Decomposition of multiple inheritance DAGs for object-oriented software measurement

T.K. Shih, C.-M. Chung, C.-C. Wang

Department of Computer Science and Information Engineering, Tamkang University, Tamsui, Taipei, Taiwan 251, R.O.C.
E-Mail: chunwang@dis.tku.edu.tw

Abstract

Software metrics are widely used to measure software complexity and assure software quality. However, research in the field of the software complexity measurement of a class hierarchy has not yet been carefully studied. In this paper, we introduce a novel factor named unit repeated inheritance (URI) and an important method named inheritance level technique (ILT) method to realize and measure the object-oriented software complexity of a class hierarchy. The approach is based on the graph-theoretical model for measuring the hierarchical complexity in inheritance relations. The proposed metrics extraction shows that inheritance has a close relation to the object-oriented software measurement and reveals that overuse of the repeated (multiple) inheritance will increase software complexity and be prone to implicit software errors.

1 Introduction

Software metrics serve as a significant strategy to measure the complexity of a software and also as a mechanism in improving the overall quality of software products. Many software complexity metrics for procedure-oriented languages based on control flow and data flow factors have been proposed. However, these traditional metrics strategies are not sufficient for software development in the object-oriented paradigm due to the new data abstractions and control abstractions, such as inheritance, instantiation, polymorphism, and message passing introduced. More recently, complexity studies have been attempted to consider object-oriented software systems. Chidamber & Kemerer have contributed six candidate metrics (e.g., WMC, DIT, NOC, CBO, RFC, LCOM) to the development and empirical validation of a set of theoretically-grounded metrics of object-oriented design. Li & Henry proposed three groups of object-oriented metrics to evaluate
the relations between their metrics and the maintenance effort in two commercial systems. Kim, Chang, Kusumoto & Kikuno also proposed complexity metrics which can be calculated directly from program codes from two viewpoints: static and dynamic complexities. However, all of these proposed object-oriented metrics mechanisms only focus on measuring the empirical relations between a set of objects, but pay little attention to the inheritance relation between objects and the complexity of an inheritance hierarchy. Thus, there is a need of developing software metrics extraction for an inheritance hierarchy.

The rest of this paper is organized as follows. Section 2 discusses a mathematical graph model of an inheritance graph, introduces the properties of a URI, and proposes mechanisms to decompose a repeated inheritance hierarchy into a number of URIs. Section 3 proposes our object-oriented software metric method, introduces a Finding_URIs_Algorithm to decompose an inheritance graph into a set of URIs, and proves the soundness and completeness of the Finding_URIs_Algorithm. Section 4 presents a vertex splitting procedure to reduce the software complexity of a nested inheritance graph. The final section concludes our contributions and describes future work.

2 Inheritance Relations VS. Graph Representations

One of the main benefits of object-oriented programming languages is that it facilitates the reuse of classes. A class is a template that defines the attributes that an object of that class will possess. A class's attributes consist of data members (instance variables) and member functions (methods). Classes are used to define new classes, or subclasses, through a relation known as inheritance. Inheritance specifies a hierarchical organization on the classes and permits a subclass to inherit attributes from its parent classes and either extend, restrict or redefine them. Generally speaking, a class hierarchy (an inheritance graph) can be one of 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 as nodes, and $E$ is a set of inheritance edges which are ordered relations such that $E = \{(x,y) \mid y$ inherits from $x, \text{and } x,y \in V\}$. However, one of the delicate problems raised by the presence of multiple inheritance is what happens when a class is an ancestor of another in more than one way. This situation is called a repeated inheritance and must be dealt with properly. Meyer suggests that the usage of repeated (multiple) inheritance could easily introduce more errors in the design and implementation phases and thus a repeated (multiple) inheritance should be avoided as much as possible. In the case when it is necessary to use repeated (multiple) inheritances, the complexity measures of an inheritance hierarchy seems to be important.

