11-28-2010

Testing coupling relationships in object-oriented programs

Roger Alexander

Jeff Offutt

Andreas Stefik

Testing Coupling Relationships in Object-Oriented Programs [*]

Roger Alexander
Schweitzer Engineering Laboratories, Inc.
Pullman, Washington, USA
rtalexander@mac.com

Jeff Offutt
George Mason University
Software Engineering
Fairfax, Virginia, USA
offutt@gmu.edu

Andreas Stefik
Department of Computer Science
Central Washington University
Ellensburg, Washington, USA
astefik@eecs.wsu.edu

16 June 2009

Abstract

As we move to developing object-oriented programs, the complexity traditionally found in functions and procedures is moving to the connections among components. Different faults occur when components are integrated to form higher level structures that aggregate behavior and state. Consequently, we need to place more effort on testing the connections among components. Although object-oriented technologies provide abstraction mechanisms for building components that can then be integrated to form applications, it also adds new compositional relations that can contain faults. This paper describes techniques for analyzing and testing the polymorphic relationships that occur in object-oriented software. The techniques adapt traditional data flow coverage criteria to consider definitions and uses among state variables of classes, particularly in the presence of inheritance, dynamic binding, and polymorphic overriding of state variables and methods. The application of these techniques can result in an increased ability to find faults and to create overall higher quality software.

KEYWORDS: Integration, Testing, Coupling, Inheritance, Polymorphism

1 INTRODUCTION

The emphasis in object-oriented languages is on defining abstractions (e.g. abstract data types) that model concepts relative to an application domain [35]. These abstractions appear in software as user-defined types

[*] This work was supported in part by the U.S. National Science Foundation under grant CCR-98-04111 to George Mason University.

that have both state and behavior. Although abstract data types and other OO classes 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 in 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 OO software is due to the nature of the relationships found in object-oriented languages [11]. The compositional relationships of inheritance and aggregation, combined with the power of polymorphism and dynamic binding, makes it harder to detect faults that result from the integration of components. This is because the integration of classes and components is different in object-oriented languages [10].

Procedure-oriented languages use procedures and functions as the primary abstraction mechanism. In contrast, both object-based and object-oriented languages use data abstraction as the primary mechanism. In addition, object-oriented languages use the integration mechanism of *inheritance*. New types created by inheritance are *descendants* of the existing type [34]. Inheritance differs from aggregation in several ways; a key difference being that the encapsulation of the inherited type may not be preserved, that is, the new type can have access to the internal representation of the ancestor types.

When combined with inheritance, polymorphism and dynamic binding can strongly affect component integration. When a call is made to a polymorphic method, which version is executed depends on the actual type of the object bound to the variable through which the call is made (e.g. *o* in *o.m*()). [35]. 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 among the related classes.

The second form, *integration of abstraction*, deals with the effects of aggregation in the presence of inheritance, dynamic binding 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 during execution. That is, the actual type of the object reference may change. Because of this, a number of substitutions must be tested to ensure the type behaves correctly, that is, the code correctly implements the abstraction.

1.1 Testing OO Software

There are a number of testing issues that are unique to object-oriented software. Several researchers have asserted that some traditional testing techniques are not effective for object-oriented software [9, 27, 23] 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 less applicable. Also, many of the interactions that occur among program components take place through complex state interactions, which, due to inheritance and polymorphism, are subtle and not always obvious. Additionally, precise determination of the set of types that objects bound to a variable may have is an undecidable problem [7]. 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.

A class is usually regarded as the basic unit of OO testing. Harrold and Rothermel [26] define three levels of testing: (1) *intra-method testing*, in which tests are constructed for individual methods; (2) *inter-method testing*, in which multiple methods within a class are tested in concert; and (3) *intra-class testing*, in which tests are constructed for a single class, usually as sequences of calls to methods within the class. Gallagher and Offutt [25] use *inter-class testing*, in which more than one class is tested at the same time.

Much of the early research in object-oriented testing focused on the inter-method and intra-class levels [22, 26, 45]. Later research focused on the testing of interactions between single classes and their users [39] and system-level testing of OO software [31]. Problems associated with the essential language features of inheritance, dynamic binding 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 OO testing has focused on one of two problems. One is the ordering in which classes should be integrated and tested [13] and the other is developing techniques and coverage criteria for selecting tests. This paper presents results on the latter problem, specifically focusing on problems that can arise because of the use of inheritance, dynamic binding and polymorphism. The result is a collection of **inter-class** testing criteria. This is a type of *integration testing*, which Beizer [8] defined to be testing interfaces between units and modules to assure that they have consistent assumptions and communicate correctly.

This paper presents a technique for finding errors in the polymorphic relationships among integrated components. The general strategy for this solution is to adapt traditional data flow coverage criteria to consider definitions and uses among state variables of classes, particularly in the presence of inheritance and polymorphic overriding of state variables and methods. We consider test criteria to be based on test requirements. *Test requirements* are specific things that must be satisfied or covered, for example, reaching statements and executing *def-use* pairs. A *testing criterion* is a rule or collection of rules that impose requirements on tests. Test engineers measure the extent to which a criterion is satisfied in terms of *coverage*, which is the percent of test requirements that are satisfied.

2 Definitions and Background

Although the concepts presented in this paper are largely independent of language, 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. OO programming relies on the *class* to encapsulate state information in a collection of variables, referred to as *state variables*. A class also has a set of behaviors that are represented by a collection of methods that operate on those state variables. The state variables and methods define a type from which *objects* can be instantiated.

Classes are composed in two ways. *Aggregation* is the traditional notion of one type containing instances (or references to instances) of another type as part of the its internal state representation. *Inheritance* allows the representation of one type to be defined in terms of the representation of one or more other types. The new type *inherits* properties (state and behavior) of its ancestors. The definition of the ancestors becomes part of the definition of the new descendant type.

Objects are accessed by using *object references* that refer to *variable instances*. *Polymorphism* allows variable instances to be bound to references of different types according to the structure of the inheritance hierarchy. In most languages, an object reference that is declared with one type (*declared type*) can be bound to variable instances of its declared type or any of its descendant types (its *actual type*). *Dynamic binding* permits different method implementations to execute depending upon the actual type of an instance that is bound to a particular reference rather than its declared type [35]. In the remainder of this paper, the phrase *polymorphic behavior* refers to the combination of dynamic binding of polymorphic calls.

2.1 Coupling-Based Testing

The work in this paper is based on previous work by Jin and Offutt [30] that presented an approach to integration testing of procedure-oriented software based on coupling relationships among procedures. Coupling was originally proposed to measure design [18, 40, 37], 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. 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 location where a procedure call is made (referred to as a *call site*) with the variable as an *actual argument*, and then to *uses* of the corresponding *formal argument* 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's paper [30], and are defined more formally in their paper. 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 components, and x and y are program variables.[1]

As in traditional data flow analysis, a path from node i to j is *def_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 *coupling-def* is a node that contains a definition that can reach a use in another program component on at least one execution path. There are three kinds of coupling-defs:

- *last-def-before-call*: the definition of a variable that is subsequently passed as an actual argument to a called method, or
- *last-def-before-return*: the definition of a formal parameter that was passed by reference, the definition of a local variable subsequently used in a return statement, or the definition of a formal parameter that was passed by value and subsequently used in a return statement, or
- *shared-data-def*: the definition of a shared (global) variable.

