JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 16, 619-647 (2000)

Flow Analysis of Class Relationships for Object-Oriented Programs

JIUN-LIANG CHEN AND FENG-JIAN WANG
Department of Computer Science and Information Engineering
National Chiao Tung University
Hsinchu, Taiwan 300, R.O.C.

Program analysis techniques have been widely applied in various fields of software engineering, such as debugging, testing, and proof of simple correctness properties. In object-oriented (OO) programs, inheritance, association, and aggregation relationships may introduce complicated dependencies concealed within classes that might obstruct program analysis. This paper proposes a class relationship flow models to provide analysis for inheritance, association, and aggregation of class relationships. The flow model consists of three flows, inheritance, association, and aggregation flows, corresponding to these relationships. A sequence of class relationships is represented as a flow path from one class to another. Along a flow path, each member within a class is associated with an operation, define or use, to represent whether its status is changed or referenced. Thereby, the concealed dependencies introduced by class relationships can be analyzed according to the flow operations. The analysis might be used as a technique for program understanding, anomaly detection, and program testing.

Keywords: program analysis, class relationship, object-oriented, software engineering, program dependence graph

1. INTRODUCTION

Program analysis predicates some properties of the dynamic behavior of a program statically. It has been extensively used to enable various optimizations and transformations in compilers. Program analysis techniques are also widely applied in software engineering, such as in static debugging, testing, proof of simple correctness properties, and so forth [1]. The object-oriented (OO) paradigm for software development has gained momentum and popularity over the years. The paradigm provides the features of object abstraction, encapsulation, inheritance, and polymorphism for program construction. More and more OO components and class libraries have become available. These libraries encourage program reuse for building (large-scale) software systems, and the analysis of OO components for reuse, testing, debugging, and maintenance is becoming important. Most program analyzers focus on detecting features likely to be bugs for a single class in light of the specific syntax of a language [2]. Nevertheless, an appropriate model for analyzing class libraries is still lacking.

In OO programs, a class encapsulates attributes and methods (both are also called members) as the state and behavior of its instances (objects). The details of programs are often hidden inside classes deeply. The relationships between classes include *inheritance, association,* and *aggregation* [3]. An *inheritance* relationship between one class and another,

Received December 17, 1997; revised April 13, 1998; accepted August 20, 1998.
Communicated by Y. S. Kuo.

called subclass, means that the subclass can possess members of the class. An *association* relationship between two classes, where one is associated with the other, means that the latter class's members can be used in the former. An *aggregation* relationship between two classes means that an instance of one class is a part of one instance of the other. On the other hand, such a relationship implies that one class can propagate some information (e.g., methods or attributes) to the other via the relationship. This propagation can be transitive via a sequence of relationships over several classes. That is, class relationships may introduce dependencies concealed within classes. The class diagrams used in most OO methodologies [4, 5] are too coarse-grained to describe the concealed information propagated via these class relationships. For example, from Fig. 1, one observes that "class B inherits from class A" and "class E inherits from class B." However, one may not know which members defined in A and B are inherited by E. As to association, one can know that "class A can be associated with class S by invoking method draw() in S." If the invocation is made by means of a polymorphic message, the methods implemented in classes T and U can be invoked potentially. That is, A has implicit association relationships with T and U. Based on aggregation, it is obvious that "class Y encapsulates class X's instance as its attribute" and "X encapsulates class E's instance as its attribute." This figure does not reveal "which members in E and X are accessible in Y." Such information propagated via class relationships implicitly might increase the difficulty of understanding, debugging, and testing OO programs [6-8].

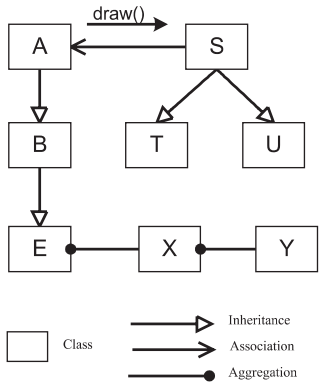

Fig. 1. An example of a class diagram.

In [9], an inheritance flow model was presented to reveal members propagated among classes via inheritance relationships. The model does not consider the other two class relationships that might incur implicit dependencies among classes. In order to analyze these implicit dependencies, this paper proposes a class relationship flow model, which consists of *inheritance*, *association*, and *aggregation* flows used to represent implicit propagation introduced by class relationships, by extending the inheritance flow model. The association flow describes the method invocations and attribute accesses along with a sequence of association relationships. The aggregation flow describes the members of a class that are accessible along with a sequence of aggregation relationships.

In the class relationship flow model, a sequence of class relationships is represented as a flow path class by class. Each member within a class is associated with an operation, *define* or *use*, to describe whether its status is changed or referenced along a flow path. Hence, the implicit information propagated among classes can be deemed as along these flow paths of class relationships. Such notation is the same as traditional data flow [10]. To illustrate the flow model in practice, Java [11] is used in sample programs through this paper. This flow model can be used in program analysis techniques in several applications, e.g., program understanding, anomaly detection, and program testing.

The remainder of this paper is organized as follows. Section 2 reviews related work on flow analysis. Section 3 proposes a class relationship flow model that consists of inheritance, association, and aggregation flows. Then, the analysis of class relationship flow is presented in the next section. Section 5 discusses several applications of flow analysis. In the final section, we draw conclusions and suggest directions for work.

2. BACKGROUND

2.1 Preliminary Definitions

In object-oriented languages, the definition of a member (i.e., a method or attribute) in a class can be divided into *signature* and *body* parts, where the signature denotes the member declaration, and the body denotes the member implementation. A member can be denoted as (t, n, p, b), where t is a member type (a returned value type for a method), n is a member name or identifier, p is a list of formal parameters (none for an attribute), and b is a member body. A member signature consists of t, n, and p. A member is *abstract* or *pure virtual* when it has no body, i.e., b is null. Let $M_1 = (t_1, n_1, p_1, b_1)$ and $M_2 = (t_2, n_2, p_2, b_2)$ be two members. The signatures of M_1 and M_2 are identical if $t_1 = t_2$, $n_1 = n_2$, and $p_1 = p_2$. These two signatures are *indistinguishable* if $t_1 \neq t_2$, $n_1 = n_2$, and $p_1 = p_2$.

In a class, let a member δ be inherited from a superclass, and let a member δ' be specified in the class definition. The signature of δ' overrides that of δ when the two signatures are indistinguishable. The body of δ' will override that of δ in the class when the two signatures are indistinguishable or identical. For example, in the program shown in Fig. 2, the signature of member id in class Vehicle and that in class Car are indistinguishable because one is of the int type, and the other is of the String type. The signatures of move() in the two classes are identical. In this case, id in Car overrides the signature and body of that inherited from Vehicle whereas move() in Car overrides only the body of that inherited from Vehicle.

```
class Vehicle{
        public int id;
        public void move(){ /* body of Vehicle move */};
  }
class Car extends Vehicle {
        public String id;
        public void move(){ /* body of Car move */ };
}
```

Fig. 2. An example of member overriding.

To represent the structure of OO programs, we define a graph, called a *Program Structure Graph* (PSG). A PSG is a multi-digraph, where vertices represent classes, interfaces, methods, and attributes, and multiple edge sets represent class inheritance, interface inheritance, public-/protected-/private-memberships, declaration, and method invocation, respectively. The vertex and multiple edge sets were first discussed in [12]. A program structure graph for an OO program can, thus, be defined as follows.

Definition 2.1: Let P be an OO program. A *Program Structure Graph* (PSG) of P is defined as $G_{PSG}(P) = (V, E)$, where:

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. $E = (E_{ext}, E_{imp}, E_{pub}, E_{pro}, E_{pri}, E_l, E_m)$, where
   - $E_{ext} \subseteq V_c \times V_c$ is a set of edges from a class to its immediate subclass representing class inheritance,
   - $E_{imp} \subseteq V_i \times (V_c \cup V_i)$ is a set of edges from an interface to its immediate subclass or subinterface representing interface inheritance,
   - $E_{pub} \subseteq (V_m \cup V_a) \times (V_c \cup V_i)$ is a set of edges from a member to its definition class representing public-membership relationships,
   - $E_{pro} \subseteq (V_m \cup V_a) \times (V_c \cup V_i)$ is a set of edges from a member to its definition class representing protected-membership relationships,
   - $E_{pri} \subseteq (V_m \cup V_a) \times (V_c \cup V_i)$ is a set of edges from a member to its definition class representing private-membership relationships,
   - $E_l \subseteq V_c \times V_a$ is a set of edges from a class to an attribute representing declaration dependencies [12], and
   - $E_m \subseteq (V_m \cup V_a) \times V_m$ is a set of edges from a member to a method that accesses the member directly.

The subsequent definition for a PSG is for convenience IN presenting our model.

Definition 2.2: Let e_1, e_2, ..., and e_k be a set of edges in a PSG. (e_1, e_2, ..., e_k) is a path in the graph if and only if the terminal vertex of e_i is the initial vertex of e_{i+1} for $1 \le i \le k-1$. Let v_I be the initial vertex e_1, and let v_T be the terminal vertex of e_k. The path from v_I to v_T is denoted as $v_I \rightarrow v_T$ for short. For a path $v_a \rightarrow v_b = (e_1, e_2, ..., e_n)$ $V_a \xrightarrow{E_x \cup E_y \cup \cdots \cup E_z} V_b$, means that $\forall j$, $1 \le j \le n$, $e_j \in E_x \cup E_y \cup ... \cup E_z$.

From Program I (see Fig. 3), we can construct a PSG as shown in Fig. 4. The classes for primitive data types (e.g., char, integer, etc.) are omitted in a PSG.

```
class Rec{  public long array[] = new long[200]; }
class C0 {
     public int stateC = 0;
}
```

```
interface FunPack {
    abstract public void funC();
}
class C1 extends C0 implements FunPack {
    public Rec nameList = new Rec();
    public void funC( ) {  ...}
}
class C2 extends C1
    private S0 objS = new S0( );
    public void setup( ){  objS.setS( ); }
}
class S0{
    private int stateS;
    public void setS( ){  stateS = 0; }
}
```

Fig. 3. Program I.

