Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

High-Level Flow Model for Anomaly Detection in Object-Oriented Programs

Jiun-Liang Chen Feng-Jian Wang

Department of Computer Science and Information Engineering
National Chiao Tung University
HsinChu, Taiwan, R.O.C.

Abstract

In this paper, a High-Level Flow model representing flow information of an object-oriented (OO) program for data flow analysis is presented. This model introduces to conventional data flow additional properties which are made by object abstraction, encapsulation, inheritance, and polymorphism. In an OO program, an object encapsulates both component data (attributes) and operational functions (methods). The model employs method path expression to represent the flow information of a method, where a method path expression is a set of path expressions of which each describes the accesses of an attribute in the method. The flow information of an OO program can be derived by concatenating method path expressions related. Due to OO features, there are two additional types of data flow anomalies, a message-sequence anomaly and a class-definition anomaly, described in this paper. These anomalies can help indicate programming errors for debugging an OO program. The algorithms of detecting these anomalies can be implemented efficiently with bit-pattern computation.

Keywords: data flow analysis, anomaly detection, object orientation.

1. Introduction

Recently, object-oriented (OO) paradigms, associated with class libraries with high-level modularity, reusability, and extendibility, have been widely applied for developing software [2]. Comparing with conventional programs, OO programs are introduced with the features of object abstraction, encapsulation, inheritance, and polymorphism; they use different languages and thus program structures from conventional ones. In past decades, a number of program analysis techniques, such as anomaly detection [9, 10], program dependency graphs [4, 8], and program slicing [16], have been developed based on data flow analysis [1, 12]. These techniques are useful for testing, debugging, and maintaining conventional programs. However, most of the analysis models they used seem insufficient for OO programs since the models lack OO features.

A data flow anomaly is often an indication of the existence of a programming error. The detection of anomalies might help users to debug a program and improve the quality of a program [5]. In an OO program, an object encapsulates both component data (attributes) and operational functions (methods). When receiving a message, an object achieves its responsibility by invoking a method operating on attributes. Rather than a variable associated with a data operation, a message passed to an object results in a sequence of operations on the object's attributes according to the invoked method defined in the object's class. The OO features embedded in a program make the data flow analysis complicated.

In the paper, we propose a High-Level Flow (HLF) model which introduces OO features into a conventional data flow for program analysis. The HLF model employs a method path expression to represent the flow information of a method in a class. A method path expression consists of a set of path expressions [5], of which each is a regular expression describing the sequences of data flow operations on an attribute. When an object receives messages, its flow information can be obtained by computing the method path expressions of invoked methods. Based on conventional data flow anomalies, two additional types of anomalies, a message-sequence anomaly and a class-definition anomaly, are specified. These anomalies can help users to examine an appropriate invocation sequence of methods, and indicate programming errors in a class. The algorithms for efficiently detecting these anomalies were developed. They can be implemented with bit-pattern computation.

The rest of this paper is organized as following: Section 2 reviews the related work of data flow analysis. The HLF model for OO programs is presented in Section 3. Then, Section 4 illustrates the anomaly detection in OO programs with the HLF model. Section 5 draws a conclusion and suggests future work.

2. Related Work

Data flow analysis is a technique to ascertain and collect information about the *define*, *use*, and *kill* operations on variables in a program [1, 12]. Various applications of data flow analysis can be found in [11]. The data flow information is the basis for the traditional program analysis techniques such as program dependency graphs [4, 8], program slicing [16], ripple effect analysis [17], and so on. In an OO program, data are not obvious since they, as well as procedures, are encapsulated in an object. One can not apply traditional data flow analysis to an OO program.

Sudholt and Steigner in [14] extended an interprocedural data flow analysis algorithm for OO languages. In their approach, one first decomposes an object into a set of procedures and global variables in detail (i.e., primitive statement and variables).

He/she can then apply Cooper's and Kennedy's algorithm (see [3]) on these procedures and variables to perform traditional data flow analysis. This approach was emphasized on low-level data flow information for compilers. It is used in the field of compiler optimization and parallelization, but might be not apt to detect high-level anomalies for debugging OO programs.

Hierarchical Data Flow Analysis, HDFA, was proposed by [15] for analyzing data flow information in OO programs. The hierarchical data flow consists of three layers: classes, objects, and attributes. The HDFA exploits data flow information by identifying class flow, object flow, and attribute flow. The operations, *kill*, *define*, and *use*, are defined for each layer of flows. The attribute flow is similar to traditional data flow analysis [7]. The class flow and the object flow can be used to analyze anomalous message sequences. The flow information to which the class flow and the object flow contribute is restricted because the two flows are abstracted from the attribute flow. In addition, HDFA does not investigate the data flow introduced by some OO features, such as inheritance and polymorphism.

3. High-Level Flow Model

We now present the High-Level Flow (HLF) model that introduces OO features into the data flow of an OO program. In an OO program, an object may encapsulate other objects as its attributes. The most primitive form of an object is a data item (such as an integer or a char) which can not be decomposed further. The HLF model defines the flow for simple and complex objects separately. In the HLF model, an object is a *simple* object if it is nothing but a data item; it is a *complex* object otherwise. That is, a complex object contains at least one object inside.

