STRENGTHEN AND SUPPORT THE MAINTENANCE OF OBJECT-ORIENTED SOFTWARE

Ming-Chi Lee
Dept. of Business Administration, National Ping Tung Institute of Commerce
Taiwan, R.O.C.
E-mail: lmc@sun1.npic.edu.tw

Timothy K. Shih and Teh-Sheng Huang
Dept. of Computer Science and Information Engineering, Tamkang University
Taiwan, R.O.C.

ABSTRACT

Inheritance is one of the most common features of object-oriented languages, and has been widely applied to develop large and complex software system. However, designing a suitable inheritance hierarchy, involving redundant inheritance, is a difficult task and easily suffers from *name-confliction* and *repeated inheritance* which are error-prone and difficult to test. In this paper, we explain how redundant inheritance makes object-oriented programs difficult to test and maintain, and we give a concrete example of the problems that arise. We show that the difficulty lies in the fact that we lack an effective detection tool suited for work with inheritance problems. Therefore, a formal checking mechanism is proposed to detect and resolve redundant inheritance. Furthermore, this checking mechanism can be easily incorporated with object-oriented CASE tool to enhance software quality.

Keywords: Inheritance Hierarchy, Redundant Inheritance, Repeated Inheritance, Inference Rule, Object-Oriented Program

INTRODUCTION

Inheritance is a relationship among classes wherein one class shares the structure or behavior defined in one (*single inheritance*) or more (*multiple inheritance*) other classes. Class hierarchy consists of a set of ordered inheritance relationships and is often denoted as a directed acyclic graph. It plays a vital role and a fabric in object-oriented design (OOD). However, designing a suitable inheritance hierarchy, especially with multiple inheritance, is a difficult task (Moises et al 1992). This is because several problems associated with multiple inheritance remain still in debate, such as *name collisions* and *repeated inheritances* (Cardelli 1984, Meyer 1988). Booch (1991) showed that we have never been able to define a class hierarchy right the first time except for trivial small cases. In practice, this design of class hierarchy is an incremental and iterative process. If you allow multiple inheritance into a language, then sooner or later someone is going to write a *redundant inheritance*. For example, given a class hierarchy $\Omega=\{ \alpha_1\rightarrow\alpha_2, \alpha_1\rightarrow\alpha_3 , \alpha_2\rightarrow\alpha_4, \alpha_3\rightarrow\alpha_4 \}$. if we add a new inheritance relationship, $\alpha_1\rightarrow\alpha_4$, then this inheritance $\alpha_1\rightarrow\alpha_4$, is called a redundant inheritance because α_4 could inherit indirectly from α_1. After adding $\alpha_1\rightarrow\alpha_4$, α_4 inherits twice (or more) from α_1. This situation suffers from *name-confliction* problem and the ambiguous method invokation. As a consequence, many implicit software faults very much difficult to test are generated (Chung & lee 1997). Although some object-oriented programming languages permits this writing, it is necessary to detect and refine them before coding (Meyer 1988). We argue that an inheritance-based checking mechanism is essential for effective testing and maintenance of object-oriented programs. In addition, it is worthy to note that an inheritance hierarchy is dynamic rather than static during the lifetime of OO software development. It is therefore almost impossible to maintain an unambiguous and nonredundant inheritance hierarchy forever without a checking mechanism.

In Section 2, we introduce the redundant inheritance problem associated with repeated inheritance and name conflictions. In Section 3, inference rules and the inheritance constraints are introduced. Meanwhile, we show how to compute the closure set of an inheritance hierarchy by *inference rule*. In Section 4, a concept of *minimal class hierarchy* is proposed to address the issue of redundant inheritance. Also, a simple but delicate method to compute the closure of a set of classes with respect to a class hierarchy is introduced. In Section 5, we derive an algorithm based on the concept of *minimal class hierarchy* to determine the redundant inheritances in an inheritance hierarchy. Three approaches to resolving redundant inheritances are presented. Finally in Section 6 the conclusion and future research is presented and discussed. .

REDUNDANT INHERITANCE AND BASIC WORKS

Basically, inheritance relationships have two different structures. One is *single inheritance* which allows every class inherits from at most one superclass. In contrast, multiple inheritance allows every class inherits from more than one superclass. In OOD, designing a suitable class hierarchy involving multiple inheritance is a difficult task. Two problems present themselves when we have multiple inheritance: how to deal with name conflictions from different superclasses, and how do we handle redundant inheritance. Name conflictions are possible when two or more different superclasses use the same name for some element of their interfaces, such as instance variables and methods (Carre &Geib 1990). In part (a) of Fig.1, class *US-Driver* and class *China-Driver* both have a method named *traffic-violation*, representing the number of traffic violations in US or China respectively. If someone has driver licenses both in US and China, then a class US-China-Drive is declared to be inherited from both of these classes, what does it mean to inherit two *traffic-violation* with the same name? There are basically three approaches to resolving this clash. First, the language semantics might regard a name confliction as illegal, and reject the compilation of the class. This is the approach taken by Smalltalk and Eiffel (Borning & Ingalls `1982). In Eiffel, however, it is possible to rename items so that there is no ambiguity. Second, the language semantics might regard the same name introduced by different classes as referring to the same traffic-violation, which is the approach taken by CLOS. Third, the language semantics might permit the confliction, but require that all references to the name fully qualify the source of its declaration. This is the approach taken by C++.