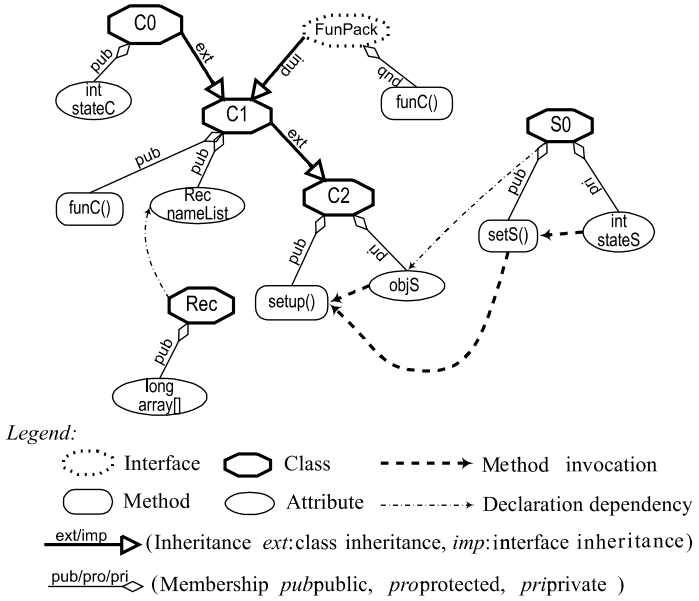

Fig. 4. The PSG of Program I.

2.2 Related Work

Data flow analysis is a technique used to ascertain and collect information about *define*, *use*, and *kill* operations on variables in a program (e.g., [10, 13]). A variable is defined in a statement if a value is assigned to the variable. A variable is *used* if the variable is referenced in a statement. A variable is *killed* if the value of the variable is no longer

available. Conventional data flow analysis often represents a program as a control flow graph, where vertices denote statements, and edges denote execution sequences between statements. Along an execution path in the graph, the flow information for a variable can be expressed as a sequence of operations. The flow information is useful not only for optimizing and parallelizing compilers [14], but also for using many program analysis techniques (e.g., program dependency graphs [15, 16], program slicing [17], ripple effect analysis [18], and so on). In OO programs, a variable represents an object with its own state and behavior defined in a class, rather than data only. The data flow information alone is insufficient to analyze the information implicitly propagated via class relationships.

Sudholt and Steigner [19] extended an inter-procedural data flow analysis algorithm for OO programs. Their approach first thoroughly decomposes an object into a set of procedures and global variables and then performs an inter-procedural data flow analysis algorithm (e.g., [20]) on these variables with the procedures. It focuses on low-level data flow information for optimization and parallelization of compilation. In [21], Hierarchical Data Flow Analysis (HDFA) was proposed to explore data flow information in OO programs for three hierarchical layers: classes, objects, and attributes. The HDFA consists of class flow, object flow, and attribute flow, in which three kinds of operations, *kill*, *define*, and *use*, are used to describe the states of attributes, objects, and classes for each of the hierarchical layers. As in traditional data flow analysis, an object flow and a class flow are derived from the state change of the variables when a number of messages are executed. The extended inter-procedural data flow analysis and HDFA focus on the state change of program execution, rather than on class relationships.

Kung in [3] proposed an object relation diagram (ORD) to represent the relationships between classes with inheritance, association, and aggregation for change impact analysis during regression testing. An example of ORD is shown in Fig. 1. The drawback of ORD is that it is too coarse-grained for exploring implicit propagation among classes via class relationships.

3. A FLOW MODEL FOR CLASS RELATIONSHIPS

This section presents a class relationship flow model that consists of inheritance, association, and aggregation flows used to express the implicit information propagated among classes via inheritance, association, and aggregation relationships.

3.1 Inheritance Flow

Classes in an OO program can be structured as a hierarchy via inheritance relationships. In a class hierarchy, a class can either use the members inherited from its superclasses without explicit declaration, or it can redefine them. Both the signature and body of an inherited member can be propagated via inheritance, and they are called *signature-inheritance* and *body-inheritance*, respectively.

In [9], an inheritance flow model was proposed to describe the members of inheritance in a class hierarchy. In this model, each member is associated with a pair of operations which stand for its signature and body defined or used in a class. For signature/body-inheritance, the operations on a member are defined as follows.

Definition 3.1: In inheritance flow, an operation on the signature of a member for a class or interface is either a *signature-inheritance define* (D_{si}) or *signature-inheritance use* (U_{si}).

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 body of the member is implemented in the class, and that the member signature is inherited from a superclass.

A member with a D_{si} indicates that the member signature is not inherited from a superclass. An operation on a member is a U_{si} when the class overrides the member body inherited from a superclass.

Definition 3.2: In inheritance flow, an operation on the body of a member for a class is either a *body-inheritance define* (D_{bi}), *body-inheritance use* (U_{bi}), or *null* (N_{bi}).

1. A D_{bi} on a member means that the body of the member is newly defined or redefined in the class.
2. A U_{bi} on a member means that the body of the member exists but is not specified in the definition of the class.
3. An N_{bi} on a member means that neither D_{bi} nor U_{bi} on the body of the member, i.e., the body does not exist.

A D_{bi} on a member indicates that the member body is implemented or re-implemented, while a U_{bi} means that the member body is inherited from a superclass, no matter whether or not it is used in a class. An N_{bi} on a member means that the member is abstract.

To simplify discussion as in [9], we can define a flow graph, called a *class relationship graph* (CRG) so as to represent inheritance, association, and aggregation relationships for our flow model. The CRG is extended from the PSG with vertex tags. A class relationship flow graph for a program can, thus, be defined as follows.

Definition 3.3: A *Class Relationship Graph* (CRG) of an OO program P is defined as $G_{CRG}(P) = (V, E, T)$, where:

1. (V, E) is $G_{PSG}(P)$.
2. $T = \{<t_{si}, t_{bi}> \mid t_{si} \in \{D_{si}, U_{si}, \varepsilon\}$ and $t_{bi} \in \{D_{bi}, U_{bi}, N_{bi}, \varepsilon\}\}$ is a set of vertex tags. $T(X) = <X.t_{si}, X.t_{bi}>$ is a pair of vertex tags associated with vertex X, where $X.t_{si}$ and $X.t_{bi}$ represent the operations of signature-inheritance and body-inheritance on X, respectively.

In a class, a member associated with $<U_{si}, U_{bi}>$ or $<U_{si}, N_{bi}>$ implies that it is inherited from a superclass or superinterface. Since the member is not defined for the class, there is no vertex with vertex tag $<U_{si}, U_{bi}>$ or $<U_{si}, N_{bi}>$ in CRG. Therefore, $T(X)$ in CRG is either $<D_{si}, D_{bi}>$, $<D_{si}, N_{bi}>$, or $<U_{si}, D_{bi}>$. Note that if an OO language provides public, protected, and public inheritances, then the CRG needs different inheritance edges to represent them. Here, our target language, Java, allows public inheritance only; the other two inheritances and related work are not discussed.

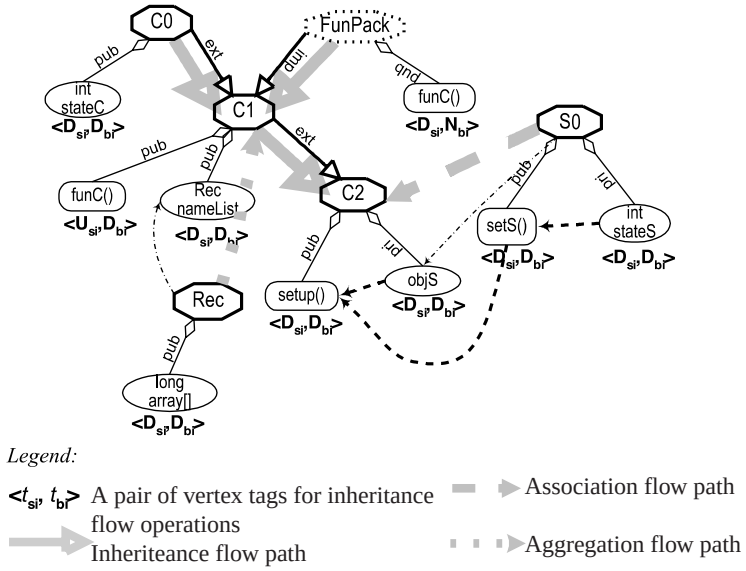

Fig. 5. The CRG of Program I.

An example of a CRG is shown in Fig. 5. The CRG is constructed from Program I. The pairs of vertex tags denoting the flow operations are attached to the bottom of each method and attribute vertice. The vertex tags for the vertices without corresponding inheritance flow operations are not shown in the figure.

For example, method funC() in interface FunPack shown in Fig. 3 is associated with $<D_{si}, N_{bi}>$, since it is abstract. For attribute attrib in C0, its associated operation is $<D_{si}, D_{bi}>$ because its signature and body have been defined. In class C1, method funC()'s body is defined, but the signature is inherited from FunPack. Therefore, funC() in C1 is associated with $<U_{si}, D_{bi}>$. Class C2 possesses attrib and funC() inherited from its superclasses implicitly. These two members are available in C2; the operations on them are $<U_{si}, U_{bi}>$, and they do not appear in the program context of C2.

An inheritance flow path from one class to another in CRG can be specified as in the following definition.

Definition 3.4: Let q_1 and q_2 be two class or interface vertices in a CRG. A flow path from vertex q_1 to vertex q_2 is an *inheritance flow path*, denoted as $q_1 \xrightarrow[IHF]{} q_2$, iff one of the following holds:

1. $q_1 \to q_2 \in E_{ext} \cup E_{imp}$; or
2. $\exists \alpha$, $\alpha \in V_c \cup V_i$, such that $q_1 \to \alpha \in E_{ext} \cup E_{imp}$ and $\alpha \xrightarrow[IHF]{} q_2$.

