An inheritance flow model for class hierarchy analysis

Jiun-Liang Chen 1, Feng-Jian Wang *

Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, 30050, Taiwan

Received 14 August 1997; received in revised form 19 March 1998

Communicated by D. Gries

Abstract

This paper presents an inheritance flow model, which represents the inheritance relationships among classes as a flow graph. A flow operation is associated with each attribute and method in a class to denote the defined (redefined) or inherited member. An inherited member can be deemed as being handled by a sequence of flow operations along a path in the flow graph. This model provides several analyses in a class hierarchy, such as implicit inherited member and polymorphic method invocation. These analyses may be applied in various fields of software engineering, such as static analysis, maintenance, and complexity measurement. © 1998 Elsevier Science B.V. All rights reserved.

Keywords: Programming language; Software engineering; Object-oriented; Class hierarchy

1. Introduction

Due to inheritance, classes in the Object-Oriented (OO) paradigm often form a hierarchical structure, called class hierarchy. In a class hierarchy, a class can inherit the members from its superclass without declaring them. The inherited members for a class are implicit, since they can be used but not defined in the class. The implicit inherited members often obstruct the investigation of changing superclasses, and joining/splitting classes during software design and maintenance [10], besides program understanding [6,8].

Most programming environments (e.g., Microsoft™ Visual C++ and Borland™ C++ Builder) employ a compiler to build a virtual table [9] to record the members of each class. However, the virtual table is a fixed structure; it is difficult to provide class investigations as mentioned above. At present, there is no formal model to describe the behavior evolution in a class hierarchy. This might complicate the understanding of a class hierarchy, since one has to navigate through all the classes.

In this paper, we present an inheritance flow model that represents the inheritance relationships among classes as a flow graph. In this model, a flow operation, which is either to define or to use is associated with each member (i.e., an attribute or method) of a class. The define operation means that the member is defined or redefined, while the use operation means that the member might be referenced. Thus, the member of a class inherited by subclasses can be deemed as being handled by a sequence of flow operations along an inheritance path in the flow graph. The sequence can be used to indicate the behavior evolution in a class hierarchy, and it may form a define-use pair, like the define-use relation in data-flow analysis [1]. Therefore, several analyses...
in a class hierarchy can be performed on this flow graph, for example, implicit inheritance analysis and polymorphic message analysis. The analyses may be applied in various fields of software engineering, such as static analysis, maintenance, and complexity measurement.

2. Background

2.1. Object-oriented programs

OO programming encapsulates related data and operational procedures as attributes and methods through an object. An object is instantiated from a class, which defines attributes and methods (called members for the both). With inheritance, a class can obtain the members from its superclass. A subclass cannot only use the inherited members, but also redefine them. Inheritance rules are different for various OO languages, and Java [5] is adopted to demonstrate the class hierarchy analysis through this paper. In addition to classes, Java provides the interface, a collection of constants and procedure prototypes (signatures). For a class, inheritance is achieved by specifying its superclass (at most one) in an extends clause and its superinterfaces in an implements clause. An interface to subinterfaces is similar to a class to subclasses. An interface can extend more than one interface at a time. Thus, the class hierarchy includes not only class inheritance but also interface inheritance. Detailed features can be found in [5].

2.2. Preliminary definitions

**Definition 1.** A digraph is an ordered pair \( G(V, E) \), where \( V \) is a finite set of vertices, and \( E \subseteq V \times V \) is a finite set of directed edges. For an edge \( e, e \in E \), from a vertex \( v_1 \) to a vertex \( v_2 \), \( v_1 \) is called the initial vertex of \( e \), denoted as \( IV(e) \), and \( v_2 \) is called the terminal vertex of \( e \), denoted as \( TV(e) \).

**Definition 2.** A multi-digraph is an \( n+1 \) tuple \( (V, (E_1, E_2, \ldots, E_n), T) \) such that \( (V, (E_1, E_2, \ldots, E_n)) \) is a multi-digraph, where

1. \( V \) is a finite set of vertices,
2. \( E_i \subseteq V \times V \) for \( 1 \leq i \leq n \) is a finite set of directed edges,
3. \( T \) is a finite set of vertex tags; \( T = \bigcup_{X}^{X \subseteq V} T(X) \), where \( T(X) \) is a set of vertex tags associated with vertex \( X \).