A *coupling-use* is a node 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-uses: A use of a formal parameter after a call (*first-use-after-call*), a use of an actual parameter inside a callee (*first-use-in-callee*), and a use of a shared variable (*shared-data-use*).

In general, a *coupling path* is any def-clear path between a location where a variable is defined and the location where it is subsequently used. These locations may reside in the same or different program components, and accounts for definitions and uses of variables that occur locally within a program component, those that

[1] Jin and Offutt used the term *program unit* to refer to complete unit of code, such as a procedure or function [30]. In this work, we use the generic *program component* to refer to individual procedures, functions, or methods, but we also use the term to generically refer to a user defined type. We explicitly make the distinction to which we are referring if it is not clear from the context.

are passed as actual arguments, and global variables that are shared among distinct program components. However, for purposes of the work described in this paper, we adopt a more restrictive definition of a coupling path: a coupling path is a def-clear path between two program components from a *last-def-before-call* to a *first-use-in-callee* or from a *last-def-before-return* to a *first-use-after-call*.

We define a *subpath set* to be the set of nodes on some subpath. There is a many-to-one mapping between subpaths and subpath sets; that is, if there is a loop within the subpath, the associated subpath set is the same no matter how many iterations of the loop are taken. A *coupling path set* is the set of nodes on a coupling path.

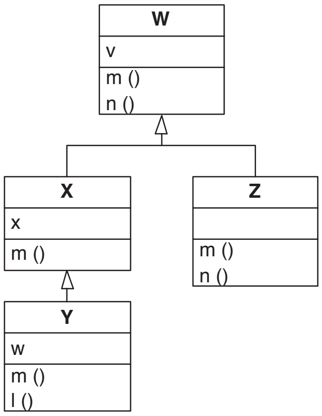
By using these definitions, traditional data flow and control flow criteria were adapted to specify four coupling-based testing criteria [30]. P_1 and P_2 are program components in a system:

- **Call coupling**: The set of paths executed by a test set must cover all call_sites in the system.
- **All-coupling-defs**: For each coupling-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-use.
- **All-coupling-uses**: For each coupling-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-use.
- **All-coupling-paths**: For each coupling-def of a variable in P_1, the set of tests executed must cover all *coupling path sets* from the coupling-def to all reachable coupling-uses. Requiring that all coupling paths be covered is impractical in general. The problem with this is that if there is a loop along a coupling path, the number of coupling paths becomes infinite. However, covering all coupling path sets does ensure that each loop body is executed at least once, but does not require all possible executions.

2.2 Coupling in the Presence of Polymorphism

Our goal is to extend CBT (and thus apply the data flow criteria) to address testing problems that arise from inheritance, dynamic binding, and polymorphism. In this context, identifying the definitions, uses and couplings is harder, thus it is necessary to consider the semantics of the OO language features very carefully.

In the following definitions, o is an *object reference*. o can reference instances whose actual types are either the declared type of o, or a descendant. When it is important to separate the object reference from the instance that it points to, we use $*o$.



(a)

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

(b)

Figure 1: 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 object-oriented 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. Type families are implemented with inheritance. Note that the view we are adopting in this paper is consistent with the notion of sub-typing as defined by Liskov and Wing [33]. That is, we assume that instances of descendant types can be freely used in a context that expects an instance of a parent type.

Every type definition T implicitly defines a set of types having a partial ordering such that an instance of any member of the equivalence class can be used where an instance of T is expected. We refer to such a set as a *type family*. Thus members of a family include the base type of a hierarchy, T, and all types that are descendants of T. Figure 1b illustrates this with four type families, each defined by one of the classes in the hierarchy shown in Figure 1a.

An instance method in a class can call another local instance method (i.e., a local method is defined, or inherited, by the same class as the calling method) implicitly without beginning qualified by an object reference. This is expressed by naming the desired method to be called. For example, the instance method $m()$ defined by class C can call another instance method defined by C simply by naming the method. For example, $m()$ can call a method p that is locally defined by C simply by giving its name in a statement or expression (i.e., $p()$).

Alternatively, an instance method can make an explicit call to another local method explicitly using a special built-in identifier that refers to the current instance receiving the call. For example, in Java, C++, and C#, an explicit call to $p()$ can be made using the identifier *this*: $this.p()$.

Many object-oriented languages also allow class methods (i.e., a `static` method in Java, C++, and C#) to be called without an explicit qualifier, provided the method is local to the same class as the calling method. As with instance methods, a class method is called simply by using its name in a statement or expression. Class methods will not be discussed further in this paper.

In a object-oriented language, all instances (objects) have a *value* regardless of the type of the objects. Usually, when dealing with an instance the so-called primitive or built-in types (e.g., `int`, `float`, `byte`, and so on, in Java, C++, and C#), we treat the value of the instance as a whole. That is, how the state of those types are constituted is immaterial, as are the effects of the operations on defined those types. For example, when we deal with a variable, say $intx = 7$, we generally do not concern ourselves with what the underlying state representation of an `int` is: it could be a 32-bit word in memory, or perhaps (though unlikely) 32 bytes of memory (one byte to represent each bit), or something else. In any case, we don't care how its state is represented, we only care about the value.[2]

So, when we use x in an expression, such as $x = x + 3$, we only care what the value of x is in the right hand side and the value of x on the left hand side. In particular, we do not concern ourselves with which portions of the state representation of `int` change as a result of the assignment, nor do we care which portions are used in the addition operation. We only care about the net result, we are simply oblivious to the state representation and the implementation of the operations. Thus, we adopt a macro view of an instance's value, which is consistent with the principles of information hiding and encapsulation.

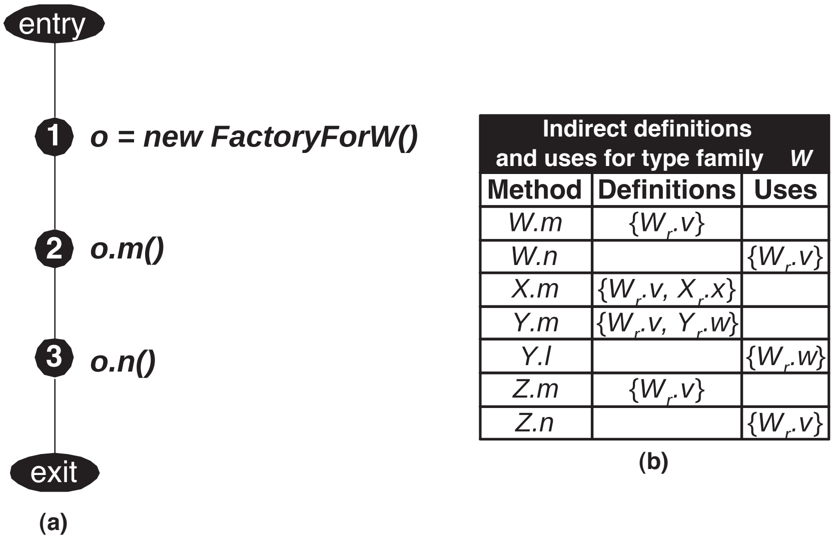
All of the above is generally true with user-defined types. Often we are oblivious to the internal representation of a user-defined type, and its implementation of operations. This is particularly true when we are using types from libraries that are externally defined. Most of the time we just assume that the types behave correctly; we do not what to have to be concerned with internal details. However, there are times when we must concern ourselves with these details, particularly if we are the author of user-defined types.