For example, an inheritance flow path from class C0 to class C2 is "$C0 \xrightarrow[Eext]{} C1 \xrightarrow[Eext]{} C2$", shown as the bold, grey arrow in Fig. 5.

3.2 Association Flow

An association relationship between two classes denotes that the execution of a method in an instance of one class might send a message to an instance of the other class to invoke the corresponding method. For a class, its method body includes the messages sent to its parameters (if they exist), class attributes, and global objects. These messages, whose receivers could be the instances of other classes, thus increase the number of method executions. Again, the executions will incur message passing. That is, a method might be invoked by a number of methods for execution along a sequence of association relationships of classes. The method invocation sequences along with association relationships constitute the *association flow*. A sequence of association relationships is called an *association flow path.*

Along an association flow path, a set of members might be invoked (or accessed) by a message. This can be described by the flow operations defined below.

Definition 3.5: In association flow, the operation on a member for a class is an *association define* (D_{as}) or *association use* (U_{as}).

1. A D_{as} on a member means that the class owns the member.
2. A U_{as} on a member means that the class contains a message that might access or invoke the member.

According to this definition, a D_{as} on a member implies that the class explicitly defines the member or inherits it from the other class. A U_{as} on a member implies that the member might be invoked by some message within the class.

For example, in Fig. 5, the method setup() and attribute objS in class C2 are both associated with D_{as} since they are defined in the class. Similarly, the methods setS() and stateS in class S0 are associated with D_{as}. There is a message in setup() to invoke setS(), and setS() contains a message to access stateS. Therefore, the operations on setS() and stateS are U_{as}.

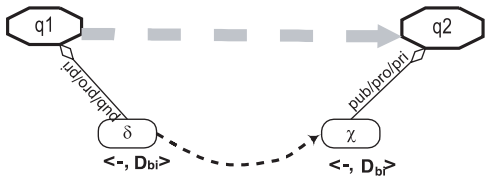

Fig. 6. An association flow path.

In CRG, an association flow path from one class to another can be defined as follows. The definition is recursive. The necessary base condition of $q_1 \xrightarrow[ASF]{} q_2$ means that member δ of class q_1 might be accessed by the method χ of class q_2. $\delta \xrightarrow[Em]{} \chi$ means that δ might be invoked or accessed by some message within χ. The base condition is illustrated in Fig. 6.

Definition 3.6: Let q_1 and q_2 be two class vertices in a CRG. A flow path from vertex q_1 to vertex q_2 is an *association flow path*, denoted as $q_1 \xrightarrow[ASF]{} q_2$, iff one of the following holds:

1. $\exists\ \delta$, $\delta \in V_m \cup V_a$, and $\exists\ \chi$, $\chi \in V_m$, such that $\delta \xrightarrow[Epub \cup Epro \cup Epri]{} q_1 \wedge \delta \xrightarrow[Em]{} \chi \wedge \chi \xrightarrow[Epuh \cup Epro \cup Epri]{} q_2$; or
2. $\exists\ \alpha$, $\alpha \in V_c$, $\exists\ \delta'$, $\delta' \in V_m \cup V_a$, and $\exists\ \chi'$, $\chi' \in V_m$, such that $\delta' \xrightarrow[Epub \cup Epro \cup Epri]{} q_1 \wedge \delta' \xrightarrow[Em]{} \chi' \wedge \chi' \xrightarrow[Epub \cup Epro \cup Epri]{} \alpha$, and $\alpha \xrightarrow[ASF]{} q_2$.

In Fig. 5, an association flow path, $S \xrightarrow[ASF]{} C2$, is shown by the dashed bold arrow because "setS() $\xrightarrow[Epub]{}$ S"; i.e., class S owns method setS(), and "setS() $\xrightarrow[Em]{}$ setup() $\xrightarrow[Epub]{}$ C2".

3.3 Aggregation Flow

An aggregation relationship between two classes means that an instance of one class is encapsulated as an attribute in the other class. A class that has an aggregation relationship with another class can be encapsulated in the latter class. With a sequence of aggregation relationships, one class's members concealed within another class under multiple layers of encapsulation might be still accessible in the latter class. The accessible members along with aggregation relationships are called an *aggregation flow*. A sequence of aggregation relationships is called an *aggregation flow path*.

In an aggregation flow path, a class's instance can be encapsulated as an attribute within another class along the sequence of aggregation relationships. The operations in aggregation flow are stated in Definition 3.7.

Definition 3.7: In an aggregation flow, the operation on a member in a class is either an *aggregation define* (D_{ag}) or *aggregation use* (U_{ag}).

1. A D_{ag} on a member means that the class owns the member.
2. A U_{ag} on a member means that the member can be directly accessed within the class.

According to the definition, in a class, a D_{ag} on a member implies that the class explicitly defines the member or inherits it from another class. A U_{ag} on a member implies that the member can be accessed in the class. For example, class Rec in Program I encapsulates

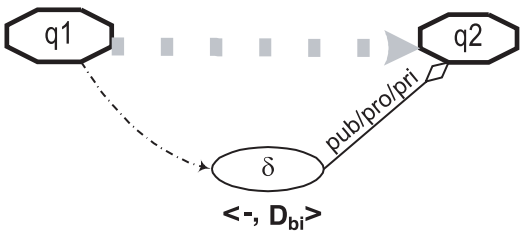

Fig. 7. An aggregation flow path.

array as a public attribute. The operation on the attribute in Rec is D_{ag}. The object nameList, an instance of Rec, is encapsulated as an attribute in class C1. Hence, Rec's attribute array is accessible in C1, and the operation on array in C1 is a U_{ag}.

In CRG, an aggregation flow path from one class to another can be described as follows. The necessary condition to form an aggregation flow path between two classes is that one class encapsulates an instance of the other as an attribute directly or indirectly. The first condition of Definition 3.8 is shown in Fig. 7.

Definition 3.8: Let q_1 and q_2 be two class vertices in a CRG. A flow path from vertex q_1 to vertex q_2 is an *aggregation flow path*, denoted as $q_1 \xrightarrow[AGF]{} q_2$, iff one of the following holds:

1. $\exists\ \delta,\ \delta \in V_a$, such that $q_1 \xrightarrow[E_l]{} \delta \wedge \delta \xrightarrow[Epub \cup Epro \cup Epri]{} q_2$; or
2. $\exists\ \alpha,\ \alpha \in V_c$, and $\exists\ \delta',\ \delta' \in V_a$, such that $q_1 \xrightarrow[E_l]{} \delta' \wedge \delta' \xrightarrow[Epub \cup Epro \cup Epri]{} \alpha$, and $\alpha \xrightarrow[AGF]{} q_2$.

For example, an aggregation path from class Rec to class C1 in Program I is shown as the dotted bold arrow via "$\mathrm{Rec} \xrightarrow[E_l]{} \mathrm{nameList} \xrightarrow[Epub]{} C1$", in Fig. 5. Note that an interface in Java can be regarded as a special abstract class because it contains method signatures and constants only. Therefore, an interface is contained in an inheritance flow only, not in association and aggregation flows.

4. FLOW ANALYSIS OF CLASS RELATIONSHIPS

With the class relationship flow model, we can analyze the flow among classes via inheritance, association, and aggregation relationships. The analysis can be performed through the CRG of a program.

4.1 Define-Use Relation

Traditional data flow provides the relations of *define* and *use* operations on variables in control flow for various applications. A define-use relation indicates that the state of a variable is changed and referenced along an execution path; this is the essential information for program parallelization, optimization, and testing [22, 23]. The flow operations in our model may form the define-use relation as in traditional data flow. Along a flow path from one class to another, a define operation on a member in the former class and a use operation on one in the latter class can be deemed as a define-use relation, called a *define-use* pair. A define-use pair formed by two flow operations is shown in Definition 4.1.

Definition 4.1: Let D_x be a define operation, and let U_x be a use operation, where the subscript x is '*si*,' '*bi*,' '*as*,' or '*ag*', corresponding to signature-inheritance, body-inheritance, association, or aggregation flows. Let Q_1 and Q_2 be two classes, and let M be a member. Two flow operations on M, one in Q_1 and the other in Q_2, form a *define-use pair* if

1. the operation on M in Q_1 is a D_x;
2. the operation on M in Q_2 is a U_x; and
3. there exists a flow path from Q_1 to Q_2, along which there is no D_x on M.

In an inheritance flow, there are two kinds of define-use pairs, a *signature define-use* (DU_{si}) pair and a *body define-use* (DU_{bi}) pair, according to the operations of signature-inheritance and body-inheritance. A define-use pair formed by the operations of an association flow is called an *association define-use* (DU_{as}) pair while that formed by the operations of an aggregation flow is called an *aggregation define-use* (DU_{ag}) pair. The corresponding characteristics of these define-use pairs in CRG can be deduced as shown in the following corollaries .

Corollary 4.1: Let q_1 and q_2 be two classes, and let m be a member. (m, q_1, q_2) is a DU_{si} pair if the following are true:

1. $x.t_{si} = D_{si}$ and $m \rightarrow q_1 \in E_{pub} \cup E_{pro}$;
2. $\exists\, m'$, $m' \in V_m \cup V_a$, such that $m' \xrightarrow[Epub \cup Epro \cup Epri]{} q_2 \wedge m'.t_{si} = U_{si}$, and the signatures of m' and m are identical; and
3. $q_1 \xrightarrow[IHF]{} q_2$ and $\forall\, \alpha, \alpha \in V_c$ and $q_1 \xrightarrow[IHF]{} \alpha \wedge \alpha \xrightarrow[IHF]{} q_2$, there is no vertex m'', $m'' \in V_m \cup V_a$, such that $m'' \xrightarrow[Epub \cup Epro \cup Epri]{} \alpha \wedge m''.t_{si} = D_{si}$ and m'''s signature and m's are indistinguishable.

