A Fault Model for Subtype Inheritance and Polymorphism *

Jeff Offutt
Department of Information and Software Engineering
George Mason University
Fairfax, VA 22030-4444 USA
ofut@ise.gmu.edu

Roger Alexander
Computer Science Department
Colorado State University
Fort Collins, CO 80523-2001 USA
rta@cs.colostate.edu

Ye Wu, Quansheng Xiao, Chuck Hutchinson
Department of Information and Software Engineering
George Mason University
Fairfax, VA 22030-4444 USA
{wuye,xiaoqs}@ise.gmu.edu, chuckhutchinson2@yahoo.com

The Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE '01), pages 84–95, Hong Kong, PRC, November 2001.

Abstract

Although program faults are widely studied, there are many aspects of faults that we still do not understand, particularly about OO software. In addition to the simple fact that one important goal during testing is to cause failures and thereby detect faults, a full understanding of the characteristics of faults is crucial to several research areas. The power that inheritance and polymorphism brings to the expressiveness of programming languages also brings a number of new anomalies and fault types. This paper presents a model for the appearance and realization of OO faults and defines and discusses specific categories of inheritance and polymorphic faults. The model and categories can be used to support empirical investigations of object-oriented testing techniques, to inspire further research into object-oriented testing and analysis, and to help improve design and development of object-oriented software.

1. Introduction

Like their procedural counterparts, programs written in object-oriented languages have data flow anomalies and faults. Occasionally one of these faults manifests a failure, and corrective measures are then usually taken to eliminate the fault. Fortunately, many of the traditional testing techniques and strategies for fault elimination are applicable to object-oriented programs, although the techniques primarily focus on the syntactic and semantic constructs that are also found in procedure-oriented languages. The power that inheritance and polymorphism brings to the expressiveness of programming languages also brings a number of new anomalies and fault types. We refer to these as *object-oriented faults*. Unfortunately, the techniques that are used to eliminate faults in procedure-oriented programs are not as applicable to those found in object-oriented programs.

The current paper restricts attention to what is commonly known as subtype inheritance, rather than subclass. If class B uses *subtype* inheritance to inherit from class A, then it will be semantically possible for any instance of class B to freely be used (substituted) when an instance of class A is expected [11, 12]. This is called "substitutability". That is, B has an "is-a" relationship with A, for example, a *car* "is a" special case of a *vehicle*. Subclass inheritance relaxes this restriction, and allows descendant classes to reuse methods and variables from ancestor classes without necessarily ensuring that instances of the descendants are type-compatible with the ancestor type. There is some disagreement over which use of inheritance is appropriate [10, 19], and the authors of this paper are **not** taking a position with regards to this debate. Rather, we recognize that professional programmers use both types of inheritance, and both can lead to faults. This paper restricts attention to faults that can occur when subtype inheritance is used, and later work will address faults that arise from subclass inheritance. For this paper, a class *extends* its parent class if it introduces a new method name and does not override any methods in an ancestor class. A class *refines* the parent class if it provides new behavior not present in the overridden method, does

*This work is supported in part by the U.S. National Science Foundation under grant CCR-98-04111.

not call the overridden method, and its behavior is semantically consistent with that of the overridden method.

This paper makes several assumptions. Unless otherwise noted, state variables that are inherited have sufficient visibility to allow direct reference by methods defined in descendant classes. For example, in the languages Java and C++, the access specifier for the state variables are **not** private.

Inheritance is used to create classes that are subtypes of their parents, not subclasses [12]. Methods may be overridden in a descendant class.

Finally, when considering the state effects of a particular method, the transitive closure of state definitions is assumed over called methods that are locally defined in a descendant class. That is, if a method *m()* calls *n()* and *n()* gives a definition to a state variable *y*, *m()* is considered to define *y* by transitivity. Without loss of generality, we can ignore those non-public methods in a descendant that affect state and that are only called by other methods also defined in the descendant. This is safe to do since the state definitions made by those methods cannot be called by any method defined outside of the descendant, and considering them would not add new fault types. Further, the effects of these methods is captured in the transitive closure mentioned above.

1.1. A fault/failure model for object-oriented programs

In standard IEEE terminology [9], a *failure* is an external, incorrect behavior of a program (an incorrect output or a runtime failure). A *fault* is the group of incorrect statements in the program that causes a failure. Based on these definitions, the fault/failure model that is widely used in the testing literature states that there are three conditions necessary for a failure to be observed [6, 15]. First, the location in the program that contains the fault must be reached (*Reachability*). Second, after executing the location, the state of the program must be incorrect (*Infection*). Third, the infected state must propagate to cause some output of the program to be incorrect (*Propagation*)[1]. Faults that result from polymorphic behavior must conform to this model, and a general failure model can be formulated in terms of this model. Figure 1 depicts a UML class diagram that shows an inheritance hierarchy and client relationships. This figure is used in the following to describe the fault model for failures that result from the use of polymorphism.