3.1 Flow Model for Simple Object

A simple object in the HLF model is similar to the traditional data flow model [5]. To access a simple object is to modify or to reference the value of the object. There are two kinds of operations to access a simple object: a *data-define* operation and a *data-use* operation. When accessing a simple object, a data-define operation (d) modifies the value of the object, while a data-use operation (u) references its value without modification.

The flow of information of a simple object in a program is a sequence of operations working on the object during execution. The operations working on an object include d, u, and l: d is a data-define operation, u is a data-use operation, and l is a null operation (no operation working the object). The sequence of operations can be represented as a *path expression*, a regular expression of the operations on a simple object along all the possible execution paths of a program [5].

Definition 3.1 Let x be a simple object, and F be a program which accesses object x. The flow information of object x in program F, denoted as $P(F; x)$, is defined as the path expression of object x in program F.

Each string in $P(F; x)$ represents the sequence of operations working on object x in one execution path of program F. For example, a simple object x is accessed in the following program.

```
foo(int x, int y){
   if ( x > 0 )          // use x
      x = y;             // define x, use y
   else y = x;           // define y, use x
}
```

Within program *foo()*, the sequence of operations working on object x is either ud or uu. Note that initiating a formal parameter with an actual parameter is not regarded as a data-define operation in this model. Since the initialization is done by a compiler, it does not represent the intention of a programmer/debugger to define a formal parameter. The path expression of x in *foo()* is

$P(foo; x) = ud + uu \equiv u(d + u)$, where symbol + means 'or'.

Flow information may help detect potential errors in a program such as an abnormal usage of an object (a data flow anomaly). The abnormal usage of a simple object can be defined in terms of path expressions as following:

Definition 3.2 Let x be a simple object and have its life time within a program F. The usage of x in F is said to be *abnormal* if $P(F; x) \equiv u\rho + \rho'$, $\rho dd\rho' + \rho''$ or $\rho d + \rho'$, where ρ, ρ', and ρ'' are arbitrary path expressions.

In the definition, $u\rho$ means that an object is used before being defined, $\rho dd\rho'$ means that an object is consecutively defined twice, and ρd means that an object is defined but not used.

3.2 Flow Model for Complex Object

The flow information of a simple object is the basis of the flow model for a complex object, since the latter is composed of simple objects. In light of object abstraction, message passing scheme, inheritance, and polymorphism, here we explore the flow information of a complex object by means of its component objects.

Object abstraction

A class is the abstraction of objects of the same kind. Attributes and methods defined in a class represent the state and behavior of an object. In other words, the flow information of a complex object can be defined from the context of its defining class. The structure of a class is defined as:

Definition 3.3 The structure of a class, C, is denoted as $\{A_1, A_2, ..., A_m, M_1, M_2, ..., M_n\}$ where A_i, $1 \leq i \leq m$, is an attribute (object) of class C; M_j, $1 \leq j \leq n$, is a method of class C with M_1 and M_n, the constructor and destructor of class C respectively.

To simplify the discussion, attributes A_1, A_2, ..., and A_m in Definition 3.3 are regarded as simple objects through this paper.

The flow information in a class involves the sequences of operations working on the objects in its method(s). The flow information of object x in method M_j can be described by a path expression, $P(M_j; x)$. To represent the flow information within a method, we define a method path expression, the union of path expressions in the method for the objects accessed.

Definition 3.4 Let M_j be a method of class C, and $P(M_j; x)$ denote x's path expression in M_j. M_j's method path expression, $MP(M_j)$, is defined as $MP(M_j) \equiv \{P(M_j; x) \mid$

object x is a component object in class C}

In Definition 3.4, object x can be M_j's local variable, or the attribute in class C. The life time of local variables of a (instance) method is the same as that of the method. To the method owner, these variables are (and their corresponding objects may be) killed when their method execution completes. For inter-method flow analysis of an object, only the attributes are thus considered in a method path expression. A method path expression for a method invoking other methods can be obtained by expanding the invoked methods in the invoking method (which is similar to interprocedural data flow analysis [11]).

With object abstraction, an *instance*-type property defined in a class, such as an instance attribute or an instance method, belongs to an individual object. A *class*-type property, such as a class attribute or a class method, is common to all objects of a same class. To distinguish the properties between instance-type and class-type, we attach an object identifier to the method path expression of an instance method, but not to that of a class method. Similarly, we attach an object identifier to the path expression of an instance attribute, but not to that of a class attribute.

Definition 3.5 If object O is instantiated from class C, then O's method path expression of instance method M_j is denoted as MP($O.M_j$), and the path expression of instance attribute A_i in method M_j is denoted as P($O.M_j$; $O.A_i$).

Each attribute or method encapsulated within an object is associated with an access specifier, *private* or *public*, to indicate the scope of access. The access of a private attribute is under the scope of an owner object, while that of a public attribute is not. Similarly, a private method can be invoked only in the scope of its owner object, but the invocation of a public method is not restricted.