**Definition 3.** A tagged multi-digraph is a tuple \( (V, (E_1, E_2, \ldots, E_n), T) \) such that \( (V, (E_1, E_2, \ldots, E_n)) \) is a multi-digraph, where

1. \( V \) is a finite set of vertices,
2. \( E_i \subseteq V \times V \) for \( 1 \leq i \leq n \) is a finite set of directed edges,
3. \( T \) is a finite set of vertex tags; \( T = \bigcup_{X}^{X \subseteq V} T(X) \), where \( T(X) \) is a set of vertex tags associated with vertex \( X \).

**Definition 4.** Let \( G = (V, (E_1, E_2, \ldots, E_n), T) \) be a tagged multi-digraph.

1. A path in \( G \) is a path in the multi-digraph \( (V, (E_1, E_2, \ldots, E_n)) \).
2. Let \( E_x \) and \( E_y \) be two sets of edges in \( G \). Given a path

\[
v_a \rightarrow v_b = (e_1, e_2, \ldots, e_k) \quad \text{and} \quad v_a \xrightarrow{E_x \cup E_y} v_b
\]

denotes that for all \( i, 1 \leq i \leq k \), \( e_i \in E_x \cup E_y \).

3. Inheritance flow model

In a class hierarchy, a path from a root class to its subclass via inheritance relationship(s) is called inheritance flow. Along an inheritance flow, public and private members defined in a class can be inherited by its subclass implicitly. The specification of a member in a class consists of signature and body parts, which denote the declaration and implementation/instantiation for the member, respectively. Thus, inheritance flow has to take the signature and body into consideration. The evolution of a member in inheritance flow can be described by flow operations as the followings.

**Definition 5.** In inheritance flow, an operation on the signature of a member in a class or interface is either a signature-inheritance define \( (D_{si}) \) or signature-inheritance use \( (U_{si}) \). In a class (or interface),

1. a \( D_{si} \) on a member means that the signature of the member is declared in the class or interface originally;
2. a \( U_{si} \) on a member means that the signature of the member is inherited from a superclass or superinterfaces, i.e., the class or interface possesses the signature without defining it.
Definition 6. In inheritance flow, an operation on the body of a member in a class is either a body-inheritance define (Dbi), body-inheritance use (Ubi), or null (Nbi). In a class,
1. a Dbi on a member means that the body of the member is newly defined or redefined in the class;
2. a Ubi on a member means that the body of the member exists, but does not be specified in the class;
3. an Nbi on a member means that neither Dbi nor Ubi on the body of the member, i.e., the body does not exit.

In inheritance flow, a member in a class can be associated with a pair of flow operations for its signature and body by \( \{D, i, U\} \times \{Dbi, Ubi, Nbi\} \). There are six combinations for a flow operation pair, but only five of them are reasonable. The excluded one is \( \{D, i, Ubi\} \), because no member can let its signature be redefined, but remain its body.

To represent the structure of an OO program, we define an Inheritance Flow Graph (IFG) based on a tagged multi-digraph.

Definition 7. Let \( P \) be a program. An Inheritance Flow Graph (IFG) of \( P \) is defined as \( G_{IFG}(P) = (V, E, T) \), where \( (V, E, T) \) is a tagged multi-digraph, and

1. \( V = V_c \cup V_i \cup V_m \cup V_a \), where
   - \( V_c \) is a set of vertices representing classes,
   - \( V_i \) is a set of vertices representing interfaces,
   - \( V_m \) is a set of vertices representing methods, and
   - \( V_a \) is a set of vertices representing attributes;
2. \( E = (E_{ext}, E_{imp}, E_{pub}, E_{pro}, E_{pri}) \), where
   - \( E_{ext} \subseteq (V_c \cup V_i) \times (V_c \cup V_i) \) is a set of edges representing class inheritance,
   - \( E_{imp} \subseteq (V_i \times (V_c \cup V_i)) \) is a set of edges representing interface inheritance,
   - \( E_{pub} \subseteq (V_c \cup V_i) \times (V_m \cup V_a) \) is a set of edges representing public-membership of attributes and methods in a class,
   - \( E_{pro} \subseteq (V_c \cup V_i) \times (V_m \cup V_a) \) is a set of edges representing protected-membership of attributes and methods in a class,
   - \( E_{pri} \subseteq (V_c \cup V_i) \times (V_m \cup V_a) \) is a set of edges representing private-membership of attributes and methods in a class;
3. \( T = \bigcup_{X \in V_a \cup V_m} T(X) \), where \( T(X) \) is a pair of vertex tags, denoted as \( \langle a, b \rangle \), associated with \( X \) to represent a pair of flow operations for the signature and body of a member.