The first condition, **reachability** has five requirements:

1. There exists an inheritance hierarchy rooted at class T with descendant D (*class D extends T*).

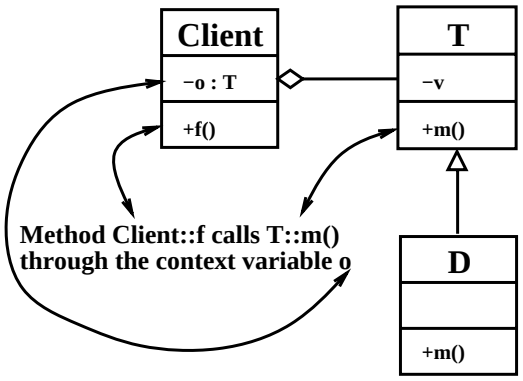


Figure 1. Example inheritance hierarchy.

2. T has a method $m()$ and D has a method $m()$ that overrides *T::m()*.
3. There is a variable *o* in a client *C* with a declared type of *T* ($T\ o$;).
4. The actual type of the instance bound to o is D (for example, $o = new\ D()$;).
5. C invokes m through the instance context provided by o ($o.m()$;).

Infection occurs under the following condition. For a polymorphic fault to exist, *T::m()* and *D::m()* must modify different portions of the state space of T. Note that *D::m()* may define a variable with an incorrect value, but we do not consider this to be a polymorphic fault, but a traditional fault[2]. To model this situation, we want to compare the portion of the state that is declared by T. Any state added by D is not relevant. Thus: *defs (D::m())* $\cup\ state(T) \neq$ *defs (T::m())*. That is, there is some state variable v such that $D : m()$ defines v but *T::m()* does not, or *T::m()* defines v but *D::m()* does not.

Propagation occurs under the following condition. There must be a variable that is defined by either *D::m()* or by *T::m()* and not both, there must be an execution path from the definition to a later use that contains no intervening definition to that variable. That is: $\exists n \in methods(T) \mid C\ calls\ o.n() \wedge \exists w \in state(T) \mid uses(n, w) \wedge ((w \in defs($ *D::m()*$) \wedge w \overline{\in} defs($ *T::m()*$)) \vee (w \overline{\in} defs($ *D::m()*$) \wedge w \in defs($ *T::m()*$))$. C need not be the same client that called $o.m()$ earlier. The only requirement is that at some future point in time, $n()$ is called in the context of the same instance that $m()$ was called in.

[1] Morell used Execution, Infection, and Propagation [14, 15]. Offutt used Reachability, Sufficiency, and Necessity [16, 6]. We choose to combine the two sets of terms by using what we consider to be the most descriptive.

[2] A definition may be direct through an assignment, as in $x = y$, or indirect through a method call whose effect is to change the state of the instance bound to the variable. Without loss of generality, the (conservative) view adopted in this paper is that any such method call always results in a state change to the instance. However, static analysis techniques can usually be used to identify some of the calls that actually define the state.

2. A Graphical Model for Polymorphic Faults

One of the most difficult aspects of developing object-oriented software is **visualizing** the often complex interactions that can occur in the presence of inheritance, polymorphism, and dynamic binding [5]. The compositional relationships of inheritance and aggregation, combined with the power of polymorphism and the inherent undecidability of dynamic binding, increase the difficulty of modeling software, detecting faults, and debugging the faults [4]. This section presents a model for visualizing these interactions, particularly with the idea of understanding actual and potential faults in object-oriented software.

2.1. Overview of object-oriented language features

The fundamental building block in object-oriented software is the *class*, which is used to define new types. A class encapsulates state information in a collection of *state variables*, and has a set of behaviors that are implemented by methods that use those state variables. Classes define types that are used to instantiate objects [13, 18].

Classes can be composed to form new types in two ways. In *aggregation*, one type contains instances of another type. Previous languages implement aggregation with records. *Inheritance* allows the representation of one type to be defined in terms of the representation of one or more existing types. The new type (the *child*) has all of the state and behavior properties of the existing (*parents*), That is, it *inherits* the parent's variables and methods.

Polymorphism allows the same pointer to reference objects of different types, subject to limitations of the inheritance hierarchy. Thus, the type of the object referenced can change during execution. A pointer's *declared type* is the type used when it is declared, and its *actual type* is the type of the object last assigned to the pointer. Most OO languages (including Java and C++) require that the type of the pointer's object be its declared type or a descendant of its declared type.

When polymorphism is combined with method overriding, the same call can execute different methods. This is called *dynamic binding*. The method that is executed depends on the **actual type** of the object when the call is reached. Thus, which method is actually executed cannot be known statically, and must be determined dynamically (during execution).

As an example, consider the UML class diagram and code fragment shown in Figure 2. The declared type of *o* is *W*, but at line 10, the actual type will be either *V* or *W*. Since *V* overrides *m()*, which version of *m()* is executed depends on whether the input flag to the method *f()* was `true` or `false`.

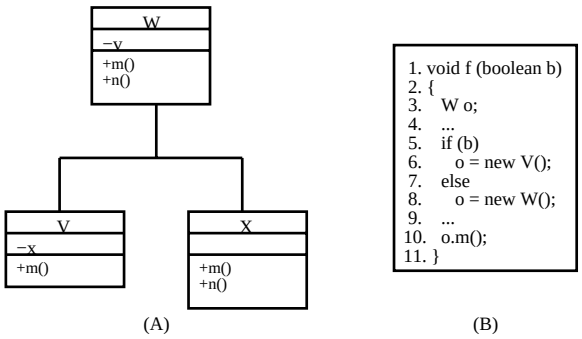


Figure 2. Example class hierarchy in UML. V and X inherit from W. V overrides method m() and X overrides m() and n(). The minuses ("-") indicate the attributes are private and the pluses ("+") indicate the attributes are non-private.

