Journal of Universal Computer Science, vol. 10, no. 4 (2004), 391-427
submitted: 17/10/03, accepted: 2/2/04, appeared: 28/4/04

Coupling-based Testing of O-O Programs

Roger T. Alexander
(Colorado State University, USA
rta@cs.colostate.edu)

Jeff Offutt
(George Mason University, USA
ofut@ise.gmu.edu)

Abstract: As we move from developing procedure-oriented to O-O programs, the complexity traditionally found in functions and procedures is moving to the connections among components. More faults occur as components are integrated to form higher level aggregates. Consequently, we need to place more effort on testing the connections among components. Although O-O technology provides abstraction mechanisms to build components to integrate, it also adds new compositional relations that can contain faults, which must be found during integration testing. This paper describes new techniques for analyzing and testing the polymorphic relationships that occur in O-O software. The application of these techniques can result in an increased ability to find faults and overall higher quality software.
Key Words: object-oriented software, coverage testing
Category: D.2.5

1 Introduction

The emphasis in O-O languages is on defining abstractions such as abstract data types that model concepts in an application domain [Meyer, 1997]. Although abstract data types and other O-O concepts can help achieve a higher quality design, how we test software needs to change. A major factor is that shifting from procedure-oriented software to object-oriented software induces a complementary shift where the complexity of the software resides. Instead of primarily being in the software units, the complexity is now primarily in the way in which we connect software components. Thus, developers are finding that we need less emphasis on unit testing and more on integration testing. Another factor that affects testing of O-O software is due to the inherent complexity in the nature of the relationships found in O-O languages [Binder, 1996].

The compositional relationships of inheritance and aggregation, combined with the power of polymorphism, makes it harder to detect faults that result from the integration of components. This is because the way classes and components are integrated is different in O-O languages [Berard, 1993]. Procedure-oriented languages use procedures and functions as the primary abstraction mechanism. In contrast, both object-based and O-O languages use data abstraction as the primary mechanism. In addition, O-O languages use the integration mechanism of inheritance. New types created by inheritance are descendants of the existing type [Meyer, 1990], but this is not aggregation. A key difference is that the encapsulation of the inherited type may not be preserved, that is, the new type can have access to the internals of the ancestor types. When combined with inheritance, polymorphism and the associated dynamic binding can strongly affect component integration. When a call is made to a polymorphic method, which version is executed depends on the current type of the object [Meyer, 1997].

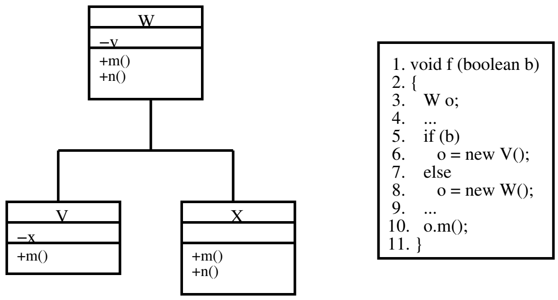
Thus inheritance and polymorphism provide two forms of integration that must be dealt with when testing objects, neither of which has a procedure-oriented counterpart. The first form, integration of representation, addresses the issues associated with combining the representation chosen for the state space of existing classes to form a representation for a new class through inheritance. Properties and behaviors that are inherited, along with state-space definitions, must be carefully combined with new and overriding methods to ensure consistency in behavior and state. The second form, integration of abstraction, deals with the effects of aggregation in the presence of inheritance and polymorphism. To integrate successfully, the aggregated type and its owner must work together correctly for all forms of representation that can exist for the aggregated type. This is not just a static language issue; dynamic binding means that the representation of an aggregated type may change dynamically, i.e, the actual type of the object reference may change. Thus, several substitutions must be tested to ensure the type behaves correctly, that is, the code correctly implements the abstraction.

This paper summarizes research on testing for software faults that can arise from the use of integration of representation and integration of abstraction. [Section 2] presents background information and concepts, and [Section 3] describes a model of faults that result from the use of inheritance and polymorphism; [Section 4] presents background for coupling-based testing and O-O concepts; [Section 5] describes how coupling-based testing is used to testing polymorphic relationships; [Section 6] presents our criteria for testing those relationships; we then summarize experimental results from evaluations of the effectiveness of the coupling-based testing criteria in [Section 7]; we discuss related work in [Section 8] and conclude the paper with [Section 9].

2 Background

Classes are the fundamental building blocks in O-O software. A class defines new types and 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 [Meyer, 1997][Parnas et al., 1976].

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 types (parents). 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 at run time. A



(A) (B)

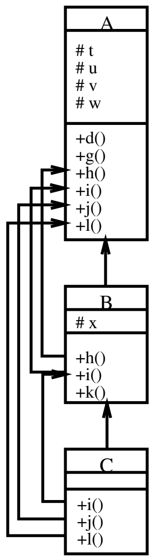
Figure 1: Example class hierarchy in UML

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 O-O languages 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 [Fig. 1]. The declared type of `o` is `W`, but at line 10, the actual type can 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`.

2.1 Problems with Overriding and Polymorphism

Consider the simple inheritance hierarchy in [Fig. 2]. The state variables of class `A` are protected, and thus are available to the descendants. 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 shows the state variable definitions and uses of some of the methods.

This small example has some very complex interactions that can yield very difficult problems. For instance 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.



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 2: SDA, SDIH: State definition anomalies

2.2 A Graphical Model for Polymorphic Faults

One of the most difficult aspects of developing O-O software is visualizing the often complex interactions that can occur in the presence of inheritance, polymorphism, and dynamic binding [Binder, 1996]. 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 [Berard, 1994].

This section presents a model for visualizing these interactions, particularly with the idea of understanding actual and potential faults in O-O software. The essential problems are that of understanding which version of a method will be executed 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 in [Binder, 1996], where a preliminary graph was introduced. 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 [Fig. 2]. Assume that in A's implemen-

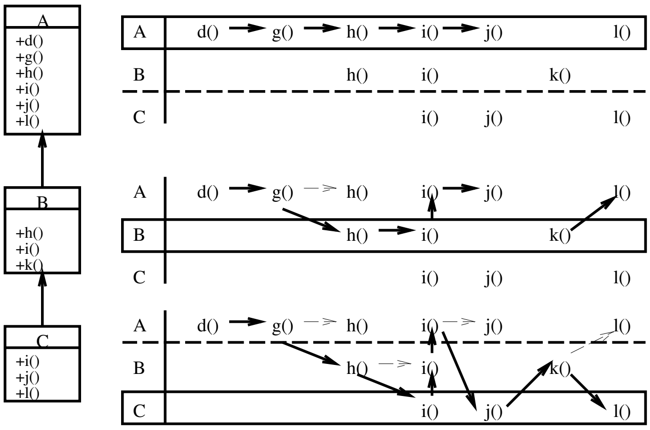


Figure 3: Calls to `d()` when the object has various actual types

tation, `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()`. [Fig. 3] 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()`.

3 Inheritance Faults and Anomalies in O-O Programs

The 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, as summarized in [Tab. 1]. Most of the types are programming language-independent, although the language that is used will affect how the faults manifest. The examples, terminology, and many of the specifics are based on Java. We try to point out where the rules would change for other languages. In all cases, we are concerned with how each

Acronym	Fault or 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

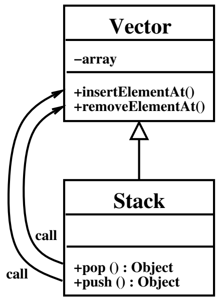
Table 1: Faults and anomalies due to inheritance and polymorphism

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.

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. Instances 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, i.e., say first as a `T`, then as a `C`, then as 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 [Fig. 4], which is based on JDK 1.2. Class `Vector` encapsulates a sequential data structure that supports direct access to its elements, and class `Stack` encapsulates a sequential LIFO data structure. 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 ("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 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 use of the instance is mixed between the `Stack` and `Vector`, be-



Method	CalledMethods
Vector::insertElementAt	
Vector::removeElementAt	
Stack::pop	Vector::removeElementAt
Stack::push	Vector::insertElementAt

State Variable Uses and Definitions	
Variable / Method	*Vector* *array*
Vector::insertElementAt	*d, u*
Vector::removeElementAt	*d, u*
Stack::pop	*d*, u**
Stack::push	*d*, u**

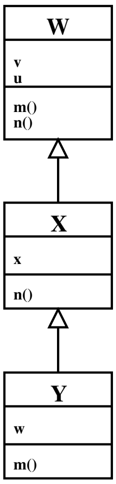
Figure 4: ITU: Descendant with no overriding methods

```
 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   // Empty stack!
16   s.pop();
17   ...
18 }

19 public void g(Vector v)
20 {
21   // Remove the last element
22   v.removeElementAt(v.size()-1);
24 }
```

Figure 5: ITU: Code example showing inconsistent type usage

havioral problems can occur. The code fragment in [Fig. 5] illustrates how behavioral anomalies can occur. 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 since the element removed from the stack is not the last element that was added.



	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: Sample class hierarchy

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 or a potential data flow anomaly exists. Whether or not an anomaly actually occurs depends on the sequences 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 [Fig. 6]. The parent is class `W`, and it has descendants `X`, and `Y`. `W` defines methods `m()` and `n()`, which have the definitions and uses shown in the table. Assume that a valid method call sequence is `W.m()` followed by `W.n()`. `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 also 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.m()` 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 by `X.n()`, that `W.v` is defined by a call to `m()` prior to a call to `n()`, no longer holds. In this 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 is not always true since the controlling factor in whether a fault has occurred is 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 way 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.

3.3 State Definition Inconsistency Due to State Variable Hiding (SDIH)

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 that 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 could implicitly reference v, but this is not usual.