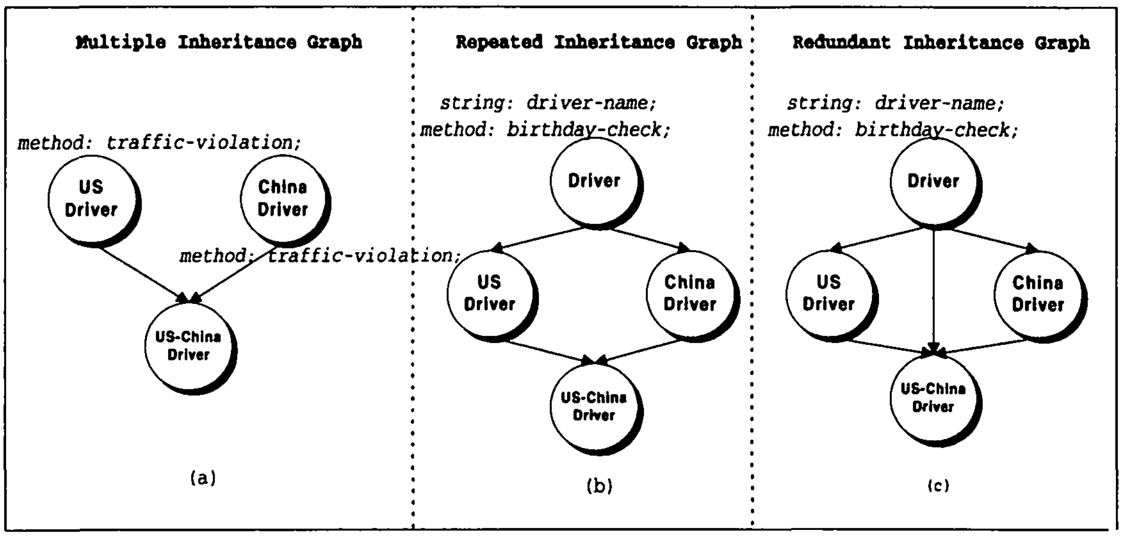

Figure 1: Mutiple, Repeated and Redundant Inheritance

The second problem is redundant inheritance which is raised by the presence of repeated inheritance when a class is an ancestor of another in more than one way. Consider the inheritance graph in Fig.1-b. A base class Driver is declared to be the parent class of both class US-Driver and class China-Driver. We find that class US-China-Driver inherits twice from class Driver. This situation is called a repeated inheritance. Furthermore, the class Driver, suppose, is declared to be the parent class of class US-China, then this inheritance is called a *redundant inheritance* because class US-China-Driver could inherits all the value and behaviors from class Driver through class US-Driver or class China-Driver (see Fig.1-c). From the programming viewpoints, the redundant inheritance is not only unnecessary but also error-prone. In this redundant inheritance graph, class US-China-Driver inherits twice (or more) from class Driver in the redundant inheritance. Redundant inheritance is often confused with repeated inheritance. To formally verify the difference, repeated and redundant inheritance are defined individually as follows:

Definition 1: Given two classes denoted as α_i and α_j in some class hierarchy Ω, if there exists at least two (or more) inheritance paths between α_i and α_j, all the classes on these inheritance paths form a repeated inheritance.

Definition 2: Given a class hierarchy Ω with *n* inheritances, for an inheritance relation $\alpha_i \rightarrow \alpha_j \in \Omega$, if there has been existed other *k* inheritances for $1 < k < n$, such that $\alpha_i \rightarrow \alpha_{i+1} \rightarrow \alpha_{i+2} \ldots \rightarrow \alpha_{i+k} \rightarrow \alpha_j$, then $\alpha_i \rightarrow \alpha_j$ is called a redundant inheritance.

Basically, redundant inheritance is generated on the extension of repeated inheritance. In this paper, we call a repeated inheritance graph which contains a redundant inheritance as a *redundant inheritance graph*. Consequently, a redundant inheritance graph must also be (contain) a repeated inheritance graph. Apparently, Fig.1-c is just a redundant inheritance graph. After the above analysis, we can derive a mathematical expression to verify the relationships among multiple inheritance graph, repeated inheritance graph and redundant inheritance graph. This verification is helpful to reduce the occurrence of controversial and error-prone inheritances.

multiple inheritance graph $\subseteq$ *repeated inheritance graph* $\subseteq$ *redundant inheritance graph* (1)
For a large OOD system, the detection of redundant inheritance is a time-consuming task. In next section, *inference rule* is used to address the redundant inheritance. This inference rule can be easily programmed and maintained in the OO software development.

INHERITANCE CONSTRAINTS AND INFERENCE RULES

One of the key issues in object-oriented design is certainly the specification of *inheritance constraints*. In the object-oriented design (OOD) phase, system designer declares inheritance constraints initially and modifies them later. Since inheritance constraints must not be ambiguous and redundant, some external checking mechanisms must be provided. The constraints detected in the OO design should be embedded in some way so that unambiguity and nonredundancy are preserved. The classification of inheritance constraints are shown as follows:

1. Single Inheritance Constraints (SIC): Given two classes, α, and β, $\alpha \rightarrow \beta$ means that β is a subclass of α, and that every instance in β is also an instance in α.

2. Multiple Inheritance Constraints (MIC): Given three classes, α, β, and γ, $\alpha\beta \rightarrow \gamma$ means that γ is a subclass of α and also a subclass of β. $\alpha\beta \rightarrow \gamma$ implies that $\alpha \rightarrow \gamma \wedge \beta \rightarrow \gamma$.