Definition 1 Given a repeated inheritance graph $G = (V,E)$, that $V$ and $E$ represented as the number of classes and edges in $G$, respectively. If it satisfies $|V| = |E|$ then $G$ is defined as a unit repeated inheritance (URI).
**Lemma 1** For each URI, let $CN$ denote the number of common nodes and $CE$ denote the number of common edges, there must exist a condition $CN - CE = 2$ for the URI.

**Proof:** Assuming that there are two inheritance paths, $p_1$ and $p_2$, construct a URI. Each of $p_1$ and $p_2$ has $m$ and $n$ nodes and thus $m - 1$ and $n - 1$ edges, respectively. According to the definition of URI, the number of nodes and the number of edges must be equal, that is,

$$m + n - CN = (m - 1) + (n - 1) - CE,$$

hence

$$CN - CE = 2.$$

Then we defined a binary operator $\oplus$ to represent the composition of two repeated inheritance graphs $G_1$ and $G_2$.

**Definition 2** Let $G = (V, E)$, $G_1 = (V_1, E_1)$, and $G_2 = (V_2, E_2)$ be graphs, $\oplus$ is a binary operator on the domain of graphs defined by the following logical equation:

$$G_1 \oplus G_2 = G \iff (V_1 \cup V_2 = V) \land (E_1 \cup E_2 = E)$$

The association of the introduced binary operator is from left to right.

**Theorem 1** Let $G = (V, E)$ be an inheritance graph. If it contains repeated inheritances, then the graph $G$ could be decomposed into a set of URIs.

**Proof:** The proof is by induction on the number of Euler regions of an inheritance graph. Let $G_r = (V_r, E_r)$ be an inheritance graph with $r$ Euler regions, suppose that we remove a common edge, $\overline{e}$, which is one of the edges between any two consecutive Euler regions, then the number of Euler regions will be decreased by one, and two URIs will disappear. We assume that the remaining graph is $G_{r-1} = (V_{r-1}, E_{r-1})$, where the number of Euler regions are $r-1$, $V_r = V_{r-1}$, and $E_r = E_{r-1} \cup \{\overline{e}\}$. Then the removed regions are the two URIs $R_1$ and $R_2$ which must contain the removed common edge.

**Induction hypothesis:** $G_r = G_{r-1} \oplus R_1 \oplus R_2$.

- If $r = 2$, then the claim is true because we observe that an inheritance graph with 2 Euler regions satisfy the base case as Figure 1.
  Thus, $G_2 = G_1 \oplus R_1 \oplus R_2$.
- We now assume that the induction hypothesis is true for an inheritance graph with $r$ Euler regions, and prove that this assumption implies that an inheritance graph with $r+1$ Euler regions holds the induction hypothesis. Let $G_{r+1} = (V_{r+1}, E_{r+1})$ and $G_r = (V_r, E_r)$, we know from the property of a graph that $V_{r+1} = V_r$ and $E_{r+1} = E_r \cup \{\overline{e}\}$,
Figure 1: An inheritance graph with 2 Euler regions

suppose that $bc$ is a common edge between the two Euler regions existing on $G_{r+1}$, not on $G_r$. By the induction hypothesis, $G_{r+1} = (V_{r+1}, E_r \cup \{bc\}) = (V_r, E_r \cup \{bc\}) = ((G_{r-1} \oplus R_1 \oplus R_2) \oplus (V_r, \{bc\})) = G_r \oplus R_1 \oplus R_2$, which is exactly what we wanted to prove. So the Theorem is complete.

For a class hierarchy, repeated inheritances may be very complex and difficult to measure. However, any complex repeated inheritance can be decomposed into a set of URIs. Theorem 1 reveals this property.

Adjacent matrix $\Omega$ is used to represent an inheritance graph $G$. The detail is described as follows.

**Definition 3** Let $\Omega$ be the adjacent matrix of an inheritance graph $G = (V, E)$, the $i, j$ entry of $\Omega^r$ is the number of different paths of length $r$ from $v_i$ to $v_j$, where $r$ is the times of transition and $v_i, v_j \in V$.