2.2. Problems with overriding and polymorphism

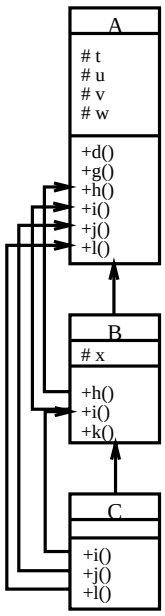
To illustrate the problems that can be caused by method overriding and polymorphism, consider the simple inheritance hierarchy that is three classes deep, shown on the left of Figure 3. The root class *A* contains four state variables and six methods. The state variables are *protected*, and thus are available to descendents of *A*. Its immediate descendant *B* specifies one state variable and three methods. Finally, class *C* specifies only three methods. The arrows on the left show the overriding: *B::h()* overrides *A::h()*, *B::i()* overrides *A::i()*, *C::i()* overrides *B::i()*, *C::j()* overrides *A::j()*, and *C::l()* overrides *A::l()*. The table of the right of Figure 3 shows the state variable definitions and uses of some of the methods in the hierarchy. The problem begins with a call to *A::d()*. This small example has some very complex interactions that potentially yield some very difficult problems to model, understand, test, and debug.

Suppose that an instance of *A* is bound to an object *o* and a call is made through *o* to *A::d()*, which calls *A::g()*, which calls *A::h()*, which calls *A::i()*, which finally calls *A::j()*. In this case, the variables *A::u* and *A::w* are first defined, then used in *A::i()* and *A::j()*, which poses no problems.

Now suppose that an instance of *B* is bound to *o*, and a call to *d()* is made. This time *B*'s version of *h()* and *i()* are called, *A::u* and *A::w* are **not** given values, and thus the call to *A::j()* can result in a data flow anomaly.

2.3. Modeling polymorphism: The yo-yo graph

One of the major difficulties with using polymorphism and dynamic binding is that of modeling and visualizing the complicated interactions. The essential problems are that of understanding which version of a method **will** be executed



Method	Defs	Uses
A::h	{A::u, A::w}	
A::i		{A::u}
A::j	{A::v}	{A::w}
A::l		{A::v}
B::h	{B::x}	
B::i		{B::x}
C::i	{C::y}	
C::j		{C::y}
C::l		{A::v}

Figure 3. Data flow anomalies with polymorphism.

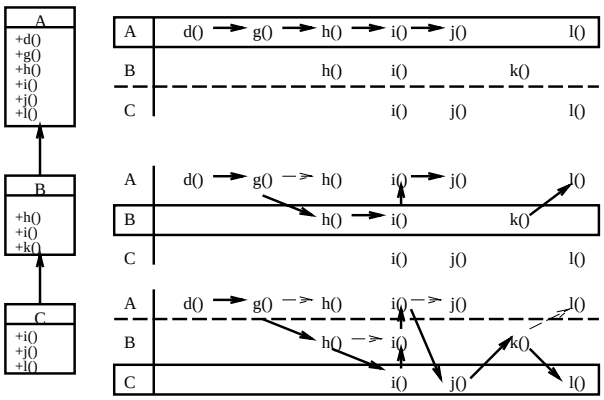


Figure 4. Calls to d() when object has various actual types.

and which versions **can** be executed. The fact that execution can sometimes "bounce" up and down among levels of inheritance has been called the yo-yo effect by Binder and he introduced a preliminary graph [5]. We have extended this notion as a basis for a graphical representation that we call the "yo-yo graph" to show all possible actual executions in the presence of dynamic binding. The *yo-yo graph* is defined on an inheritance hierarchy that has T_0 as root and descendants T_1 through T_n. For each class T_i, all new, inherited, and overridden methods are shown. Method *calls* in the source are represented as arrows from caller to callee. Each class T_i is given a *level* in the yo-yo graph that shows the actual calls made if an object has the actual type T_i. Bold arrows are actual calls and light arrows are calls that cannot be made due to overriding.

Consider the inheritance hierarchy shown in Figure 3. Assume that in *A*'s implementation, *d()* calls *g()*, *g()* calls *h()*, *h()* calls *i()*, and *i()* calls *j()*. Further, assume that in *B*'s implementation, *h()* calls *i()*, *i()* calls its parent's (that is, *A*'s) version of *i()*, and *k()* calls *l()*. Finally, assume that in *C*'s implementation, *i()* calls its parent's (this time *B*'s) version of *i()*, and *j()* calls *k()*.

Figure 4 is a yo-yo graph of this situation and expresses the **actual** sequence of calls if a call is made to *d()* through an instance of actual type A, B, and C. At the top level of the graph, it is assumed that a call is made to method *d()* through an object of actual type A. In this case, the sequence of calls is simple and straightforward. In the second level, where the object is of actual type *B*, the situation starts to get more complex. When *g()* calls *h()*, the version of *h()* defined in *B* is executed (the light dashed line from *A::g()* to *A::h()* emphasizes that *A::h()* is **not** executed). Then control continues to *B::i()*, *A::i()*, and then to *A::j()*.