We can characterize the constraint membership problem in an inference rule system. In what follows, we assume that we are given a set of classes denoted by Ψ and a class hierarchy denoted by Ω, involving only classes in Ψ. The inference rules are:

Axiom 1: The inference rules for single inheritance constraints are

1. Transitivity: given three classes, α, β and $\gamma \in \Psi$, if $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$ hold, then $\alpha \rightarrow \gamma$.
2. Union: if $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$.hold, then $\alpha \rightarrow \beta\gamma$.
3. Decomposition: if $\alpha \rightarrow \beta\gamma$ hold, then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$.

There are several other inference rules that follows from axiom 1. We state two of them in the next theorem.

Theorem 1: The inference rules for multiple inheritance constraints are
1. The composition rule: given classes $\beta, \alpha_1, \alpha_2, ..., \alpha_n \in \Psi$, if $\alpha_i \rightarrow \beta$ for every $i= 1, ..., n$, then $\alpha_1 \alpha_2 ... \alpha_n \rightarrow \beta$.
2. The pseudotransitivity rule: given classes α, β, γ and ϕ, if $\alpha \rightarrow \beta$ and $\phi\beta \rightarrow \gamma$ hold, then $\alpha\phi \rightarrow \gamma$ holds.

Proof:
1): We are given $\alpha_1 \rightarrow \beta$, $\alpha_2 \rightarrow \beta$. By composition rule, $\alpha_1 \rightarrow \beta$, $\alpha_2 \rightarrow \beta$ deduce $\alpha_1 \alpha_2 \rightarrow \beta$. By induction, $\alpha_1 \alpha_2 ... \alpha_n \rightarrow \beta$ holds.

2): By *decomposition rule*, $\phi\beta \rightarrow \gamma$ tells us $\phi \rightarrow \gamma$, $\beta \rightarrow \gamma$ holds. By union rule, $\alpha \rightarrow \gamma$, $\phi \rightarrow \gamma$ implies $\alpha\phi \rightarrow \gamma$.

Before tackling the main issues in this paper, it is important to introduce *transitivity rule* and the *closure* computation with respect to a given class hierarchy.

A. Transitivity Rule for Inheritance Hierarchy

Given a class hierarchy consists of $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$. Then, we could claim that $\alpha \rightarrow \gamma$ must also hold in Ω. This proof is easy by the property of transitivity. In general, let Ω be a set of inheritance relations. We say $\xi \rightarrow \eta$ is indirectly inherited from Ω, written $\Omega \Rightarrow \xi \rightarrow \eta$. We saw above that if Ω contains $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \bullet \gamma$ is indirectly inherited from Ω. That is, $\{ \alpha \rightarrow \beta , \beta \rightarrow \gamma\} \Rightarrow \alpha \bullet \gamma$. Let Ω^+, the closure of Ω, be the set of inheritance relations that are indirectly inherited from Ω, i.e., $\Omega^+ = \{ \xi \rightarrow \eta \mid \Omega \Rightarrow^+ \xi \rightarrow \eta\}$.

For example, Let $\Phi=\{\alpha, \beta, \gamma\}$, and $\Omega=\{ \alpha \rightarrow \beta , \beta \rightarrow \gamma\}$. Then Ω^+ consists of all those inheritances $\xi \rightarrow \eta$ such that either

1. ξ contains α, e.g., $\alpha\beta \rightarrow \gamma$, $\alpha \rightarrow \beta$, or $\beta \rightarrow \gamma$,
2. ξ contains β but not α, and η does not contain α, e.g., $\beta \rightarrow \gamma$, or $\beta \rightarrow \phi$, and
3. $\xi \rightarrow \eta$ is $\gamma \rightarrow \phi$.

It turns out that computing the closure set for a class hierarchy Ω is a time-consuming task in general, simply because the set of relationships in Ω^+ can be large even though Ω itself is small. Consider the set $\Omega = \{\alpha_1 \rightarrow \alpha_2, \ldots, \alpha_{n-1} \rightarrow \alpha_n\}$. Then Ω^+ includes all the transitive inheritances $\alpha_i \rightarrow \Psi$, where Ψ is a subset of $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$. Obviously, the number of all derived inheritances on the Ψ by inference rules could be C(n,1) + C(n,2) + ... + C(n, n-1) + C(n, n). The number is equal to 2^n. As there are 2^n such sets of Ω, we can not expect to list Ω^+ conveniently, even for a reasonably sized *n*. By contrast with Ω^+, computing ξ^+, for a set of classes ξ, is not hard; it takes time proportional to the length of all inheritances in Ω, written out. Instead of computing the tedious Ω^+, ξ^+ is informative enough to tackle the cyclic and redundant inheritance issues. In the next section, an algorithm to compute ξ^+ will be presented.

B. Equivalence of Two Class Hierarchies

Let Ω and Θ be two class hierarchies; Ω and Θ are said to be *equivalent* if and only if $\Omega^+ = \Theta^+$. To test whether Ω and Θ are equivalent, we must verify whether both $\Omega^+ \subseteq \Theta^+$ and $\Theta^+ \subseteq \Omega^+$ hold. The verification of the equivalence of two class hierarchies by inference rules is relatively time-consuming. An illustration is shown as follows:

<u>*Example A:*</u>
Given a class hierarchy Ω consists of two multiple inheritance constraints (MIC) and two single inheritance constraints (SIC)as follows:

$$\Omega = \begin{cases} SIC : \alpha_1 \rightarrow \alpha_2\alpha_3\alpha_4, \alpha_6 \rightarrow \alpha_5 \\ MIC : \alpha_1\alpha_2\alpha_3 \rightarrow \alpha_5, \alpha_3\alpha_4 \rightarrow \alpha_6 \end{cases}$$

We can find other *equivalent* class hierarchies whose closure is equal to Ω. Suppose Θ is an equivalent one which contains four single inheritance constraints and one multiple inheritance constraint as follows:

$$\Theta = \begin{cases} SIC : \alpha_1 \rightarrow \alpha_2\alpha_3\alpha_4\alpha_5, \alpha_6 \rightarrow \alpha_5, \alpha_3 \rightarrow \alpha_6, \alpha_4 \rightarrow \alpha_6 \\ MIC : \alpha_2\alpha_3 \rightarrow \alpha_5 \end{cases}$$

By use of Axiom 1 and Theorem 1, we can verify that Ω and Θ are equivalent by checking whether Ω^+ is equal to Θ^+. However, we cannot expect to list Ω^+ and Θ^+ conveniently. A useful alternative equivalence is addressed in the next theorem.