When we define a type, we must be concerned with not only the value its instances present to the world, but we must also be concerned with how that value is represented and how it is determined. Thus, we must concern ourselves with how individual operations affect the internal state representation we have specified for the type.

When testing user-defined types, each operation that alters the value of an instance must be considered from both micro and macro perspective.[3] This perspective concerns itself with how each operation affects the value of an instance of a user-defined type, and how those operations interact with one another as a result.

[2] There are circumstances where we care about how much physical memory the type uses, but this is a separate concern.

[3] The macro perspective is equally important in testing. However, the research presented in this paper focuses primarily on the micro perspective.



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	$\{W_r.v, Y_r.w\}$	
Y.l		$\{W_r.w\}$
Z.m	$\{W_r.v\}$	
Z.n		$\{W_r.v\}$

Figure 2: Control flow graph fragment (a) and associated definitions and uses (b).

To do requires that we focus on the definitions and uses of the variables (fields or attributes) that comprise the state of a user-defined type.

An object (instance) $*o$ is considered to be altered (i.e., assigned a value) when one of its state variables is changed. Changes to publicly visible state variables can immediately occur via reference through $*o$ (e.g., $o.v = 7$), or indirectly through method calls also referenced through $*o$ (e.g., $o.m()$). We refer to latter type of definitions as *indirect definitions*.

An indirect definition, or *i-def*, occurs when a method m defines one of $*o$'s variables. Similarly, an *indirect use* (*i-use*) occurs when m references the value of one of $*o$'s variables.

Again, consider the class diagram shown in Figure 1 and assume that in hypothetical code under test, W includes a method $FactoryForW()$ that returns an instance of W. Figure 2a 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 given in Figure 2b shows that $W.m()$ defines v so an indirect definition occurs at node 2 through $o.m()$. Thus, any call to $m()$ with respect to an instance of W bound to o results in an indirect definition of the state of $*o$, where o is a reference to $*o$. Note that there are no indirect uses by $m()$. The table in Figure 2b 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-use* of v.

One difficult challenge of data flow analysis for OO programs is *static non-determinism* induced by dynamic binding and polymorphism. Polymorphism allows one call to refer to multiple methods, depending on the actual 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 consider not just the syntactic call that is made, but the set of methods that can potentially execute. That set depends on the type of the instance that is bound to the object reference. We introduce the term satisfying set:

> **Definition 2.1.** ***T***:he **satisfying set** of a polymorphic call to method $m()$, through a reference o to objects of type T, contains all overriding methods $m()$ of descendants of T, plus $m()$ itself.

Note that our definition of satisfying set is a gross approximation, and is based on the assumption that the all the methods in a satisfying set can be executed at the corresponding call site. However, while this assumption is safe in the sense that all methods that could possibly execute at a call are contained in the satisfying set, in many cases, the set of methods that can actually execute will likely be a subset of the satisfying set.

This is due to the fact that it may not be feasible at a given call site, s, for o to be bound to all the possible types that contribute a method to the satisfying set. The type of objects that o can be bound to is constrained by the set of paths that lead to s, and the particular bindings that occur along those paths. For example, suppose there is a single path p that leads to s, and that the nearest binding of o on p is the result of a constructor invocation (e.g., $o = newT())$, then the only method $m()$ that can be executed at s is the definition of $m()$ in T (or inherited by T from one of its ancestors). Thus, the only method in the satisfying set of $o.m()$, will be the $m()$ defined by T. Other overriding definitions of $m()$ is the satisfying can never be executed.

The practical effect of our definition of a satisfying set is that any testing strategy that requires (complete) coverage of the satisfying set will generate test requirements that can never be satisfied. A more precise approximation can possibly be obtained using static analysis techniques, such as points-to-analysis [36].

When considering the set of indirect definitions or indirect uses that can occur at a call site, it is first necessary to determine the set of methods that can satisfy the call. For each method in the satisfying set (satisfying methods), 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 Figures 1 and 2a, the satisfying set for the call to $m()$ at node 2 is $\{W.m(), X.m(), Y.m(), Z.m()\}$ and the *i-def* set is the following set of ordered pairs:

$$\begin{aligned} i\text{-}def\ (2,*o,m()) = \{&(W.m(),\{*W.v\}),\ (X.m(),\{*W.v,*X.x\}),\\ &(Y.m(),\{*W.v,*Y.w\}),\ (Z.m(),\{*W.v\})\} \end{aligned}$$

Each pair indicates a satisfying method for $m()$ and the corresponding set of state variables that the method defines. As such, this definition makes sense in this context if $FactoryForW()$ can return a variable of type W, X, Y, or Z, but is not if it returns an object bound directly to W. In this example, $X.m()$ defines state variables v and x contained in classes W and X.

From Figure 2b, the *i-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 $o.n()$ that have non-empty *i-use* sets (but their *i-def* sets are empty), which yields the following *i-use* set:

$$\textit{i-use}\,(3, *o, n()) \;=\; \{(W.n(), \{*W.v\}),\, (Z.n(), \{*W.v\})\}$$

2.3 Differences in Coupling Paths in OO Programs

When dynamic binding occurs, it is difficult for programmers to keep track of which methods are called, and if not careful, it is easy to allow data flow anomalies and other integration faults to creep into the implementation. From a coupling and integration perspective, the two primary issues are determining what calls can be executed dynamically, and what effects the calls have on the corresponding state space. Alexander's dissertation proposal categorized the calls into twelve separate cases, which are summarized here in three categories [1].

Most method calls in OO programs are made through *explicit instance contexts* (e.g. $o.m()$). Such calls are subject to dynamic binding as the type of the instance bound to o may vary across different executions. These are Category I calls. It is also possible to for methods to be called without an explicit reference to an instance context. This occurs, for example, when an instance method, $k()$, calls another instance method, $l()$, without using the keyword *this* (i.e., $this.l()$) (where $l()$ is defined by the same class as $k()$). These type of calls form Category II. Lastly, Category III calls also do not reference an instance context, but may be polymorphic. For example, consider types A and B, where B is a subtype of A. Suppose that A defines methods $p()$ and $m()$, such that $p()$ calls $m()$, and that B defines methods $k()$ and $m()$, where the later overrides method $m()$ defined by A. Further suppose that $k()$ calls $p()$. When this call is made, $p()$ will then call $m()$ with the resulting being the execution of B's definition of $m()$. Thus, $k()$ has made an indirect polymorphic call to B's m without without referencing an instance context directly. A complete analysis of these categories are given in Alexander's dissertation proposal [1].

Finally, note that Category II cases are handled by the original CBT definitions. Category I and Category III cases, which involve dynamic binding and polymorphism, require extensions to the definitions and additional analysis techniques.

3 Extended Coupling Definitions

The original coupling-based testing definitions of Jin and Offutt [30] were extended by Alexander and Offutt [3] to account for the various calling contexts that occur in object-oriented programs. In the following definitions, $m()$ refers to a program component, 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 (possibly empty) set of formal arguments and a range given as a return type. The notation $\mathbb{P}T$ represents a set of elements each of which is an instance of T, where T is the name of some type (e.g. integer, class, string, etc.).