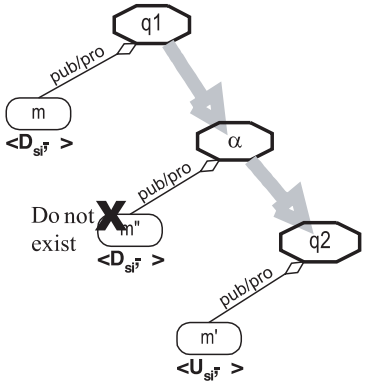

Fig. 8. A signature define-use pair.

Corollary 4.1 indicates that there are three conditions in a CRG for a signature define-use pair of member m in the class q_1 and class q_2. The first condition is that a D_{si} exists on the member m in the class q_1, and that m can be inherited by q_1's subclasses. The second is that a U_{si} exists on m' in q_2. Since the signatures of m' and m are identical; i.e., the operation on method m is a U_{si}, the result is that m's signature in class q_1 is not redefined before being inherited by class q_2 along an inheritance flow path. If m'' exists, the signature of m in q_1 is overridden by m'''s signature. Fig. 8 illustrates the above description.

The conditions for a DU_{bi} pair in a CRG are given in Corollary 4.2. The first condition is that a D_{bi} exists on member m in class q_1, and that m can be inherited by q_1's subclasses. The second is that a U_{bi} exists on member m in class q_2. That is, class q_2 may own member m without any explicit declaration. The final result is that m's body in class q_1 is not redefined before being inherited by class q_2 along an inheritance flow path. If m'' exists, then the body of m inherited by class q_2 is not the one specified in class q_1. The corollary is depicted by Fig. 9.

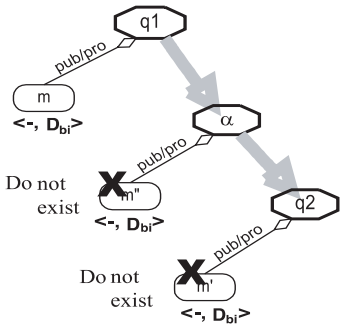

Fig. 9. A body define-use pair.

Corollary 4.2: Let q_1 and q_2 be two classes, and let m be a member. (m, q_1, q_2) is a DU_{bi} pair if the following are true:

1. $x.t_{bi} = D_{bi}$ and $m \rightarrow q_1 \in E_{pub} \cup E_{pro}$;
2. there is no vertex m', $m' \in V_m \cup V_a$, such that $m' \xrightarrow[Epub \cup Epro \cup Epri]{} q_2 \wedge m'.t_{bi} = D_{bi}$, and m''s signature and m's are identical or indistinguishable; and
3. $q_1 \xrightarrow[IHF]{} q_2$ and $\forall\, \alpha, \alpha \in V_c$ and $q_1 \xrightarrow[IHF]{} \alpha \wedge \alpha \xrightarrow[IHF]{} q_2$, there is no vertex m'', $m'' \in V_m \cup V_a$, such that $m'' \xrightarrow[Epub \cup Epro \cup Epri]{} \alpha \wedge m''.t_{bi} = D_{bi}$, and m'''s signature and m's are identical or indistinguishable.

In the CRG of Program I (see Fig. 5), (funC(), FunPack, C1) is a DU_{si} pair, and (stateC, C0, C1) is a DU_{bi} pair. Note that a member associated with U_{si} or U_{bi} in a class does not mean that it is invoked (or accessed) in the class. Therefore, the define-use pairs can be used to compute whether a class possesses the signatures and bodies of inherited members from superclasses.

For an association flow, a define-use pair can be regarded as the invocation relation between a sender and a receiver of a message. The define-use pairs include direct and indirect method invocations or attribute accesses. Since an association flow does not involved interfaces, these define-use pairs exist between classes. For a member, only one occurrence of a D_{as} exists along an association flow path.

Corollary 4.3: Let q_1 and q_2 be two classes, and let m be a member. (m, q_1, q_2) is a DU_{as} pair if the followings are true:

1. $m.t_{bi} = D_{bi}$ and $m \rightarrow q_1 \in E_{pub} \cup E_{pro} \cup E_{pri}$;
2. $\exists\, \delta, \delta \in V_m$, such that $m \xrightarrow[Em]{} \delta \wedge \delta \xrightarrow[Epub \cup Epro \cup Epri]{} q_2$; and
3. $q_1 \xrightarrow[ASF]{} q_2$.

Corollary 4.3 shows the conditions for an association define-use pair. The first condition is that the class q_1 may own member m, namely, a D_{as} on m in q_1. The second is that class q_2 has some method δ containing a message to invoke/access method m; i.e., a U_{as} is on m in q_2. The final result is that there is an association flow path from class q_1 to class q_2. The overriding of method m does not have to be considered in an association flow path. Fig. 10 depicts a DU_{as} of member m in class q_1 and class q_2. In the CRG of Program I, (setS(), S, C2) is an association define-use pair since setS() defined in S might be invoked by method setup() in C2.

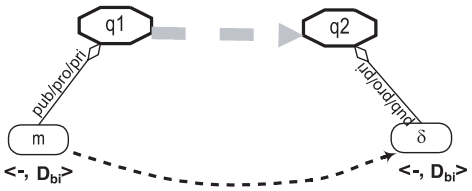

Fig. 10. An association define-use pair.

For an aggregation flow, a define-use pair is a special whole-part relation between a class and its encapsulated members. An aggregation define-use pair implies that a member of a class encapsulated inside another class is accessible by the latter class. The encapsulation might be across multiple encapsulation layers along an aggregation flow path.

Corollary 4.4: Let q_1 and q_2 be two classes, and let m be a member. (m, q_1, q_2) is a DU_{ag} pair if the following are true:
1. $m.t_{bi} = D_{bi}$ and $m \rightarrow q_1 \in E_{pub}$;
2. $\exists\ \delta, \delta \in V_a$, such that; $q_1 \xrightarrow[E_l \cup Epub]{} \delta \wedge \delta \xrightarrow[Epub \cup Epro \cup Epri]{} q_2$ and
3. $q_1 \xrightarrow[AGF]{} q_2$.

Corollary 4.4 shows the conditions for an aggregation define-use pair in a CRG. The first condition is that class q_1 owns member m, i.e., a D_{ag} on m in q_1, and member m must be accessible from outside of class q_1; i.e., m is a public member of class q_1. The second is that class q_1's instance must be encapsulated within some attribute δ of class q_2. The third is that there is an aggregation flow path from class q_1 to class q_2. A D_{ag} on m along an aggregation flow path occurs only once. It is not necessary to consider whether member m will be overridden along the flow path. The define-use pair is illustrated in Fig. 11. In the CRG of Program I, a DU_{ag} pair, for example, is the attribute array in classes Rec and C1.

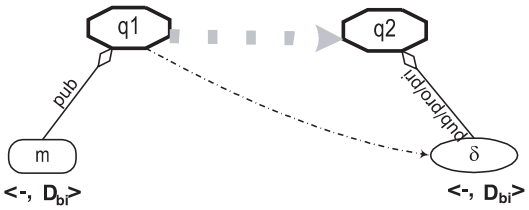

Fig. 11. An aggregation define-use pair.

4.2 Hybrid Class Relationships

Association flow with inheritance

The members propagated via an inheritance flow may introduce implicit association relationships between classes. In other words, one class can associate with another without any association flow path between them. Such an association can be achieved by means of inherited members or polymorphism. For example, class B in Fig. 12 owns method mA(E) from class A. Method mA(E) contains the message shape.draw() that invokes the method draw(), whose receiver's class is E. Class E inherits the method draw() from class D. Therefore, class B has an association relationship with class E.

```
class A{
    public void mA(E shape){ shape.draw(); }
}
class B extends A
     ...
}
class D{
    public void draw(){ ... }
}
class E extends D
     ...
}
```

Fig. 12. Program II.

To model an association flow in a class hierarchy, inheritance relationships have to be taken into consideration. An association flow path from one class to another in a CRG is restated by Definition 4.2. The flow path may include inheritance edges, besides the membership and member access edges considered in Definition 3.6.

Definition 4.2: Let q_1 and q_2 be two class vertices in a CRG. A flow path from vertex q_1 to vertex q_2 is an *association flow path*, denoted as $q_1 \xrightarrow[ASF]{} q_2$, iff one of the following holds:

1. $\exists\ \delta, \delta \in V_m \cup V_a$, and $\exists\ \chi, \chi \in V_m$, such that $\delta \xrightarrow[Epub \cup Epro \cup Epri \cup Eext]{} q_1 \wedge \delta \xrightarrow[Em]{} \chi \wedge \chi \xrightarrow[Epub \cup Epro \cup Epri \cup Eext]{} q_2$; or
2. $\exists\ \alpha, \alpha \in V_c, \exists\ \delta', \delta' \in V_m \cup V_a$, and $\exists\ \chi', \chi' \in V_m$, such that $\delta' \xrightarrow[Epub \cup Epro \cup Epri \cup Eext]{} q_1$, $\delta' \xrightarrow[Em]{} \chi' \wedge \chi' \xrightarrow[Epub \cup Epro \cup Epri \cup Eext]{} \alpha$, and $\alpha \xrightarrow[ASF]{} q_2$.

This definition is extended from Definition 3.6. The base condition involves the association relationships incurred by inherited members as Fig. 13 shows. Thus, the association relationship of classes B and E mentioned above can be modeled as an association flow path from E and B, $E \xrightarrow[ASF]{} B$. The flow path shown as the dashed bold arrow in Fig. 14 is formed by "draw() $\xrightarrow[Epub]{} D \xrightarrow[Eext]{} E$" and "draw() $\xrightarrow[Em]{}$ mA(E) $\xrightarrow[Epub]{} A \xrightarrow[Eext]{} B$".