Theorem 2: For each class hierarchy Ω, there is an equivalent class hierarchy Θ in which no both sides of an inheritance have more than one class.
Proof:
(1) Let Θ be the set of inheritance relations $\alpha \rightarrow \phi_i$ such that for some $\alpha \rightarrow \psi$ in Ω, $\phi_i \in \psi$ where ϕ_i is a single class. The $\alpha \rightarrow \phi_i$ follows from ψ by the decomposition rule. Thus, $\Theta \subseteq \Omega^+$ holds. On the other hand, $\Omega \subseteq \Theta^+$ also hold, since if $\psi = \phi_1 \ldots \phi_n$, then $\alpha \bullet \psi$ follows from $\alpha \rightarrow \phi_1, \ldots, \alpha \rightarrow \phi_n$ using the union rule.

(2) Let Θ be the set of inheritance relations $\phi_i \rightarrow \beta$ such that for some $\psi \rightarrow \beta$ in Ω, $\phi_i \in \psi$. The $\phi_i \rightarrow \beta$ follows from ψ by the decomposition rule. Thus, $\Theta \subseteq \Omega^+$ holds. On the other hand, $\Omega \subseteq \Theta^+$ also holds, since if $\psi = \phi_1 \ldots \phi_n$, then $\psi \rightarrow \beta$ follows from $\phi_1 \rightarrow \beta, \phi_2 \rightarrow \beta, \ldots, \phi_n \rightarrow \beta$ using the union rule.

<u>Example B</u>
Consider class hierarchy Ω in *Example A* again, By Theorem 2, we can find an equivalent class hierarchy denoted Θ whose both sides has only one class and is shown as follows:

$$\Omega = \begin{cases} SIC : \alpha_1 \rightarrow \alpha_2\alpha_3\alpha_4, \alpha_6 \rightarrow \alpha_5 \\ MIC : \alpha_1\alpha_2\alpha_3 \rightarrow \alpha_5, \alpha_3\alpha_4 \rightarrow \alpha_6 \end{cases} \xrightarrow{\text{equivalent}} \Theta = \begin{cases} \alpha_1 \rightarrow \alpha_2, \alpha_1 \rightarrow \alpha_3, \alpha_1 \rightarrow \alpha_4 \\ \alpha_1 \rightarrow \alpha_5, \alpha_6 \rightarrow \alpha_5, \alpha_2 \rightarrow \alpha_5 \\ \alpha_3 \rightarrow \alpha_5, \alpha_3 \rightarrow \alpha_6, \alpha_4 \rightarrow \alpha_6 \end{cases}$$

It turns out to be useful, when we develop a class hierarchy design, to consider a stronger restriction on equivalence than that both sides have but one class.

MINIMAL CLASS HIERARCHY

In this section, we propose a useful concept called *minimal class hierarchy* to address redundant inheritance. This minimal class hierarchy plays a vital role in our checking mechanism for redundant inheritance.

Definition 3: A class hierarchy is said to be *minimal* if:
1. Every both sides of an inheritance relation in Ω is a single class.

2. No $\xi \rightarrow \eta \in \Omega$ such that the set $(\Omega - \{\xi \rightarrow \eta\})$ is equivalent to Ω.

Theorem 3: If a class hierarchy Ω contains redundant inheritance relationships, then there is a *minimal* class hierarchy Θ such that $\Theta \subset \Omega$ and $\Omega^+ = \Theta^+$.
Proof: We have to show that Θ satisfy the two conditions mentioned above.
 1. For each redundant inheritances $\xi \rightarrow \eta \in \Omega$, $(\Omega - \{\xi \rightarrow \eta\})^+ = \Omega^+$ must hold, and we can remove $\xi \rightarrow \eta$ from Ω. Let $\Theta = \Omega - (\xi \rightarrow \eta)$. Because Ω contain redundant inheritance relationships, there exists at least one redundant relationships $\xi \rightarrow \eta$ such that $\Omega - \{\xi \rightarrow \eta\}$ is equivalent to Ω.

 2. By Theorem 3, all the inheritance relationships in Ω can be decomposed into a class hierarchy Θ, in which no both sides have more than one class. Therefore, Θ can be done in the same way.