Message passing scheme

When a sequence of messages are passed to an object (a receiver), the flow information of the receiver can be described by concatenating the corresponding method path expressions. Let a sequence of messages passed to object O invoke methods $O.M_{j_1}$, $O.M_{j_2}$, ..., and $O.M_{j_k}$ sequentially. The flow information of object O can be represented by MP($O.M_{j_1}$) MP($O.M_{j_2}$) ... MP($O.M_{j_k}$). MP($O.M_{j_1}$) MP($O.M_{j_2}$) ... MP($O.M_{j_k}$) contains a set of the path expressions of O's component objects accessed in $O.M_{j_1}$, $O.M_{j_2}$, ..., and $O.M_{j_k}$ sequentially.

Definition 3.6 Let a sequence of messages invoke methods M_{j_1}, M_{j_2}, ..., and M_{j_k} of object O. O's flow information for the sequence of messages, denoted as MP($O.M_{j_1}$) MP($O.M_{j_2}$) ... MP($O.M_{j_k}$), is {P($O.M_{j_1}$, $O.M_{j_2}$, ..., $O.M_{j_k}$; x) | object x is O's component object}, where P($O.M_{j_1}$, $O.M_{j_2}$, ..., $O.M_{j_k}$; x) is equal to P($O.M_{j_1}$; x) P($O.M_{j_2}$; x)...P($O.M_{j_k}$; x).

The method path expressions of an object can also represent a number of message sequences passed to the object like a regular expression. For example, given an instance, O, of class C, the method path expressions representing the flow information for all possible message sequences received by object O is denoted as MP($O.M_1$) (MP($O.M_{j_1}$) + MP($O.M_{j_2}$) + ... + MP($O.M_{j_k}$))* MP($O.M_n$), where methods M_{j_1}, M_{j_2}, ..., and M_{j_k} are O's public methods, symbol * denotes repeating zero or more times for a method, and symbol + denotes operator 'or'. Each method invocation sequence begins with constructor $O.M_1$, and ends with destructor $O.M_n$.

Inheritance

The inherited properties, attributes and methods, of a class are defined in its superclass(es). To obtain the flow information of the inherited attributes, one has to consider the properties of superclasses. The method path expressions of a class's constructor/destructor include the path expressions of the inherited attributes in the constructor/destructor of its superclass, respectively. This can be expressed as below. To simplify the discussion, if object x is a class-type attribute, its flow information includes the path expressions in the methods of all class B's instances that access x.

Definition 3.7 Let class C be a subclass of class B. The method path expressions of class C's constructor and destructor are

MP(M_1) ≡ {P(M_{B_c}, M_1; x) | object x is an inherited attribute from class B} ∪ {P(M_1; x) | object x is a component object in class C}, and

MP(M_n) ≡ {P(M_n, M_{B_d}; x) | object x is an inherited attribute from class B} ∪ {P(M_n; x) | object x is a component object in class C}.

Here, M_{B_c} and M_{B_d} represent the constructor and destructor of class B invoked by class C, respectively.

Polymorphism

A virtual method may have multiple implementations, of which each is defined in different classes of an inheritance hierarchy. The uncertainty about which implementation will be invoked by a virtual method call arises from the existence of polymorphic object references. It is determined by the class to which the object reference is bound to at run-time. Definition 3.8 shows that all of the multiple implementations have to be taken into consideration for a virtual method call.

Definition 3.8 Let B be a super class with a virtual method m, and C_1, C_2, ..., C_p be the subclasses of B with their implementations, m^1, m^2, ..., and m^p, of m. Let O be an object reference of class B. When $O.m$ is invoked, its method path expression includes the method path expressions of m^1, m^2, ..., and m^p. That is, MP($O.m$) ≡ MP($O.m^1$)+MP($O.m^2$)+...+ MP($O.m^p$).

3.3 An Example

To demonstrate the HLF model, we give an example shown in Figure 1. The program in Figure 1 is the definition of class *Stack* written in C++ [Stro91]. There are six methods, *Stack* (constructor), *~Stack* (destructor), *push*, *pop*, *full*, and *empty*, and three attributes, *size*, *top* and *s*, in class *Stack*. In this example, attribute *s*, an array, is assumed to be a simple object. The method path expressions of class *Stack*'s methods are illustrated in Table 2. Note that the sequence of operations working on attribute *s* in statement *delete* at line 10 is treated as *du* for simplifying the discussion of anomaly analysis.

```
/* Definition of a class Stack */
line #   code
1.  class Stack (
2.  private:
3.    int size, top;
4.    char *s;
5.  public:
6.    Stack(int &sz){
7.      size=sz;              //sz:u; size:d;
8.      s= new char[size];    //s:d; size:u;
9.      top = 0;  };          //top:d;
10.   ~Stack() { delete s;};//s:du;
11.   void push(char &c)
12.     s[top]= c;           //c:u; top:u; s:d;
13.     top++; };            //top:ud;
14.   char pop(){
15.     top--;              //top:ud;
16.     return s[top]; };   //top:u; s:u;
17.   int empty()
18.     if (top == 0) return 1;//top:u;
19.     else return 0; };
20.   int full() {
21.     if(size==top+1) return 1;//top:u; size:u;
22.     else return 0; };
23. }
```