As is usual with data flow analysis [24, 44], $defs(i)$ 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 $m()$, $exit(m)$ 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. [4]

The following definitions are introduced to handle the effects of dynamic binding and polymorphism. The set of classes that belong to the same type family specified by c is $family(c)$; c is the 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.[5] $i\text{-}defs(m)$ is the set of variables that are indirectly defined within $m()$ and $i\text{-}uses(m)$ is the set of variables used by $m()$.

3.1 Coupling Sequences

To allow for inheritance, dynamic binding, 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*. We give the following formal definition of a coupling sequence as:

Definition 3.1. A ***coupling sequence*** is a pair of methods, $o.m()$ and $o.n()$, that, when called by a third method, $m()$ defines a state variable v on the instance bound to o, such that v is subsequently used by $n()$. There must be a definition clear path with respect to o, and to v, between the calls to $m()$ and $n()$.

[4] In this work, we only consider paths that are associated with the normal flow of control through a program. In particular, we leave the investigation of coupling relationships in the presence of exceptions for future work.

[5] Note that $state(c)$ contains only the identifiers that correspond to the declared variables that represent the state of c. In particular, this is independent of the declared type of any state variable, whether that type is a primitive, array, pointer, or user defined type. Thus fine-grained definitions and uses of state variables is not considered by the analysis presented in this paper.

The calling method, referred to as the *coupling method*, first calls the *antecedent method*, which defines one (or more) of the instance's state variables. Later, the coupling method calls the *consequent method*, which then uses one (or more) of those state variables defined by the antecedent method.

Note that a coupling sequence must have at least one definition clear path between the definition(s) in the antecedent method and the corresponding use(s) in the consequent method. Coupling sequences are denoted using the notation $s_{j,k}$, where j and k refer to the statements (or control flow graph nodes) that correspond to calls to the antecedent and consequent methods, respectively.

Programs in which the antecedent or consequent method is the same as the coupling method are special cases and are handled implicitly. An *indirect call* is defined to be when the antecedent and consequent methods are called from another method that is, in turn, called by the coupling method. We do not consider indirect calls further in this paper.

The *control flow schematic* shown in Figure 3 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 consists of 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 Figure 3 from h to i to j to k and finally l forms a *transmission path* with respect to $o.v$. The identifier o is called the *context variable*. Formal derivations of coupling sequences for the prototypical situation and special cases are in Alexander's dissertation [2].

3.2 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 *indirect-def* or *i-def*) and used by $n()$ (an *indirect use* or *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*.

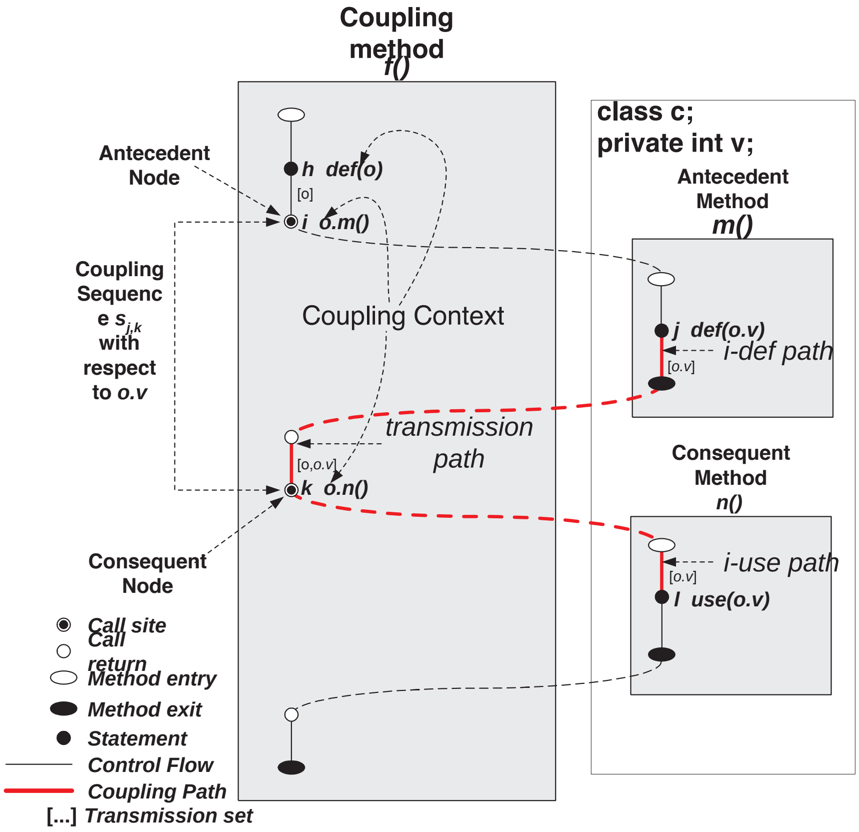


Figure 3: Control flow schematic for prototypical coupling sequence.

3.3 Binding Mechanisms

For a coupling sequence to exist, its context variable o must be *bound* to an instance that reaches the coupling sequence's antecedent node. The location where the binding of o occurs (i.e., the definition of o to a reference to a particular object) is called a *binding site*; there must be at least one definition clear path with respect to o from the binding site to the antecedent node.

Not all bindings are possible for o at a given binding site. The type of an instance bound to o must be a member of the type family induced by o's declared type, T. The *binding set* of o is the set of all possible types of instances that can be bound (assigned) to o. The binding set is determined by the *binding mechanism* used to assign a instance reference to o. There are four kinds of binding mechanisms:

1. **Explicit instance creation**: $o = new\ W()$, where W is a member of the type family induced by T.
2. **Parameter passing**: o is passed to a method as an actual parameter (e.g., $m(o)$).
3. **Assignment of another variable to the context variable**: $o = p$, where p's declared type also is a member of the type family induced by T.
4. **Assignment of the return value of a method call to o**: $o = f()$, where the return type of $f()$, X, is a member of the type family induced by T.

For the type 1 binding mechanism, the type of the instance bound to o can only be the specific type of instance being instantiated, W. Hence, any feasible coupling sequence must have its context variable bound to an instance of W.

For the remaining binding mechanisms, the type of the instance bound to o is restricted by the declared type used in the mechanism. In a type 2 binding mechanism, o is passed as an actual argument to the called method, $m()$, and T must be a member of the type family induced by the declared type of $m()$'s corresponding formal argument. Furthermore, the *actual type* of the instance bound to o is restricted to be a member of the set of o's possible bindings, U, that can reach the location where $m()$ is called. The elements of U will be the union of the binding sets for each binding site of o that reaches $m()$'s call site.

For a type 3 binding mechanism, the resulting type of the binding is limited to the set, Q, of possible bindings to p, which is determined in turn by the binding mechanisms used at each binding site of p. Q is the union of all possible types that can be bound to p at binding sites that reach the assignment $o = p$.

Finally, in a type 4 binding mechanism, the resulting type is limited to the set of types R in the type family induced by X. If $R \neq T$, then R must be a proper subset of T's type family. There will be types in T's type family that $f()$ cannot possibly return.

