Cyclic Inheritance Detection for Object-Oriented Database

Ding-An Chiang and Ming-Chi Lee

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

Abstract

Inheritance is the main theme of schema design for the object-oriented software and object-oriented database. 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 [Cox86][Meyer88]. However, misuse of inheritance will lead to cyclic inheritance which suffers from redundant classes and endless self-inheritance. Unfortunately, for a class hierarchy with cyclic inheritance, to detect all the cyclic inheritances is a NP-complete problem. This paper describes a graph-theoretical reduction methodology to reduce them in a polynomial time. An algorithm to support this reduction is also presented.

1 Introduction

Object-oriented design strategy is a new promising approach for developing databases to reduce data redundancy and enhance data reusability. One of the 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; a class inherits all properties defined for its superclasses, and can have additional properties local to itself. The notion of property inheritance and class hierarchy enhances application programmer’s productivity by facilitating top-down design of the database as well as applications. The class hierarchy is usually represented by a directed graph, called inheritance graph.

Although inheritance mechanism supports the good properties of reusability and incremental design [Cox86][Meyer88], misuse of it will lead to three trouble problems: function/method name-confliction, redundant inheritance, redundant is-a relationships and cyclic inheritance. The first two problems will be introduced briefly in this paper, instead of being solved. We focus on the third problem — cyclic inheritance.

The first problem, function name-confliction is caused by multiple inheritance. As the classes are arranged to represent the is-a relationships among them, it is possible for a class to inherit properties from several superclasses. This situation is called multiple inheritance [Sed86]. It leads to possible function name-conflictions between properties inherited from various superclasses [Bor82][Car88]. Another source of conflict arises from the possibility that a locally-defined class variable or method may have the same name as the one that is inherited. These conflicts are partially resolved by giving the local definition precedence. Other conflicts are resolved based upon a user-supplied total ordering of the superclasses [Car92].

The second problem is the redundant inheritance. Now, consider the situation in which the user, for three given classes A, B, and C, declares A to be a subclass of B, B to be a subclass of C, and A to a subclass of C. Since is-a relationship is transitive, the last declaration “A is a subclass of C” is redundant in the sense that it can be derived from other is-a relationships. We call such declaration, “A is a subclass of C” a redundant inheritance.

The third problem is cyclic inheritance problem. Now, consider the situation in which the user, for three given classes A, B and C, declares A to be a superclass of B (denoted by A → B), B to be a superclass of C (denoted by B → C), and C to be a superclass of A (denoted by C → A). By the property of transitivity, we find that such declaration will lead to a circuit (A → B → C → A). A, B and C are actually the same class (i.e., A = B = C). The circuit is also called, cyclic inheritance and these classes on the circuit are called redundant classes [Horowitz 81].

The situation of cyclic inheritance is undesirable because it suffers from the endless self-inheritance. The class hierarchy design of object-oriented database (OODB) tends to be modified frequently during the software or database lifetime and users tend to arrive at a preliminary design through trial and error using the schema.
change operations [Banerjee 87]. After the user modifies the class hierarchy, the resulting class hierarchy is prone to enter the cyclic inheritance which leads to be in an inconsistent and redundant state.

To enhance the correctness of a class hierarchy design, the cyclic inheritance should be detected and eliminated. One approach to solving the problem is finding out all the cyclic inheritances and getting rid of them. However, the detection of all the cyclic inheritances is proved to be computationally hard. Instead of doing so, we reduce these redundant classes to one class by utilizing the refine or rename strategies [Meyer 88]. This reduction will help us to get rid of cyclic inheritances in a polynomial time for a class hierarchy. For most cases, the redundant classes could be reduced to one class. However, in some cases, we cannot reduce these redundant classes into one class successfully, because they involve the problems of type-classing and semantics. In this paper, we are not concerned with these problems but assume that redundant classes could be reduced to class without any prework.

How to relax the assumptions is left to further researches.

In the next section, we use graph-representation to introduce the core concepts of inheritance mechanism, including single inheritance, multiple inheritance and cyclic inheritance. In section 3, a cyclic inheritance detection and reduction algorithm is proposed. Although detecting whether a class hierarchy contains cyclic inheritance is O(k) where k is the number of inheritance edges, finding out all cyclic inheritances (cycles) is an NP-complete problem. We propose a graph-theoretical reduction process to reduce all the cyclic inheritance in a polynomial time O(nk^4). In section 4, we analyse the time complexity of this algorithm. Future researches are recommended in the final section.