Figure 1. An example of a class `Stack`

Table1. The method path expression for methods defined in class `Stack`

Method	*Method path expression*	*Path expression*
Stack	{P(Stack;size), P(Stack;s), P(Stack; top)}	P(Stack;size)=*du*; P(Stack;s)=*d*; P(Stack;top)=*d*
~Stack	{P(~Stack;size), P(~Stack; s), P(~Stack; top)}	P(~Stack;size)=*l*; P(~Stack; s)=*du*; P(~Stack; top)=*l*
push	{P(push; size), P(push; s), P(push; top)}	P(push; size)=*l*; P(push; s)=*d*, P(push; top)=*uud*
pop	{P(pop;size), P(pop;s),P(pop; top)}	P(pop; size)=*l*; P(pop;s)=*u*, P(pop; top)=*udu*
empty	{P(empty;size), P(empty;s), P(empty; top) }	P(empty;size)=*l*; P(empty;s)=*l*; P(empty; top)=*u*
full	{P(full; size), P(full; s), P(full; top)}	P(full; size)=*u*; P(full; s)=*l*; P(full; top)=*u*

4. Anomaly Detection

An anomaly in a program might indicate the existence of a programming error. With the HLF model, we can perform anomaly detection through the flow information of an OO program. The anomaly for an object concerned here is an abnormal usage of its attribute. For a complex object, two operations that make an abnormal usage of its attribute is called *intra-method anomaly* if they are within a method. If the operations are in two methods of the object respectively, the abnormal usage is called *inter-method anomaly.* An intra-method anomaly can be detected by analyzing the method path expression of an invoked method, like traditional data flow anomaly detection [9, 10, 3]. Detecting an inter-method anomaly for an object is more complicated because it depends on the sequences of received messages. This section focuses on the inter-method anomaly analysis and detection.

4.1 Anomaly Analysis

A complex object has a data-flow anomaly when its attribute has an abnormal usage along one path from construction to destruction through some methods. In a complex object, the number of execution paths could be very large since the sequence of messages to invoke its public methods is not restricted. In general, only some of the paths are significant to programmer. Due to this, a data-flow anomaly is weak to indicate a programming error. By extending the traditional data-flow anomaly, we define an *inter-method data-flow anomaly,* also called an *inter-method anomaly* for short, for anomaly analysis. An inter-method anomaly exists in two public methods with respect to a private attributes when one of the followings succeeds: (1) There is a path ending with a data-define operation in the first method, and each path in the second method begins with a data-define operation. (2) The first method has a path ending with a data-define operation, and the second method is the destructor whose paths either begin with a data-define operation, or are null operations. (3) The first method is the constructor whose paths contain no data-define operation, and the second method has a path beginning with a data-use operation. These statements can be formally represented as the following definition.

Definition 4.1 Let a class $C = \{(A_1, A_2, ..., A_m, M_1, M_2, ..., M_n) \mid A_1, A_2, ..., A_m$ are private attributes, $M_2, M_3, ..., M_{n-1}$ are public methods, and M_1 and M_n are the constructor and destructor respectively.$\}$. Two public methods, M_j and M_k, of class C have an *inter-method anomaly* with respect to an attribute A_i if one of the following cases is satisfied:

(1) $P(M_j; A_i) \equiv (\rho d + \rho')$ for some j, $1 \le j \le n$-1, and $P(M_k; A_i) \equiv d\rho''$ for some k, $2 \le k \le n$-1,

(2) $P(M_j; A_i) \equiv (\rho d + \rho')$ for some j, $1 \le j \le n$-1, and $P(M_k; A_i) \equiv (d\rho'' + 1)$ for $k = n$, or

(3) $P(M_j; A_i) \equiv u^*$ for $j = 1$, and $P(M_k; A_i) \equiv (u\rho + \rho')$ for some k, $2 \le k \le n$.

In Definition 4.1, ρ, ρ', and ρ'' are arbitrary path expressions. The first two cases of the definition imply that a useless data-define operation (the value defined by the operations is never referred to) always exists in the first method when the two methods are invoked consecutively. The third implies that an illegal data-use operation (which is not preceded by any data-define operation) always exists in the second method. Although an inter-method anomaly can evidently to indicate a programming error, it does not tell what incurs the error yet. (An abnormal usage of an object within a complex object can be incurred by a wrong sequences of message passing, or erroneous definition of the object's class.) Hence, two new types of anomalies, a *message-sequence anomaly* and a *class-definition anomaly,* based on the inter-method anomaly are defined for detecting a wrong sequence of message passing and an erroneous class definition.

A message-sequence anomaly exists in two consecutively public methods (which could be identical) if they has an inter-method anomaly with respect to all private attributes which they access.

Definition 4.2 Let M_j and M_k be two public methods of class C, and $A_s, A_{s+1}, ..., A_t$ be the private attributes accessed in M_j and M_k. Two methods, M_j and M_k, have a *message-sequence anomaly* if $\forall\ i,\ s \le i \le t,\ A_i$ such that M_j and M_k have an *inter-method anomaly* with respect to A_i

In accordance with Definition 4.2, we can summarize the following property.

Property 4.1 Given two messages passed to an object, the sequence of the two messages might be *inappropriate* if the corresponding invoked methods of the object have a message-sequence anomaly.