3.4 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 the definition of a variable to a use.

Each path consists of up to three sub-paths, or segments. The *indirect-def sub-path* is the portion of the coupling path that occurs in the antecedent method $m()$, extending from the last (indirect) definition of the coupling variable to the exit node of $m()$. The *indirect-use 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 the value of the coupling variable is not modified and the context variable is not modified.

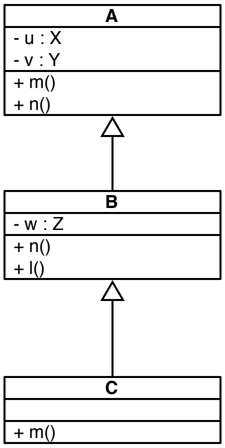
For a given coupling sequence and a coupling variable of the coupling sequence, there is a set of coupling paths for each type of *coupling sub-path.* These sets are used to form coupling paths by matching together elements of each set. For each given coupling variable, the set of coupling paths of the variable is formed by taking the cross product of these sets. The complete set of coupling paths is the union of the coupling sets for all coupling variables.

Definition 3.2. A **coupling path** of a coupling sequence $s_{j,k}$ is a sequence of nodes beginning with a node j having a call to the antecedent method of $s_{j,k}$ and ending at a node k that has a call to the consequent method of $s_{j,k}$, such that there is a definition clear path between j and k for the coupling $s_{j,k}$'s coupling and context variable(s). A coupling sequence will have at least one coupling path.

3.5 The Effects of Inheritance, Dynamic Binding and Polymorphism On Coupling

Thus far, the analysis can be applied to arbitrary data types, irrespective of inheritance and polymorphism. This establishes *def-use* pairs for abstract data types. This subsection adds dynamic binding with inheritance and polymorphism into the analysis, which requires the analysis to handle static *non-determinism*.

To see the effects of dynamic binding with inheritance and polymorphism on path sets, consider the class diagram shown in Figure 4a. 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 is shown in Figure 4*b*.



(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.v}	

(b)

Figure 4: Sample class hierarchy and *def-use* table. Part (a) shows an inheritance hierarchy, the notation '+' means public and '-' means private. Part (b) shows which variables are defined and used by each public method, information that is **not** included in any standard documents, including UML.

Figure 5 shows coupling paths assuming a coupling method that uses the hierarchy in Figure 4a. Figure 5(a) shows the declared type of the coupling variable o is A, and Figure 5(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 $\Theta^{A}_{s_{j,k}} = \{A.v\}$. 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 Figure 5c. 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 $\Theta^{B}_{s_{j,k}} = \{A.u\}$. In turn, this results in a different set of coupling paths. The coupling paths now extend 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, Figure 5d depicts the coupling sequence that results when o is bound to an instance of C. First, observe that execution of the 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

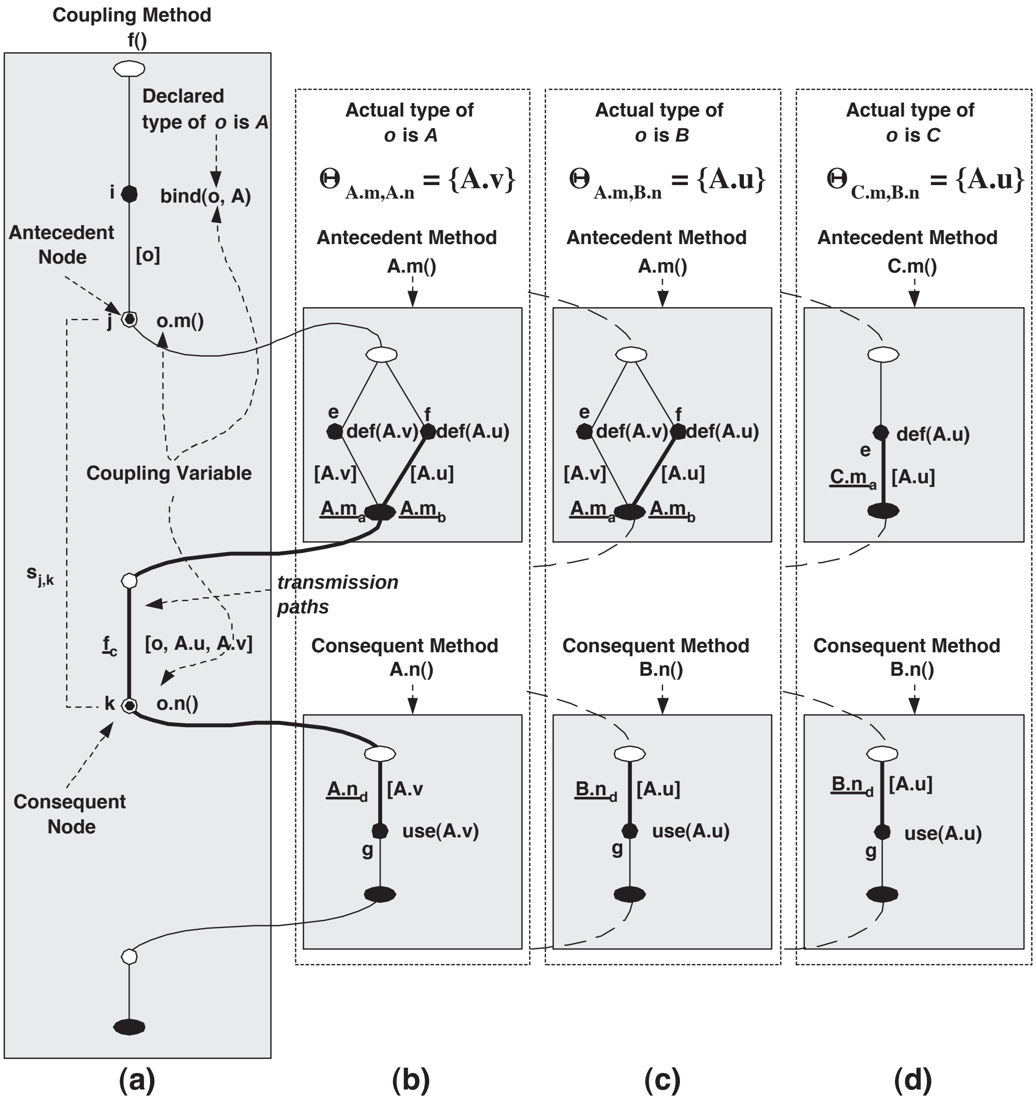


Figure 5: Coupling sequence when *o* is declared as type *A* (a), bound to an instance of *A* (b), *B* (c) and *C* (d).

Table 1: Summary of sample coupling paths

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

$C.m()$ and the consequent method $B.n()$, which yields $\Theta^{C}_{s_{j,k}} = \{A.u\}$. The corresponding coupling path set includes those paths that begin at node e in $C.m()$ and extend to the exit node of $C.m()$, then back node j of the coupling method, and through the entry node of $B.n()$ to node g, also in $B.n()$.

Table 1 summarizes the coupling paths for the examples shown in Figure 5. 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 appear in the coupling paths of other ancestor classes.