2 Inheritance and Graph Theorems

For a class hierarchy design, there is an inheritance relationship C1 → C2, if class C1 is a superclass of C2. The class-superclass relationship C1 → C2 is an "is-a" relationship in the sense that every instance of a class is also an instance of the superclass. Using the terminology of the entity-relationship model, we say that C1 is a generalization of C2 and C2 is a specialization of C1. We map the is-a relationship into an edge which connects vertices C1 and C2 in a graph representation.

An inheritance graph could be 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 → y | y inherits from x, where x and y ∈ V }. For an inheritance graph, it could be divided into three basic types: single inheritance graph, multiple inheritance graph, and cyclic inheritance graph.

First, a single inheritance is that each class inherits uniquely from one parent class. It is a tree structure and no one class inherits from more than one parent class.

Second, if a class is permitted to inherit from more than one parent class, it is called a multiple inheritance [Stefik 86] [Cardelli 84]. For example, if a class A inherits from two parent classes, B and C (B → A, C → A), this class is called multiple inheritance and prone to lead to name-conflcitions. If, for example, both B and C contain a function push(stack, element), it is an ambiguity for class A, because A cannot distinguish it [Meyer 88].

Third, for a class hierarchy with n classes, a cyclic inheritance is a sequence of relationships C1 → C2, C2 → C3, ... , Cn-1 → Cn, Cn → C1. (see figure 1)

Figure 1. An example of cyclic inheritance.

In this case, we find that such declaration will lead to a circuit in a class hierarchy. This is an improper design because it suffers from endless self-inheritance by the property of transitivity. What is much worse still is that it leads to redundant classes (C1 = C2 = ... = Cn). All the classes on the circuit are the same. In the following, we derive a theorem to characterize the redundant class problem.

* Theorem 1: If G = (V, E) is an inheritance graph with cyclic inheritance, then G must contain at least one circuit (closed region).

Theorem 1 reveals that whether a class hierarchy contains a cyclic inheritance is equivalent to checking whether it contains a circuit. Although it shows that cyclic inheritance leads to a circuit, how to detect a circuit is an another problem. Fortunately, this detection is not difficult because it could be done easily by depth-first search. In the following, we propose a graph theorem to show that time complexity of detecting a cyclic inheritance in
a class hierarchy is proportional to the number of inheritance edges. This theorem will help to reduce all cyclic
inheritances in a polynomial time in the next section.

Theorem 5: Let \( G = (V, E) \) be a directed graph. Then, the time complexity of determining whether it contains
a cyclic inheritance is \( O(k) \), where \( k \) is number of inheritance edges in \( E \).

By theorem 2, we already show that detecting whether a class hierarchy contains cyclic inheritances is equivalent
to checking whether it contains circuits. Therefore, finding all cyclic inheritances is equivalent to detecting all
circuits. However, for given a graph, finding out all circuits is a \( \text{NP-complete} \) problem [Manber 89]. That is, if we
want to detect all cyclic inheritances to removes them by resetting all the inheritances between classes and their
superclasses, this approach will suffer from a \( \text{NP-complete} \) problem.

Theorem 5: Finding out all cyclic inheritances is a \( \text{NP-complete} \) problem.

In the next section, instead of actually finding out all cyclic inheritances, we present a graph-theoretical
methodology to reduce these circuits.

3 Cyclic Inheritance Reduction

In this section, a graph-theoretical methodology to reduce cyclic inheritances in a polynomial time is presented.
Theorem 2 shows that finding a cyclic inheritance is \( O(k) \), where \( k \) is the number of inheritance edges. We will
take the advantage of this property to help to develop the reduction methodology. The idea is that we find a
cyclic inheritance each time and reduce these redundant classes to one class. Repeat this process until there is
no cyclic inheritance.

An inheritance graph \( G = (V, E) \) containing a universe (dummy) class is built to represent the class hierarchy
design, where \( V \) are the set of classes, and \( E \) are edges representing all the inheritance relationships between all
classes. This universe class is used to connect all the separated class hierarchies such that the \( G \) is a connected
graph [Tsuda 91] (see figure 2). For an OODB, a class hierarchy design is usually composed of a set of subgraphs
(separated class hierarchies), the advantage for adding this universe class is that it ensures a connected graph
which will help to reduce the cyclic inheritance by depth-first traverse.

\[
\text{universe class}
\]

\[
\text{class hierarchy-1} \quad \ldots \quad \text{class hierarchy-n}
\]

Fig.2 universe class connects all separated class hierarchies.

This reduction must start from the universe class and traverse all the class hierarchies from 1 to \( n \), suppose
there are \( n \) class hierarchies. We add this universe class to connect all the separated class hierarchies.