Let $\Omega$ be the adjacent matrix of an inheritance graph $G$. Every entry $m_{i,j}$ is either 1 or 0, where

$$
\Omega = [m_{i,j}]_{n \times n} \begin{cases} 
m_{i,j} = 1, & \text{if there exists a directed edge from } v_i \text{ to } v_j \text{ and } i \neq j. \\
m_{i,j} = 0, & \text{otherwise.}
\end{cases}
$$

We have $\Omega^r(i, j)$ representing the $m_{i,j}$ entry of $\Omega^r$.

**Definition 4** Let $G = (V, E)$ be an inheritance graph, $i$ and $j$ associate with two vertices of $G$ ($v_i$ and $v_j$, respectively), and $r$ be an arbitrary positive integer. The following notations are used:

1. Root is a set of vertices of an inheritance graph with no in-edges.
2. $|\Omega^r(i, j)|$ denotes the number of different inheritance paths of length $r$ between vertex $v_i$ and vertex $v_j$.
3. $L_r(i, j)$ denotes a set of inheritance paths of length $r$ from vertex $v_i$ to vertex $v_j$, where vertex $v_i \in \text{Root}$.
4. $R_r(i, j)$ defines a set of repeated inheritances obtained by the union of different paths in $G$, where $|\Omega^r(i, j)| \neq 0$. 

5. \( \text{Region}_{r}(i, j) \) denotes a set of repeated inheritances rooted at vertex \( v_i \) and terminated at vertex \( v_j \) in \( G \), where vertex \( v_i \in \text{Root} \).

**Theorem 2** A set of all repeated inheritances, denoted \( \text{Region}_{r}(i, j) \), can be found by the following formula:

If \( 2 < r \leq n \),

\[
\text{Region}_{r}(i, j) = R_r(i, j) \lor \bigcup_{p=2}^{r-1} (R_r(i, j) \lor \text{Region}_p(i, j)) \lor \bigcup_{p=1}^{r-1} (R_r(i, j) \lor L_p(i, j)) \lor \bigcup_{p=2}^{r-1} (L_r(i, j) \lor \text{Region}_p(i, j)) \lor \bigcup_{p=1}^{r-1} (L_r(i, j) \lor L_p(i, j))
\]

where \( \lor \) is a disjunction operator representing different alternatives, \( i \in \text{Root} \), and \( n \) is the length of the longest path starting from \( v_i \).

If \( r = 2 \),

\[
\text{Region}_{2}(i, j) = R_2(i, j) \lor (R_2(i, j) \lor L_1(i, j)) \lor (L_2(i, j) \lor L_1(i, j)), \text{ where } i \in \text{Root}.
\]

Otherwise, \( \text{Region}_{r} \) is undefined.

Theorem 2 proposes a method to find all repeated inheritances from an inheritance graph. In fact, by Theorem 1, a repeated inheritance can be decomposed into a set of unit repeated inheritances (URIs) as the basis of an object-oriented software metric in a class hierarchy. In the next section, we discuss how to use URIs to develop an object-oriented software metric.

### 3 An Inheritance Level Technique Method

Suppose \( G=(V,E) \) is an inheritance graph, by graph theory, \( G \) possesses the property of transitivity. Therefore, we can develop an inheritance level-based metric to count number of URIs within each level, named \( \text{ILT}(0) \), \( \text{ILT}(1) \), \( \text{ILT}(2) \), ..., \( \text{ILT}(n) \), where \( n \) is the length of the longest inheritance path in an inheritance graph plus 1. The higher the value of \( n \) is, the more complex and error prone the inheritance hierarchy will be. By Theorem 1 and 2 as the bases of our inheritance level technique discussed in this section, both \( \text{ILT}(0) \) and \( \text{ILT}(1) \) cannot contain any URIs. Thus we start to count the number of URIs at each inheritance level from \( \text{ILT}(2) \).
• ILT(2): Every URI with inheritance path of length 2 at this level needs to be counted.