As an example, consider again the hierarchy in [Fig. 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 probably 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 indirect 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 [Fig. 7.a]. Since `e()` is an extension method, it cannot be directly called

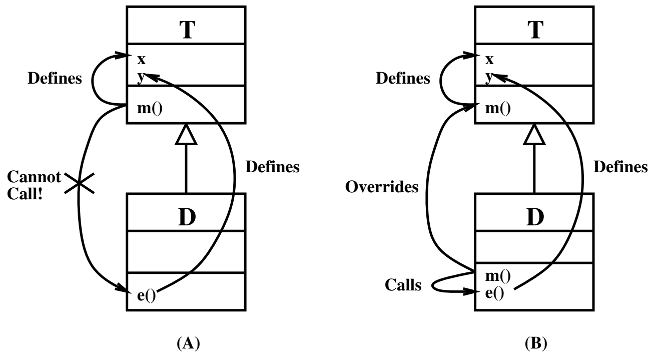


Figure 7: IISD: Example of indirect inconsistent state definition

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()` in [Fig. 7.b]) 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 variable `T.y`. 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 I (ACB1)

Consider the class hierarchy shown in the left half of [Fig. 8]. Class C's constructor calls `C.f()`. Class `D` contains the overriding method `D.f()` that 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 O-O languages, e.g., Java and C#, 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 [Fig. 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 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 whether default initializations have been specified for the variables used by `f()`. Furthermore, a fault is likely to occur if the assumptions or preconditions of `D.f()` have not been satisfied prior to construction [Alexander et al., 2000]. This is particularly insidious if `D` accidentally overrides `f()`, which is possible if the programmer does not have access to `C`'s source.

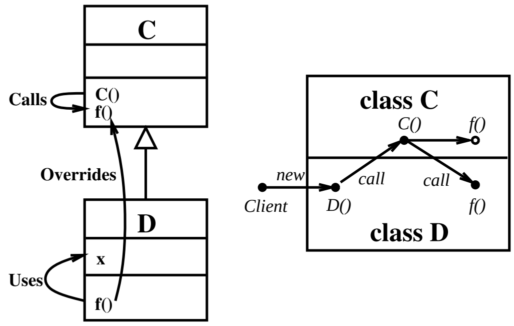


Figure 8: ACB1: Example of anomalous construction behavior

3.7 Anomalous Construction Behavior II (ACB2)

Similar to ACB1, the constructor of `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.

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. The role of the constructor is to establish the initial state conditions and the state invariant for new instances of the class. By the time the constructor has finished, the state of the instance should be well defined. There are two possibilities for faults here: (i) the construction process may have assigned an incorrect initial value to a state variable, i.e., the computation used to determine the initial value is wrong; (ii) the initialization of a state variable may have been overlooked, i.e., 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 [Fig. 9]. Class `AbstractFile` contains the state variable `fd` that is not initialized. The intent 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

```
abstract class AbstractFile
{
  FileHandle fd;

  abstract public
    void open();
  public void read()
    { ... fd.read(); ... }
  public void write()
    { ... fd.write(); ... }
  abstract public
    void close();
}
```

```
class SocketFile
  extends AbstractFile
{
  public void open()
  {
    fd = new Socket(...);
  }

  public void close()
  {
    fd.flush();
    fd.close();
  }
}
```

Figure 9: IC: Incomplete construction of state variable fd

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 at all. Suppose that the designer of `AbstractFile` also designed and implemented `SocketFile`, i.e, he or she ensures that the data flow anomaly that exists in `AbstractFile` is avoided 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)

Consider the example in [Fig. 10.a], where the state variables in an ancestor class `A` are declared private, and a polymorphic method `A.m()` defines `A.v`. Furthermore, `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` [see Fig. 10.b]. 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 to be sure of avoiding a data flow anomaly 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.

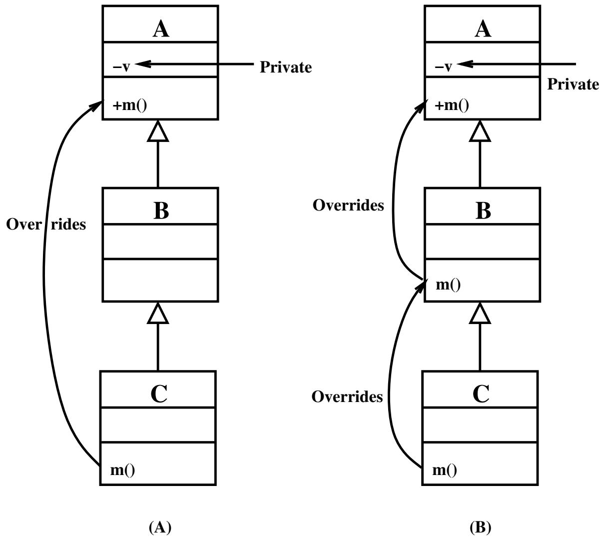


Figure 10: SVA: State visibility anomaly

4 Coupling, Testing, and Polymorphism

Our work in O-O testing builds on [Jin and Offutt, 1998], where the authors introduced a novel approach to integration testing of procedure-oriented software based on coupling relationships among procedures. Coupling was originally proposed to measure design [Constantine and Yourdon, 1979] [Page-Jones, 1980][Offutt et al., 1993], and the original papers presented up to twelve types of coupling in lists that were ordered in terms of estimated severity. Only three unordered types are needed for testing: parameter coupling, shared data coupling, and external device coupling. Parameter couplings occur whenever one procedure passes parameters to another, shared data couplings occur when two procedures reference the same global variable, and external device couplings occur when two procedures access the same external storage device. Jin and Offutt's approach, called coupling-based testing (CBT), is an application of data flow testing to the integration level. It requires that programs execute from definitions of a variable in a caller to a call site, and then to uses of the corresponding formal arguments in the called procedure. The execution path from the definition to the use must be definition-clear, that is, the variable must not be redefined along the path.

The following CBT definitions are taken from [Jin and Offutt, 1998]:

> V_P is the set of variables that are referenced by program component `P`, and N_P is the set of nodes in P's control flow graph. P_1 and P_2 are specific program units, and x and y are program variables. As in traditional data flow analysis, a path from node i to j is $def\text{-}clear$ with respect to x if there is no definition of x

along the path. A call site is a node $i \in N_{P_1}$ that contains a call from P_1 to P_2. A node $i \in N_{P_1}$ that contains a definition that can reach a use in P_2 on some execution path is a $coupling\text{-}def$. There are three kinds of $coupling\text{-}defs$: a definition of a formal parameter before a call ($last\text{-}def\text{-}before\text{-}call$), a definition of an actual parameter before a return ($last\text{-}def\text{-}before\text{-}return$), and a definition of a shared (global) variable ($shared\text{-}data\text{-}def$). A $coupling\text{-}use$ is a node $i \in N_m$ that contains a use that can be reached by a definition in another unit on at least one execution path. There are three kinds of $coupling\text{-}uses$: A use of a formal parameter after a call ($first\text{-}use\text{-}after\text{-}call$), a use of an actual parameter inside a callee ($first\text{-}use\text{-}in\text{-}callee$), and a use of a shared variable ($shared\text{-}data\text{-}use$). A coupling path between two program units is a path from a $coupling\text{-}def$ to a $coupling\text{-}use$. The path must be $def\text{-}clear$.

Traditional data flow and control flow criteria were adapted to specify four coupling-based testing criteria. If P_1 and P_2 are program units, then:

Call coupling: The set of paths executed by a test set must cover all $call\text{-}sites$.

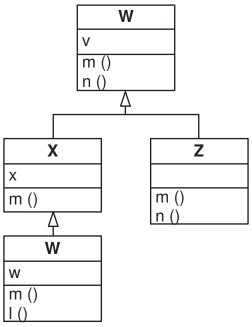
All-coupling-defs: For each $coupling\text{-}def$ of a variable in P_1, the set of paths executed by a test set must cover at least one coupling path to at least one reachable $coupling\text{-}use$.

All-coupling-uses: For each $coupling\text{-}def$ of a variable in P_1, the set of paths executed by a test set must cover at least one coupling path to each reachable $coupling\text{-}use$.

All-coupling-paths: For each $coupling\text{-}def$ of a variable in P_1, the set of tests executed must cover all coupling path sets from the $coupling\text{-}def$ to all reachable $coupling\text{-}uses$. A coupling path set is a set of nodes that can appear on sub-paths through a program unit between a $coupling\text{-}def$ and a $coupling\text{-}use$. This accounts for the case where the program unit has loops. Requiring that all coupling paths be covered is impractical in general; however, covering all coupling path sets ensures that each loop body is executed at least once, but does not require all possible executions.

4.1 Coupling in the Presence of Polymorphism

We extended CBT to apply the data flow criteria to address testing problems that arise from inheritance, polymorphism and dynamic binding. Identifying the definitions, uses and couplings is more complex, thus it is necessary to consider the semantics of the O-O language features very carefully. In the following definitions, o is an identifier whose type is a reference to an instance of an object, pointing to a memory location that contains an instance of some type. The reference o can only refer to instances whose actual instantiated types are either the base type of o or a descendant type.



(a)

Base Type	Type Family
W	{ W, X, Y, Z }
X	{ X, Y }
Y	{ Y }
Z	{ Z }

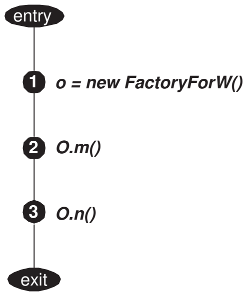
(b)

Figure 11: Sample class hierarchy (a) and associated type families (b)

Programmers can define new types in procedural languages such as C and Pascal and object-based languages such as Ada 83 and Modula-2. Strongly typed O-O languages such as Java, C++, and Ada 95 not only allow new types, but user defined types can be grouped into families of types. All members of a given type family share some common behavior, which allows instances of any member of a type family to be substituted for an instance of any other member. That is, we assume that instances of descendant types can be freely used in a context that expects an instance of a parent type [Liskov and Wing, 1994]. Every type definition by a class defines a type family. Members of the family include the base type of a hierarchy and all descendants of that base type. [Fig. 11.a] illustrates this with four type families, each defined by one of the classes in the hierarchy, and summarized in [Fig. 11.b].