Cyclic Inheritance Reduction

This reduction task is proceeded in a \textit{shift-and-reduce} manner. This manner simplifies the relationships between
classes but it needs an extra memory (stack) to keep those shifted classes before finding a cyclic inheritance. We
shift in the classes in the order of the is-a relationships and check whether a cycle is found. If a circuit is found,
a reduction is performed.

A \textit{shift} action is defined as travelling the inheritance graph along the inheritance edges by depth-first traverse.
For each being traversed class \( v \), we must check whether it has appeared in stack or not, if it does not appear
yet and \( v \) is not terminal class, then push it into stack. Otherwise, if it is a terminal class, then pop all terminal
classes in the top of the stack. Here, we define the \textit{terminal class} as a class whose descendant classes have been
traversed already. If \( v \) does not satisfy both the two conditions, then there must be a cyclic inheritance in the
stack and we have to pop classes from the stack until \( v \) is encountered [i.e., a cyclic inheritance]. In fact, at this
moment all the popped classes are just equal to the set of redundant classes on this reduced cyclic inheritance
and they could be reduced into one class. After the reduction, the reduced class should be pushed into stack and
we shift in next is-a relationship. Note that checking whether a being traversed class \( v \) has appeared or not needs
an examination to the stack. That is we need to search for the stack to check whether the class \( v \) has appeared
in the stack. The search time which is proportional to the numbers of classes in \( V \) is \( O(n) \). The \textit{shift-and-reduce}
algorithm is described as follows:
Cyclic Inheritance Algorithm

A depth-first search of G is carried out beginning at the universe class p. A stack is initialized with p. A class array visited[] initially set false.

Input: an inheritance graph G = (V, E), where V is the class sets and E is a set of is-a relationships

Output: a directed acyclic graph G' = (V', E')

begin

visited[p]=true;

for each vertex v adjacent to p and not visited[v]

begin

shift an is-a relationship (p -> v) along the inheritance edges by depth-first traverse

check whether vertex v appeared in stack;

if v is not in stack and v is terminal node then

push it into stack;

CIR(v);

else if v is terminal class then

visited[v]=true;

pop terminal classes;

close

while (stack[top] # p and stack[top] is not U)

check whether vertex v appeared in stack;

if stack[top] ≠ p and stack[top] is not v then

pop stack;

reduce all the popped classes (redundant classes) to a new class;

push the new class into stack.

end {for}

end

The algorithm of the reduction of a cyclic inheritance is presented above. In the following, a theorem to ensure that the resulting graph must be a DAG is presented.

Theorem 4. Let G = (V, E) be a connected directed graph with cyclic inheritances, and let G' = (V', E') be the reduced graph of G by the reduction process above. Then, G' must be a directed acyclic graph.

4 Time Complexity Analysis

In the previous section, we have proved that the algorithm ensures a DAG. In the following, we want to analyze the performance (time complexity) of the algorithm. The performance is proportional to the number of inheritance edges O(k). When k = 2, if there exists a cyclic inheritance on them (A -> B and B -> A), in this case, class A is equivalent to class B, we can reduce either one of them. The reduction time is a constant. The reduction is shown as below.

Now consider the case of k ≥ 3 and assume that G' = (V', E') is the resulting graph after reducing a circuit (a set of redundant classes) from the G=(V,E), where k1 is the number of E' and k2 is number of reduced inheritance edges. By theorem 2, we already know that time complexity of detecting a circuit is O(k). Therefore, since G' ⊆ G, the reduction of G' is O(k1). Repeat this step until the inheritance graph contains no cyclic inheritance. Suppose t cyclic inheritances are found to be reduced by depth-first traverse and the remaining inheritance edges for each reduction are k1, k2, ..., kn respectively, where i < n. Then, the total reduction time is O(k1) + O(k2) + ... + O(kn) = O(nk). Therefore when a cyclic inheritance graph, we can reduce it to a directed cyclic graph (DAG) in a polynomial time O(nk) under a worst case, where n are the number of classes and k are the number of inheritance edges.

5 Conclusion

In this paper, we reveal that improper class hierarchy design will lead to cyclic inheritance. Detecting all the cyclic inheritances is a NP-complete problem. We propose a graph-oriented reduction methodology to reduce all the cyclic inheritances in a polynomial time O(nk). An algorithm to support this reduction is also presented. However, we ignore the problems of type checking and semantics between those redundant classes during the reduction. The problem of how to ensure the correctness of the reduction is left to future researches. In addition,
the redundant is-a problem is undiscussed in this paper. In fact, to detect all the redundant is-a relationships is another NP-complete problem. However, with the limitation of space, we will introduce it in another paper.

References