Example C: Consider the class hierarchy Θ in *Example B* again. We apply Theorem 3 to Θ, then get a minimal class hierarchy denoted Δ equivalent to Θ shown as follows.

$$\Theta = \left\{\begin{array}{lll} \alpha_1 \rightarrow \alpha_2 & \alpha_1 \rightarrow \alpha_3 & \alpha_1 \rightarrow \alpha_4 \\ \alpha_1 \rightarrow \alpha_5 & \alpha_6 \rightarrow \alpha_5 & \alpha_2 \rightarrow \alpha_5 \\ \alpha_3 \rightarrow \alpha_5 & \alpha_3 \rightarrow \alpha_6 & \alpha_4 \rightarrow \alpha_6 \end{array}\right. \xrightarrow{\text{minimal}} \begin{array}{lll} \alpha_1 \rightarrow \alpha_2 & \alpha_1 \rightarrow \alpha_3 & \alpha_1 \rightarrow \alpha_4 \\ & \alpha_6 \rightarrow \alpha_5 & \alpha_2 \rightarrow \alpha_5 \\ & \alpha_3 \rightarrow \alpha_6 & \alpha_4 \rightarrow \alpha_6 \end{array}$$

By the comparison of *Example A* and *Example C*, we can conclude that Ω, Θ, and Δ are equivalent mutually.(i.e., $\Omega^+ = \Theta^+ = \Delta^+$.) Theorem 3 derives a useful property that a minimal class hierarchy itself is a non-redundant class hierarchy. This non-redundancy is just what we are seeking for. We can derive a redundant inheritance detection algorithm straightforward from the Theorem 3. The idea of this algorithm is that for each inheritance ξ • $\eta \in \Omega$, if Ω^+ is equal to $(\Omega - \{\xi \rightarrow \eta\})^+$, then $\xi \rightarrow \eta$ is a redundant inheritance.

Algorithm 1: Find out Redundant Inheritances with respect to a class hierarchy
Input: A class hierarchy $\Omega=\{e_1, e_2, \ldots, e_n\}$ with *n* inheritances.
Output: A redundant inheritance set Γ

```
1. $\Gamma = \emptyset$; initialize $\Gamma$ with empty
2. for each $e_i \in \Omega$, $i = 1,2,\ldots,n$
       if $\Omega^+ = \{\Omega - e_i\}^+$ then add $e_i$ to $\Gamma$
```

However, this algorithm is not efficient because the computing of Ω^+ is an exponential time. Fortunately, there is an alternative approach to replace Ω^+. At the other extreme, computing ξ^+, for a set of classes ξ, is not hard; it takes time proportional to the length of all inheritances in Ω, written out. A more efficient algorithm based on ξ^+ will be proposed in the next section. Now we define ξ^+ formally as follows:

Definition 4: Let Ω be a class hierarchy on a set of classes denoted by Ψ, and let ξ be a subset of Ψ. Then the closure of ξ with respect to Ω, denoted by ξ^+, is defined as the set of classes η such that $\xi \rightarrow \eta$ can be deduced from Ω by inference rule.

The redundant inheritance detection could be achieved by checking whether ξ^+ still contains η after removing $\xi \rightarrow \eta$. If it does, then $\xi \rightarrow \eta$ is a redundant inheritance relationship. A simple algorithm to compute ξ^+ is shown in the following.

Algorithm 2: Computation of the Closure of a set of classes with respect to a class hierarchy
Input: A finite set of classes Ψ, a class hierarchy Ω on Ψ, and a set $\xi \subseteq \Psi$.
Output: ξ^+, the closure of ξ with respect to Ω
Method: We compute a sequence of sets of classes $\xi^{(0)}, \xi^{(1)}, \ldots,$ by the rules:
1. $\xi^{(0)} \leftarrow \xi$
2. $\xi^{(i+1)}$ is $\xi^{(i)}$ plus the set of classes η such that there is some inheritance $Y \to Z$ in Ω, η is in Z, and $Y \subseteq \xi(i)$.
Since $\xi = \xi^{(0)} \subseteq \ldots \subseteq \xi^{(i)} \subseteq \ldots \subseteq \Psi$, and Ψ is finite, we must eventually reach *i* such that $\xi^{(i)} = \xi^{(i+1)}$. It then follows that $\xi^{(i)} = \xi^{(i+1)} = \xi^{(i+2)} = \ldots$. There is no need to compute beyond $\xi^{(i)}$ once we discover $\xi^{(i)} = \xi^{(i+1)}$. We can (and shall) prove that ξ^+ is $\xi^{(i)}$ for this value of *i*.

In the following, we use an example to illustrate the algorithm. To illustrate the algorithm clearly, a complex class hierarchy is chosen and executed instead of a trivial one. Although the class hierarchy may be very complex and contains cyclic inheritance which is controversial in semantics, the major goal is to demonstrate the process of algorithm execution. Therefore, we do not request its programming feasibility in practice.

Example: Let Ω consists of the following eight inheritances:

$$\alpha_1\alpha_2\alpha_7 \to \alpha 3 \quad \alpha_3 \to \alpha_1 \quad \alpha_2\alpha_3 \to \alpha_4$$
$$\alpha_1\alpha_3\alpha_7 \to \alpha_2 \quad \alpha_4 \to \alpha_5\alpha_6 \quad \alpha_2\alpha_5 \to \alpha_3$$
$$\alpha_3\alpha_6 \to \alpha_2\alpha_4 \quad \alpha_3\alpha_5 \to \alpha_1\alpha_6$$