O-O languages allow method calls both with and without respect to an instance. Instance methods are called with respect to instance variables, and class methods have no instance. Instance methods can make the instance explicit, as in `o.m()`, or implicit, as in `p()`. For the call `o.m()`, `m()` executes in the context of the instance that is bound to the reference `o`. For a shorthand convenience, we say that `m()` executes in the context of `o`, or `o` is `m()`'s instance context. There must be an implicit object reference for the call to `p()`, i.e., `p()` must appear in the program text of a method that was called through an explicit instance, and `p()` must be defined in the same class with `o`. An object instance `o` is considered to be defined when one of the state variables `v` of the object is defined. An indirect definition, or $i\text{-}def$, occurs when a method `m` defines `v`. Similarly, an indirect use ($i\text{-}use$) occurs when `m` references the value of `v`.

Again, consider the class diagram shown in [Fig. 11] and assume that `W` includes a method `FactoryForW()` that returns an instance of `W`. [Fig. 12.a] shows a control flow fragment with an instance of `W` bound to `o`. This is a local definition of the object reference `o` that results from the call to the method `FactoryForW()`. The table in [Fig. 12.b] shows that `W.m()` defines `v`, so an indirect definition occurs at node 2



(a)

Indirect definitions and uses for type family *W*		
Method	**Definitions**	**Uses**
W.m	$\{W_r.v\}$	
W.n		$\{W_r.v\}$
X.m	$\{W_r.v, X_r.x\}$	
Y.m	$\{W_r.v, X_r.w\}$	
Y.l		$\{W_r.w\}$
Z.m	$\{W_r.v\}$	
Z.n		$\{W_r.v\}$

(b)

Figure 12: Control flow graph fragment (a) and associated definitions and uses (b)

through `o.m()`. Thus, any call to `m()` with respect an instance of `W` bound to `o` results in an indirect definition of the state the object bound to `o`. Note that there are no indirect uses by `m()`. The table in [Fig. 12.b] also shows all of the indirect definitions and uses that can occur for any instance that is a member of the type family defined by `W`. Node 3 contains no $defs$, but an $i\text{-}use$ of `v`.

The most difficult part of the problem of data flow analysis of O-O programs is the static non-determinism introduced by polymorphism and dynamic binding. Polymorphism allows one call to refer to multiple methods, depending on the current type of the object reference, and dynamic binding means that we cannot know which method is called until run-time. When discussing the indirect definitions and uses that can occur at call sites through object references, we must also consider the set of methods that can potentially execute. That set depends on the type of the instance that is bound to the object reference. However, a key insight is that the set of potential methods is finite and can be determined statically. To analyze this, we introduce the term satisfying set:

Definition 1. The satisfying set of a polymorphic call to method `m()` through an object reference o contains all methods that override `m()`, plus `m()` itself.

When considering the set of indirect definitions or indirect uses that can occur at a call site, it is necessary to determine which methods can satisfy the call. For each such method, identify all state variables that are defined and used. The result is the set of definitions and uses for each satisfying method. Returning again to [Figs. 11 and 12.a], the satisfying set for the call to `m()` at node 2 is $\{$`W.m()`, `X.m()`, `Y.m()`, `Z.m()`$\}$ and the $i\text{-}def$ set is the following set of ordered pairs $i\text{-}def$(2, `o`, `m()`) = $\{$(`W.m()`, $\{$`W.v`$\}$), (`X.m()`, $\{$`W.v`, `X.x`$\}$), (`Y.m()`, $\{$`W.v`, `Y.w`$\}$), (`Z.m()`, $\{$`W.v`$\}$)$\}$. Each pair indicates a satisfying method for `m()` and the corresponding set of state variables that the method defines. In this example, `X.m()` defines state variables `v` from class `X` and `x` from `W`.

From [Fig. 12.b], the $i\text{-}use$ set for node 2 is the empty set, as none of the satisfying methods for `m()` reference any state variable. However, considering node 3, the table shows that there are two methods that satisfy the call to `o.n()` and have non-empty $i\text{-}use$ sets (but their $i\text{-}def$ sets are empty), which yields the following $i\text{-}use$ set: $i\text{-}use$(3, `o`, `n()`) = {(`W.n()`, {`W.v`}), (`Z.n()`, {`W.v`})}.

4.2 Differences in Coupling Paths in O-O Programs

From a coupling and integration perspective, the two primary issues are determining what calls can be executed in the presence of inheritance and polymorphism, and what effects the calls have on the corresponding state space. In [Alexander, 2001], the calls are categorized into twelve separate cases, which are summarized here in three categories. Most method calls in O-O programs are made through explicit instance contexts or implicit instance contexts as with `o.m()`. These calls allow polymorphic behavior because the instance `o` may be bound to a different type on different executions. These are called Category I calls. It is also possible to make method calls in the absence of an instance, for example when using a static method in Java. If a static method call cannot be polymorphic, it is a Category II call. If a static method call can be polymorphic, it is a Category III call. Category II cases are handled by the original CBT definitions. Category I and Category III cases, which involve polymorphism, require extensions to the definitions and additional analysis techniques.

5 Polymorphism and Coupling-based Testing

The original coupling-based testing definitions in [Jin and Offutt, 1998] were extended in [Alexander and Offutt, 1999] to account for the various calling contexts that occur in O-O programs. In the following definitions, $m()$ refers to a program unit, including methods that appear in class specifications. V_m is the set of variables that are referenced by $m()$, and N_m the set of nodes in the control flow graph for $m()$. Each definition is expressed as a function whose domain is given by a set of formal arguments and a range given as a return type. As is usual with data flow analysis [Frankl and Weyuker, 1988][Rapps and Weyuker, 1985], $defs(i)$ is the set of variables defined at node i and $uses(i)$ is the set of variables used. $entry(m)$ refers to the entry node of method $m()$, $code(m)$ refers to the exit node, $first(p)$ refers to the first node in path p, and $last(p)$ refers to the last node in p.

The following definitions are introduced to deal with inheritance and polymorphism: the set of classes that belong to the same type family specified by c is $family(c)$, where c is the base ancestor class. $type(m)$ is the class that defines method $m()$ and $type(o)$ is the class c that is the declared type of variable o. o must refer to an instance of a class that is in the type family of c. $state(c)$ is the set of state variables for class c, either declared in c or inherited from an ancestor. $i\text{-}defs(m)$ is the set of variables that are indirectly defined within $m()$ and $i\text{-}uses(m)$ is the set of variables used by $m()$.

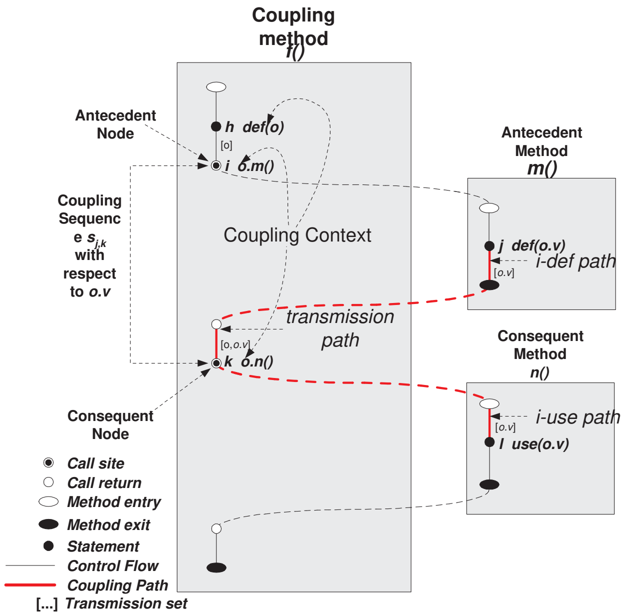


Figure 13: Control flow schematic for prototypical coupling sequence

5.1 Coupling Sequences

To allow for inheritance and polymorphism, we consider the case where two methods that define and use the same variable are called by a third method. Both calls must be made through the same instance, and this is said to make a coupling sequence. The calling method is called the coupling method $f()$, and it calls $m()$, the antecedent method to define x, and $n()$, the consequent method to use x. Programs in which the antecedent or consequent method is the same as the coupling method are special cases and are handled implicitly. (Programs in which the antecedent or consequent methods are called from another method that is called by $f()$ have not yet been considered.)

Coupling sequences are pairs of nodes within the body of $f()$. The control flow schematic shown in [Fig. 13] illustrates our prototypical situation where the coupling method calls both the antecedent and consequent methods. The schematic abstracts away the details of the control flow graph and shows only nodes that are relevant to coupling analysis. The thin line segments represent control flow and the thicker lines indicate control flow that is part of a coupling path. The line segments can represent multiple sub-paths. A path may be annotated with a transmission set such as $[o, o.v]$, which contains variables for which the path is definition-clear. Assuming that the intervening sub-paths are def-$clear$ with respect to the state variable $o.v$, the path in [Fig. 13] from h to i to j to k and finally l forms a transmission path with respect to $o.v$. The object o is called the context variable. Formal derivations of coupling sequences for the prototypical situation and special cases were presented in [Alexander, 2001].

5.2 Coupling Variables and Coupling Sets

Every coupling sequence $s_{j,k}$ has an associated set of state variables that are defined by the antecedent method and then used by the consequent method with respect to the coupling type t. This set of variables is referred to generically as the coupling set $\Theta^t_{s_{j,k}}$ of $s_{j,k}$ and is defined as the intersection of those variables defined by $m()$ (an i-def) and used by $n()$ (an i-use) through the instance context provided by a context variable o that is bound to an instance of t. Note that which versions of $m()$ and $n()$ execute is determined by the actual type t of the instance bound to o. The members of the coupling set are called coupling variables.