3.6 Polymorphic Coupling Sequences and Coupling Sets

Inheritance increases the number of potential bindings for a given coupling sequence. When combined with dynamic binding and polymorphism, the actual 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; that is, the classes that are in the inheritance hierarchy. The following subsections present modified definitions of coupling sequences and coupling sets that take polymorphism into account.

3.6.1 Polymorphic coupling sequences

To account for the possibility of dynamic polymorphic behavior at a call site, the definition of a coupling sequence given in Section 3.1 must be amended to handle all methods that can execute. To accomplish this, we introduce the notion of a *binding triple*. A binding triple for a coupling sequence 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

Table 2: Binding triples for coupling sequence from class hierarchy in Figure 4

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}*

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 that 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 Figure 5 is given in Table 2. 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 Figure 4.

3.6.2 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, dynamic binding and polymorphism can result in different methods being executed, thus this is no longer sufficient. Instead, the coupling sets of all possible method combinations must be combined to form an aggregate coupling set for the sequence. Thus, the 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.

3.7 Coupling Paths in Object-oriented Programs

Procedure-oriented programs have couplings that occur between procedures in terms of parameters explicitly passed as arguments or through shared global data [30]. Data abstraction, which is naturally and heavily used in object-oriented programs, results in coupling paths that originate at last-definitions in an antecedent method and that terminate at first-uses in a consequent method of a user defined type.

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. The second is when there is a possibility of polymorphic behavior. Polymorphic behavior allows for the possibility that, at runtime, the actual type of the instance bound to a variable can be different from the declared type of that variable, subject to the constraint that the actual type must be a member of the type family defined by the declared type. Due to this possible variation of the actual types, it may not be possible to statically determine which methods will actually execute at a given call site. However, it is possible to statically determine all the methods that could possibly execute. The following subsections discuss the coupling paths that result from each of these cases.

3.7.1 Non-Polymorphic coupling paths

Consider again the Type I coupling sequence shown in Figure 3 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.

Ignoring polymorphism for the moment, 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-def* in an antecedent method to a node in a consequent method that contains a *first-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-def path set*, the *t-path set*, and the *i-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-def-paths*, *t-paths*, and *i-use-paths*), subject to the constraint that p be definition-clear with respect to a particular coupling variable v.

3.7.2 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 bound 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.

3.7.3 Data flow anomalies

To understand the effects that dynamic binding and polymorphism can have on the data flow relationships between methods, again consider the table in Figure 4*b*. This table summarizes the definitions and uses of the methods for the class hierarchy shown in Figure 4a. Method $A.m()$ defines state variables $A.v$ and $A.u$ and method $A.n()$ uses $A.v$. From a data flow perspective, there is no problem when o's type is A and these methods are executed as shown in Figure 5b. Similarly, $B.n()$ uses $A.u$. Again there is no problem when o's type is B since $A.u$ has been defined by the call at statement three. But consider the situation where o's type is C, and method $C.m()$ defines $A.v$ but not $A.u$. The call at statement three results in the execution of $C.m()$, which in turn defines $A.v$, but not $A.u$. Later, when the call at statement five is made, $B.n()$ is executed, which uses $A.u$.

There is a potential problem in that a data flow anomaly may exist since $A.u$ is apparently used without a preceding definition. It appears that the implementation of $C.m()$ has violated an assumption made by $B.n()$ with respect to $A.u$. The problem is only potential because it is possible that $A.u$ was defined by an earlier method invocation or as part of construction of the instance bound to o.

The preceding discussions illustrate that the dynamic binding and polymorphism can cause the data flow graph to change dynamically. This makes the analysis more difficult, and would probably be prohibitive if the number of potential types was unbounded. Happily, the number of potential types for an object is finite and usually small. The difficulty lies in the fact that there are multiple graph representations that must be considered for a given method, depending upon the presence of dynamic binding and the possible polymorphism that can occur. We solve this problem by doing an exhaustive analysis to compute all possible coupling data flows.

The above discussion raises another issue, that of infeasible intra-method coupling sequences. An intra-method coupling sequence is *infeasible* if there is no input that can execute the sequence. This is a problem for testers because infeasible coupling sequences result in test requirements that cannot be satisfied. Moreover, it is extremely difficult to determine whether a coupling sequence is infeasible or if it is simply difficult to find an appropriate test case.

A key factor in the identification of infeasible coupling sequences is the precision of the estimate of the satisfying set for an object reference at a given call site.

As noted in section 2.2, the approach we present in this paper relies on a conservative estimation of possible bindings. While this estimate is safe in that it identifies all possible coupling sequence, it does have the risk of identifying infeasible coupling sequences that would generate testing requirements that are impossible to satisfy. A more precise estimation will lead to fewer coupling sequences that involve bindings that cannot occur during execution and a corresponding reduction in the number of infeasible test requirements. We are at present investigating techniques for determining a more reliable estimate.

4 COUPLING CRITERIA

The analysis in Section 3 allows coupling definitions and uses to be identified in the presence of inheritance, dynamic binding and polymorphism. Now we consider how to use this information to support testing. We adapt the data flow criteria [24, 26, 30, 44] to define subpaths in the program that must be tested.

Testing criteria can be used in one of two ways, as a mechanism to help testers mechanically or manually 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, that is, a set of tests already exist. The issue of mechanical or automatic test data generation is not part of the current research.

The following definitions, like most test criteria definitions, allow infeasible testing requirements (we discuss this briefly in section 2.2). We could define the criteria so they only apply to feasible coupling sequences, as Frankl and Weyuker did in their paper [24], but we choose not too.

4.1 Object-oriented Coupling Criteria

In this section we present the coupling-based test adequacy criteria for object-oriented programs [4]. In the following subsections, *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 $s_{j,k}$.

All-Coupling-Sequences (ACS) The first criterion is based on an assumption that during integration testing each coupling sequence should be covered. Accordingly, the *All-Coupling-Sequences* requires that every coupling sequence in *f()* be covered by at least one test case.

Definition 4.1. ***All-Coupling-Sequences***: For every 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 $f(t)$.

All-Poly-Classes (APC) The *All-Poly-Classes* criterion is added to include instance contexts of calls, because ACS does not consider inheritance or polymorphism. This is achieved by ensuring there is at least one test for every class c 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, the *All-Poly-Classes* criterion requires that, for each *f()*, there is at least one test case t for every combination $(c, s_{j,k})$, where c is in the type family defined by the instance context of $s_{j,k}$. The combination (c, $s_{j,k}$) 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 that declared type and defines an overriding method for the antecedent or consequent method. Thus, only classes that override the antecedent or consequent methods are considered.

Definition 4.2. ***All-Poly-Classes***: For every coupling sequence $s_{j,k}$ in method $f()$, and for every class c in the family of types defined by the context of $s_{j,k}$, where either the antecedent or consequent methods are overridden by c, 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 execution trace of $f(t)$.

All-Poly-Coupling-Defs-Uses (APDU) The final criterion, *All-Poly-Coupling-Defs-Uses*, adds strength to the APC criterion by requiring that all coupling paths be executed for every member of the inheritance hierarchy defined the a type providing the instance context of a coupling sequence.

Definition 4.3. ***All-Poly-Coupling-Defs-Uses***: For every coupling sequence $s_{j,k}$ in method $f()$, for every class in the family of types defined by the context of $s_{j,k}$ (where either the antecedent or consequent methods are overridden by c), for every coupling variable v of $s_{j,k}$, for every node m that has a last definition of v and every node n that has a first-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 execution trace of $f()$.

