parents.

   (b) Union ((1,2,6,8), (1,3,8)) =(1,2,3,6,8)



   Union ((1,2,6,8), (1,2,7,8)) = (1,2,6,7,8)



   Union ((1,3,8),(1,2,7,8)) = (1,2,3,7,8)

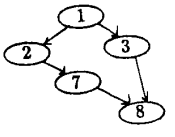


Fig. 8 (continued)

The time complexity of this algorithm is $O(n^3)$. Let n be the numbers of total classes, t be the numbers of terminal classes and $1 \leq t \leq n$, *Ancestor(i)* be the ancestor sets of the ith class, and S_i be the number of the elements in *Ancestor(i)*, for all class $i \in$ terminal classes and $1 \leq S_i \leq n$. For each terminal objects, we perform the repeated inheritance algorithm to find all the basic repeated inheritance component (unit closed region). The time complexity of this algorithm is

$$\begin{aligned}\sum_{i=1}^{t} C\binom{s_i}{2} &= C\binom{s_1}{2} + C\binom{s_2}{2} + \cdots + C\binom{s_t}{2} \\ &\leq C\binom{n}{2} + C\binom{n}{2} + \cdots + C\binom{n}{2} \\ &= t \times n(n+1)/2 \\ &\leq n^2(n+1)/2 \\ &\equiv O(n^3)\end{aligned}$$

URIs Hierarchy Testing Methodology

Suppose that G=(V,E) is an inheritance graph which contains repeated inheritance and its *euler's region number is r*. By the theorems as we have seen above, G could be successfully decomposed into a set of URIs whose region number is 2 to help to detect name-confliction errors and some certain software errors. However, there may be many hidden errors existed in those subgraphs of G whose region numbers are between 3 and *r*. To address these hidden software errors, we have to find all the repeated inheritances, all set of closed regions with different *euler's region numbers* of an inheritance graph. All the repeated inheritances could be found by modifying the algorithm mentioned above. We need only to modify the *union* operation to permit it to union three, four, $\cdots$, and s_i classes from the terminal classes each time, but it is a time consuming task. All the cyclic inheritances could be found as below:

$$\bigcup_{i=1}^{t} Union\binom{s_i}{2}(Ancestor(i)) \cup \bigcup_{i=1}^{t} Union\binom{s_i}{3}(Ancestor(i)) \cup \cdots$$

$$\bigcup_{i=1}^{t} Union\binom{s_i}{n-1}(Ancestor(i)) \cup \bigcup_{i=1}^{t} Union\binom{s_i}{s_i}(Ancestor(i))$$

The time complexity of finding all the closed regions is as follows:

$$\begin{aligned}&C\binom{s_i}{2} + C\binom{s_i}{3} + \cdots + C\binom{s_i}{s_i} \\ &\leq C\binom{n}{2} + C\binom{n}{3} + \cdots + C\binom{n}{n} \\ &= 2^n - C\binom{n}{1} - C\binom{n}{0} \\ &= 2^n - n - 1 \\ &\equiv O(2^n)\end{aligned}$$

Although it outlines all the repeated inheritances, the time complexity is exponetial. To specify these hidden software errors, we classify all the repeated inheritances according to their different *euler region numbers* respectively. We build a hierarchy testing prototypes to cover these hidden software errors corresponding to repeated inheritances with different euler's region numbers. First, we divide all the repeated inheritances into a set of closed regions denoted as URIs(n), where $1 \leq n \leq r$. The hierarchy testing prototypes represented by URI(n) are shown as following:

- URIs(1): require every class in a repeated inheritance graph needs to be exercised at least once.
- URIs(2): require every *unit repeated inheritance* with r = 2 needs to be exercised at least once.
- URIs(3): require every closed region with r = 3 needs to exercised at least once.

$\vdots$

- URIs(r): This testing level is equivalent to test the original inheritance graph.

After finding the hierarchy testing prototypes, we can apply *McCabe' cyclomatic testing strategy* [17] mentioned in section 2 to help test those hidden errors.

URIs(1) is also called *object-testing* [6]; requires every class needs to be tested at least once, clearly, it is equivalent to testing all classes. It is the basic unit testing for any inheritance graph. To proceed on URIs(r) testing for r $\geq$ 2, URIs(1) should be tested in advance. URIs(2) requires every closed region r with r = 2 to be tested at least once. URIs(3) requires every closed region with r = 3 to be tested at least once. URIs(r) faces the original inheritance graph. It is equivalent to testing all inheritance paths of an inheritance graph. To identify the correctness of this hierarchy testing methodology, we should prove that URIs(2), URIs(3), ... ,and URIs(r) contains the same unit repeated inheritances.

Theorem 3: For an inheritance graph, the repeated inheritances could be found from the terminal classes by union of 2, 3, ..., and n classes, respectively. For each union, they will contain the same unit repeated inheritances .i.e.,

$$URIs\left(\bigcup_{i=1}^{t} Union\binom{s^i}{2}(Ancestor(i))\right)$$

$$= URIs\left(\bigcup_{i=1}^{t} Union\binom{s^i}{3}(Ancestor(i))\right)$$

$$= \cdots = URIs\left(\bigcup_{i=1}^{t} Union\binom{s^i}{n}(Ancestor(i))\right)$$

Proof: Suppose $G_r = (V, E)$ is an repeated inheritance graph. By URIs algorithm, we find that

$$G_r = \bigcup_{i=1}^{t} Union\binom{s^i}{2}(Ancestor(i))$$

when the union number is 2,

$$G_r = \bigcup_{i=1}^{t} Union\binom{s^i}{3}(Ancestor(i))$$

when the union number is 3,

$$\vdots$$

$$G_r = \bigcup_{i=1}^{t} Union\binom{s^i}{n}(Ancestor(i))$$

when the union number is n.