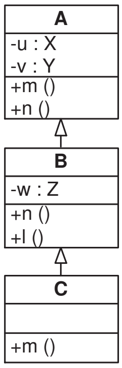
5.3 Coupling Paths

Coupling sequences require that there be at least one def-$clear$ path between each node in the sequence. Identifying these paths as parts of complete sequences of nodes results in the set of coupling paths. A coupling path is considered to transmit a definition of a variable to a use. Each path consists of up to three sub-paths, or segments. The i-def sub-path is the portion of the coupling path that occurs in the antecedent method $m()$, extending from the last (indirect) definition of a coupling variable to the exit node of $m()$. The i-def sub-path is the portion of the consequent method $n()$ that extends from the entry node of $n()$ to the first (indirect) use of a coupling variable. The transmission sub-path is the portion of the coupling path in the coupling method that extends from the antecedent node to the consequent node, with the requirement that neither the value of the coupling variable nor the context variable is modified.

For a given coupling sequence, there is a single set of coupling paths for each type of coupling sub-path. These sets are used to form coupling paths by matching together elements of each set. The set of coupling paths is formed by combining elements of the i-def sub-path set with an element from the transmission sub-path set, and then adding an element of the i-use sub-path set. The complete set of coupling paths is formed by taking the cross product of these sets.

5.4 The Effects of Inheritance and Polymorphism on Coupling

To see the effects of inheritance and polymorphism on path sets, consider the class diagram shown in [Fig. 14.a]. The type family contains the classes `A`, `B`, and `C`. Class `A` defines methods `m()` and `n()` and state variables `u` and `v`. Class `B` defines method `l()` and overrides `A`'s version of `n()`. Likewise, `C` overrides `A`'s version of `m()`. Definitions and uses for each of these methods are shown in [Fig. 14.b]. [Fig. 15] shows coupling paths assuming a coupling method that uses the hierarchy in [Fig. 14.a]. [Fig. 15.a],

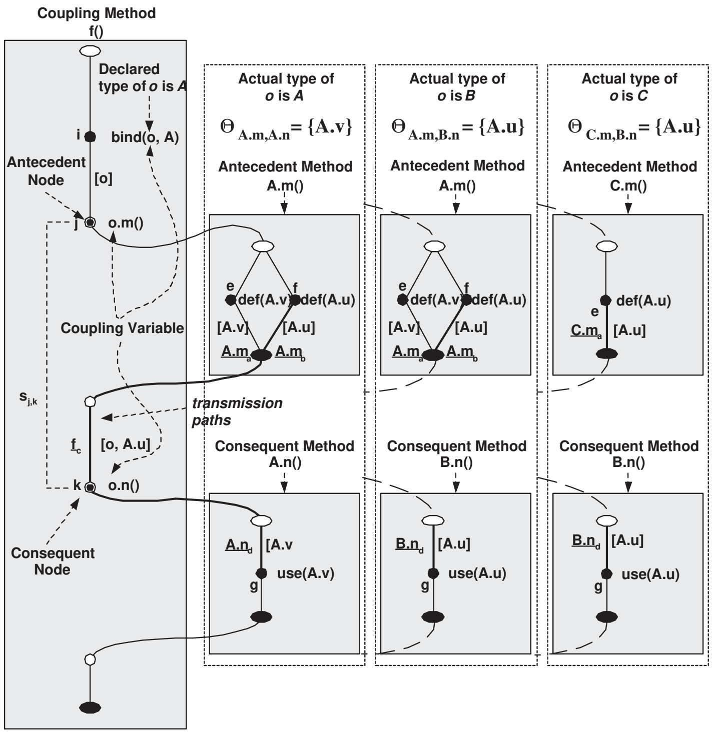


(a)

Method	Defs	Uses
A.m ()	{ A.u, A.v }	
A.n ()		{ A.v }
B.n ()		{ A.u }
B.l ()		{ A.v }
C.m ()	{ A.u }	

(b)

Figure 14: Sample class hierarchy and def-use table



(a) (b) (c) (d)

Figure 15: Coupling sequence when o is declared as type A (a), bound to an instance of A (b), B (c) and C (d)

Type	Coupling Path
A	$\langle \texttt{A.m}(e), \texttt{A.m}(exit), \texttt{f}(j.return), \texttt{f}(k.call), \texttt{A.n}(entry), \texttt{A.n}(g)\rangle$
B	$\langle \texttt{A.m}(e), \texttt{A.m}(exit), \texttt{f}(j.return), \texttt{f}(k.call), \texttt{B.n}(entry), \texttt{B.n}(g)\rangle$
C	$\langle \texttt{C.m}(e), \texttt{C.m}(exit), \texttt{f}(j.return), \texttt{f}(k.call), \texttt{B.n}(entry), \texttt{B.n}(g)\rangle$

Table 2: Summary of sample coupling paths

shows the declared type of the coupling variable `o` is `A`, and [Fig. 15.b] shows the antecedent and consequent methods when the actual type is also `A`. The coupling sequence $s_{j,k}$ extends from the node j where the antecedent method `m()` is called to the call site of the consequent method at node k. As shown, the corresponding coupling set for $s_{j,k}$ when `o` is bound to an instance of `A` is $\Theta^{A}_{s_{j,k}} = \{\texttt{A.v}\}$. Thus, the set consists of the coupling paths for $s_{j,k}$ that extend from node e in `A.m()` to the exit node of in `A.m()`, back to the consequent node k in the coupling method, and through the entry node of `A.n()` to node g. There is no coupling path with respect to `A.u` because `A.u` does not appear in the coupling set for `A.m()` and `A.n()`.

Now, consider the effect on the elements that comprise the set of coupling paths when `o` is bound to an instance of `B`, as shown in [Fig. 15.c]. The coupling set for this case is different from when `o` was bound to an instance of `A`. This is because `B` provides an overriding method `B.n()` that has a different use set than the overridden method `A.n()`. Thus, the coupling set is different with respect to the antecedent method `A.m()` and the consequent method `B.n()` yielding $\Theta^{B}_{s_{j,k}} = \{\texttt{A.u}\}$. In turn, this results in a different set of coupling paths. The set of coupling paths now extends from node f in `A.m()` back through the call site at node k in the coupling method, and through the entry node of `B.n()` to node g of `B.n()`. Finally, [Fig. 15.c] depicts the coupling sequence that results when `o` is bound to an instance of `C`. First, observe that execution of node j in the coupling method results in the invocation of the antecedent method, which is now `C.m()`. Likewise, execution of node k results in the invocation of the consequent method `n()`. Since `C` does not override `m()` and because `C` is a descendant of `B`, the version of `n()` that is invoked is actually `B.n()`. Thus, the coupling set for $s_{j,k}$ is taken with respect to the antecedent method `C.m()` and the consequent method `B.n()`, which yields $\Theta^{C}_{s_{j,k}} = \{\texttt{A.u}\}$. The corresponding coupling path set includes the paths that begin at node e in `C.m()` and extend to the exit node of `C.m()`, then back to node j of the coupling method, and through the entry node of `B.n()` to node g, also in `B.n()`.

[Tab. 2] summarizes the coupling paths for the examples shown in [Fig. 15]. Paths are represented as sequences of nodes. Each node is of the form $method(node)$, where $method$ is the name of the method that contains the node, and $node$ is the node identifier within the method. Note that the prefixes call- or return- are appended to the names of nodes that correspond to call or return sites.

It is possible that some classes in a type family will not be in any coupling sequences. This happens when the class does not override any methods that are called in a coupling sequence. Thus, as an optimization, we can safely ignore consideration of such classes from the coupling analysis. This is possible since any coupling path that could be executed through such a class will necessarily be in the coupling paths of other ancestor classes.

5.5 Polymorphic Coupling Sequences and Coupling Sets

Inheritance increases the number of potential bindings for a given coupling sequence. When combined with polymorphism, the number of methods that execute and the variables indirectly defined and used can vary at run-time. Since the depth and breadth of inheritance is always finite, the actual methods and variables referenced can be tightly bound. They are limited by the members of the type family of the declared type. The following subsections present modified definitions of coupling sequences and coupling sets that take polymorphism into account.

Polymorphic Coupling Sequences: To account for the possibility of polymorphic behavior at a call site, the definition of a coupling sequence given in [Section 5.1] must be changed to handle all methods that can execute. To accomplish this, we introduce the notion of a binding triple, which consists of the antecedent method $m()$, the consequent method $n()$, and the set of coupling variables that result from the binding of the context variable to an instance of a particular type. The triple matches together a pair of methods $p()$ and $q()$ that can potentially execute as the result of executing the antecedent and consequent nodes j and k. Neither method is required to be from the class c that provides the instance context for the coupling sequence. Each may be from different classes that are members of the type family defined by c, provided that $p()$ is an overriding method for $m()$ or $q()$ is an overriding method for $n()$. Note that there will be exactly one binding triple for each class d in $family(c)$ that defines an overriding method for either $m()$ or $n()$. Classes that do not define such overriding methods are excluded. A coupling sequence induces a set of binding triples. This set always includes the binding triple that corresponds to the antecedent and consequent methods, even when there is no method overriding. In this case, the only member of the binding triple set will be the declared type of the context variable, assuming the type is not abstract. If the type is abstract, an instance of the nearest concrete descendant to the declared type is used.