and let $\xi = \alpha_2\alpha_4$. To apply algorithm 2, we let $\xi^{(0)} = \alpha_2\alpha_4$. To compute $\xi^{(1)}$ we look for Ω that have a left side α_2, α_4, or $\alpha_2\alpha_4$. There is only one, $\alpha_4 \to \alpha_5\alpha_6$, so we adjoin α_5 and α_6 to $\xi^{(0)}$ and make $\xi^{(1)} = \alpha_2\alpha_4\alpha_5\alpha_6$. For $\xi^{(2)}$, we look for left sides contained in $\xi^{(1)}$ and find $\alpha_4 \to \alpha_5\alpha_6$ and $\alpha_2\alpha_5 \to \alpha_3$. Thus $\xi^{(2)} = \alpha_2\alpha_3\alpha_4\alpha_5\alpha_6$. Then, for $\xi^{(3)}$ we look for left sides contained in $\alpha_2\alpha_3\ \alpha_4\alpha_5\alpha_6$ and find, in addition to the two previously found, $\alpha_3 \to \alpha_1$, $\alpha_2\alpha_3 \to \alpha_4$, $\alpha_3\alpha_6 \to \alpha_2\ \alpha_4$, and $\alpha_3\alpha_5 \to \alpha_1\alpha_6$. Thus $\xi^{(3)} = \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5\alpha_6$. It therefore comes as no surprise that $\xi^{(3)} = \xi^{(4)} = \ldots$. Thus $(\alpha_2\alpha_4)^+ = \alpha_1\ \alpha_2\ \alpha_3\ \alpha_4\ \alpha_5\ \alpha_6$.

Time Complexity and Data Structure Analysis for Algorithm 2
Algorithm 2 can be implemented to run in proportional to the sum of the lengths of the inheritances in Ω if we keep, for each inheritance $Y \to Z$, a count of the number of classes in Y that are not yet in $\xi^{(i)}$. We must also created a list, for each class η, of the inheritances on whose left side η appears. When η is adjoined to some $\xi^{(i)}$, we decrement by one that count for each inheritance on η's list. When the count for $Y \to Z$ becomes 0, we know $Y \subseteq \xi^{(i)}$. Lastly, we must maintain $\xi^{(i)}$ as a Boolean array, indexed by class numbers, so when we discover $Y \subseteq \xi^{(i)}$, where $Y \to Z$ is an inheritance in Ω, we can tell in time proportional to the size of Z those classes in Z that need to be adjoined to $\xi^{(i)}$. When computing $\xi^{(i+1)}$ from $\xi^{(i)}$ we have only to set to true the entries of the array corresponding to classes added to $\xi^{(i)}$; there is no need to copy $\xi^{(i)}$.
Now the problem of proving that algorithm 2 is correct must be addressed. It is easy to prove that every class placed in some $\xi^{(k)}$ belongs in ξ^+, but harder to show that every class in ξ^+ is placed in some $\xi^{(k)}$

Theorem 4: Algorithm 2 correctly computes ξ^+.
Proof: First we show by induction on *k* that if η is placed in $\xi^{(k)}$, then η is in ξ^+.
Basis: $k=0$. Then η is in ξ, so by reflexivity, $\xi \to \eta$.
Induction: Let $k>0$ and assume that $\xi^{(k-1)}$ consists only of class in ξ^+. Suppose η is placed in $\xi^{(k)}$ because η is in Z, $Y \to Z$ is in Ω, and $Y \subseteq \xi^{(k-1)}$). Since $Y \subseteq \xi^{(k-1)}$, we know $Y \subseteq \xi^+$ by the inductive hypothesis. Thus $\xi \to Y$ by Lemma 1. By transitivity, $\xi \to Y$ and $Y \to Z$ imply $\xi \to Z$. By reflexivity, $Z \to \xi$, so $\xi \to \eta$ by transitivity. Thus η is in ξ^+.

REDUNDANT INHERITANCE DETECTION AND RESOLUTION

In this section, a detection algorithm for redundant inheritance will be presented. Also, three approaches to resolving redundant inheritances are proposed. In Section 3, we have shown that the computation of a closure set of a class hierarchy, for example Ω, is very time consuming. In contrast, computing ξ^+, for a set of classes ξ, is not hard. Previously, ξ^+ is defined as the closure of a set of classes ξ on some class hierarchy. Because there may exist various different class hierarchies in an OO design, the definition of the closure ξ^+ is not precise enough to specify to which class hierarchy it belongs. For the sake of precision, the term ξ^+_Ω is used to strengthen the previous definition. Simply stated, the closure of ξ on Ω is denoted as ξ^+_Ω. The idea of the algorithm is that for each $\xi \to \eta \in \Omega$, if $\eta \in \xi^+_{\Omega-\{\xi \bullet \eta\}}$ then $\xi \to \eta$ is a redundant inheritance.

Algorithm 3: Find out Redundant Inheritances

```
Input: A class hierarchy $\Omega$
Output: A set of redundant inheritance $\Gamma$
1. Apply Theorem 2 to $\Omega$, then get an equivalent class hierarchy $\Theta$
2. For each $\xi \to \eta \in \Theta$ do
   (1) Apply Algorithm 2 to compute $\xi^+_{\Theta-\{\xi \bullet \eta\}}$
   (2) if $\eta \in \xi^+_{\Theta-\{\xi \bullet \eta\}}$ then add $\xi \to \eta$ to $\Gamma$
```

Example D: Given a class hierarchy Ω, find out its redundant inheritances. Let Ω consists of the following four inheritances:

$$\Omega = \begin{cases} \alpha_1 \to \alpha_3\alpha_5 & \alpha_2 \to \alpha_3 \\ \alpha_3\alpha_4 \to \alpha_2\alpha_3 & \alpha_3 \to \alpha_4 \end{cases}$$

Step 1. To apply Theorem 2 to Ω, we then get a class hierarchy Θ as follows:

$$\Omega = \begin{cases} \alpha_1 \to \alpha_3 & \alpha_1 \to \alpha_5 & \alpha_2 \to \alpha_3 \\ \alpha_3 \to \alpha_4 & \alpha_3 \to \alpha_5 & \alpha_4 \to \alpha_5 \end{cases}$$