When the object is of actual type *C*, it becomes clear why the term "yo-yo" is used. Control proceeds from *A::g()* to *B::h()* to *C::i()*, then back up through *B::i()* to *A::i()*, back to *C::j()*, back up to *B::k()*, and finally down to *C::l()*.

This example illustrates some the complexities that can result in object-oriented programs due to method overriding and polymorphism. Along with this induced complexity comes more difficulty and required effort in testing.

3. Categories of Inheritance Faults and Anomalies

Benefits of using inheritance include more creativity, efficiency, and reuse. Unfortunately, it also allows a number of anomalies and potential faults that anecdotal evidence has shown to be some of the most difficult problems to detect, diagnose, and correct. This section presents a list of fault types that can be manifested by polymorphism. Table 1 summarizes the fault types that result from inheritance and polymorphism. The goal is a complete list of faults, though we do not make this claim. Most of the types are programming language-independent, although the language that is used will affect how the faults manifest. In all cases, we are concerned with how each anomaly or fault is manifested through polymorphism in a context that uses an instance of the ancestor. Thus, we assume that instances of descendant classes can be substituted for instances of the ancestor.

Acronym	Fault/Anomaly
ITU	Inconsistent Type Use (context swapping)
SDA	State Definition Anomaly (possible post-condition violation)
SDIH	State Definition Inconsistency (due to state variable hiding)
SDI	State Defined Incorrectly (possible post-condition violation)
IISD	Indirect Inconsistent State Definition
ACB1	Anomalous Construction Behavior (1)
ACB2	Anomalous Construction Behavior (2)
IC	Incomplete Construction
SVA	State Visibility Anomaly

Table 1. Faults and anomalies due to inheritance and polymorphism.

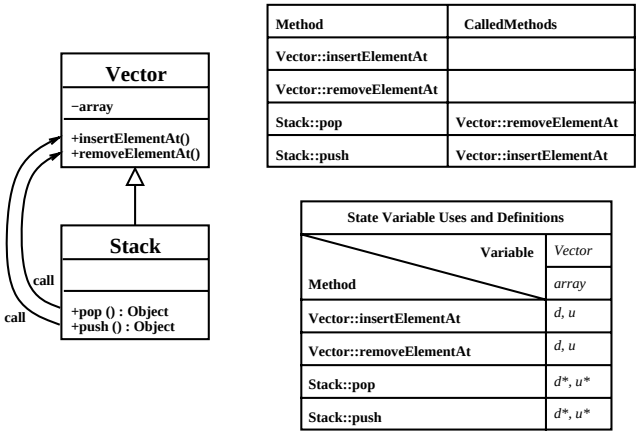


Figure 5. ITU: Descendant with no overriding methods.

3.1. Inconsistent type use (ITU)

For this fault type, a descendant class does not override any inherited method. Thus, there can be no polymorphic behavior. Every instance of a descendant class C that is used where an instance of T is expected can only behave exactly like an instance of T. That is, only methods of T can be used. Any additional methods specified in C are hidden since the instance of C is being used as if it is an instance of T. However, anomalous behavior is still a possibility. If an instance of C is used in multiple contexts (that is, through coercion, say first as a T, then as a C, then a T again), anomalous behavior can occur if C has extension methods. In this case, one or more of the extension methods can call a method of T or directly define a state variable inherited from T. Anomalous behavior will occur if either of these actions results in an inconsistent inherited state.

As an example, consider the class hierarchy shown in Figure 5[3]. Class *Vector* encapsulates a sequential data structure that supports direct access to its elements. Class *Stack* also encapsulates a sequential data structure that has a "last-in/first-out" access policy. As shown, *Stack* uses methods inherited from *Vector* to implement its behavior. The top table summarizes the calls made by each method, and the bottom table summarizes the definitions and uses (represented as "d" and "u", respectively) of the state space of *Vector*.

The extension method *Stack::pop()* calls *Vector::removeElementAt()*, and *Stack::push()* calls *Vector::insertElementAt()*. Clearly these two classes have different semantics. As long as an instance of *Stack* is used solely as an instance of *Stack*, there will be no behavioral problems. Alternatively, the *Stack* instance could be used solely as a instance of *Vector*, again without experiencing behavioral problems. However, if the usage of the instance is mixed between the *Stack* and *Vector*, behavioral problems can occur.

The code fragment in Table 2 illustrates how behavioral anomalies can occur when the type system is used to manipulate the manner in which instances of classes are used. For the method $f()$, the instance bound to the formal argument s is used only as a $Stack$ in lines 3 through 9. However, at line 11, s is passed as an actual argument to method g, which expects an instance of $Vector$. This is syntactically correct because an instance of $Stack$ is also an instance of *Vector*. There is a potential behavioral problem that begins at line 21 where the last element of s is removed. The fault is manifested when control returns and reaches the first call to *Stack::pop()* at line 14. Here, the element removed from the stack is **not** the last element that was added, thus the stack integrity constraint will be violated.

3.2. State definition anomaly (SDA)

In general, for a descendant class to be behaviorally compatible with its ancestor, the state interactions of the descendant must be consistent with those of its ancestor. That is, the refining methods implemented in the descendant must leave the ancestor in a state that is equivalent to the state that the ancestor's overridden methods would have left the ancestor in. For this to be true, the refining methods provided by the descendant must yield the same net state interactions as each public method that is overridden. From a data flow perspective, this means that the refining methods must provide definitions for the inherited state variables that are consistent with the definitions in the overridden method. If not, then a potential data flow anomaly exists. Whether or not an anomaly actually occurs depends upon the sequences

[3] This example is based on the library provided with the Java Development Kit version 1.2.