A member associated with \( \langle U_{si}, U_{bi} \rangle \) or \( \langle U_{si}, N_{bi} \rangle \) in a class implies that it is inherited from a superclass or superinterface. Since the member is implicit for the class, there is no membership edge from the vertex of the class to that of this member in an IFG. That is, \( T(X) \) in an IFG is either \( \langle D_{si}, D_{bi} \rangle \), \( \langle D_{si}, N_{bi} \rangle \), or \( \langle U_{si}, D_{bi} \rangle \). Note that if an OO language provides public, protected, and public inheritances, the IFG needs different inheritance edges to represent them. Since Java language allows public inheritance only, the other two inheritances and related work are not discussed here.

For a program in Fig. 1, its IFG is illustrated in Fig. 2. Attribute attrib in \( C0 \) is inherited by \( C1 \), but does not appear in the definition of \( C1 \). In the graph, there is no edge from \( C1 \) to attrib (i.e., \( C1 \rightarrow attrib = \varepsilon \)).

4. Class hierarchy analysis

4.1. Define-use pair analysis

In a class hierarchy, the inheritance of a member from one class to its subclass can form a define-use pair in terms of flow operations. According to the inheritance flow model, we can define a define-use pair with respect to the signature and body of an inherited member.

Definition 8. Let \( c_1 \) and \( c_2 \) be classes or interfaces, \( \{c_1, c_2\} \subseteq V_c \cup V_i \). Let \( m \) be a member, \( m \in V_m \cup V_a \). Given two operations on \( m \) in \( c_1 \) and \( c_2 \), \( m, c_1, c_2 \) is said to be a signature define-use \( \langle DU_{si} \rangle \) pair iff the following conditions are true:
1. \( c_1 \rightarrow m \in E_{pro} \cup E_{pub} \) and \( T(TV(c_1 \rightarrow m)) \in \{\langle D_{si}, D_{bi} \rangle, \langle D_{si}, N_{bi} \rangle\} \),
2. either \( T(TV(c_2 \rightarrow m)) = \langle U_{si}, D_{bi} \rangle \) or \( c_2 \rightarrow m = \varepsilon \),
3. \( c_1 \rightarrow c_2 \), and
interface T0 {
    public void f1(); // Dsi, Nbi
    public void f2(); // Dsi, Nbi
}

interface T1 extends T0 {

abstract class C0 {
    protected int attrib; // Dsi, Dbi
    public void f2() {attrib=2;} // Dsi, Dbi
}

class C1 extends C0 implements T1 {
    public void f2() {
        attrib = 100;
        ...
    }
    public void f1() {
        attrib = 10;
        ...
    }
}

// Definition of class C1
class C2 extends C1 {
    public void f2() {
        ...
    }
}

Fig. 1. An example program.

(iv) for all $\alpha$,

$$\alpha \in V_c \cup V_i \wedge c_1 \xrightarrow{E_{ext}\cup E_{imp}} \alpha \wedge \alpha \xrightarrow{E_{ext}\cup E_{imp}} c_2,$$

such that either $T(TV(\alpha \rightarrow m)) = (U_{si}, D_{bi})$ or $\alpha \rightarrow m \neq \varepsilon$.

In Definition 8, condition (i) denotes $m$ is associated with a $D_{si}$ in $c_1$, and can be inherited by $c_1$'s subclasses. Condition (ii) denotes $m$ is associated with a $U_{si}$ in $c_2$. Condition (iii) is that there is an inheritance path from $c_1$ to $c_2$. Condition (iv) indicates that $m$'s signature defined in $c_1$ does not be refined by other classes (or interfaces) before the signature is inherited by $c_2$. The $D_{U_{si}}$ pair indicates that the evolution of $m$ in the class hierarchy begins at $c_1$. $m$ may have more than one implementation from $c_1$ to $c_2$. These implementations belong to $m$'s evolution. When a programmer redefines $m$'s body in $c_2$, the evolution will be updated.