• ILT(3): Every URI with inheritance path of length 3 at this level needs to be counted.

...:

• ILT(n): This level signifies that the inheritance level-based metric is complete and then we sum up the number of URIs at each level.

Therefore, the complexity of an inheritance graph is defined to be the number of URIs shown as follows:

**Definition 5** Let \( G=(V,E) \) be an inheritance graph.

1. A root class is a class node with no in-edges in \( G \).
2. A terminal class is a class node with no out-edges in \( G \).
3. \( \text{Ancestor}(v) \) is a set which records all different inheritance paths that start at a root (class) and end at \( v \).

**Algorithm**: Finding Unit Repeated Inheritances (URIs);

**Let** \( CN \) denote the number of common nodes and \( CE \) be the number of common edges;

**Input**: A directed graph \( G = (V,E) \);

**Output**: A set of Unit Repeated Inheritances;

**Step 1**: Build a directed graph consisting of classes and inheritance edges and initialize \( \text{Ancestor}(v)=\{v\} \), for all vertices in \( V \);

**Step 2**: Going breadth-first, traverse for all root classes; in the process of traversal, every element of \( \text{Ancestor}^{-1}(v) \) is added into \( \text{Ancestor}(v) \), where \( v \) is a parent class of \( v' \);

**Step 3**: For all terminal classes do

- if the number of the ancestor sets \( \geq 2 \) then
  - begin
    - Union \( \text{set1}, \text{set2} \) \( \{\text{set1} \neq \text{set2}\} \);
    - if there exist common parents, record the union set then
      - if \( CN - CE = 2 \) then
        - A unit repeated inheritance is found;
      - else no URI exists;
    - endif;
  - endif;
- else no repeated inheritances exist, return an empty set;
endif;
In the Algorithm, we utilize the *breadth-first search* to traverse all classes in order to obtain the *Ancestor* set of each class. In the process of traversal, all ancestor classes of each class are added into the *Ancestor* set of the class. Then, we check whether each terminal class has an *Ancestor* set of cardinality greater than 2. If the *Ancestor* set of a terminal class contains two or more elements, we unite any two elements in the *Ancestor* set. Next, if every union set exists common parents and the number of nodes of the union set is equal to the number of edges, a URI is found; otherwise, we discard the union set because it cannot satisfy the property of a URI. If the number of elements of an *Ancestor* set of the terminal class is less than 2, there are no repeated inheritances, and the algorithm returns an empty set. In the following subsection, we prove that the algorithm is *sound* and *complete*.

3.1 **Soundness and Completeness of the Algorithm**

**Lemma 2 (Soundness of the Finding_URIs_Algorithm).** The repeated inheritances generated from the inheritance graph $G$ by the algorithm must be URIs. 

*Proof:* By breadth-first traversal search (BFS), we can associate BFS numbers with nodes. That is, a vertex has a BFS number $l$ if it was the $l$th vertex to be visited by the BFS.

Assuming that there are two arbitrary complete inheritance paths, $set1$ and $set2$, in $Ancestor(v)$, where $v$ is one of the terminal classes in the inheritance graph, each has $m$ and $n$ nodes and thus $m-1$ and $n-1$ edges, respectively. We unite the two paths. If the union set satisfies $CN-CE=2$, then

- total number of nodes = $m + n - CN$,
- total number of edges = $(m-1)+(n-1)-CE = m+n-2-(CN-2) = m+n-CN$.

We find that the number of nodes is equal to the number of edges. Thus the repeated inheritance generated from the Algorithm is a URI. As a result, the Algorithm is sound.

**Lemma 3 (Completeness of the Finding_URIs_Algorithm).** If there exists a URI in the inheritance graph $G$, the URI must be in the set of URIs obtained by the Finding_URIs_Algorithm.