As an example, the set of binding triples for the coupling sequence $s_{j,k}$ shown in [Fig. 15] is given in [Tab. 3]. The first column gives the type `t` of the context variable of $s_{j,k}$, the next two columns correspond to the antecedent and consequent methods that actually execute for a particular `t`, and the final column gives the set of coupling variables induced when the context variable is bound to an instance of `t`. The type hierarchy corresponding to the coupling type `t` is shown in [Fig. 14].

t	p	q	S
A	A.m()	A.n()	{A.v}
B	A.m()	B.n()	{A.u}
C	C.m()	B.n()	{A.u}

Table 3: Binding triples for coupling sequence from class hierarchy in [Fig. 14]

Polymorphic Coupling Sets: The original definition of a coupling set only considers the antecedent and consequent methods in the context of the coupling variable's declared type. Inheritance and polymorphism makes this insufficient. Instead, the coupling sets of all possible method combinations must be combined to form an aggregate coupling set for the sequence. Thus, the polymorphic coupling set is defined as the union of all the coupling sets for each binding triple. The coupling set for a sequence is the union of all the coupling sets for the individual pairs of methods that could potentially execute through the call sites at the antecedent and consequent nodes.

5.6 Coupling Paths in O-O Programs

Procedure-oriented programs have couplings that occur between procedures in terms of parameters or through shared global data [Jin and Offutt, 1998]. O-O programs contain coupling paths that originate at $last\text{-}definitions$ in an antecedent method and that terminate at $first\text{-}uses$ in a consequent method. There are two general cases in which coupling paths can occur. The first is when there is no possibility of polymorphic behavior at the call sites. In this case, the methods that execute are specified by the declared type of the context variable. Second is when there is a possibility of polymorphic behavior at the call sites. Polymorphic behavior means it is not possible to statically determine which methods will execute. However, it is possible to statically determine the set of methods that can execute. The following subsections discuss the coupling paths that result from each of these cases.

Non-Polymorphic Coupling Paths: Consider again the Type I coupling sequence in [Fig. 13] where the body of method `f()` contains an object reference `o` of declared type `T`. Assume that `o` is bound to an instance whose actual type is `T`. There is no possibility of polymorphic behavior when the declared and actual types are the same. An instance coupling occurs wherever an object reference is used to access methods or state variables of an instance.

If we ignore polymorphism, we are interested in all of the indirect definitions that can reach indirect uses with respect to a particular instance context. Thus, we desire to identify all non-polymorphic coupling paths that extend from a node containing a $last\text{-}def$ in an antecedent method to a node in a consequent method that contains a

$first\text{-}use$ with respect to the coupling variable of interest. Collectively, this set of paths is the coupling path set for the coupling sequence $s_{j,k}$. We form these paths by taking the cross product of the $i\text{-}def$ path set, the $t\text{-}path$ set, and the $i\text{-}use$ path set for a coupling sequence. Each non-polymorphic coupling path is formed by concatenating a single path p from each of the coupling path segments ($i\text{-}def\text{-}paths$, $t\text{-}paths$, and $i\text{-}use\text{-}paths$), subject to the constraint that p be $definition\text{-}clear$ with respect to a particular coupling variable v.

Polymorphic Coupling Paths: The instance coupling paths above do not allow for polymorphic behavior when the actual type differs from the declared type. This requires that an instance coupling results in one path set for each member of the type family. The number of paths is limited by the number of overriding methods, either defined directly or inherited from another type. The polymorphic coupling paths are formed by considering each binding triple.

6 O-O Testing Criteria

The analysis in [Section 5] allows coupling definitions and uses to be identified in the presence of inheritance and polymorphism. This information is used to support testing by adapting the data flow criteria to define sub-paths in the program to be tested (see [Frankl and Weyuker, 1988], [Harrold and Rothermel, 1994], [Jin and Offutt, 1998] or [Rapps and Weyuker, 1985]). Testing criteria can be used to help testers generate tests (test generation), or to measure the quality of pre-existing tests (coverage analysis). This work currently assumes the criteria will be used as coverage analyzers, i.e., tests already exist. When using the testing criteria, it is assumed that the antecedent and consequent methods have been tested individually before the method containing the coupling sequence. This allows the developer to assume that any discovered failures are related to the interfaces.

6.1 O-O Coupling Criteria

In this section, we present four coupling-based test adequacy criteria for O-O programs [Alexander and Offutt, 2000][1]. The first criterion builds on an assumption that each coupling sequence should be covered during integration testing. It requires that every coupling sequence in $f()$ be covered by at least one test case.

Definition 2 All-Coupling-Sequences (ACS). For each coupling sequence $s_{j,k}$ in $f()$, there is at least one test case $t \in T_{s_{j,k}}$ such that there is a coupling path induced by $s_{j,k}$ that is a sub-path of the execution trace of $f(t)$.

[1] In the definitions, $f()$ represents a method being tested, $s_{j,k}$ is a coupling sequence in $f()$, where j and k are nodes in the control flow graph of $f()$, and $T_{s_{j,k}}$ represents a set of test cases created to satisfy $s_{j,k}$.

ACS does not consider inheritance or polymorphism, so this criterion is added to include instance contexts of calls. This is achieved by ensuring there is at least one test for every class that can provide an instance context for each coupling sequence. The idea is that the coupling sequence should be tested with every possible type substitution that can occur in a given coupling context. Thus, this criterion requires that, for each $f()$, there is at least one test case t for every combination $(s_{j,k}, c)$, where c is in the type family defined by the instance context of $s_{j,k}$. The combination $(s_{j,k}, c)$ is feasible if and only if c is the same as the declared type of the context variable for $s_{j,k}$, or c is a child of the declared type and defines an overriding method for the antecedent or consequent method. That is, only classes that override the antecedent or consequent methods are considered.

Definition 3 All-Poly-Classes (APC). For each coupling sequence $s_{j,k}$ in method $f()$, and for every class in the family of types defined by the context of $s_{j,k}$, there is at least one test case t such that when $f()$ is executed using t, there is a path p in the set of coupling paths of $s_{j,k}$ that is a sub-path of the execution trace of $f(t)$.

ACS requires that coupling sequences be covered but does not consider the state interactions that can occur when multiple coupling variables are involved. Thus some definitions or uses of coupling variables may not be covered during testing. This criterion addresses these limitations by requiring that every $last\text{-}def$ of a coupling variable v in an antecedent method of $s_{j,k}$ reaches every first use of v in a consequent method of $s_{j,k}$. Thus, there must be at least one test case that executes each feasible coupling path p with respect to each coupling variable v[2]. That is, every feasible coupling path between each coupling-definition and coupling-use pair for v must be executed by at least one test case.

Definition 4 All-Coupling-Defs-Uses (ACDU). For every coupling variable v in each coupling $s_{j,k}$ of t, there is a coupling path p induced by $s_{j,k}$, such that p is a sub-path of the execution trace of $f(t)$ for at least one test case $t \in T_{s_{j,k}}$.

APC requires multiple instance contexts to be used, and ACDU requires definitions to reach uses. The final criterion merges these requirements. In addition to inheritance and polymorphism, this criterion requires that all coupling paths be executed for every member of the type family defined by the context of a coupling sequence.

Definition 5 All-Poly-Coupling-Defs-Uses (APDU). For each coupling sequence $s_{j,k}$ in method $f()$, for every class in the family of types defined by the context of $s_{j,k}$, for every coupling variable v of $s_{j,k}$, for every node m that has a $last\text{-}def$ of v and every node n that has a $first\text{-}use$ of v, there is at least one test case t such that when $f()$ is executed using t, there is a path p in the coupling paths of $s_{j,k}$ that is a sub-path of the trace of $f()$.

[2] Coupling path p is not feasible if no input exists that results in the execution of p.

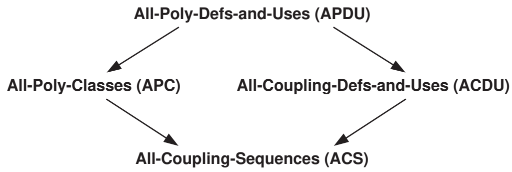


Figure 16: O-O coupling testing subsumption hierarchy

6.2 Subsumption of the Criteria

A well-known method to compare different testing criteria is the subsumption relationship [Frankl and Weyuker, 1988][Weiss, 1989][Zhu, 1996]. Criterion A subsumes criterion B if and only if every test set that satisfies A also satisfies B. Although there are certainly exceptions, the common assumption is that criteria at higher levels in a subsumption hierarchy have more testing power, but come at a higher cost. The subsumption hierarchy for the criteria presented in this paper is shown in [Fig. 16].

7 Evaluation

This section summarizes some of the experimental results. One controlled experiment evaluated ACS, APC and APDU with Branch Coverage used as the control. Branch testing is a unit-level white box testing technique, and seeks to execute enough tests to assure that every branch alternative has been executed at least once [Beizer, 1990].

7.1 Experimental Design

There are many questions about the efficacy of these criteria. The most immediate question is whether tests derived from these criteria will help software testers find O-O faults. Beyond that, it is important to understand whether they will help find faults better than existing techniques do, and also how much difference there is among the criteria. The following sections describe the design of an experiment to address these questions.

Subject Programs: The subject programs are collections of classes that are integrated with a client method under test. Each class includes at least one method that has one or more coupling sequences with respect to a particular class hierarchy, referred to as the subject hierarchy. [Tab. 4.a] summarizes the subject programs used in these experiments. Column "`f`" identifies the method under test, and $|s_{j,k}|$ is the number of coupling sequences contained within `f`. Each coupling sequence has a context variable that defines a type family. Column "$F_{s,f}$" gives the number of classes in the type family (inheritance hierarchy) for the program. The term program includes `f` (the method

f	$(\|s_{j,k}\|)$	F_{s_f}	**Description**
P_1	4	4	Polymorphic Example
P_2	5	5	Polymorphic Example
P_3	1	5	Polymorphic Example
P_4	1	4	Student Developer
P_5	3	4	Polymorphic Example
P_6	3	5	Polymorphic Example
P_7	6	4	Professional Developer
P_8	20	5	Professional Developer
P_9	11	16	Open Source (ANTLR)
P_{10}	7	9	Open Source (JMK)

(a)

f	ACS	APC	APDU	BC
P_1	2	4	6	1
P_2	2	5	320	2
P_3	2	5	80	2
P_4	1	3	3	1
P_5	2	5	75	1
P_6	2	5	105	1
P_7	1	2	64	1
P_8	4	2	42	4
P_9	6	15	95	6
P_{10}	4	9	27	4
Total	26	55	817	23

(b)

Table 4: Subject program characteristics (a) and number of test cases per subject program and criterion (b)

under test), the class that specifies `f`, and all classes in the type family specified by the context variable of each coupling sequence. Column "Description" indicates the source from which each program was obtained. Five programs (P_1, P_2, P_3, P_5, and P_6) were examples created specifically to ensure that all of the subject faults were tested by at least one experiment. Of the remaining five subject programs, one was developed by a graduate student (P_4), two were developed by a professional programmer with 15 years of experience (P_7 and P_8) and the others are open source products: ANTLR (a parser generator) and JMK (a build system, similar to make).