Two methods with a message-sequence anomaly imply that all accessed attribute in the two methods are associated with a useless data-define or an illegal data-use operation. Because of the operation, to invoke the two methods consecutively could be an error very possibly. Thereby, one can use this property to examine whether one method is appropriately invoked before the other.

To indicate a programming error in the definition of a class, we define another type anomaly, class-definition anomaly.

Definition 4.3 Let a class $C = \{(A_1, A_2, ..., A_m, M_1, M_2, ..., M_n) \mid A_1, A_2, ..., A_m$ are private attributes, $M_2, M_3,.., M_{n-1}$ are public methods, and M_1 and M_n are the constructor and destructor respectively.$\}$. Class C contains a *class-definition anomaly* in a method M_j if there exists a private attribute A_i such that one of the following cases holds:

(1) If $j \neq n$, then for all k, $2 \le k \le n$, M_j and M_k have an inter-method anomaly with respect to A_i.
(2) If $j \neq 1$, then for all k, $1 \le k \le n$-1, M_k and M_j have an inter-method anomaly with respect to A_i.

Observing the definition above, one can obtain the following property:

Property 4.2 A programming error might exist in a method of a class if the class contains a class-definition anomaly in the method.

In a class, a method with a class-definition anomaly implies that all methods are invoked inappropriately before (except the destructor) or after (except the constructor) the method. The execution of the method always comes with a useless data-define operation or an illegal data-use operation. It is obvious that a class-definition anomaly indicates a programming error more precise then that an inter-method anomaly does. For example, according to Definition 4.1, class *Stack* shown in Figure 1 contains several inter-method anomalies, such as P(*Stack, push, ~Stack*; *top*) = *duud*, P(*Stack, empty, ~Stack*; *s*) = *ddu*, and so on. These anomalies seem not harmful for the class, and these cases do not form a class-definition anomaly.

4.2 Anomaly Detection

In the HLF model, all the paths along which an attribute is accessed in a method are described by a path expression. It is impractical to traverse all possible paths in a method when detecting inter-method anomalies. For example, a loop in a path can be regarded as zero and two iterations for static data flow analysis [9]. Like exposed operations in [1], an inter-method anomaly is caused by exposable operations in a method. Reducing the loops and the operations that are not exposable in a method can simplify the inter-method anomaly detection.

To distinguish the exposable operations from the operations on an attribute in a method, we define three types of *exposable* operations.

Definition 4.4 Let m and a be a method and an attribute of a class, respectively. Let ρ and ρ' be arbitrary path expressions.

(1) A data-use operation, x, on a in m is an *upward-exposable use* (UEU) if $P(m; a) \equiv x\rho + \rho'$.
(2) A data-define operation, y, on a in m is a *downward-exposable define* (DED) if $P(m; a) \equiv \rho y u^* + \rho'$.
(3) A data-use operation, z, on a in m is a *downward-exposable use* (DEU) if $P(m; a) \equiv \rho d u^* z + \rho'$.

A UEU is the first operation on an attribute along one path in a method. A DED is the last data-define operation on an attribute along one path in a method. A DEU is the last operation that is preceded by a data-define operation on an attribute along one path in a method. According to Definitions 4.3, one path along which an attribute is accessed in a method contains at most three exposable operations: one for UEU, one for DED, and one for DEU, sequentially. A DEU always follows a DED because the path in which a DEU exists contains at least one data-define operation. Other operations are called *non-exposable*. The exposable flow information of a method can be defined by eliminating non-exposable operations from the flow information of the method.

The sequence of operations along a path after simplifying the loops [9] and eliminating the non-exposable operations is one of the following path expressions: *udu*, *ud*, *du*, *u*, *d*, and *l*. To facilitate the inter-method anomaly analysis, a path expression in a method is then reduced as a subset of (or equal to) {*udu*, *ud*, *du*, *u*, *d*, *l*}. In the set, symbol *u* which is the first of *udu*, *ud*, and *u* is a UEU, and that being the last of *udu* and *du* is a DEU.

According to Definition 4.1, an inter-method anomaly is determined in five kinds of path expressions, $\rho d+\rho'$, $d\rho''$, $d\rho''+l$, u^*, and $u\rho+\rho'$. $\rho d+\rho'$ means that there exists one string ending with d, a DED. $d\rho''$ denotes all strings beginning with d. It implies that there is a DED in each string of the path expression, and no UEU. In the reduced path expression of $d\rho''$, all elements still begin with d. $d\rho''+l$ is similar to $d\rho''$ and the reduced path expression contains one more element l. u^* can be regarded as $l+uu$. For the detection, uu has the same power as u. Both u^* and $u\rho+\rho'$ have a UEU. Let P'(M; x) denote the reduced path expression of P(M; x). For an object, when method M_k is invoked after method M_j, a inter-method anomaly, a useless data-define operation, in M_j can be detected if the first two cases in Definition 4.1 holds. The first case can also be detected with reduced path expression. Here, P'(M_j; A_i) contains d, ud, or both, and P'(M_k; A_i) is {du}, {d}, or {du, d}. The DED in P'(M_j; A_i) is a useless data-define operation because each element in P'(M_k; A_i) begins with a DED. The second case holds with P'(M_k; A_i) that may contain one more element l. An illegal data-use operation is detected when the third case occurs. Now, P'(M_j; A_i) is {u}, {l}, or {u, l}, and P'(M_k; A_i) contains at least one element in {udu, ud, u}. The UEU in P'(M_k; A_i) is an illegal data-use operation since there is no DED in P'(M_j; A_i). Therefore, a inter-method anomaly can be found with reduced path expressions.