*Proof:*

1. Due to the property of repeated inheritance, there must exist two or more different inheritance paths in the repeated inheritance. Therefore, if there is only one element in the *Ancestor* set of a terminal class, we cannot get any repeated inheritances. So, the "if the number of the ancestor set $\geq 2$" is a necessary condition.
2. If there exist no common parents of any two paths, that is, the only common node in the union set is the terminal class, we have \( CN = 1 \) and \( CE = 0 \). Thus, the total number of nodes is \( m + n - 1 \), and the total number of edges is \( (m - 1) + (n - 1) - 0 = m + n - 2 \). This does not satisfy the property of URIs. We can conclude that if there exists no common parent of two paths, there is no URI. Therefore, the condition that “if there exist common parents” in the algorithm is necessary.

3. According to the property of URIs, it must be true that \( CN - CE = 2 \) for each repeated inheritance. That is, we can deduce from \( CN - CE = 2 \) that the number of common nodes is equal to the number of common edges within the repeated inheritance. Therefore, the “if \( CN - CE = 2 \)” is a necessary condition.

Per the above discussion, the three “if” expressions in the algorithm are all necessary conditions. If there exists a URI in the repeated inheritance, it must be selected by the Algorithm. Therefore, the Algorithm is complete.

3.2 An Illustration of the Algorithm

Let \( n \) be the number of total classes, \( t \) be the number of terminal classes, where \( 1 \leq t \leq n \), and \( S_i \) represents a set containing all elements in \( \text{Ancestor}(i) \), for each class \( i \in \text{terminal classes and } 1 \leq |S_i| \leq n \). For each terminal class \( i \), we can unite the set of \( \text{Ancestor}(i) \) to find all URIs. Therefore, given a repeated inheritance, the number of URIs are derived.

\[
\text{URI}(R) \in \bigcup_{i=1}^{t} U_i,
\]

where \( U_i = \{x|a, b \in S_i, a \cup b = x\} \), and \( R \) is a given repeated inheritance.

And the complexity of the repeated inheritance is:

\[
|\bigcup_{i=1}^{t} U_i|
\]

Let us illustrate the two equations using an example. In Figure 2, the root class is class 1, and the terminal class is class 7. First, we use the breadth-first traversal algorithm to start from the root class 1 pass its children classes 2, 3, and 4, and then add class 1 into the sets of \( \text{Ancestor}(2) \), \( \text{Ancestor}(3) \), and \( \text{Ancestor}(4) \) (That is, the set of \( \text{Ancestor}(2) \) is \( \{1,2\} \), \( \{1,3\} \) is for \( \text{Ancestor}(3) \), and \( \{1,4\} \) is for \( \text{Ancestor}(4) \)). Next, we continue on traversing class 2, whose child is class 6; and \( \text{Ancestor}(2) \) is added into \( \text{Ancestor}(6) \). We obtain that the set of \( \text{Ancestor}(6) \) is \( \{1,2,6\} \). The ancestor sets of 3 and 4 can be found in a similar manner. For each terminal class, with union of two sets having a common parent number, a repeated inheritance is found. If there is no common parent number, there is no repeated inheritance. Therefore,
Figure 2: An illustration of an inheritance level-based metric. There are three URIs: (1,2,3,6,7), (1,3,4,7), and (1,2,4,6,7) there are ancestor sets, (1,2,6,7), (1,3,7), and (1,4,7) in Ancestor(7) (i.e., \(|S_i| = 3\)). When there is a union of (1,2,6,7) and (1,3,7), there exists a common parent class 1 between (1,2,6,7) and (1,3,7). And then a union set (1,2,3,6,7) containing a URI is found. After performing all union operations on Ancestor(7) of terminal class 7, by Eq. (1) and (2), there are three URIs in Figure 2 as follows.

\[
| \bigcup_{i=1}^{1} U_i | = 3
\]

4 Vertex Splitting in Inheritance Graphs

When we design a class hierarchy, the repeated inheritance graph can be divided into two types: parallel inheritance graph and nested inheritance graph. Both are different in terms of structure and complexity. The inheritance-based metric shows that the complexity of a nested inheritance will be more complex than that of a parallel inheritance. The difference between their complexities can help us reduce the software complexity of the class hierarchy while we design a class hierarchy.