Test Data: The test data used in the experiments were created randomly according to a uniform distribution. The data itself was produced from custom test data generators developed in Perl for each criterion. Enough tests were generated to achieve 100% coverage for each criterion. The strategy used to select test cases is similar to how test cases are normally selected for the Branch Coverage test adequacy criterion. For each coupling sequence, the path expression [Beizer, 1990] necessary to execute the sequence was identified. These expressions were then used to create Perl programs that would generate the test data necessary to execute the set of sequences for the method under test. A similar procedure was followed to test the state space interactions between antecedent and consequent methods. These path expressions ensured that the required coupling paths were covered.

[Tab. 4.b] summarizes the number of test cases for each combination of subject program and test adequacy criterion. For ACS, the number of test cases is determined by the number of coupling sequences and control flow paths present in the method under test. For APC, the number of test cases is also determined by the size of the type family for the coupling variable. Finally, for APDU, it is determined by adding the number of control flow paths in the antecedent and consequent methods to the test cases for APC and ACS.

7.2 Conduct of Experiments

The testing and evaluation procedure consisted of four steps: the first step created a test oracle that can be used to evaluate the results of subsequent tests; the second step injected faults into each subject program; the third step executed each subject program on each test case; the final step used the test oracle to determine if the outcome of each execution for the corresponding test case detected a fault.

Test Oracle Derivation: For each combination of subject program and criterion (f, C) and each coupling sequence $s_{j,k}$ in f, the following procedure was used:

1. Execute f using at least one test case $c \in S_{C,f}$ taken from the test set $S_{C,f}$, such that the context variable o of $s_{j,k}$ is bound to an instance of the declared type of o.
2. Record this result and add it to the test oracle for f, Ω_f.
3. For the $All\text{-}Poly\text{-}Classes$ and $All\text{-}Poly\text{-}Def\text{-}Uses$ execute f with at least one test case $c \in S_{C,f}$ for each combination of (t, v, d_v, u_v, p), where t is a descendant of the declared type of the context variable of $s_{j,k}$, v is a variable in $s_{j,k}$'s coupling set, d_v is a last definition of v by the antecedent method of $s_{j,k}$, u_v is a first use of v in the consequent method of $s_{j,k}$, and p is a $def\text{-}clear$ path from d_v to u_v.
4. For $All\text{-}Poly\text{-}Classes$ and $All\text{-}Poly\text{-}Def\text{-}Uses$ record the combination of t and v in Ω_f. Also, for $All\text{-}Poly\text{-}Def\text{-}Uses$, include the state of the instance bound to the context variable after execution of the antecedent method and immediately after each $first\text{-}use$ in the consequent method.

Fault Injection: For each coupling sequence $s_{j,k}$ in a subject program f and each type t that is a subtype of the declared type T of $s_{j,k}$'s coupling variable, the following procedure was used:

1. Inject faults into each method of t that overrides the antecedent or consequent methods of $s_{j,k}$. This yields the fault-seeded subtype of T and results in a shadow inheritance hierarchy rooted at T [see Fig. 17]. The shadow hierarchy mirrors the original hierarchy in structure below the root, but is seeded with faults.
2. For $All\text{-}Poly\text{-}Def\text{-}Uses$, for each coupling variable in $s_{j,k}$, inject faults into the antecedent and consequent methods, yielding the fault seeded type t'' (also a subtype of T). This results in a shadow inheritance hierarchy rooted at T [see Fig. 18].

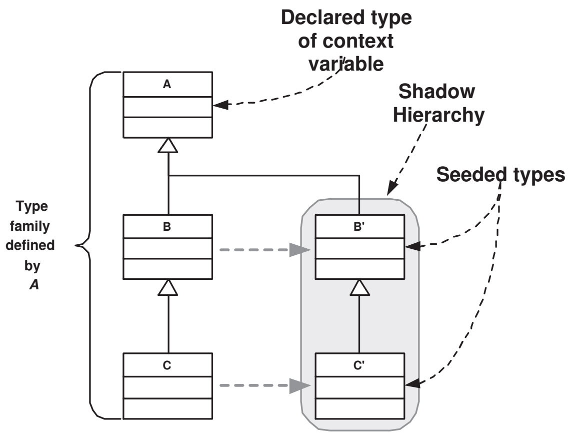


Figure 17: Seeded shadow hierarchy

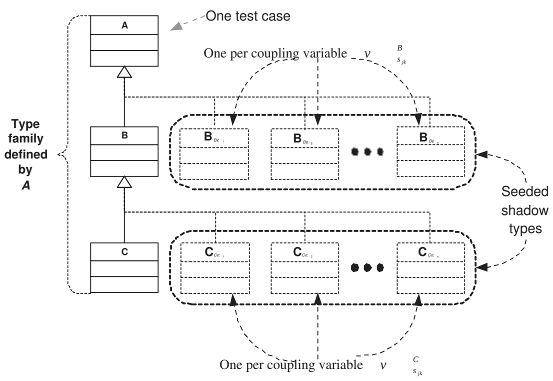


Figure 18: Seeded shadow hierarchy for $All\text{-}Poly\text{-}Def\text{-}Uses$

Test Execution: For each coupling sequence $s_{j,k}$ in f, and each type t that is a subtype of the declared type T of $s_{j,k}$'s coupling variable:

1. Execute f using a test case c that binds $s_{j,k}$'s context variable to the corresponding fault-seeded type t'. Record the result in the test result set for f, Ψ_f.

2. For each test case $c \in S_{C,f}$ execute f using c, and record the state of the instance bound to $s_{j,k}$'s context variable for the pairs of last-definitions and first-uses of each coupling variable. Add this result to Ψ_f.

Result Evaluation: For each coupling sequence $s_{j,k}$ in f, and each type t that is a subtype of the declared type T of $s_{j,k}$'s coupling variable:

1. Compare each test result in Ψ_f with the corresponding result in the test oracle Ω_f. If the two results are equal, then the test passed. This ascertains whether or not an instance of the descendant type t can be substituted freely for an instance of the declared type T of the context variable.

2. For All-$Poly$-Def-$Uses$, compare each test result in Ψ_f for each coupling variable v in $s_{j,k}$ with the corresponding pair in the test oracle Ω_f. If the results are equal, the test passed. This ascertains if the method under test preserves the fidelity of the interactions between the antecedent and consequent methods when the context variable o is bound to an instance of a particular type that is a subtype of the declared type of o.

7.3 Experimental Results

The results for the experiments are shown in [Tab. 5]. For each of the five fault types, the columns show the number of faults seeded, the number of faults detected, and the detection effectiveness (percentage found). The last column presents the average number of faults found over the five types. The rows represent the criteria applied to each program. For example, for P_1, APDU found 82% of all faults, while BC did not find any. Some subject programs did not exhibit the structural characteristics necessary to support the syntactic pattern for the fault type. Empty cells represent combinations of programs and fault types that were not tested. The last group of rows in the table summarizes the total number of faults that were seeded, the total number of faults detected, and the average detection effectiveness for each criterion.

[Fig. 19] shows a plot of the detection effectiveness per criterion for each fault type averaged (using the mean) over all programs. The individual data points were weighted to reflect the differences in the number of faults seeded for each combination of program and test adequacy criterion. Thus, the data points are comparable. A cursory examination of the plot reveals that the most effective coupling-based test adequacy criterion is All-$Poly$-Def-$Uses$, which has average detection effectiveness across fault types of $\overline{X}_{APDU} = 0.82$. The other coupling-based criteria have average detection effectiveness of 0.63 (APC) and 0.37 (ACS), with Branch Coverage having the lowest detection effectiveness of 0.12.