The first case in Definition 4.1 can formally be described with reduced path expression as

$P'(M_j; A_i) \cap \{ud, d\} \neq \varnothing$ for $P(M_j; A_i) \equiv (\rho d + \rho')$, and
$P'(M_k; A_i) \cap \{udu, ud, u, l\} = \varnothing \wedge P'(M_k; A_i) \cap \{du, d\} \neq \varnothing$ for $P(M_k; A_i) \equiv d\rho''$.

The rest can be seen in Algorithm 4.1 which detects an inter-method anomaly between two methods. The result of Algorithm 4.1 is *d*, *u*, or *n*, where *d* denotes a useless data-define operation on the attribute, *u* denotes an illegal data-use operation, and *n* means no inter-method anomaly.

```
Algorithm 4.1  Inter-MethodAnomalyDetection
  Let $P'(M; x)$ denote the reduced path expression of $x$ in $M$,
    and $M_1$ and $M_n$ be the constructor and destructor of a class.
  Input $P_j$, $P_k$, Case: $P_j$ and $P_k$ two path expression methods,
    Case is a flag to indicate which case of detecting an inter-
    method anomaly to be applied.
  Output result: $d$ for a useless data-define operation; $u$ for
    an illegal data-use operation; $n$: for no anomaly.

  Begin
    result := $n$;
    if (Case = 1) then // case (1) in Definition 4.1
      if ($P_j \cap \{ud, d\} \neq \varnothing \wedge P_k \cap \{udu, ud, u, l\} = \varnothing \wedge$
          $P_k \cap \{du, d\} \neq \varnothing$) then
          result := $d$;
      endif
    endif
    if (Case = 2) then // case (2) in Definition 4.1
      if ($P_j \cap \{ud, d\} \neq \varnothing \wedge P_k \cap \{udu, ud, u\} = \varnothing \wedge$
          $P_k \cap \{du, d, l\} \neq \varnothing$) then
          result := $d$;
      endif
    endif
    if (Case = 3) then // case (3) in Definition 4.1
      if ($P_j \cap \{udu, ud, du, d\} = \varnothing \wedge P_j \cap \{u, l\} \neq \varnothing \wedge$
          $P_k \cap \{udu, ud, u\} \neq \varnothing$) then
          result := $u$;
      endif
    endif
    output result;
  End.
```

An algorithm to detect a message-sequence anomaly derived from Definition 4.2 is illustrated in Algorithm 4.2. For two given methods, according to the case of Definition 4.1 to which they belong, the reduced path expressions of each attribute in the two methods are examined by the inter-method anomaly detection. The result of this algorithm indicates whether the two methods have a message-sequence anomaly or not.

```
Algorithm 4.2 Message-SequenceAnomalyDetection
  Let $MP'(M)$ denote the reduced flow information of $M$,
    $P'(M; x)$ denote the reduced path expression of $x$ in $M$, and
    $M_1$ and $M_n$ be the constructor and destructor of a class.
  Input $M_j$, $M_k$: $M_j$ and $M_k$ are two methods.
  Output result: true for a message-sequence anomaly; false
    for none.

  Begin
    result := true;
    if ($M_j \neq M_n \wedge M_k \neq M_1$) then
      if ($M_k \neq M_n$) then // case (1) in Definition 4.1
        for each attribute $x$ such that $P'(M_j; x) \in$
            $MP'(M_j) \wedge P'(M_k; x) \in MP'(M_k)$ do
          if (Inter-MethodAnomalyDetection(
              $P'(M_j; A_i)$, $P'(M_k; A_i)$, 1) = $u$) then
              result := false;
          endif
        endfor
      else  // case (2) in Definition 4.1
        for each attribute $x$ such that $P'(M_j; x) \in$
            $MP'(M_j) \wedge P'(M_k; x) \in MP'(M_k)$ do
          if (Inter-MethodAnomalyDetection(
              $P'(M_j; A_i)$, $P'(M_k; A_i)$, 2) = $u$) then
              result := false;
          endif
        endfor
      endif
    endif
    if ($M_j = M_1 \wedge M_k \neq M_1$) then // case (3) in Definition 4.1
      for each attribute $x$ such that $P'(M_j; x) \in MP'(M_j) \wedge$
          $P'(M_k; x) \in MP'(M_k)$ do
        if (Inter-MethodAnomalyDetection(
            $P'(M_j; A_i)$, $P'(M_k; A_i)$, 3) = $u$) then
            result := false;
        endif
      endfor
    endif
    output result;
  End.
```