Definition 6

- A parallel inheritance graph is a graph such that there are no URIs containing any URIs.
- A nested inheritance graph is a graph such that there exists a URI containing at least one URI.
(a) Parallel Inheritance and its Complexity

Let us consider the following inheritance graph (Figure 3), which is a parallel inheritance graph because both URIs, (abcf) and (adeg), do not contain any URIs.

(b) Nested Inheritance and its Complexity

Let us consider the following inheritance graph (Figure 4). It is a nested inheritance because there exists a URI (abge) containing a URI (acdfg).

Although the number of classes in Figure 3 and Figure 4 are the same, the URI complexity of Figure 4 is higher than that of Figure 3. The comparative result shows that the nested inheritance graph is more complex than the parallel inheritance graph. The difference of the complexity indicates that the parallel inheritance is preferable to the nested inheritance when we design a class hierarchy. Based on this result, we had better, if possible, translate the nested inheritance into the parallel inheritance. In the following, we present a rule, based on this inheritance-based metric, to simplify a class hierarchy. This rule is a translation procedure to transform a nested inheritance graph into a parallel inheritance graph in order to reduce the complexity of the class hierarchy. The main step is to partition the root class into several disjoint classes by sacrificing part of the reuse functions (methods). Note that just some nested inheritances can satisfy the rule because of the fact that the split of root classes involves the problems of semantic checking and function renaming. We leave this topic to further research. In this paper, we only present a strategy to reduce the complexity of a class hierarchy when the nested inheritance graph encounters.
A Vertex Splitting Translation Procedure:

Step 1: By BFS, record each the out-edge of the root vertex, denoted by $out(root)$.

Step 2: Select a root vertex with the highest degree of $out(root)$, denoted $root_r$, from the set of the root vertices.

Step 3: If $out(root_r) = 1$, then the root vertex cannot be split.

else if $(out(root_r) \mod 2 = 0)$, then we partition the root vertex equivalently into two vertices, $r_1$ and $r_2$, where $r = r_1 \cup r_2$.

else if $(out(root_r) \mod 2 \neq 0)$, we partition the root vertex into $out(root_r)$ vertices, $r_1, r_2, \ldots, r_k$, where $k$ is an odd number and $r = r_1 \cup r_2 \cup \ldots \cup r_k$.

Step 4: Repeat step 2 until the set of root vertices is empty.

The following is an illustration of the translation procedure. In the translation process, a nested inheritance graph will be transformed into a parallel inheritance graph. In Figure 5, its complexity is 6 before using the translation procedure.

After the translation procedure, the complexity of the graph is reduced to 2 in Figure 6. The reduction process is illustrated as follows:

A Reduction Process:

Step 1: Root vertex $a$ contains a nested inheritance graph.

Step 2: The vertex $a$ is the only one to be selected.

Step 3: Partition $a$ into $a_1, a_2,$ and $a_3$, where $a = a_1 \cup a_2 \cup a_3$.

Step 4: The reduction process terminates because there are no root vertices containing nested inheritance graphs.

5 Conclusion

In this paper, the new concept named URI was introduced and the ILT method based on URIs was proposed to help programmers sum up the number of URIs at each level, which makes the complexity measures of class hierarchies become easy. Therefore, the inheritance-based level metric
will be a good blueprint to improve the correctness and quality of object-oriented software systems.

Our approach is very different from the McCabe’s cyclomatic complexity discussed in the following two ways. First, a class hierarchy is declarative while cyclomatic complexity is based on the operational semantics of condition statements. Moreover, even if one uses cyclomatic complexity to measure a class hierarchy, a node containing more than two children nodes needs to be further decomposed and the issue of cyclomatic complexity needs to be refined. Therefore, our approach deals with problems different from cyclomatic complexity.

References