4.2 Subsumption of the Criteria

The above criteria impose certain requirements on the testing process, and each comes at a particular cost. A common method to compare testing criteria is the subsumption relationship [24, 48, 50]. Criterion A *subsumes* criterion B if and only if every test set that satisfies A also satisfies B. In posterior testing, in which a test adequacy criterion is only used as a stop rule, according to Zhu, an adequacy criterion higher on the subsumption hierarchy means better in fault detecting ability in various measurements of detecting ability and also has the higher cost. However, in prior testing (where test adequacy criteria are used to generate test cases), there are exceptions [50].

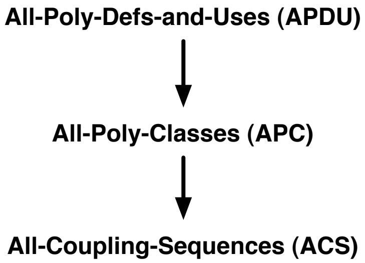


Figure 6: OO Coupling Testing Subsumption Hierarchy.

A general issue with many test adequacy criteria occurs when the unit under test does not contain the properties covered by a particular criterion. For example, the criteria we propose below do not generate any test requirements for program components that do not have coupling sequences, meaning that the unit under test trivially satisfy the criteria without testing, and thus not satisfying Weyuker's Inadequate Empty Set axiom [49]. Ammann and Offutt handle this issue by explicitly defining graph criteria so that empty cases subsume "lower level" criteria. For example, edge coverage is defined to include individual nodes – a distinction that matters when a graph has one node and no edges [6]. The criteria in this paper do not contain properties covered the "lower level" criteria. This facilitated clear definitions and a clean experimental design. As this work goes forward, modifying the definitions along the lines of Ammann and Offutt's book would be helpful.

The subsumption hierarchy for the criteria presented in this paper is shown in Figure 6. The practical issue for testers, then, is to determine which criterion to select for a particular situation. This includes determining how to balance such factors as effectiveness and cost. The next section presents experimental data to address this question.

5 Validation

This section describes the experiments used to empirically validate the efficacy of the object-oriented coupling-based testing criteria described in section 4. We begin with a discussion of the methods and procedures used to conduct the experiments, which includes a description of the experimental subjects, the test adequacy criteria used for comparative purposes, the test data, and the fault types that are used for evaluating the effectiveness of the criteria. Next we present the results of the experiments, and conclude with a discussion of their significance.

5.1 Methods and Procedures

This experiment evaluated the three coupling-based test adequacy criteria [5]. Branch Coverage [8] is used as the control to determine if the other criteria are effective at detecting faults.

5.1.1 Subject Programs

Ten programs were tested using these criteria. Some were pre-existing and some written specially for this experiment. 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*.

Table 3 summarizes the subject programs. Column *f* identifies the method under test and $\left|S_f\right|$ is the number of coupling sequences contained within *f*. Each coupling sequence has a context variable whose declared type *T* defines a family of types that are descendants of *T*. The column labeled F_{S_f} gives the number of classes in this type family (inheritance hierarchy) for the corresponding program. The column labeled *Description* indicates the source from which each program was obtained.

Not all object-oriented programs in the wild will necessarily have instances of each of the subject fault types. However, we felt it important to determine if the proposed criteria, or even Branch Coverage, were powerful to detect the subject faults if present. Thus, five of the subject programs, P1, P2, P3, P5, and P6, were engineered to collectively ensure the presence of all the subject fault types (see section 5.3.5 for a discussion of threats to validity). Each of these programs was implemented by different programmers having varying levels of experience (three were students, two were professional software engineers).

Of the remaining five subject programs, P4 was developed by a graduate student, and P7 and P8 were developed by a professional programmer with 15 years of experience. The remaining two are open source products: ANTLR is a parser generator [29] and JMK is a build system, similar to the Unix tool *make* [46].

Since the APDU criterion is concerned with indirect definitions reaching indirect uses, knowledge of the internal state of objects is needed to evaluate and compare APDU's effectiveness. To achieve this, we instrumented each subject program in a side-effect free manner to to increase its observability. Thus, the output of each instrumented subject program included additional information in addition to what it normally produced, but did not alter program state.

5.1.2 Test Data

The test data used in the experiments was produced from custom test data generators. For each combination of subject program *P* and test adequacy criterion *C*, the input space of *P* was analyzed, and custom Perl program was developed to generate test inputs to satisfy *C*. In all cases, the test data was drawn randomly from a uniform distribution, and sufficient data was generated to achieve 100% coverage for *C*. Note that in general, automatic test data generation is an unsolved problem. The procedure we used in this work is, at best, a semi-automatic process. A human analyzed each subject program and determine the necessary path expressions required to achieve coverage for the subject criteria. That knowledge was then used to manually write Perl programs to generate the data. The problem of automatic test generation is left for future research.

The strategy used to select test cases is similar to how test cases are selected for the Branch Coverage test adequacy criterion [8, 19]. For each coupling sequence, the path expression [8] 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 for testing the state space interactions between antecedent and consequent methods. These path expressions ensured that the required coupling paths were covered.

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, the number of test cases is determined by adding the number of control flow paths in the antecedent and consequent methods to the test cases for APC and ACS. The result of this process was a set of test cases for each subject program.

Once we had a set of test data, the next step was to create oracles that could be used to evaluate the results of subsequent tests and determine if a given test was passed or failed. Each subject method in our experiment had a corresponding oracle. An oracle consisted of a set of test cases sufficient to cover the test adequacy criteria described in section 4.1, along with Branch Coverage. A single test case consisted of a set of test inputs and their expected outputs. The oracle for each subject program was derived by executing each subject program with the corresponding test set and recording the result of each execution.

5.1.3 Injected Faults

Faults were injected into the programs based on a categorization of OO faults. Each subject program *P* was modified by injecting faults into the bodies of the antecedent and consequent methods for each member of each type family induced by the declared type of the coupling sequences in *P*. The types of faults injected into each unit under test are described in detail in our previous work [38]. The number of faults was

Table 3: Subject program characteristics and number of test cases.

f	$\lvert S_f \rvert$	F_{S_f}	**Description**	**ACS**	**APC**	**APDU**	**BC**
P1	4	4	Polymorphic Example	2	4	6	2
P2	5	5	Polymorphic Example	2	5	320	2
P3	1	5	Polymorphic Example	2	5	80	2
P4	1	4	Student Developer	1	3	3	2
P5	3	4	Polymorphic Example	2	5	75	2
P6	3	5	Polymorphic Example	2	5	105	2
P7	6	4	Professional Developer	1	2	64	2
P8	20	5	Professional Developer	4	2	42	4
P9	11	16	Open Source (ANTLR)	6	15	95	6
P10	7	9	Open Source (JMK)	4	9	27	4

$\lvert S_f \rvert$ is the number of coupling sequences in the program and F_{S_f} is the number of classes in the type family. **ACS**, **APC**, **APDU** and **BC** are the numbers of tests to satisfy each criterion.