Step 2. for each inheritance $\xi \bullet \eta$ in Θ we do:

1. $\alpha_1 \to \alpha_3$ is chosen: we apply algorithm 2 to compute the closure of $\alpha_1{}^+$ on $(\Theta\text{-}\{\alpha_1 \to \alpha_3\})$, and we get $\alpha_1{}^+=\{\alpha_5\}$. We find that $\alpha_3 \in \alpha_1{}^+$ does not hold, then discard $\alpha_1 \to \alpha_3$.
2. $\alpha_1 \to \alpha_5$ is chosen: we apply algorithm 2 to compute the closure of $\alpha_1{}^+$ on $(\Theta\text{-}\{\alpha_1 \to \alpha_5\})$, and we get $\alpha_1{}^+=\{\alpha_3, \alpha_4, \alpha_5\}$. We find that $\alpha_5 \in \alpha_1{}^+$ holds, then we adjoin $\alpha_1 \to \alpha_5$ to Γ.
3. $\alpha_2 \to \alpha_3$ is chosen: we apply algorithm 2 to compute the closure of $\alpha_2{}^+$ on $(\Theta\text{-}\{\alpha_2 \to \alpha_3\})$, and we get $\alpha_2{}^+=\{\varnothing\}$ (denoting empty set). We find that $\alpha_3 \in \alpha_2{}^+$ does not hold, then discard $\alpha_2 \to \alpha_3$.
4. $\alpha_3 \to \alpha_4$ is chosen: we apply algorithm 2 to compute the closure of $\alpha_3{}^+$ on $(\Theta\text{-}\{\alpha_3 \to \alpha_4\})$, and we get $\alpha_3{}^+=\{\alpha_5\}$. We find that $\alpha_4 \in \alpha_1{}^+$ does not hold, then discard $\alpha_3 \to \alpha_4$.
5. $\alpha_3 \to \alpha_5$ is chosen: we apply algorithm 2 to compute the closure of $\alpha_3{}^+$ on $(\Theta\text{-}\{\alpha_3 \to \alpha_5\})$, and we get $\alpha_3{}^+=\{\alpha_4, \alpha_5\}$. We find that $\alpha_5 \in \alpha_3{}^+$ holds, then we adjoin $\alpha_3 \to \alpha_5$ to Γ.
6. $\alpha_4 \to \alpha_5$ is chosen: we apply algorithm 2 to compute the closure of $\alpha_4{}^+$ on $(\Theta\text{-}\{\alpha_4 \to \alpha_5\})$, and we get $\alpha_4{}^+=\{\varnothing\}$. We find that $\alpha_5 \in \alpha_4{}^+$ does not hold, then discard $\alpha_4 \to \alpha_5$.

After the execution of algorithm 3, Γ will contain a set of redundant inheritances $\{\alpha_1 \to \alpha_5, \alpha_3 \to \alpha_5\}$. For this redundant inheritance $\alpha_1 \to \alpha_5$ in Γ, it means that class α_5 could inherit directly or indirectly from α_1. Equation (1), shown in Section 2, reveals that α_5 would inherit twice (or more) from α_1. To reduce the occurrence of implicit errors caused by this redundant inheritance, we had better refine all the ancestor classes of class α_5. The depth-first or breadth-first traversal algorithm can be used to determine these ancestor classes. The refinement depends on the actual inheritance relations. We can combine related classes to remove these unnecessary duplicates. This behavior is also called *class aggregation* (Hendler 1986). The redundant inheritance $\alpha_3 \to \alpha_5$ can be dealt by the same way.

Analysis for Algorithm 3

Algorithm 3 basically consists of two parts. First part is to apply Theorem 3 to a given class hierarchy. As the proof of Theorem 3 mentioned above, we use *decomposition rule* and *union* rule to get a class hierarchy whose both sides contain only one class. For an inheritance $\xi \to \eta$, the decomposition rule can be implemented to run in time proportional to the length of η, and union rule is in proportional to the length of ξ. Second part is to apply algorithm 2 to compute the closure set of ξ on Θ for each inheritance $\xi \to \eta \in \Theta$. The analysis of algorithm 2 has been shown above. Integration of the analysis of the two parts, the algorithm 3 runs in proportional to the sum of the lengths of inheritances in a given class hierarchy.

Resolutions for Redundant Inheritance

After the redundant inheritance have been identified, it is important to resolve them before coding. Basically, there are three approaches to dealing with the problem of redundant inheritance. First, we can treat occurrences of redundant inheritance as illegal. This is the approach taken by Java, Smalltalk and Eiffel (with Eiffel permitting renaming to disambiguate the duplicate references). Second, we can permit duplication of superclasses, but require the use of fully qualified names to refer to members of a specific copy. This is one of the approaches taken by C++. Third, we can treat multiple reference to the same class as denoting the same class. This is the approach taken by C++ when the repeated superclass is introduced as a virtual base class. A virtual base class exists when a subclass names another class as its superclass and marks that superclass as virtual, to indicate that it is a shared class. The redundant inheritance in Fig.1-c can be resolved by declaring superclass *Driver* as a *virtual base class*:

```
class Driver {....};

class US-Driver: virtual public Driver { ... };

class China-Driver: virtual Public Driver { ... };

class US-China-Driver : public  US-Driver,  public China-Driver { ... };
```

Similarly, in CLOS repeated classes are shared, using a mechanism called the *class precedence list*. This list, calculated whenever a new class is introduced, includes the class itself and all of its superclasses, without duplication, and is based upon the following rules:

1. A class always has precedence over its superclass
2. Each class sets the precedence order of its direct superclasses

In this approach, the inheritance graph is flattened, duplicates are removed, and the resulting hierarchy is resolved using single inheritance (Dussud 1993). This is akin to the computation of a topological sort of classes. If a total ordering of classes can be calculated, then the class that introduce the redundant inheritance is accepted. Note that this total ordering may be unique, or there may be several possible orderings. If no ordering can be found (for example, when there exist cyclic inheritance), the class is rejected. In Fig.1-c, the class *US-China-Driver* would be accepted, because there is a unique ordering of superclasses; the superclass hierarchy includes exactly one (shared) appearance of the class *Driver*.