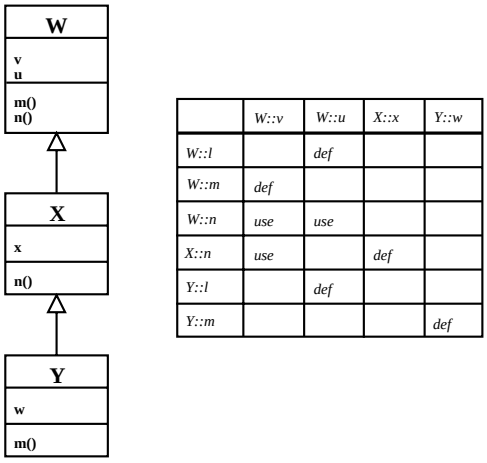
```
 1  public void f (Stack s)
 2  {
 3    String s1 = "s1";
 4    String s2 = "s2";
 5    String s3 = "s3";
 6     ...
 7    s.push (s1);
 8    s.push (s2);
 9    s.push (s3);
10
11    g (s);
12
13    s.pop();
14    s.pop();
15    // Oops!  The stack is empty!
16    s.pop();
17    ...
18  }

19  public void g (Vector v)
20  {
21    // Remove the last element
22    v.removeElementAt (v.size()-1);
23  }
```

Table 2. ITU: Code example showing inconsistent type usage.

of methods that are valid with respect to the ancestor.

As an example, consider the class hierarchy and tables of definitions and uses shown in Figure 6. The parent of the hierarchy is class W, and it has descendants X, and Y. W defines methods $m()$ and $n()$, each of which has the definitions and uses shown in the table. Assume that a valid method call sequence is *W::m()* followed by *W::n()*. As the table of definitions and uses shows, *W::m()* defines state variable *W::v* and *W::n()* uses it. Now consider the class X and its refining method *X::n()*. As the table shows, it too uses state variable *W::v*, which is consistent with the overridden method and with the method sequence given above. Thus far, there is no inconsistency in how *X* interacts with the state of W. In fact, because a use can never affect future state-dependent behavior, *X::n()* could just as easily have used a different variable.

Now consider class Y and the method *Y::m()*, which overrides *W::n()* through refinement. Observe that *Y::m()* does not define *W::v*, as *W::m()* does; but defines *Y::w* instead. Now, a data flow anomaly exists with respect to the method sequence $m();n()$ for the state variable *W::v*. When an instance of Y is subjected to this sequence, *Y::w* is defined first (because *Y::m()* executes), but then *W::v* is used by method *X::n()*. Thus, the assumption made in the implementation of *X::n()* that *W::v* is defined by a call to $m()$ prior to a call to $n()$ no longer holds, and a data flow anomaly has occurred. In this particular example, a failure occurs since there is no prior definition of *W::v* when Y is the type of an instance being used. Note that this will not be true in the general case since the controlling factor



	W::v	*W::u*	*X::x*	*Y::w*
W::l		*def*		
W::m	*def*			
W::n	*use*	*use*		
X::n	*use*		*def*	
Y::l		*def*		
Y::m				*def*

Figure 6. SDA, SDIH: State definition anomalies.

in whether a fault has occurred will be a function of what prior method invocations have occurred, any default initializations that were applied, and how individual state variables are handled during instance construction.

Any extension method that is called by a refining method must also interact with the inherited variables of the ancestor in a manner that is consistent with the ancestor's current state. Since the extension method provides a portion of the refining method's net effects, to avoid a data flow anomaly the extension must not define inherited state variables in a way that would be inconsistent with the method being refined. Thus, the net effect of the extension method cannot be to leave the ancestor in a state that is logically different from when it was invoked. For example, if the logical state of an instance of a stack is currently not-empty/not-full, then execution of an extension method cannot result in the logical state spontaneously being changed to either empty or full. Doing so would preclude the execution of *pop* or *push* as the next methods in sequence.

3.3. State definition inconsistency due to state variable hiding (SDIH)

The introduction of an indiscriminately named local state variable can easily result in a data flow anomaly where none would otherwise exist. If a local variable is introduced to a class definition where the name of the variable is the same as an inherited variable v, the effect is the inherited variable is hidden from the scope of the descendant (unless explicitly qualified, as in $super.v$). A reference to v by an extension or overriding method will refer to the descendant's v. This is not a problem if all inherited methods are overridden since no other method would be able to implicitly reference the inherited v. However, this pattern of inheritance is the exception rather than the rule. There will typically be one or more inherited methods that are not overridden. There is a

possibility for a data flow anomaly to exist if a method that normally defines the inherited v is overridden in a descendant when an inherited state variable is hidden by a local definition.

As an example, again consider the class hierarchy shown in Figure 6. Suppose the definition of class Y has the local state variable v that hides the inherited variable *W::v*. Further suppose method *Y::m()* defines v, just as *W::m()* defines *W::v*. Given the method sequence $m();n()$, a data flow anomaly exists between W and Y with respect to *W::v*.