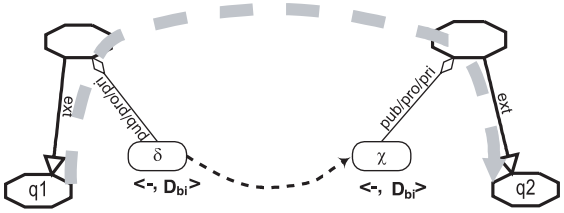

Fig. 13. Ann association flow path with inheritance

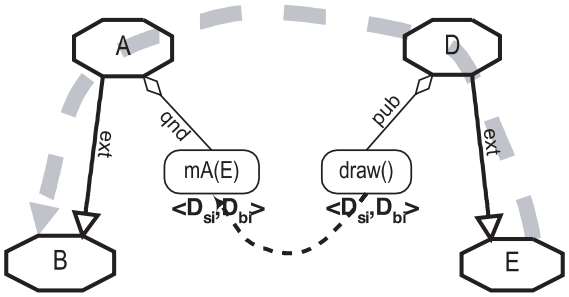

Fig. 14. An association flow path from E to B.

A D_{as} on a member in a class means that the class owns the member. The member can be defined by the class or inherited from the class's superclass. A D_{as} in a CRG can be stated as follows.

Corollary 4.5: Let q be a class and m be a member. The association flow operation on m is a D_{as} in q iff one of the following conditions holds:

1. $m.t_{bi} = D_{bi}$ and $m \rightarrow q \in E_{pub} \cup E_{pro} \cup E_{pri}$; or
2. $\exists\ \alpha$, $\alpha \in V_c$, such that (m, α, q) is a DU_{bi} pair.

In Corollary 4.5, the first condition implies that member m is specified within class q while the second imp;ies that there exists one superclass α of q defining m inherited by class q. For example, the association flow operation on draw() in class E (see Program II) is a D_{as} because (draw(), D, E) is a DU_{bi} pair.

A U_{as} on a member in a class is caused by the method that sends a message to access the member within the class. The method can be defined within the class or inherited from other class. Such a U_{as} in a CRG can be found as follows.

Corollary 4.6: Let q be a class and m be a member. The association flow operation on m is a U_{as} in q iff one of the following conditions holds:

1. $\exists\ \delta$, $\delta \in V_m$, such that $m \xrightarrow[Em]{} \delta \wedge \delta \xrightarrow[Epub \cup Epro \cup Epri]{} q$; or
2. $\exists\ \alpha$, $\alpha \in V_c$, and $\exists \chi$, $\chi \in V_m$, such that $m \xrightarrow[Em]{} \chi$, and ($\chi$, α, q) is a DU_{bi} pair.

The corollary above shows the conditions for a U_{as} on a member. The first condition implies that a method δ in class q contains a message to invoke m. The second one implies that there exists a class α, from which class q inherits a method χ containing a message to invoke m. For example, a U_{as} is on draw() in class B (see Program II), since draw() $\xrightarrow[Em]{}$ mA (E) and (mA(E), A, B) is a DU_{bi} pair.

Aggregation flow with inheritance

An inheritance flow may also introduce an implicit aggregation flow such that one class may aggregate another without any aggregation flow path to the latter. For example, the public attribute array in class Rec (see Program I) is accessible in class C1 because Rec's instance is encapsulated as nameList in C1. Class C2 inherits nameList from C1 and

is allowed to access array defined in Rec. C2 is aggregated in class F (see Program III in Fig. 15), so the public members in C2 might also be accessible in F. Therefore, F may aggregate Rec via aggregation and inheritance relationships.

```
class F
    public C2 objC2 = new C2();
        // class C2 is defined in Program I
        ...
}
```

Fig. 15. Program III.

Due to inheritance relationships, the flow path between classes for aggregation flow has to be redefined as follows.

Definition 4.3: Let q_1 and q_2 be two class vertices in a CRG. A flow path from vertex q_1 to vertex q_2 is an *aggregation flow path*, denoted as $q_1 \xrightarrow[AGF]{} q_2$, iff one of the following holds:

1. $\exists\ \delta, \delta \in V_m \cup V_a$, such that $q_1 \xrightarrow[E_l \cup Eext]{} \delta \wedge \delta \xrightarrow[Epub \cup Epro \cup Epri \cup Eext]{} q_2$; or
2. $\exists\ \alpha, \alpha \in V_c$, and $\exists\ \delta', \delta' \in V_m \cup V_a$, such that $\alpha \xrightarrow[E_l \cup Eext]{} \delta' \wedge \delta' \xrightarrow[Epub \cup Epro \cup Epri \cup Eext]{} q_2$, and $\alpha \xrightarrow[AGF]{} q_2$.

This definition is recursive, and the base condition involves the aggregated classes via inheritance edges besides membership and declaration edges in Definition 3.8. Such a case is shown in Fig. 16. For example, the aggregation of classes F and Rec in Program III can be an association flow path from Rec and F. The flow path, $\mathrm{Rec} \xrightarrow[AGF]{} F$, is formed by "$\mathrm{Rec} \xrightarrow[E_l]{} \mathrm{nameList} \xrightarrow[Epub]{} C1 \xrightarrow[Eext]{} C2$" and "$C2 \xrightarrow[E_l]{} objC2 \xrightarrow[Epub]{} F$". The dotted bold arrow in Fig. 17 shows the flow path.

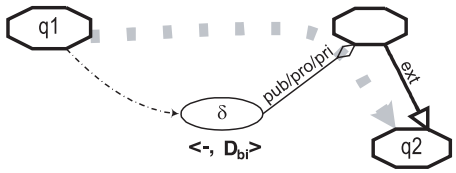

Fig. 16. An aggregation flow path with inheritance.

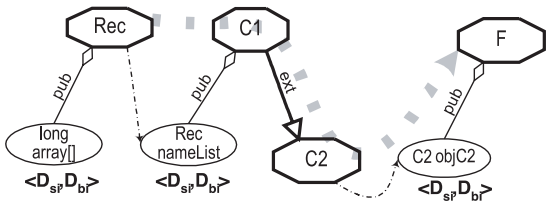

Fig. 17. An aggregation flow path from Rec to F.

According to Definitions 3.5 and 3.7, the meaning of D_{as} is the same as that of D_{ag}. Therefore, the necessary conditions in Corollary 4.5 can be applied to a D_{ag} on a member in a class. The necessary conditions for U_{ag} with an inheritance flow in a CRG is shown in Corollary 4.7.

Corollary 4.7: Let q be a class and m be a member. The aggregation flow operation on m is a U_{ag} in q iff one of the following conditions holds:

1. $\exists\ \delta$, $\delta \in V_a$, such that $m \xrightarrow[E_I \cup Epub]{} \delta \wedge \delta \xrightarrow[Epub \cup Epro \cup Epri]{} q$; or
2. $\exists\ \alpha$, $\alpha \in V_c$, and $\exists \chi$, $\chi \in V_a$, such that $m \xrightarrow[E_I \cup Epub]{} \chi \wedge \chi \xrightarrow[Epub \cup Epro \cup Epri]{} q$ and (χ, α, q) is a DU_{bi} pair.

This corollary shows the necessary conditions for a U_{ag} on a member. The first condition is that there is an attribute δ of class q, whose class encapsulates m as a public member. That is, m is accessible within class q. The second one is that there exists a class α, from which class q inherits an attribute χ whose class as a public member. For example, a U_{ag} is an array in class F (see Program III), because (draw(), D, E) is a DU_{bi} pair.

4.3 Flow Information

For a class, the information propagated via class relationships can be divided into *input, generated,* and *output flows*. The input flow of a class that includes the members is from the prior classes in the flow paths. The generated flow of a class is referred to as the newly defined or redefined members in the class. The output flow of a class subsumes the input and generated flows that can be propagated to the immediate post classes in the flow paths. In a CRG, we can define the flow information of a class with respect to inheritance, association, and aggregation flows as follows.

The signature-inheritance flow information of a class is stated in Definition 4.4. The input flow includes the member signatures defined in superclasses or superinterfaces that are inherited by the class. The generated flow denotes the newly defined or redefined signatures in the class. The output flow involves the signatures, including those from superclasses, of the class that can be inherited by subclasses.

Definition 4.4: In an inheritance flow, the input, generated, and output signature-inheritance flows of a class q are defined as $SIF_{in}(q)$, $SIF_{gen}(q)$, and $SIF_{out}(q)$, where

$SIF_{in}(q) = \{(m, \alpha) \mid (m, \alpha, q)$ is a DU_{si} pair$\}$;
$SIF_{gen}(q) = \{(m, q) \mid m \in V_m \cup V_a, m.t_{si} = D_{si}$, and $m \rightarrow q \in E_{pub} \cup E_{pro} \cup E_{pro}\}$; and
$SIF_{out}(q) = SIF_{in}(q) \cup SIF_{gen}(q) - \{(m, q) \mid (m, q) \in SIF_{gen}(q)$ and $m \rightarrow q \in E_{pri}\}$.

For class C2 in Program I, SIF_{in}(C2) = {(funC(), FunPack), (stateC, C0), (nameList, C1)}, and SIF_{gen}(C2) = {(objS, C2), (setup(), C2)}. Then, SIF_{out}(C2) = {(funC(), FunPack), (stateC, C0), (nameList, C1), (setup(), C2)}.

The definition of the input, generated, and output flows for body-inheritance in Definition 4.5 is the similar to that for signature-inheritance. Only their corresponding flow operations and define-use pairs are different. This definition is shown as follows.