Based on Definition 4.3, we designed an algorithm for detecting class-definition anomalies in a class as below. The reduced path expressions of all attributes in any two methods of a class are examined for the inter-method anomaly detection. The result of the algorithm is a set of class-definition anomalies existing in a class. For example, (M_j, A_i, d) in the set denotes that a class-definition anomaly in method M_j with respect to attribute A_i is a useless data-define operation.

```
Algorithm 4.3  Class-DefinitionAnomalyDetection
  Let a class $C = \{(A_1, A_2, ..., A_m, M_1, M_2, ..., M_n) \mid A_1, A_2, ...,$
    $A_m$ are private attributes, $M_2, M_3, .., M_{n-1}$ are public
    methods, and $M_1$ and $M_n$ are the constructor and
    destructor respectively.}, and $P'(M; x)$ be a reduced path
    expression of $x$ in $M$.
  Input $C$: a class with its flow information modeled with the
    HLF model
  Output AnomalySet: $\{(M_j, x, a) \mid$ A class-definition
    anomaly in $M_j$ with respect to attribute $x$ is a useless data-
    define operation for $a=d$, or an illegal data-use operation
    for $a=u$.}
```

```
Begin
  AnomalySet = ∅;
  for $i$ := 1 to $m$ do
    for $j$ := 1 to $n$ do
      AnomalyFlag_D:= true;
      AnomalyFlag_U:= true;
      for $k$ := 1 to $n$ do
        if ($j \neq n \wedge 2 \leq k \leq n$-1) then
          if (Inter_MethodAnomalyDetection(
              P'($M_j$; $A_i$), P'($M_k$; $A_i$), 1) $\neq d$) then
              AnomalyFlag_D:= false;
          endif
        endif
        if ($j \neq n \wedge k = n$) then
          if (Inter_MethodAnomalyDetection(
              P'($M_j$; $A_i$), P'($M_k$; $A_i$), 2) $\neq d$) then
              AnomalyFlag_D:= false;
          endif
        endif
        if ($j \neq 1$ and $1 \leq k \leq n$-1) then
          if (Inter_MethodAnomalyDetection(
              P'($M_k$; $A_i$), P'($M_j$; $A_i$), 3) $\neq u$) then
              AnomalyFlag_U:= false;
          endif
        endif
      endfor
      if (AnomalyFlag_D) then
        AnomalySet := AnomalySet ∪ {($M_j$, $A_i$, $d$)};
      endif
      if (AnomalyFlag_U) then
        AnomalySet := AnomalySet ∪ {($M_j$, $A_i$, $u$)};
      endif
    endfor
  endfor
  output AnomalySet;
End.
```

Since the size of a reduced path expression will not increase as the number of path does, the inter-method anomaly detection algorithm can be implemented with cost-effective bit-pattern computation (see Appendix). To demonstrate the class-definition anomaly detection, we employ class *Stack* in Figure 1 with some modification shown in Figure 2 as an example. Suppose an error exists at line 21 in class *Stack* once the symbol '==' is mis-typed as '=' (see Figure 2). The path expression of attribute *size* in method *full()* is *d*. The reduced path expressions and the corresponding bit patterns are shown in Table 3 (see Appendix). After the execution of Algorithm 4.3, a class-definition anomaly set is generated, *AnomalySet* = {(*full*, *size*, *d*)}. As a result, a class-definition anomaly exists in method *full()* and indicates that there is a useless data-define operation on attribute *size*.

```
20.   int full(){
21.         if (size = top+1) return 1;//top:u; size:d;
22.         else return 0; };
```

Figure 2. An example of a class-definition anomaly

5. Conclusion and Future Work

In this paper, we have presented the High-Level Flow model for OO programs. The HLF model introduces additional flow information associated with object orientation, message passing, encapsulation, inheritance, and polymorphism features. Besides conventional data flow anomalies, two types of anomalies have been defined due to OO features. A message-sequence anomaly can help users to examine an appropriate method invocation sequence, whereas a class-definition anomaly can indicate a programming error in a class. The algorithms for detecting these anomalies can be implemented efficiently with bit-pattern computation. Currently, the HLF model has not considered alias and global variables yet, because these variables seldom appear in programs with well object-orientation. The HLF model will be extended with the use of *self* [6] or *this* [13] variable. In addition, we will apply these anomalies for OO program testing and debugging.

Acknowledgment

This research was partly sponsored by National Science Council, Taiwan, ROC, under contract No. NSC85-2213-E009-030.

References:

[1] F.E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," *Communications of the ACM*, vol. 19(3), pp.137~147, 1976.

[2] W. Berg, M. Cline, and M. Girou, "Lessons Learned from the OS/400 OO Project," *Communications of the ACM*, vol. 38(10), pp.54~64, 1995.

[3] K.D. Cooper and K. Kennedy, "Interprocedural Side-Effect Analysis in Linear Time," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp.57~66, 1988.

[4] 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, pp.319~349, July 1987.

[5] L.D. Fosdick and L.J. Osterweil, "Data Flow Analysis in Software Reliability," *ACM Computing Surveys*, vol. 8 (3), pp.305~330, Step. 1976.

[6] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its implementation*, Addison-Wesley, Reading, 1983.

[7] M.S. Hecht, *Flow Analysis of Computer Programs*, Elsevier North-Holland, Amsterdam, 1977.

[8] S. Horwitz and T. Reps, "The Use of Program Dependence Graphs in Software Engineering," *Proceedings of the 14th International Conference on Software Engineering*, pp.392~411, 1992.