3.4. State defined incorrectly (SDI)

Suppose an overriding method defines the same state variable *v* that the overridden method defines. If the computation performed by the overriding method is not semantically equivalent to the computation of the overridden method with respect to v, then subsequent state dependent behavior in the ancestor will likely be affected, and the externally observed behavior of the descendant will be different from the ancestor. While this problem is not a data flow anomaly, it is a potential behavior anomaly.

3.5. Indirect inconsistent state definition (IISD)

An inconsistent state definition can occur when a descendant adds an extension method that defines an inherited state variable. For example, consider the class hierarchy shown in Figure 7A, where Y specifies a state variable x and method $m()$, and the descendant D specifies method $e()$. Since $e()$ is an extension method, it cannot be directly called from an inherited method, in this case *T::m()*, because $e()$ is not visible to the inherited method. However, if an inherited method is overridden, the overriding method (such as *D::m()* as depicted in Figure 7B) can call $e()$ and introduce a data flow anomaly by having an effect on the state of the ancestor that is not semantically equivalent to the overridden method (e.g. with respect to the variable *T::y* in the example). Whether an error occurs depends on which state variable is defined by $e()$, where $e()$ executes in the sequence of calls made by a client, and what state dependent behavior the ancestor has on the variable defined by $e()$.

3.6. Anomalous construction behavior(1) (ACB1)

The constructor of an ancestor class C calls a locally defined polymorphic method $f()$. Because $f()$ is polymorphic, a descendant class D can provide an overriding definition of $f()$. If this is so, then D's version of $f()$ will execute when the constructor of C calls $f()$, not the version defined by C. To see this, consider the class hierarchy shown in the left half of Figure 8. Class C's constructor calls *C::f()*. Class D contains the overriding method *D::f()* that

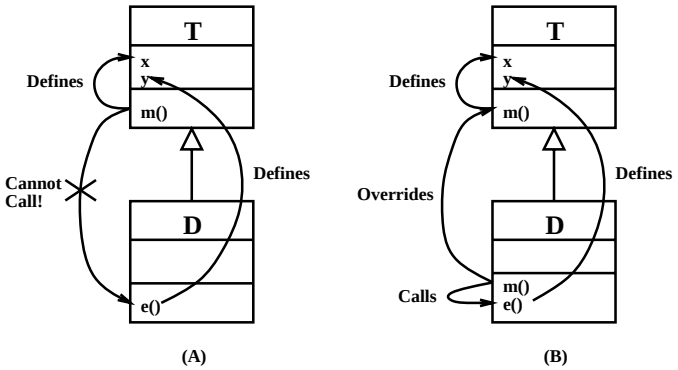


Figure 7. IISD: Example of indirect inconsistent state definition.

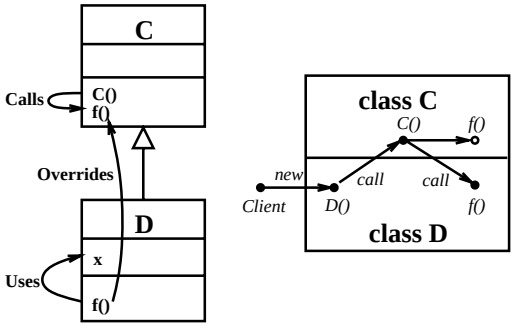


Figure 8. ACB1: Example of anomalous construction behavior.

defines the local state variable *D::x*. There is no apparent interaction between D and C since *D::f()* does not interact with the state of C. However, C interacts with D's state by virtue of the apparent call that C's constructor makes to *C::f()*. In some object-oriented languages (e.g. Java and C-Sharp), constructor calls to polymorphic methods execute the method that is closest to the instance that is being created. For the class C in the hierarchy in Figure 8, the closest version of $f()$ to C is specified by C itself, and thus executes when an instance of C is being constructed. For D, the closest version is *D::f()*, which means that when an instance of D is being constructed, the call made to $f()$ in C's constructor actually executes *D::f()* instead of its own locally specified $f()$. This is illustrated by the partial yo-yo graph in the right half of Figure 8.

The result of the behavior shown in Figure 8 can easily result in a data flow anomaly if *D::f()* uses variables defined in the state space of D. Because of the order of construction, D's state space will not have been constructed. Whether or not an anomaly exists depends on if default initializations have been specified for the variables used by $f()$. Furthermore, a fault will likely occur if the assumptions or precon-

ditions of *D::f()* have not been satisfied prior to construction [2].

3.7. Anomalous construction behavior(2) (ACB2)

Similar to ACB1 (Section 3.6), the constructor of an ancestor class C calls a locally defined polymorphic method $f()$. A data flow anomaly can occur if $f()$ is overridden in a descendant class D and if that overriding method uses state variables inherited from C. The anomaly occurs if the state variables used by *D::f()* have not been properly constructed by *C::f()*. This depends on the set of variables used by *D::f()*, the order in which the variables in the state of C are constructed, and the order in which $f()$ is called by C's constructor. Note that it is not generally possible for the programmer of class C to know in advance which version of $f()$ will actually execute, and which state variables that the executing version depends on. Thus, the invocation of polymorphic method calls from constructors is unsafe and introduces non-determinism into the construction process. This is true of both ACB2 and ABC1.