All three of the coupling-based testing criteria exhibit a similar fault detection pattern. For example, they do reasonably well at detecting faults of type IC and IIS, with the corresponding detection effectiveness across this sequence being monotonically increasing. In contrast, all three are much less effective at detecting faults of type DSDA, SDI, and SDIH. Note that in all cases, across all fault types, all four criteria appear

Program	Criterion	Faults Seeded					Faults Detected					Detection Effectiveness					Average
		SDA	IC	SDI	IISD	SDIH	SDA	IC	SDI	IISD	SDIH	SDA	IC	SDI	IISD	SDIH	
P1	APDU	9		6	3	3	7	0	3	3	3	0.78		0.50	1.00	1.00	0.82
	ACS	9		6	3	3	7	0	3	3	3	0.78		0.50	1.00	1.00	0.82
	APC	9		6	3	3	7	0	3	3	3	0.78		0.50	1.00	1.00	0.82
	BC	9		6	3	3	0	0	0	0	0	0.00		0.00	0.00	0.00	0.00
P2	APDU	39	6	39		39	10	3	10		10	0.26	0.50	0.26		0.26	0.32
	ACS	39	6	39		39	0	0	0		0	0.00	0.00	0.00		0.00	0.00
	APC	39	6	39		39	5	3	1		3	0.13	0.50	0.03		0.08	0.18
	BC	39	6	39		39	8	0	9		9	0.21	0.00	0.23		0.23	0.17
P3	APDU	36	3	33		36	36	3	30		36	1.00	1.00	0.91		1.00	0.98
	ACS	36	3	33		36	7	3	3		7	0.19	1.00	0.09		0.19	0.37
	APC	36	3	33		36	9	3	5		12	0.25	1.00	0.15		0.33	0.43
	BC	36	3	33		36	0	0	0		0	0.00	0.00	0.00		0.00	0.00
P4	APDU	24		24		18	11		12		8	0.46		0.50		0.44	0.47
	ACS	24		24		18	0		4		0	0.00		0.17		0.00	0.06
	APC	24		24		18	11		12		8	0.46		0.50		0.44	0.47
	BC	24		24		18	5		5		2	0.21		0.21		0.11	0.18
P5	APDU	36	3	36		36	36	3	31		33	1.00	1.00	0.86		0.92	0.94
	ACS	36	3	36		36	7	0	8		6	0.19	0.00	0.22		0.17	0.15
	APC	36	3	36		36	8	3	10		7	0.22	1.00	0.28		0.19	0.42
	BC	36	3	36		36	0	0	0		0	0.00	0.00	0.00		0.00	0.00
P6	APDU	18		18		18	18		13		18	1.00		0.72		1.00	0.91
	ACS	18		18		18	0		0		0	0.00		0.00		0.00	0.00
	APC	18		18		18	13		13		16	0.72		0.72		0.89	0.78
	BC	18		18		18	0		0		0	0.00		0.00		0.00	0.00
P7	APDU			55		30			37		26			0.67		0.867	0.77
	ACS			55		30			32		26			0.58		0.867	0.72
	APC			55		30			34		26			0.62		0.867	0.74
	BC			55		30			14		8			0.25		0.267	0.26
P8	APDU			76		30			34		23			0.45		0.767	0.61
	ACS			76		30			5		2			0.07		0.067	0.07
	APC			76		30			12		2			0.16		0.067	0.11
	BC			76		30			30		21			0.39		0.7	0.55
P9	APDU	42		42	12	42	38		37	12	39	0.90		0.88	1.00	0.93	0.93
	ACS	42		42	12	42	4		10	7	15	0.10		0.24	0.58	0.36	0.32
	APC	42		42	12	42	15		26	12	31	0.36		0.62	1.00	0.74	0.68
	BC	42		42	12	42	3		9	2	5	0.07		0.21	0.17	0.12	0.14
P10	APDU	27		27	6	27	27		26	6	23	1.00		0.96	1.00	0.85	0.95
	ACS	27		27	6	27	6		12	5	7	0.22		0.44	0.83	0.26	0.44
	APC	27		27	6	27	12		17	6	8	0.44		0.63	1.00	0.30	0.59
	BC	27		27	6	27	4		7	3	5	0.15		0.26	0.50	0.19	0.27
Summary	APDU	231	12	356	21	279	183	9	233	21	219	0.80	0.83	0.67	1.00	0.80	0.82
	ACS	231	12	356	21	279	31	3	77	15	66	0.19	0.33	0.23	0.81	0.29	0.37
	APC	231	12	356	21	279	80	9	133	21	116	0.42	0.83	0.42	1.00	0.49	0.63
	BC	231	12	356	21	279	20	0	74	5	50	0.08	0.00	0.16	0.22	0.16	0.12

Table 5: Experimental results

to exhibit an ordering with respect to the average detection effectiveness across fault types, i.e., BC $<$ ACS $<$ APC $<$ APDU.

Explanation of Effects: The variation in the detection effectiveness among the coupling criteria is of no surprise. ACS, the weakest criterion, does not consider the effects on state space interactions caused by inheritance and polymorphism, and this could account for its relatively poor performance as compared to the others. According to the first condition of the fault/failure model [DeMillo and Offutt, 1991][Morell, 1988], a location that contains a fault must be reached before the fault can manifest a failure. The shortcoming of ACS is that not all locations that can contain faults due to inheritance and polymorphism will be executed. By their very nature, these faults will be located within the hierarchy associated with the objects being integrated, not in the

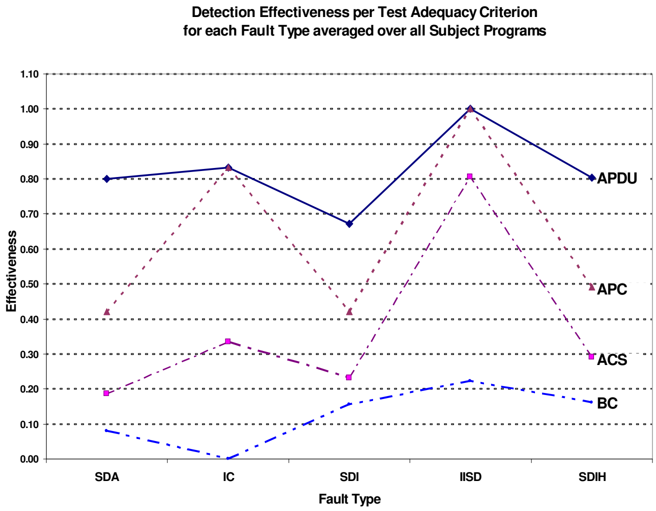


Figure 19: Average detection effectiveness by fault type

method under test. Thus, faults at these locations will not necessarily be executed as a result of testing according to the ACS criterion. As expected, the APC criterion performs better than ACS since it requires that all possible type substitutions be tested for each coupling sequence appearing in the method under test. Thus, the possibility of executing a fault located in the hierarchy being integrated is increased simply because control flow enters each type at least once. However, this is not sufficient to ensure all feasible locations containing faults will be executed.

The most effective of the three coupling-based test adequacy criteria is APDU, which is not surprising since it requires that all state interactions be tested with respect to the coupling variable for each coupling sequence, and for all types of instances that can be bound to the coupling variable. This supports our theory that state interactions need to be explicitly tested.

Hypothesis Tests: Log-linear analysis permits categorical data to be analyzed in much the same manner as in analysis of variance. The distribution underlying [Tab. 6] is a product of independent multinomials. According to [Bishop et al., 1975], the kernel of the appropriate likelihood function is the same as that for a simple multinomial or a simple Poisson. Therefore the estimation procedures for the simpler sampling distributions may be used, at least for large samples. The resulting estimates are close to the correct maximum likelihood estimates and the usual goodness of fit statistics are asymptotically chi-square.

The data were analyzed by first fitting the experimental results to a model that cor-

N	**Hypothesis**	χ^2	*df*	$\Delta\chi^2$	Δdf	**Conclusion**
1	H_0: APDU is not more effective than BC	91.74	164	816.74	36	Reject H_0
	H_1 :APDU is more effective than BC					
2	H_0 :APC is not more effective than BC	35.93	68	175.00	12	Reject H_0
	H_1 :APC is more effective than BC					
3	H_0 :ACS is not more effective than BC	19.00	63	97.94	12	Reject H_0
	H_1 :ACS is more effective than BC					
4	H_0 :APDU is not more effective than APC	51.87	68	441.47	12	Reject H_0
	H_1 :APDU is more effective than APC					
5	H_0 :APDU is not more effective than ACS	47.89	68	103.88	12	Reject H_0
	H_1 :APDU is more effective than ACS					
6	H_0 :APC is not more effective than ACS	69.28	68	256.97	12	Reject H_0
	H_1 :APC is more effective than ACS					

Table 6: Results of hypothesis tests

responded to a 4-way contingency table with the i and k marginals fixed. The model consists of the dimensions Fault, Response, Fault, Program, Criterion, Response, and all lower level nested factors. The factor Response consists of two levels, each corresponding to success or failure of a particular test case. We represent these four factors by $u1$ (Program), $u2$ (Fault Type), $u3$ (Criterion), and $u4$ (Response), and represent cell counts by $m_{i,j,k,l}$, where i, j, k, and l correspond to the four factors. The best fitting model was found to be $\log(m_{i,j,k,l}) = u_o + u_1 + u_2 + u_{1,3} + u_{1,4} + u_{2,4} + u_{1,2} + u_{3,4} + u_{1,3,4} + \ldots$. The terms with one subscript represent the main effects; the terms with two subscripts represent two-factor interactions; and the terms with three subscripts represent three-factor interactions. [Fig. 20] shows that the fitted cell counts closely match the observed cell counts.

8 Related Work

Several testing issues are unique to O-O software. Several researchers have asserted that a number of traditional testing techniques are not effective for O-O software systems [Berard, 1994][Firesmith, 1993][Hayes, 1994] and that traditional software testing methods test the wrong things. Specifically, methods tend to be smaller and less complex, so path-based testing techniques are often less valuable. Additionally, inheritance and polymorphism introduce undecidability [Barbey and Strohmeier, 1994]. The execution path is no longer a function of the class's static declared type, but a function of the dynamic type that is not known until run-time.

Usually, classes are the basic unit of O-O testing. In [Harrold and Rothermel, 1994], the authors defined three levels of testing: (i) intra-method testing, in which tests are

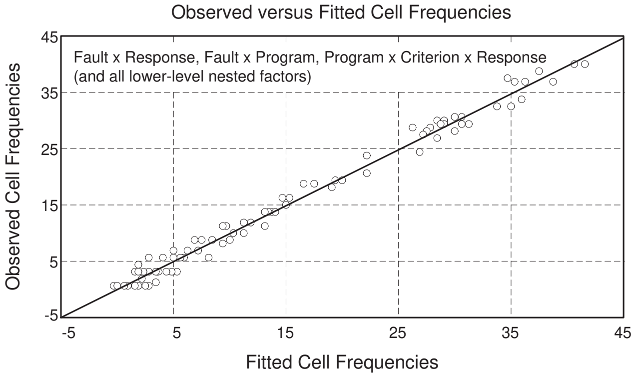


Figure 20: Observed versus fitted cell frequencies

constructed for individual methods; (ii) inter-method testing, in which multiple methods within a class are tested in concert; and (ii) intra-class testing, in which tests are constructed for a single class, usually as sequences of calls to methods within the class. In [Gallagher and Offutt, 2004] inter-class testing was added. Much of the early research in O-O testing focused on the inter-method and intra-class levels [Fiedler, 1989] [Harrold and Rothermel, 1994][Smith and Robson, 1990]. Later research focused on the testing of interactions between single classes and their users [Overbeck, 1994] and system-level testing of O-O software [Jorgenson and Erickson, 1994]. Problems associated with the essential language features of inheritance and polymorphism cannot be addressed at the inter-method or intra-class levels. These require multiple classes that are coupled through inheritance and polymorphism, which can only be addressed via inter-class testing.