By theorem 2, we could derive the fact that

$$URIs\text{-}of\text{-}G_r = URIs\left(\bigcup_{i=1}^{t} Union\binom{s^i}{x}(Ancestor(i))\right)$$

is true for all $2 \leq x \leq n$, the theorem holds.

5 Conclusion

A graph-theoretical testing methodology for object-oriented software is proposed and an algorithm for finding *name-confliction errors* is presented. This testing methodology could detect object-oriented software errors efficiently and reduce the object-oriented software development costs to enhance software quality. Furthermore, the proposed algorithm could be implemented to become an useful tool in object-oriented design phase to detect improper inheritance structures.

Further studies based upon this research are : 1. developing a new object-oriented software metric based on the algorithm to measure the object-oriented software complexity. 2. integrating object-testing to invent a new testing tool to help to develop object-oriented software development environment.

References

[1] Berge, C., *Graph and hypergraphs,* North-Holland, Amsterdam, The Netherlands, 1973.

[2] Grady Booch, "Object-Oriented Development," *IEEE Trans. Software Eng.,* vol.2 no.2 ,Feb. 1986.

[3] Chung, C.M., *A Software Metrics Based Testing Enviroment,*Ph.D. Dissertation, The advanced Center for Computer Studies, Univ, of Southwester Louisisana, Lafayette, Louisiana, 1988.

[4] Chung, C.M. and Yang, M.G., "Software maintability," *Proceedings of Science, Technologies, and Engineering,* Houston, pp. v4-12 May 1988.

[5] Chung, C.M, Edwards, W.R., and Yang, M.G., "A Software Metrics Based Software Environment," *Proceedings of International Computer Symposium,* Dec, 1988, vol.1 pp. 696-703. R.O.C.

[6] Chung, C.M, "Object-oriented Concurrent Programming from Testing View," *Proceedings of National Computer Symposium 1989.* pp 555-565, R.O.C.

[7] Chung, C.M, Yang, M.G., "Testing-Specific Metrics," *Proceedings of Science, Technologies, and Engineering,* Houston, pp. T3-15, May 1989.

[8] Chung, C.M., "Software Development Techniques –Combining Testing and Metrics," *IEEE Region 10 Conference on Communication Systems* , Spet, 1990, Hong Kong.

[9] Chung, C.M., " A Family of Testing Path Selection Criteria", Aug. 1991, *International Journal on Mini and Micro Computers.*

[10] Coad, P. and Yourdon, E. *Object-oriented Analysis.* Yourdon Press, 1990.

[11] Cox, B. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, New York, 1986.

[12] Gannon, J. "Data Abstraction Implementation Specification, and Testing," *ACM Trans on Programming Language System,* vol.3, July, 1981, pp 211-223.

[13] Ellis Horowize and Rajiv Gupta, *Object-Oriented Database with Applications to Case, Networks, and VLSI CAD,* Prenteice Hall 1991.

[14] M. Jackson, *System Development.* Englewood Cliffs, NJ:Prentice Hall, 1983.

[15] Laski J. W. and Korel Bogdan, "A data flow oriented program testing strategy," *IEEE Trans. on software eng.,* Vol. 9, No. 3, May 1983.

[16] McCabe,
T.J. "A Complexity Measurement," *IEEE Transaction on Software Engineering,* SE-2 (4), 1976, pp 308-302.

[17] McCabe, T. J., "Structured Testing: A Testing Methodology Using the McCabe Complexity Metric," *NBS Special Publication, Contract NB82NAAK5518,* U.S. Department of Commerce, National Bureau of Standards, 1982.

[18] McCabe, Thomas.J. "Design Complexity Measurement and Testing," *CACM volume 32* Dec 12, 1989, 1415-1425.

[19] Betrand Meyer, *Object-Oriented Software Construction,* Prentice Hall 1988.

[20] Betrand Meyer, "Eiffel: A Language and Environment for Software Engineering," *The Journal of Systems and Software 1988.*

[21] Ntafos, S.C., "On Required Element Testing," *IEEE Trans. on Softwar ENg,.* vol, SE-10, no.6, pp 795-803, Nov,1984.

[22] Geoffrey A. Pascoe "Elements of Object-Oriented Programming," *BYTE August,1986,* pages 139-144.

[23] Lewis J. Pinson. Richard S. Wiener, *An Introduction to Object-oriented Prgramming and Smalltalk,* Addison-Wesley pp 49-60, 1988.

[24] Rapps S. and Weyuker E. J., "Selection software test data using data flow information," *IEEE Trans. on SE,* vol. SE-11, No. 4, April 1985, pp. 367-375.

[25] Seidewitz, E.*General object-oriented software development: background and experience,* J. Syst. and Software. 19, (1989), 95-108.

[26] Seidewitz, E. and Stark, M." Towards a general object-oriented software development methodology." *Ada Letters, 7* (July/August 1987) pp 54-67.

[27] Ward, P. How to integrate object orientation with structured analysis and design. *IEEE Softw.March, 1989), 74-82.*

[28] Woodward, M.R, Hennel, M.A, and Hedley, D., " A Measure of Control Flow Complexity in Program Text," *IEEE Trans. on Software Eng.,* vol. SE-5, no.1 pp45-50,1979.