3.8. Incomplete (failed) construction (IC)

In some programming languages, the value of the variables in the state space of a class before construction is undefined. This is true, for example, in C++ but not in Java. The role of the constructor is to establish the initial state conditions and the state invariant for new instances of the class. To do so, the constructor will generally have statements that define every state variable. In some circumstances, again depending upon the programming language, default or other explicit initializations may be sufficient. In either case, by the time the constructor has finished, the state of the instance should be well defined. There are two possibilities for faults here. First, the construction process may have assigned an initial value to a particular state variable, but it is the wrong value. That is, the computation used to determine the initial value is in error. Second, the initialization of a particular state variable may have been overlooked. In this case, there is a data flow anomaly between the constructor and each of the methods that will first use the variable after construction (and any other uses until a definition occurs).

An example of incomplete construction is shown by the code fragment in Table 3. Class $AbstractFile$ contains the state variable *fd* that is not initialized by a constructor. The intent of the designer of $AbstractFile$ is that a descendant class provide the definition of *fd* prior to its use, which is done by method $open()$ in the descendant class $SocketFile$. If any descendant that can be instantiated defines *fd*, and no method is called that uses *fd* prior to the definition, there is no problem. However, a fault will occur if either of these conditions is not satisfied.

Observe that while the designer's intent is for a descendant to provide the necessary definition, a data flow anomaly exists within *AbstractFile* with respect to *fd* for methods $read()$ and $write()$. Both of these methods use *fd*, and if either are called immediately after construction, a fault will occur. Note that this design introduces an element of non-determinism into $AbstractFile$ since it is not known at design time what type of instance *fd* will be bound to, or if it will be bound (i.e. defined) at all. Suppose that the designer of $AbstractFile$ also designed and implemented $SocketFile$, as also shown in Table 3. By doing so, the designer has ensured that the data flow anomaly that exists in $AbstractFile$ is abated by the design of $SocketFile$. However, this still does not eliminate the problem of non-determinism and the introduction of faults since, at some point in time in the future, a new descendant can be added that fails to provide the necessary definition.

3.9. State visibility anomaly (SVA)

The state variables in an ancestor class A are declared private, and a polymorphic method *A::m()* defines *A::v*. Suppose that B is a descendant of A, and C of B, as depicted in Figure 9A. Further, C provides an overriding definition of *A::m()* but B does not. Since *A::v* has private visibility, it is not possible for *C::m()* to properly interact with the state of A by directly defining *A::v*. Instead, *C::m()* must call *A::m()* to modify *v*. Now suppose that B also overrides m (Figure 9B). Then for *C::m()* to properly define *A::v*, *C::m()* must call *B::m()*, which in turn must call *A::m()*. Thus, *C::m()* has no direct control on whether the data flow anomaly is resolved! In general, when private state variables are present, the only way that avoiding a data flow anomaly can be ensured is for every overriding method in a descendant to call the overridden method in its ancestor class. Failure to do so will quite possibly result in the manifestation of a fault in the state and behavior of A.

4. Applications and Implications

By considering this OO fault model, and specifically the types of faults that can be generated from OO constructs, we can gain insight into a number of issues in object-oriented analysis. In the following subsections, we discuss how this fault/failure model and the fault types impact several areas in object-oriented analysis.

4.1. Fault seeding for OO test evaluations

Fault seeding refers to artificially introducing faults into programs, usually to measure the quality of testing or to empirically compare testing strategies. When we seed faults

```
 1  Class abstract AbstractFile
 2  {
 3    FileHandle fd;
 4
 5    abstract public open();
 6
 7    public read() {fd.read ( ... ); }
 8
 9    public write() {fd.write ( ... ); }
10
11    abstract public close();
12  }
```

```
14  Class SocketFile extends AbstractFile
15  {
16    public open()
17    {
18      fd = new Socket ( ... );
19    }
20
21    public close()
22    {
23      fd.flush();
24      fd.close();
25    }
26  }
```

Table 3. IC: Incomplete construction of state variable fd.

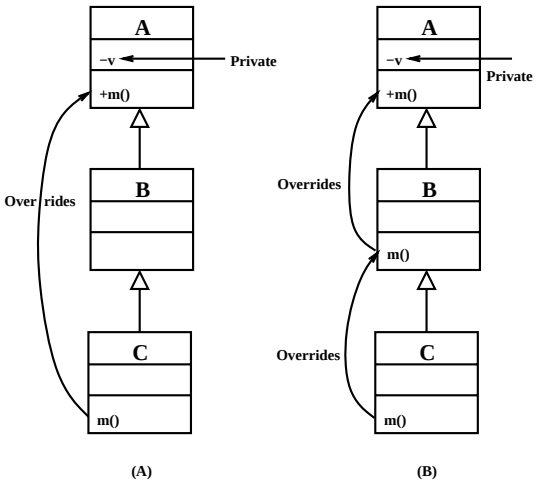


Figure 9. SVA: State visibility anomaly.

into programs we desire to have faults that are representative of real faults that could appear in software. Previously, the idea of fault *semantic size* was introduced as the relative number of inputs on which the program behaves incorrectly [17]. If the semantic size of a seeded fault is too small, then the fault will introduce a bias against the testing strategy being evaluated; if the semantic size of a seeded fault is too large, the fault will introduce a bias in favor of the testing strategy. It can be reasonably assumed that realistic faults, that is, faults that are representative of naturally occurring faults, are likely to have semantic sizes that are neither too large or too small.