Most research in O-O testing has focused on one of two problems. One is the ordering in which classes should be integrated and tested [Briand et al., 2003] and the other is developing techniques and coverage criteria for selecting tests. This paper presents results on the latter problem, specifically focusing on problems caused by the use of inheritance and polymorphism. The result is a collection of inter-class testing criteria, which is a type of integration testing [Beizer, 1990].

9 Conclusions and Future Work

This paper summarizes research on O-O software that combines two streams of concepts that originated in the 1970s. Parnas introduced the basic notions of data abstraction, which eventually led to O-O design and programming [Parnas, 1972], then Constantine and Yourdon introduced the idea of coupling as a fundamental way to evaluate the integration relations among modules [Constantine and Yourdon, 1979]. The O-O testing criteria in this paper build on several key insights. Our first insight was to base integration tests of functional software on the data and control couplings among methods. A subsequent insight was that inheritance and polymorphism can introduce new kinds of faults that were not well understood before. A final insight was that couplings can also be used to test software that uses inheritance and polymorphism.

This paper has summarized new data flow analysis techniques for O-O software, new testing criteria to address problems that can arise from using inheritance and polymorphism, and results from experimental validations. The traditional notion of software coupling has been updated to apply to O-O software, handling the relationships of aggregation, inheritance and polymorphism. This allows the introduction of a new integration analysis and testing technique for data flow interactions within O-O software. A key contribution is a technique for analyzing and testing polymorphic relationships. The foundation of this technique is the coupling sequence, which is a new abstraction for representing state space interactions between pairs of method invocations. The coupling sequence provides the analytical focal point for methods under test, and is the foundation for the algorithms for identifying and representing polymorphic relationships for both static and dynamic analysis. With this abstraction and the algorithms, both testers and developers of O-O programs can now analyze and better understand the interactions within their software. Though the coupling sequence has been cast for testing problems involving inheritance and polymorphism, is generally applicable to any program that makes uses of encapsulated data types, e.g. Modula-2 or Ada 83. We also summarized a set of test-adequacy criteria that take inheritance and polymorphism into account. These criteria provide the tester and developer with a way of judging when a testing goal has been achieved. The criteria naturally vary in their effectiveness, but this variation also correlates with the required level of testing effort and is reflected by the subsumptive relationship among the criteria. In ideal circumstances, the effort required to achieve perfect or near-perfect software would be expended. In this case, only a single criterion would be necessary. However, in practice, limited amounts of effort can be expended. The variation of the criteria allow the tester and developer to develop test requirements that reflect this reality.

This paper has focused on testing polymorphic relationships that are manifested through state space interactions that result from pairs of method invocations within the same method. However, as described in [Section 5], there are other interactions that can occur between methods that are not invoked from the same methods. These are inter-method coupling sequences and represent interactions that occur indirectly as the result of two or more separate method invocations. To accommodate this, the definition of the types of coupling sequences described in [Section 5.1] must be expanded along with the definitions for the coupling method, antecedent node and method, and consequent node and method. This can result in the ability to detect more faults, but the analysis will be more expensive. Another key area of related research is automatic generation of test cases. The research reported in this paper relied on hand generated tests. While this is acceptable for a scientific investigation, it is of limited applicability in practical settings. These techniques cannot be used in practice without automatic test generation.

A number of questions naturally result from the application of the coupling-based testing approach, such as how effective is the testing effort expended thus far, how much effort is required to test a given program using a criterion, and so on. The coupling-based testing approach naturally yields a number of artifacts, and O-O programs also have a distinct set of artifacts. There is the potential to combine these and use them as the basis of a measurement theory for the approach. For example, there may be a strong positive correlation between the depth of an inheritance hierarchy and the number of overridden methods with the number of test requirements generated from the coupling-based test adequacy criteria. Having this theory along with a practical process for its use would add significantly to the practical application of the coupling-based testing approach.

References

[Alexander, 2001] Alexander, R. T. (2001). *Testing the Polymorphic Relationships of Object-Oriented Programs*. Ph.D. Dissertation, George Mason University.

[Alexander et al., 2000] Alexander, R. T., Bieman, J. M., and Viega, J. (2000). Coping with Java programming stress. *Computer*, 33(4):30–38.

[Alexander and Offutt, 1999] Alexander, R. T. and Offutt, J. (1999). Analysis techniques for testing polymorphic relationships. In *Proceedings of the 30th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS30)*, pages 104–114, Santa Barbara, CA.

[Alexander and Offutt, 2000] Alexander, R. T. and Offutt, J. (2000). Criteria for testing polymorphic relationships. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, pages 15–23, San Jose CA. IEEE Computer Society Press.

[Barbey and Strohmeier, 1994] Barbey, S. and Strohmeier, A. (1994). The problematics of testing object-oriented software. In *Proceedings of the 2nd Conference on Software Quality Management*, volume 2, pages 411–426, Edinburgh, Scotland, UK.

[Beizer, 1990] Beizer, B. (1990). *Software Testing Techniques*. Van Nostrand Reinhold, New York, New York.

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

[Berard, 1993] Berard, E. V. (1993). *Essays on Object-Oriented Software Engineering*, volume 1. Prentice Hall.

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

[Bishop et al., 1975] Bishop, Y. M. M., Fienberg, S. E., and Holland, P. W. (1975). *Discrete Multivariate Analysis: Theory and Practice*. MIT Press, Cambridge, Massachusetts.

[Briand et al., 2003] Briand, L. C., Labiche, Y., and Wang, Y. (2003). An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(7):594–607.

[Constantine and Yourdon, 1979] Constantine, L. L. and Yourdon, E. (1979). *Structured Design*. Prentice-Hall, Englewood Cliffs NJ.

[DeMillo and Offutt, 1991] DeMillo, R. A. and Offutt, A. J. (1991). Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910.

[Fiedler, 1989] Fiedler, S. P. (1989). Object-oriented unit testing. *Hewlett-Packard Journal*, 40(2):69–75.

[Firesmith, 1993] Firesmith, D. G. (1993). Testing object-oriented software. In *Proceedings of the 11th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA, '93)*, pages 407–426. Prentice-Hall, Englewood Cliffs, New Jersey.

[Frankl and Weyuker, 1988] Frankl, P. G. and Weyuker, E. J. (1988). An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498.

[Gallagher and Offutt, 2004] Gallagher, L. and Offutt, A. J. (2004). Integration testing of object-oriented components using finite state machines. *Software Testing, Verification and Reliability*. To be published.

[Harrold and Rothermel, 1994] Harrold, M. J. and Rothermel, G. (1994). Performing data flow testing on classes. In *Software Engineering of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 154–163. ACM Press, New York, New York.

[Hayes, 1994] Hayes, J. H. (1994). Testing of object-oriented programming systems (OOPS): A fault-based approach. In Urban, E. B. and S., editors, *Object-Oriented Methodologies and Systems*, volume LNCS 858. Springer-Verlag.

[Jin and Offutt, 1998] Jin, Z. and Offutt, A. J. (1998). Coupling-based criteria for integration testing. *The Journal of Software Testing, Verification, and Reliability*, 8(3):133–154.

[Jorgenson and Erickson, 1994] Jorgenson, P. C. and Erickson, C. (1994). Object-oriented integration testing. *Communications of the ACM*, 37(9):30–38.

[Liskov and Wing, 1994] Liskov, B. and Wing, J. M. (1994). A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841.

[Meyer, 1990] Meyer, B. (1990). *Introduction to the Theory of Programming Languages*. Prentice Hall.

[Meyer, 1997] Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, New Jersey.

[Morell, 1988] Morell, L. J. (1988). Theoretical insights into fault-based testing. In *Software Engineering of the 2nd Symposium on Software Testing Analysis and Verification (TAV2)*, pages 45–62, Banff Alberta.

[Offutt et al., 1993] Offutt, A. J., Harrold, M. J., and Kolte, P. (1993). A software metric system for module coupling. *The Journal of Systems and Software*, 20(3):295–308.

[Overbeck, 1994] Overbeck, J. (1994). *Integration Testing for Object-Oriented Software*. Ph.D. Thesis, Vienna University of Technology.

[Page-Jones, 1980] Page-Jones, M. (1980). *The Practical Guide to Structured Systems Design*. YOURDON Press, New York, NY.

[Parnas, 1972] Parnas, D. (1972). On the criteria to be used in decomposing a system into modules. *Communications of the ACM*, 15(12):1053–1058.

[Parnas et al., 1976] Parnas, D. L., Shore, J. E., and Weiss, D. (1976). 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.

[Rapps and Weyuker, 1985] Rapps, S. and Weyuker, W. J. (1985). Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375.

[Smith and Robson, 1990] Smith, M. D. and Robson, D. J. (1990). Object-oriented programming: The problems of validation. In *Proceedings of the 6th International Conference on Software Maintenance*, pages 272–282. IEEE Computer Society Press, Los Alamitos, CA.

[Weiss, 1989] Weiss, S. N. (1989). What to compare when comparing test data adequacy criteria. *ACM SIGSOFT Notes*, 14(6):42–49.

[Zhu, 1996] Zhu, H. (1996). A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Transactions on Software Engineering*, 22(4):248–255.