Definition 4.5: In an inheritance flow, the *input*, *generated*, and *output* body-inheritance flows of a class q are defined as $BIF_{in}(q)$, $BIF_{gen}(q)$, and $BIF_{out}(q)$, where

$BIF_{in}(q) = \{(m, \alpha) \mid (m, \alpha, q)$ is a DU_{bi} pair$\}$;
$BIF_{gen}(q) = \{(m, q) \mid m \in V_m \cup V_a, m.t_{bi} = D_{bi}$, and $m \rightarrow q \in E_{pub} \cup E_{pro} \cup E_{pri}\}$; and
$BIF_{out}(q) = BIF_{in}(q) \cup BIF_{gen}(q) - \{(m, q) \mid (m, q) \in BIF_{gen}(q)$ and $m \rightarrow q \in E_{pri}\}$.

In Program I, the body-inheritance flows of C2 are BIF_{in}(C2) = {(funC(), C1), (stateC, C0), (nameList, C1)}, BIF_{gen}(C2) = {(objS, C2), (setup(), C2)}, and BIF_{out}(C2) = {(funC(), C1), (stateC, C0), (nameList, C1), (setup(), C2)}.

The association flow of a class can be defined as in Definition 4.6. The input flow of class q, $ASF_{in}(q)$, is a set of member-class pairs, (m, α). Each of the pairs denotes that member m in class α will be invoked directly or indirectly by a message within q. The generated flow of class q includes not only the members specified in q, but also those inherited from q's superclasses. That is, there is a D_{as} on each of these members in q. The output flow involves the member-class pairs (m, α), where member m of class α can be invoked (or accessed) directly or indirectly by some message from the outside (scope) of q. These pairs are a subset of the union of $ASF_{gen}(q)$ and $ASF_{in}(q)$.

Definition 4.6: In an association flow, the *input*, *generated*, and *output* flows of a class q are defined as $ASF_{in}(q)$, $ASF_{gen}(q)$, and $ASF_{out}(q)$, where

$ASF_{in}(q) = \{(m, \alpha) \mid (m, \alpha, q)$ is a DU_{as} pair$\}$;
$ASF_{gen}(q) = \{(m, q) \mid m \in V_m \cup V_a$, and either (i) $m.t_{bi} = D_{bi}$ and $m \rightarrow q \in E_{pub} \cup E_{pro} \cup E_{pri}$, or (ii) $\exists\ \alpha, \alpha \in V_c$, such that (m, α, q) is a DU_{bi} pair$\}$; and
$ASF_{out}(q) = \{(m, \alpha) \mid (m, \alpha) \in ASF_{gen}(q) \cup ASF_{in}(q)$, and either (i) $m \xrightarrow[Epub \cup Epro \cup Eext]{} q$ or (ii) $\exists\ \delta, \delta \in V_m \wedge \delta \xrightarrow[Epub \cup Epro \cup Eext]{} q$, such that $m \xrightarrow[Em]{} \delta\}$.

For example, the association flows of class C2 in Program I are ASF_{in}(C2) = {(setS(), S), (stateS, S)}, ASF_{gen}(C2) = {(objS, C2), (setup(), C2), (stateC, C0), (funC(), C1), (nameList, C1)}, and ASF_{out}(C2) = {(setS(), S), (stateS, S), (objS, C2), (setup(), C2), (stateC, C0), (funC(), C1), (nameList, C1)}.

In an aggregation flow, the flow information of a class can be defined as in Definition 4.5. The input flow of class q includes the members that are encapsulated in other classes and are accessible in q. The generated flow of class q includes the members specified in q and those inherited from q's superclasses. The aggregation flow operation on each of these members in q is a D_{ag}. The output flow of class q involves the members in the input and generated flows that are accessible outside of q.

Definition 4.7: In an aggregation flow, the *input*, *generated*, and *output* flows of a class q can be defined as $AGF_{in}(q)$, $AGF_{gen}(q)$, and $AGF_{out}(q)$, where

$AGF_{in}(q) = \{(m, \alpha) \mid (m, \alpha, q)$ is a DU_{ag} pair$\}$,
$AGF_{gen}(q) = \{(m, q) \mid m \in V_m \cup V_a$, and either (i) $m.t_{bi} = D_{bi}$ and $m \rightarrow q \in E_{pub} \cup E_{pro} \cup E_{pri}$, or (ii) $\exists\ \alpha, \alpha \in V_c$, such that (m, α, q) is a DU_{bi} pair$\}$; and
$AGFout(q) = \{(m, \alpha) \mid (m, \alpha) \in ASF_{gen}(q) \cup ASF_{in}(q)$, and either (i) $m \xrightarrow[Epub \cup Eext]{} q$ or (ii) $\exists\ \delta, \delta \in V_a \wedge \delta \xrightarrow[Epub \cup Eext]{} q$, such that $m \xrightarrow[Epub \cup El \cup Eext]{} \delta\}$.

For example, the aggregation flows of class C2 in Program I are AGF_{in}(C2) = {(array, Rec), (stateC, C0), (nameLlist, C1), (funC(), C1)}, AGF_{gen}(C2) = {(objC2, C2), (setup(), C2)}, and AGF_{out}(C2) = {(array, Rec), (stateC, C0), (nameLlist, C1), (funC(), C1), (setup(), C2)}. Note that access of attribute objC2 from outside of C2 is not allowed, so (objC2, C2) is not included in AGF_{out}(C2).

4.4 Flow Computation

Since an OO program can be represented as a CRG, computing the flow information of the program can be performed on the CRG. For a given class, the steps in computing its flow information can be derived from the definitions given in subsection 4.3.

In an inheritance flow, Algorithm 4.1 shows how to compute the input signature-inheritance flow for a given class c, i.e., $SIF_{in}(c)$. The algorithm performs a breadth-first traversal from *c* backward along the edges of E_{ext} and E_{imp}, i.e., ascending to *c*'s superclasses or superinterfaces. The traversal is controlled by a queue *work_list*, in which a vertex can be stored and popped with first-in-first-out order using the *enqueue* and *dequeue* methods. The traversal finds the member signatures and classes (or interfaces) that form DU_{si} pairs with class *c*. Because an inheritance flow path in a CRG never forms a cycle, the algorithm stops after all the superclasses of *c* have been examined. The output, *SIF_IN*, is *SIFin*(*c*).

Algorithm 4.1 *Computing_SIF_IN*(c, G_{CRG})
Input: (c, G_{CRG}), $c \in V_c \cup V_i$ and G_{CRG} is a CRG
Output: SIF_IN /* Input signature-inheritance flow of c */
Begin
SIF_IN := ϕ;
work_list.enqueue (c); /* Initialize the value of work_list to be 'c' */
while work_list is not empty **do**
v := work_list.dequeue (); /* Pop a class or interface vertex v from work_list */
for each s, $s \in V_c \cup V_i \wedge s \rightarrow v \in E_{ext} \cup E_{imp}$, **do**
for each m, $m \in V_m \cup V_a \wedge m.t_{si} = D_{si} \wedge m \rightarrow s \in E_{pub} \cup E_{pro}$, **do**
if (there is no (x, q), (x, q) $\in$ SIF_IN, such that the signatures of x and m are *indistinguishable*)
then
SIF_IN := SIF_IN $\cup$ {(m, s)}; /* Signature DU pair (m, s, c) */
endif
endfor
work_list.enqueue(s); /* Store c's superclass s in work_list */
endfor
endwhile
output SIF_IN /* $SIF_{in}(c)$ */
End.

In Algorithm 4.1, each vertex is visited at most once, and the visited method and attribute vertices are compared with the elements in *SIF_IN*. Let $|V|$ denote the number of vertices in a CRG. The time complexity of the algorithm is $O(|V|^2)$ for the worst case.

The algorithm used to compute the input flow information for the body-inheritance of a class is similar to that for signature-inheritance. The difference is that the bodies of inherited members are from superclasses only. That is, it is not necessary to visit interface vertices. Algorithm 4.2 shows the computation of the input body-inheritance flow of a class. The time complexity of this algorithm for the worst case is also $O(|V|^2)$.

Algorithm 4.2 *Computing_BIF_IN*(c, G_{CRG})
Input: (c, G_{CRG}), $c \in V_c$ and G_{CRG} is a CRG
Output: BIF_IN /* Input signature-inheritance flow of c */
Begin
BIF_IN := ϕ;
work_list.enqueue (c); /* Initialize the value of work_list to be 'c' */
while work_list is not empty **do**
v := work_list.dequeue (); /* Pop a class or interface vertex v from work_list */
for each s, $s \in V_c \wedge s \rightarrow v \in E_{ext}$ **do**
for each m, $m \in V_m \cup V_a \wedge m.t_{si} = D_{si} \wedge m \rightarrow s \in E_{pub} \cup E_{pro}$, **do**
if (there is no (x, q), (x, q) $\in$ BIF_IN, such that the signatures of x and m are *indistinguishable*)
then
BIF_IN := BIF_IN $\cup$ {(m, s)}; /* Signature DU pair (m, s, c) */
endif
endfor
work_list.enqueue(s); /* Store c's superclass s in work_list */
endfor
endwhile
output BIF_IN /* $BIF_{in}(c)$ */
End.

The computations between the generated flows for signature-inheritance and body-inheritance are similar. Algorithm 4.3 shows how the generated body-inheritance flow is computed for a given class c, i.e., $BIF_{gen}(c)$. The computation is done vertex by vertex, where the vertex whose t_{bi} tag is D_{bi} and which has an E_{pub}, E_{pro}, or E_{pri} edge to c is included in *BIF_GEN*. The complexity of this algorithm is $O(|V|)$ for the worst case.

Algorithm 4.3 *Computing_BIF_GEN*(c, G_{CRG})
Input: (c, G_{CRG}), $c \in V_c$ and G_{CRG} is a CRG
Output: BIF_GEN
Begin
BIF_GEN := ϕ;
for each m, $m \in V_m \cup V_a \wedge m.t_{si} = D_{si} \wedge m \rightarrow c \in E_{pub} \cup E_{pro} \cup E_{pri}$, **do**
BIF_GEN := BIF_GEN $\cup$ {(m, c)};
endfor
output BIF_GEN /* $BIF_{gen}(c)$ */
End.