[9] J.C. Huang, "Detection of Data Flow Anomaly Through Program Instrumentation," *IEEE Transactions on Software Engineering*, vol. SE-5(3), pp.226~236, May 1979.

[10] J. Jachner and V.K. Agarwal, "Data Flow Anomaly Detection," *IEEE Transactions on Software Engineering*, vol. SE-10(4), pp.432~437, 1984.

[11] S.S. Muchnick and N.D. Jones, *Program Flow analysis: Theory and Applications*, Prentice-Hall Inc., 1981.

[12] B.K. Rosen, "High-Level Data Flow Analysis," *Communications of the ACM*, vol. 20(10), pp.712~724, 1977.

[13] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, MA, second edition, 1991.

[14] M. Sudholt and C. Steigner, "On Interprocedural Data Flow Analysis for Object Oriented Languages," *Lecture Notes in Computer Science*, vol. 641, pp.156~162, 1992.

[15] S. Subramanian, W.T. Tsai, and S.H. Kirani, "Hierarchical Data Flow Analysis for O-O Programs," *Journal of OOP*, vol. 7(2), pp.36~46, 1994.

[16] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, vol. 10(4), pp.352~357, July 1984.

[17] S.S. Yau and S.S. Liu, *Some Approaches to Logical Ripple-effect Analysis*, Software Engineering Research Center, SERC-TR-24F, University of Florida, October 1988.

Appendix:

The inter-method anomaly detection algorithm can be implemented efficiently with bit-pattern computation. A reduced path expression contains at most six types of strings; it can be represented as a 6-bit pattern, where each bit of the pattern denotes the existence of one type of reduced path expression. The mapping of a bit to a path is shown in Table 2. For example, the reduced path expressions of Figure 2 corresponding 6-bit pattern forms are shown in Table 3.

With the mappings in Table 2, we illustrate Algorithm 4.1 with pseudo code as follows.

```
Inter-MethodAnomalyDetection($P_j$, $P_k$, A-Case)
Input $P_j$, $P_k$, A-Case: $P_j$ and $P_k$ are two reduced path
expressions of 6-bit pattern forms, and A-Case is a flag to
indicate which case of detecting an inter-method anomaly to
be applied.
Output AnomalyType: $d$ for a useless data-define operation;
$u$ for an illegal data-use operation; $n$ for none anomaly.
{
    AnomalyType := $n$;
    switch(A-Case)
    {
      case 1: // case (1) in Definition 4.1
        if (($P_j$&010010)$\neq$000000 $\wedge$ ($P_k$&110101)=000000
            $\wedge$ ($P_k$&001010)$\neq$000000)
            AnomalyType := $d$;
      break;
      case 2: // case (2) in Definition 4.1
        if (($P_j$&010010)$\neq$000000 $\wedge$ ($P_k$&110100)=000000
            $\wedge$ ($P_k$&001011)$\neq$000000)
            AnomalyType := $d$;
      break;
      case 3: // case (3) in Definition 4.1
        if (($P_j$&111010)=000000 $\wedge$ ($P_j$&000101) $\neq$ 000000
            $\wedge$ ($P_k$&110100)$\neq$000000)
            AnomalyType := $u$;
        break;
    }
    return AnomalyType;
}
```

In the pseudo code above, (P_j&010010) $\neq$ 000000 means that $P'(M_j;\ x) \cap \{ud,\ d\} \neq \varnothing$. Then, ($P_k$&110101) = 000000 means that $P'(M_k;\ x) \cap \{udu,\ ud,\ u,\ l\} = \varnothing$, whereas ($P_k$&001010) $\neq$ 000000 means that $P'(M_k;\ x) \cap \{du,\ d\} \neq \varnothing$. The other bit-pattern computation can be interpreted by similar way.

Table 2. The mapping of a 6-bit pattern to a reduced path expression

Bit Pattern	b_6		b_5		b_4		b_3		b_2		b_1	
Value	0	1	0	1	0	1	0	1	0	1	0	1
Reduced path exp	$\varnothing$	*udu*	$\varnothing$	*ud*	$\varnothing$	*du*	$\varnothing$	*u*	$\varnothing$	*d*	$\varnothing$	*l*

Table 3. Bit pattern for the reduced path expressions of class `Stack`

Attr.	*Reduced path exp*	$b_6b_5b_4b_3b_2b_1$
size	P'(Stack; size)=*du*	001000
	P'(push; size)=*l*	000001
	P'(pop; size)= *l*	000001
	P'(empty; size)= *l*	000001
	P'(full; size)=*d*	000010
	P'(~Stack; size)= *l*	000001
s	P'(Stack; s) = *d*	000010
	P'(push; s) = *d*	000010
	P'(pop; s) = *u*	000100
	P'(empty; s) = *l*	000001
	P'(full; s) = *l*	000001
	P'(~Stack; s) = *du*	001000
top	P'(Stack; top)=*d*	000010
	P'(push; top)=*ud*	010000
	P'(pop; top)=*udu*	100000
	P'(empty; top)=*u*	000100
	P'(full; top)=*u*	000100
	P'(~Stack; top)= *l*	000001