Hamlet [7] pointed out that empirical comparisons of testing techniques face two problems with regards to subjects: choosing a representative collection of programs, and choosing a representative collection of tests. Both of these are examples of internal controls on the empirical process and represent problems that are always present in any experiment. Problems with internal control can mean that the results of the experiment may not scale up to be true in all situations. We suggest that another potential problem is that if the techniques are compared based on the faults they find, a representative collection of faults must be used.

It is hoped that the types of faults presented in this paper can allow empirical researchers in the testing area to seed faults that are representative of naturally occurring faults. In Alexander's dissertation [1], these fault types were used to empirically compare criteria that he developed for integration testing of object-oriented software. The fault types were used to create a large number of specific faults that were seeded into ten subject programs. The fault types presented in this paper can be used for other empirical comparisons involving object-oriented software.

4.2. Metrics for polymorphism use

One thing that is currently missing in object-oriented software development is that of metrics that measure complexity of software that uses inheritance and polymorphism [3, 8]. The fault/failure model, the yo-yo graph, and the fault types can be used as a basis for which to measure object-oriented software. If a class, or a collection of classes in an inheritance hierarchy, is structured in such a way as to allow many of the problems discussed in this paper, then that should represent a negative measurement of that class or inheritance hierarchy.

4.3. Standards for appropriate use of polymorphism

Software is increasingly being built by combining and extending pre-existing software components. In particular, we often create new classes through inheritance by extending from pre-existing classes. Moreover, we often do not have access to the source for these library classes! A common example in web software is Java Servlets, which are created by extending the pre-existing *HttpServlet* class. Although this is a very powerful abstraction mechanism, the implementation can be somewhat problematic. In particular, careless inheritance and overriding can create problems

in the state space interactions of the resulting objects (as described in Section 2). Unfortunately, providers of class libraries often want to keep the implementation proprietary, and thus do not provide the source. Since the developer may not **know** the internals of the parent class, there is no way to know what type of inheritance and polymorphism is "careless"!

4.4. Summary

This paper has presented a model for faults and failures in object-oriented programs, a graphical model for polymorphic faults, and a collection of specific object-oriented fault types. In the future, we plan to extend this work to model faults that can appear when subtyping is used. These models and fault types can be used in a variety of situations, as has been outlined. One obvious subject of future work is to ask questions such as how often the faults presented in this paper appear and what percentage of failures in OO software is due to these faults. Although few simple programs will use dynamic binding as freely as the manufactured example in Section 2, our experience is that this type of software design is used frequently and these types of faults occur regularly.

5. Acknowledgments

We would like to thank Len Gallagher of NIST for a number of helpful comments.

References

[1] R. T. Alexander. *Testing the Polymorphic Relationships of Object-oriented Programs*. PhD thesis, George Mason University, Fairfax VA, 2001. Technical report ISE-TR-01-04, http://www.ise.gmu.edu/techrep.

[2] Roger T. Alexander, James M. Bieman, and John Viega. Coping with Java programming stress. *Computer*, 33(4):30–38, 2000.

[3] V. Basili, L. Briand, and W. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, October 1996.

[4] Edward Berard. Issues in the testing of object-oriented software. In *Electro'94 International*, pages 211–219. IEEE Computer Society Press, 1994.

[5] Robert V. Binder. Testing object-oriented software: A survey. *Journal of Software Testing, Verification & Reliability*, 6(3/4):125–252, 1996.

[6] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991. http://ise.gmu.edu/faculty/ofut/rsrch/abstracts/cbt.html.

[7] D. Hamlet. Theoretical comparisons of testing methods. In *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification*, pages 28–37, Key West Florida, December 1989. ACM SIGSOFT 89.

[8] R. Ibba and D. Natale. Structure-based clustering of components for software reuse. In *Proceedings of the International Conference on Software Maintenance 1993*, pages 210–215. IEEE Computer Society Press, September 1993.

[9] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. ANSI/IEEE Std 610.12-1990, 1996.

[10] W. LaLonde and J. Pugh. Subclassing != subtyping != is-a. *Journal of Object Oriented Programming*, 3(5), January 1991.

[11] B. Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison Wesley, New York NY, 2000.

[12] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. and Sys.*, 16(1):1811–1841, November 1994.

[13] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, New Jersey, 1997.

[14] L. J. Morell. *A Theory of Error-Based Testing*. PhD thesis, University of Maryland, College Park MD, 1984. Technical Report TR-1395.

[15] L. J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[16] A. J. Offutt. *Automatic Test Data Generation*. PhD thesis, Georgia Institute of Technology, Atlanta GA, 1988. Technical report GIT-ICS 88/28.

[17] A. J. Offutt and J. H. Hayes. A semantic model of program faults. In *Proceedings of the 1996 International Symposium on Software Testing, and Analysis*, pages 195–200, San Diego, CA, January 1996. ACM Press. http://ise.gmu.edu/faculty/ofut/rsrch/abstracts/synsem.html.

[18] D. L. Parnas, J. E. Shore, and D. Weiss. Abstract types defined as classes of variables. In *Proceedings of Conference on Data: Abstraction, Definition and Structure*, pages 22–24, Salt Lake City, UT, USA, 1976.

[19] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, 1996.