According to Definitions 4.4 and 4.5, the output flows for signature-inheritance and body-inheritance of a class c, $SIF_{out}(c)$ and $BIF_{out}(c)$, can be obtained from c's input flow and generated flow. These output flows can be computed by excluding the private members from the union of the input and generated flows corresponding to signature/body-inheritance.

The input flows of association and aggregation for a class c, $ASF_{in}(c)$ and $AGF_{in}(c)$, can be computed by backward traversing the corresponding flow paths from c in a CRG. The traversal finds the members and classes that form DU_{as}/DU_{ag} pairs with c. As explained in Subsection 4.2, some association and aggregation flows may be introduced by inherited members of a class. To get such association and aggregation flows, we can apply Algorithm 4.2 (by invoking *Computing_BIH_IN*()) to find the inherited members of a class. The detailed steps in computing the input association and aggregation flows of a class are shown in Algorithms 4.4 and 4.5, respectively. In these algorithms, the statement *Computing_BIH_GEN*() is used to invoke Algorithm 4.3 to get the members specified within a class.

Algorithm 4.4 *Computing_ASF_IN*(c, G_{CRG})
Input: (c, G_{CRG}), $c \in V_c$ and G_{CRG} is a CRG
Output: ASF_IN
Begin
ASF_IN := ϕ;
BIH_IN := *Computing_BIH_IN*(c, G_{CRG}); /* call *Computing_BIH_IN*(c, G_{CRG}) */
BIH_GEN := *Computing_BIH_GEN*(c, G_{CRG}); /* call *Computing_BIH_GEN*(c, G_{CRG}) */
use_set := {$m \mid \exists\, q$, $q \in V_c$, such that $(m, q) \in$ BIH_IN $\cup$ BIH_GEN};
for each v, $v \xrightarrow[ASF]{} c$, **do**
  BIH_IN := *Computing_BIH_IN*(v, G_{CRG}); /* call *Computing_BIH_IN*(v, G_{CRG}) */
  BIH_GEN := *Computing_BIH_GEN*(v, G_{CRG}); /* call *Computing_BIH_GEN*(v, G_{CRG}) */
  def_set := BIH_IN $\cup$ BIH_GEN;
  for each (x, s), (x, s) $\in$ def_set **do**
    if ($\exists\, m$, $m \in$ use_set, **and** $x \xrightarrow[Em]{} m$) **then**
      ASF_IN := ASF_IN $\cup$ {(x, s)};
    endif
  endfor
endfor
output ASF_IN /* $ASF_{in}(c)$ */
End.

In Algorithm 4.4, the worst case is that all incoming E_m edges of each vertex have to be traversed once when *Computing_BIH_IN*() is not considered. Let $|E|$ denote the number of vertices in a CRG. The complexity for the worst case is $O(|V| \times |E|)$. In Algorithm 4.5, each vertex is visited at most once in computing the input aggregation flow of a class, excluding the invocation of *Computing_BIH_IN*(). Hence, the complexity of the algorithm for the worst case without considering inheritance is $O(|V|)$.

Algorithm 4.5 *Computing_AGF_IN*(c, G_{CRG})
Input: (c, G_{CRG}), $c \in V_c$ and G_{CRG} is a CRG
Output: ASF_IN

```
Begin
 ASF_IN := ϕ;
 BIH_IN := Computing_BIH_IN(c, G_CRG);    /* call Computing_BIH_IN(c, G_CRG) */
 BIH_GEN := Computing_BIH_GEN(c, G_CRG); /* call Computing_BIH_GEN(c, G_CRG) */
 work_set := {q | ∃ (a, z), (a, z) ∈ BIH_IN ∪ BIH_GEN, such that q → a ∈ E_l};
 done_set := ϕ;                                  /*  a set of checked classes */
 while work_set ≠ ϕ do
   select a vertex v from work_set;
   work_set := work_set − {v};
   done_set := done_set ∪ {v};
   BIH_IN := Computing_BIH_IN(v, G_CRG);    /* call Computing_BIH_IN(v, G_CRG) */
   BIH_GEN := Computing_BIH_GEN(v, G_CRG); /* call Computing_BIH_GEN(v, G_CRG) */
  def_set := BIH_IN ∪ BIH_GEN;
   for each (x, s) in def_set do
        if (x → s ∈ E_pub) then
            AGF_IN := AGF_IN ( {(x, s)};
        endif
        if (x ∈ V_a) then
             work_set := work_set ∪ {r | r → x ∈ E_l and r ∉ done_set };
        endif
   endfor
 endwhile
 output AGF_IN  /* AGF_in(c) */
End.
```

The generated association and aggregation flows for a class are similar since their definitions are identical (see Definitions 4.6 and 4.7). Hence, computing the two generated flows is done to get all the members, including those specified in the class and those inherited from superclasses. From the input and generated flows for association and aggregation of a class c, we can obtain the corresponding output flows of c, $ASF_{out}(c)$ and $AGF_{out}(c)$, in accordance with Definitions 4.6 and 4.7.

These algorithms show that the analysis of class relationships can be reduced to the graph reachability problem. They are not optimal but are used to show how our flow information in an OO program can be computed.

5. APPLICATIONS

The flow model can provide the flow analysis of class relationships for program understanding, anomaly detection, and program testing.

5.1 Program Understanding

When reusing a class, a programmer often needs to understand not only the class, but also its relations with another class [24]. Thus, it is necessary to navigate in a class library to find the relations. Inheritance flow information can help one understand what implicit members a class owns and a class hierarchy [9]. The input flow of a class in an association

flow can help one identify that the members defined in other classes might be invoked or accessed by the class. For a complicated composition class, its available members from other classes can be obtained by computing the input flow via aggregation relationships.

The flow analysis tool of this flow model for program understanding is realized in the Microsoft Windows® 95/NT environment. The display and computation of class relationship flows were developed in C++ [25]. To implement the source code scanner, parser, and CRG constructor for Java programs, we employ the tools *flex* and *bison* (both are shareware developed by GNU™). Fig. 18 shows a hierarchical view of Program I. A user can select a class to perform inheritance, association, and aggregation analysis. The result of analysis is displayed in the window in the middle of Fig. 19. The analyzer helps the user understand class libraries by providing flow analysis of class relationships.

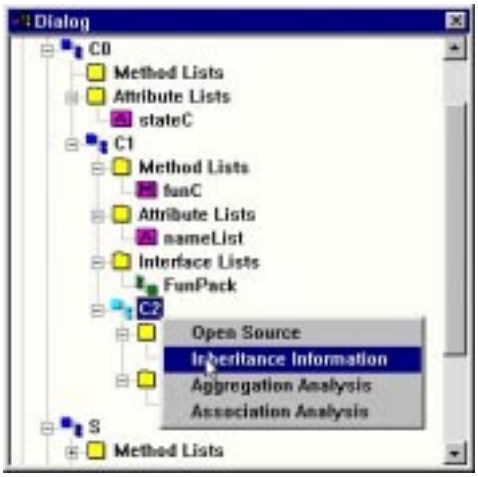

Fig. 18. Hierarchical view of classes.

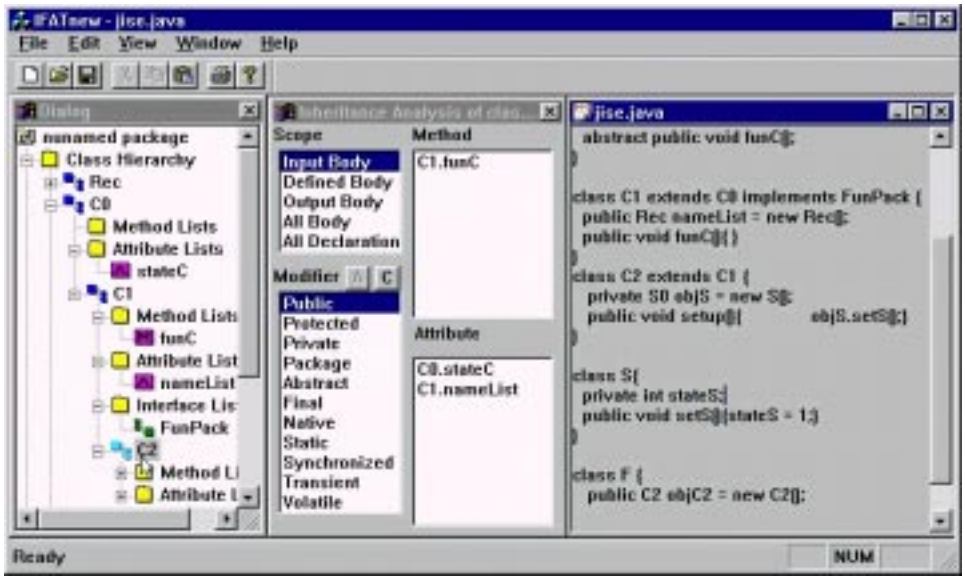

Fig. 19. Flow analyzer of class relationships for Java Programs.

5.2 Anomaly Detection

An anomaly in a program is often an indication of the existence of a programming error or an inappropriate design. The flow operation sequences in this class relationship flow model can be used to detect anomalies in OO programs, such as method interface conflicts and unimplemented methods [26]. These anomalies can be detected as in traditional data flow anomaly detection [27].

A member propagated along a flow path, from class q_1, class q_2, ... to class q_k, can be regarded as a sequence of flow operations, op_1, op_2, ..., op_k. A method interface conflict occurs when a superclass introduces a new method while one of its subclasses has previously introduced a method with the same name. The new method is overridden and, therefore, can not be inherited by subclasses. In this flow model, we can detect the conflict by finding a sequence of signature-inheritance flow operations that contains '$D_{si}D_{si}$'. An unimplemented method occurs when a class inherits an abstract method but does not define the method's body. This can be detected from a sequence of body-inheritance flow operations that contains '$N_{bi}U_{bi}$'. In addition, flow operation sequences of association and aggregation might help to determine whether the reusing approach, by means of inheritance or composition, is appropriate for existing components.