Invoking a Method in Redundant Inheirtance

In traditional programming languages, invoking a subprogram is a completely static activity. In Pascal for example, for a statement that calls the subprogram P, the compiler can generate code that creates a new stack frame, places the proper arguments on the stack, and then changes the flow of control to being executing the code associated with P. However, in OO languages that support *polymorphism*, invoking a method is dynamic because the class of the object being operated upon may not be known until runtime. Matters are even more complicated when we add redundant inheritance to this situation. Redundant inheritance with polymorphism requires a much more sophisticated technique. Consider the class hierarchy shown in Fig.1-c, which shows the base class Driver along with two subclasses named *US-Driver*, and *China-Driver*. Both of them has a subclass, named *US-China-Driver*. In class Driver, the method *birthday-check* is common to all subclasses, and therefore need not be redefined. However, the method *traffic-violation* must be redefined by each of the subclasses, since either in China or US, the number of violations must be mutually independent. Thus, since the class *Driver* is an abstract class[1], *traffic violation* has an empty implementation (it is a pure virtual function, in C++ terminology).

In C++, the developer can decide if a particular method is to be bound late by declaring it to be *virtual*; all other methods are considered to be bound early, and thus the compiler can statically resolve the method call to a simple subprogram call. In this redundant inheritance, we might have declared *traffic-violation* as a virtual

[1] A class with no instances is called *abstract data class*.

member function and the method *birthday-check* as nonvirtual because the birthday of anyone is unique and unexchangeable. Therefore, birthday-check need not be redefined.

CONCLUSION

In this paper, we propose a checking mechanism to determine redundant inheritance in a given class hierarchy. Inference rule system is used to specify the inheritance constraints. A minimal class hierarchy concept is presented to address the redundant inheritance occurrence. An algorithm based on the minimal class hierarchy is derived to detect redundant inheritances. This work will contribute to object-oriented software testing and maintenance. It is note worthy that this checking mechanism can be easily incorporated with OO CASE tool to facilitate OO software development and quality. The occurrence of cyclic inheritance which causes *self-inheritance* and endless type-checking is another problem associated with multiple inheritance in the class hierarchy design. For most theoretical papers, it is strictly prohibited and is assumed to never happen. However, the occurrence of cyclic inheritance due to careless design or specific purpose is inevitable and inherent for software designers. For the sake of space limitation, we leave it to future discussion and research.

REFERENCES

Borning, A.H. and Ingalls, D.H.(1982), ``Multiple Inheritance inSmalltalk-80," in **Proc. of the AAAI-82,** Pittsburgh, 1982, pp. 234-237.

Cardelli, Luca (1984), `` A semantics of Multiple Inheritance, " in **Semantics of Data Types**, Lecture Notes in Computer Science 173, Springer-Verlag New York, 1984, pp. 51-67.

Carre, B. and Comyn, G. (1988), `` On Multiple Classification, Points of View and Object Evolution," in Demongeot, Herve, T., Rialle, V., and Roche, C. (eds.), **Artificial Intelligence and Cognitive Science**, Manchester University Press, 1988.

Carre, B. and Geib, J.M.(1990), ``The Point of View Notion forMultiple Inheritance, " **OOPSLA**, 1990, pp. 312-321.

Chung, C.M. and Lee, M.C.(1997), `` Integration Object-Oriented Software Testing and Metrics," **International Journal of Software Engineering and Knowledge Engineering**, vol. 7, no. 1, 1997, pp. 125-144.

Dussud, P. (1993)`` TICLOS: An Implementation of CLOS for the Explorer Family." **SIGPLAN Notices** vol. 24(10), 1993.

Freeman, Lee, ed.(1991), **Computer Aided Software Engineering** (CASE*)*, James Martin Insight Inc, Naperville, Ill., The Martin Reprt, Vol. VI, updated quarterly, 1991.

Grady Booch (1991), Object-Oriented Design with Application, pp. 105-113, 1991, The Benjamin.

Hendler, J. (1986)``Enhancement for Multiple Inheritance," **SIGPLAN Notices**, vol. 21 (10), Oct. 1986, pp. 100.

Kim, Won **Modern Database Systems (1995), The Object Model, nteroperability, and Beyond**, ACM Press. 1995. pp.211-222.

Lee, M.C. and Chiang, D.A.(1992), ``Cyclic Inheritance Detection for Object-oriented Database," **IEEE Region 10 Conference, TENCON'92** Australia, November, 1992, pp. 633-637.

Meyer, Bertrand, **Object-oriented Software Construction**, New York, Prentice-Hall 1988, pp. 274.

Moises Lejter, Scott Meyers, and Steven P. Reiss, (1992)``Support for Maintaining Object-Oriented Programs," **IEEE Trans. on Software Engineering**, vol. 18, no. 12, Dec. 1992, pp. 1045-1052.

Ullman, J.D., (1988) *Principles of Database and Knowledge-Base Systems,* Vol. 1, Computer Science Press, Maryland 1988, pp. 391-392.