**Definition 9.** Let $c_1$ and $c_2$ be classes, \{c_1, c_2\} $\subseteq V_c$. Let $m$ be a member, $m \in V_m \cup V_a$. Given two operations on $m$ in $c_1$ and $c_2$, $(m, c_1, c_2)$ is said to be a body define-use ($DU_{bi}$) pair iff the following conditions are true:

(i) $c_1 \rightarrow m \in E_{pro} \cup E_{pub}$ and $T(TV(c_1 \rightarrow m)) \in \{(D_{si}, D_{bi}), (U_{si}, D_{bi})\}$,

(ii) $c_2 \rightarrow m \neq \varepsilon$,

(iii) $c_1 \rightarrow$, and

(iv) for all $\alpha$,

$$\alpha \in V_c \wedge c_1 \xrightarrow{E_{ext}} \alpha \wedge \alpha \xrightarrow{E_{ext}} c_2,$$

such that $\alpha \rightarrow m \neq \varepsilon$.
This definition is similar to the previous one. The operations on \( m \) are a \( D_{bi} \) and a \( U_{bi} \), and \( m \)'s body here is inherited via classes only. The \( DU_{bi} \) pair means that \( m \) is implicit inherited by \( C_2 \) and its body is defined in \( C_1 \). The implicit inheritance of \( m \) vanishes if \( m \)'s body is redefined in \( C_2 \).

For example, in Fig. 2, the operations on \( f_1 \) in \( T_0 \) and \( C_1 \) form a \( DU_{si} \) pair, while those on \( attrib \) in \( C_0 \) and \( C_1 \) form a \( DU_{bi} \) pair. All the \( DU_{si} \) and \( DU_{bi} \) pairs in the example program are shown in Table 1.

### 4.2. Inheritance analysis

The implicit members via inheritance relationships can be represented as the flow information in class hierarchies. For a class, its flow information includes (a) what members can be inherited by the class, and (b) which superclasses/super-interfaces bring these members. The flow information of a class includes the signatures and bodies of inherited members can be presented with \( DU_{si} \) and \( DU_{bi} \) pairs.

**Definition 10.** For a class or interface \( C, C \in V_c \cup V_i \), the inheritance flow of member signatures is \( \{(\alpha, x) \mid \alpha \in V_c \cup V_i \text{ and } x \in V_m \cup V_a \text{ such that } (x, \alpha, C) \text{ is a } DU_{si} \text{ pair}\} \).

This definition shows that the flow of member signatures entering a class (or interface) includes the member signatures originally defined in superclasses or superinterfaces. Similarly, the flow of member bodies entering a class is described in Definition 11.

**Definition 11.** For a class \( C, C \in V_c \), the inheritance flow of member bodies is \( \{(\alpha, x) \mid \alpha \in V_c \text{ and } x \in V_m \cup V_a \text{ such that } (x, \alpha, C) \text{ is a } DU_{bi} \text{ pair}\} \).
5. Applications

There are several potential applications of the analyses with the inheritance flow model, such as static analysis, maintenance, complexity measurement. For example, navigation in class hierarchies is inevitable for a programmer during maintaining and reusing OO systems [10]. The class hierarchy analysis can facilitate class navigation by collecting the define-use pairs of members. These pairs show what implicit inherited members a class can possess, and where these members are from.

Like data flow anomaly detection [4], the inheritance flow model can provide static analysis for detecting anomalies in class hierarchies, such as unimplemented member [11] and repeated inheritance [2]. A member propagated along an inheritance path, from $C_1$, $C_2$, ..., to $C_k$, can be regarded as a sequence of flow operations, $(a_1, b_1)\{a_2, b_2\}...\{a_k, b_k\}$. The operation sequence can be represented in terms of a regular expression with '|' denoting 'or' and '+' denoting repetition at least once. An unimplemented member anomaly occurs when a member owns its signature without body in a subclass. The anomaly can be detected by examine if an operation sequence of a member contains the expression $(D,i, N_{bi}) (Usi, N_{bi})$. A repeated inheritance occurs when there are two or more inheritance paths, along which there exits an identical DU_{bi} pair. For example, given two inheritance paths from a class $C_1$ to another $C_2$, a repeated inheritance anomaly can be detected when the operation sequences of a member corresponding to the two paths can be expressed as $(D_{si}, D_{bi}) | (U_{si}, D_{bi})$. This expression implies that the member in $C_1$ and $C_2$ forms a DU_{bi} pair.

Software complexity measurement captures the programming difficulties during development and predicts the maintainability and testability of the software. Current measurement approaches for OO programs, such as [3,7], do not consider the factor of behavior evolution in class hierarchies. The behavior evolution, for example, includes the overridden and inherited members and the multiple implementations of a method. However, the behavior evolution can be represented as some specific sequences of flow operations in this model. The number of these sequences might thus be one index for the complexity measurement.
Acknowledgements

The authors would like to thank the referees, whose comments helped to improve the overall presentation. This research was sponsored by MOEA and supported by Institute for Information Industry, Taiwan, R.O.C.

References