5.3 Program Testing

Before doing regression testing, it is very important to identify the potentially affected parts to be re-tested when a program is modified. The potentially affected parts can be bound by ripple effect analysis with respect to the modification [3, 18]. In order to reduce the cost of testing, the re-tested parts should be as small as possible. The more precise ripple effect analysis is, the smaller the re-tested parts are. The approach proposed in [3] uses class relationships to identify affected classes. This approach can be deemed as finding the classes that can be reached from a modified class via flow paths in a CRG. Our flow model can improve the precision of their approach by using define-use relations.

```
class S1{
    public void m1(){};
 }
class S2{
    public void m2(){};
     public void n2 (S1 objS1) { objS1.m1()};
 }
class S3{
      public void m3(S2 objS2){ objS2.m2()};
 }
```

Fig. 20. Program IV.

For example, classes S1, S2, and S3 are defined in Fig. 20, and their CRG is shown in Fig. 21. Assume that class S1 is modified, but that the relationships among the three classes are unchanged. According to the analysis in [3], the classes that need to be re-tested are S1, S2, and S3 because there is an association flow path from S1 to S3. In fact, the modification in S1 does not affect S3 via S2. This can be detected by our flow analysis since there is no DU_{as} and DU_{ag} pair between S1 and S3. Hence, only classes S1 and S2 need to be re-tested.

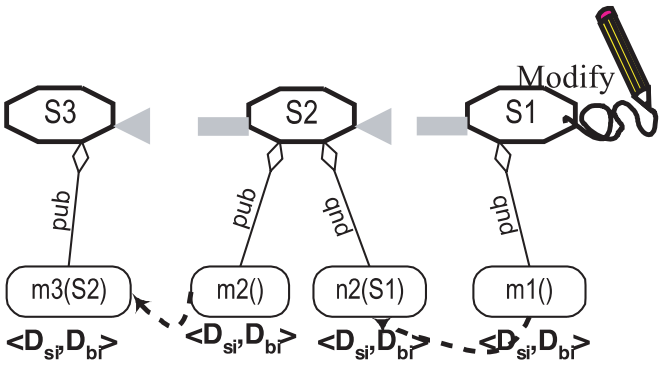

Fig. 21. The CRG of Program IV.

During structural testing (also called white-box testing), a sufficient number of test cases should be fed to a target program in order to satisfy some degree of a coverage criterion [28]. The criterion is defined on a model that represents a program, e.g., all-execution paths or all-branches in the control flow model. Thus, coverage criteria are an important factors affecting test case generation for structural testing. Many OO program testing techniques, e.g., [23, 29-32], focus on a single class and lack proper criteria for inter-classes. In the class relationship flow model, the define-use pairs can indicate inter-class testing criteria, for example, all-flow-paths, all-define-use-pairs, etc.

6. CONCLUSIONS AND FUTURE WORK

In this paper, the class relationship flow model, consisting of inheritance, association, and aggregation flows, has been proposed to analyze class libraries. With respect to these flows, each member within a class is associated with an operation to represent whether its status is defined or used. The concealed dependencies propagated along class relationships can be represented as a sequence of flow operations along a flow path. By representing a program as a class relationship graph, the flow analysis can be reduced to the graph reachability problem.

The interpretation of OO features can vary with program constructs in different languages, such as inheritance rules and object representation. Although Java programs were used for demonstration purposes in this paper, this flow model can be tailored to fit specific OO languages with minor modification for the interpretation of OO features. In addition, this flow model can give a user the ability to interpret behavior evolution, message passing, and object encapsulation in programs as sequences of flow operations. These operation sequences could be applied in various fields of OO software engineering, such as program understanding, anomaly detection, complexity measurement, and program testing.

Currently, we are improving the efficiency of the flow computation algorithms for the whole set of classes in a program. In the future, we plan to develop testing and maintenance tools based on this model, and plan to embed them within an integrated visual-programming environment [33].

ACKNOWLEDGEMENT

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

REFERENCES

1. D. L. Metayer, "Program analysis for software engineering: new applications, new requirements, new tools," *ACM SIGPLAN Notices*, Vol. 32, No. 1, 1997, pp. 86-88.
2. S. Meyers and M. Klaus, "Examining C++ program analyzers," *Dr. Dobb's Journal*, No. 262, 1997, pp. 68-75.
3. D. C. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "On regression testing of object-oriented programs," *Journal of Object-Oriented Programming*, Vol. 8, No. 2, 1995, pp. 51-65.
4. G. Booch, *Object-oriented Analysis and Design with Applications*, Redwood City: Benjamin/Cummings, 1994.
5. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, NJ, 1991.
6. J. L. Chen and F. J. Wang, "Encapsulation in object-oriented programs," *ACM SIGPLAN Notices*, Vol. 31, No. 7, 1996, pp. 30-32.
7. M. Lejter, S. Meyers, and S. P. Reiss, "Support for maintaining object-oriented programs," *IEEE Transactions on Software Engineering*, Vol. 18, No. 12, 1992, pp. 1045-1052.
8. P. K. Linos and V. Courtois, "A tool for understanding object-oriented program dependencies," in *Proceedings of IEEE Third Workshop on Program Comprehension*, 1994, pp. 20-27.
9. J. L. Chen and F. J. Wang, "An inheritance flow model for class hierarchy analysis," *Information Processing Letters*, Vol. 66, No. 6, 1998, pp. 309-315.
10. M. S. Hecht, *Flow Analysis of Computer Programs*, Elsevier North-Holland, 1977.
11. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley, Reading, Mass, 1996.
12. J. L. Chen, F. J. Wang, and Y. L. Chen, "An object-oriented dependency graph for program slicing," in *Proceedings of the 24th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 24)*, 1997, pp. 147-156.
13. F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Communications of the ACM*, Vol. 19, No. 3, 1976, pp. 137-147.
14. S. S. Muchnick and N. D. Jones, *Program Flow Analysis: Theory and Applications*, Prentice-Hall Inc., 1981.
15. J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, 1987, pp. 319-349.
16. S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," in *Proceedings of the 14th International Conference on Software Engineering*, 1992, pp. 392-411.
17. M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, 1984, pp. 352-357.

18. S. S. Yau and S. S. Liu, *Some Approaches to Logical Ripple-effect Analysis*, Software Engineering Research Center, SERC-TR-24F, University of Florida, 1988.
19. M. Sudholt and C. Steigner, "On interprocedural data flow analysis for object oriented languages," *Lecture Notes in Computer Science*, Vol. 641, 1992, pp. 156-162.
20. K. D. Cooper and K. Kennedy, "Interprocedural side-effect analysis in linear time," in *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988, pp. 57-66.
21. S. Subramanian, W. T. Tsai, and S. H. Kirani, "Hierarchical data flow analysis for O-O programs," *Journal of OOP*, Vol. 7, No. 2, 1994, pp. 36-46.
22. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers – Principles, Techniques, and Tools*, Addison-Wesley, 1986.
23. M..J. Harrold and G. Rothermel, "Performing data flow testing on classes," in *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering of ACM*, 1994, pp. 154-163.
24. J. L. Chen, F. J. Wang, and Y. L. Chen, "The development of a flow analysis tool for software reuse," in *Proceedings of the Third Symposium on Computer & Communication Technology*, 1997, pp. 200-207.
25. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, MA, second edition, 1991.
26. P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt, "Reuse contracts: managing the evolution of reusable assets," in *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications of ACM*, 1996, pp. 268-285.
27. J. C. Huang, "Detection of data flow anomaly through program instrumentation," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 3, 1979, pp. 226-236.
28. S. C. Ntafos, "A comparison of some structural testing strategies," *IEEE Transaction on Software Engineering*, Vol. 14, No. 6, 1988, pp. 868-874.
29. R. K. Doong and P. G. Frankl, "Case studies on testing object-oriented programs," in *Proceedings of the ACM SIGSOFT '91 Fourth Symposium on Software Testing, Analysis, and Verification (TAV4)*, 1991, pp. 165-177.
30. A. S. Parrish, R. B. Borie, and D. W. Cordes, "Automated flow graph-based testing of object-oriented software modules," *Journal of System and Software*, Vol. 23, No. 2, 1993, pp. 95-109.
31. M. D. Smith and D. J. Robson, "A framework for testing object-oriented programs," *Journal of Object-Oriented Programming*, Vol. 5, No. 3, 1992, pp. 45-53.
32. C. D. Turner and D. J. Robson, "The state-based testing of object-oriented programs," in *Proceedings of IEEE Conference on Software Maintenance*, 1993, pp. 302-310.
33. C. H. Hu and F. J. Wang, "Constructing an integrated visual programming environment," *Software – Practice and Experience*, 1998, to appear.

Feng-Jian Wang (ÍõÝÃ×°¼) graduated from National Taiwan University, Taipei, Taiwan, R.O.C. in 1980. He received his M.S. and Ph.D. degree in E.E.C.S. from Northwestern University, U.S.A., in 1986 and 1988. He is currently a Professor in the Department of Computer Science and Information Engineering, National Chiao Tung University. His interests include Software Engineering, Compiler, Object-Oriented Techniques, and Distributed System Software and Internet.

Jiun-Liang Chen (³¯«Ï¨}) was born in Taipei, Taiwan, R.O.C., in 1969. He received his B.S. degree in Computer Science & Information Engineering from National Chiao Tung University, R.O.C., in 1992, and Ph.D. degree in Computer Science & Information Engineering from National Chiao Tung University, R.O.C., in 1998. He has been a student member of ACM, IEEE, USENIX, and R.O.C. OOTSIG since 1994. His research interests include object-oriented programming, program analysis, distributed systems, software testing, and software maintenance.