Table 4: Faults due to inheritance, dynamic binding and polymorphism, divided by the five categories of faults [5].

f	**SDA**	**IC**	**SDI**	**IISD**	**SDIH**
P1	9	0	6	3	3
P2	39	6	39	0	39
P3	36	3	33	0	36
P4	24	0	24	0	18
P5	36	3	36	0	36
P6	18	0	18	0	18
P7	0	0	55	0	30
P8	0	0	76	0	30
P9	42	0	42	12	42
P10	27	0	27	6	27

determined by the syntactic characteristics of each program and are described in Alexander's dissertation [2].

Table 4 summarizes both the number and type of faults that were injected. Five categories of faults were used. State Definition Anomaly (SDA) faults model possible post-condition violations of methods. Incomplete Construction (IC) faults model constructors that failed to properly initialize the entire state. State Defined Incorrectly (SDI) faults model overriding methods that define a state variable incorrectly. Indirect Inconsistent State Definition (IISD) faults model methods that are added to descendants and that define a state variable incorrectly. State Definition Inconsistency (SDIH) faults model mistakes where a descendant class introduces a variable that accidentally overrides an inherited variable. More details of these types of faults are provided in our previous paper [38].

5.1.4 Test Execution and Evaluation

Each fault-seeded subject program was executed using the previously set of test cases (section 5.1.2). The result of each trial execution was compared to the oracle created for the subject program. If the result of a trail execution equals the corresponding oracle result, then the test was deemed to pass. A summary of the experimental results is given in table 5.

5.2 Results

Table 5 shows that for each fault type the number of faults seeded, the number of faults detected, and the detection effectiveness. The last column presents the average detection effectiveness per combination of criterion and fault type for each program. Effectiveness is defined as a ratio of the number of faults detected to the number of faults seeded. The shaded blocks correspond to combinations of program and fault type that were not tested. In these cases, the subject programs did not exhibit the structural characteristics necessary to support the syntactic pattern for the fault type. The last group of rows in the table summarizes by criterion the total number of faults that were seeded, the total number of faults detected, and the average detection effectiveness.

5.3 Analysis

Figure 7 shows a plot of the detection effectiveness per criterion for each fault type averaged (i.e., 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.

An examination of the plot reveals that the most effective of the coupling-based test adequacy criteria within the experimental is *All-Poly-Def-Uses* (APDU), which, as shown in Table 5, 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; each is more or less effective for the same fault types. For example, all three do reasonably well at detecting faults of type IC and IISD, with the corresponding detection effectiveness across this sequence being monotonically increasing. In contrast, all three are much less effective at detecting faults of type SDA, SDI, and SDIH. Note that in all cases, across all fault types, all four criteria appear to exhibit an ordering with respect to the average detection effectiveness across fault types (i.e., BC $<$ ACS $<$ APC $<$ APDU).

Table 5: Experimental Results [5].

		Faults Seeded					Faults Detected					Detection Effectiveness					
Program	Criterion	SDA	IC	SDI	IISD	SDIH	SDA	IC	SDI	IISD	SDIH	SDA	IC	SDI	IISD	SDIH	Average
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

The number of faults seeded and detected, and the percent detected, is given for each program, each test criterion, and each fault category. The *Average* column is the effectiveness over all five categories, and the *Summary* row combines all programs.

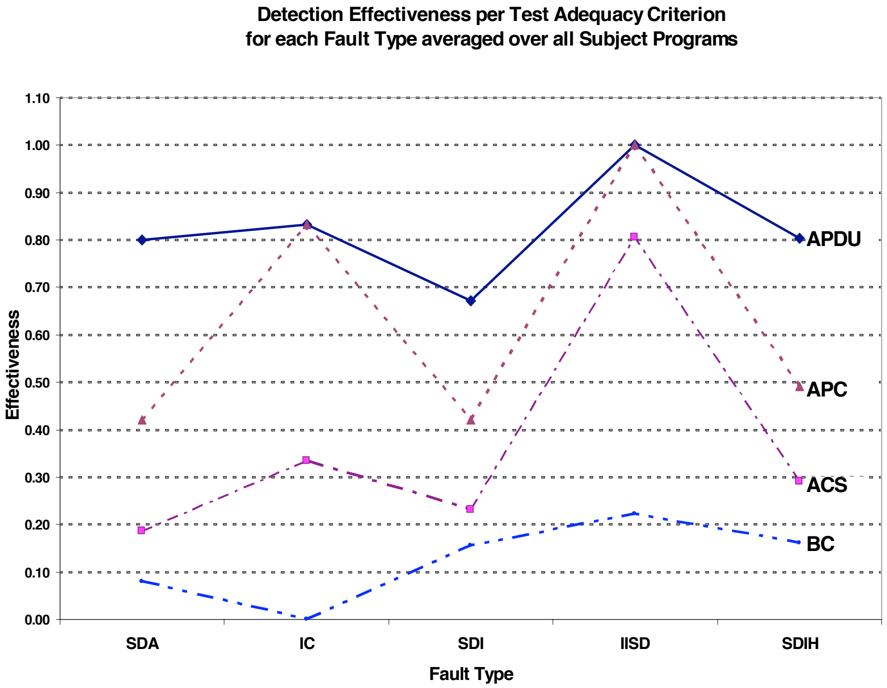


Figure 7: Average detection effectiveness by fault type [5].

5.3.1 Analysis of the coupling-based criteria

APDU has an average detection effectiveness of 0.80 for SDIH, suggesting that it is most effective of the three coupling-based criteria at detecting faults of this type. In comparison, criterion APC has a detection effectiveness of 0.49 for the SDIH faults while ACS has a detection effectiveness of only 0.29. The average detection effectiveness of Branch Coverage is approximately 0.16.

For the SDI fault type, the average detection effectiveness of APDU is 0.67 which is approximately an 18% less, making it not quite as effective as for SDIH faults. Similarly, the remaining coupling criteria also reflect a reduced average detection effectiveness. APC is reduced by approximately 14%, yielding 0.42, and for ACS, the reduction is sightly more at 21%, yielding a detection effectiveness of 0.23. The detection effectiveness for Branch Coverage remains the same at 0.16.

For the SDA fault type, APDU remains the most effective, having the same average detection effectiveness as SDIH fault types (0.80). Both APC and ACS suffer reductions, having a detection effectiveness of approximately 0.42 and 0.19. This represents a decrease of approximately 14% for APC as compared to its average detection effectiveness for SDIH faults. APC did no worse for SDA faults than it did for SDI Faults. For ACS, the reduction is approximately 34% from its detection effectiveness for SDIH faults, and a decrease of approximately 17% as compared to its effectiveness for SDI faults. Branch coverage drops to a detection effectiveness of 0.08, a decrease of 50%.

For the IISD fault type, both APDU and APC have an average effectiveness of 1.0. ACS has an effectiveness of 0.81, and Branch Coverage has the lowest average effectiveness, 0.22. For fault type IC, both APDU and APC have an average effectiveness of 0.83, while ACS drops to 0.33 as compared to IISD. Branch Coverage again has the lowest detection effectiveness at 0.00.

Compared to the other coupling-based criteria, APDU did the best job of detecting the type of faults that were seeded, having an average detection effectiveness of 0.82. In contrast, APC has an average detection effectiveness across all fault types of 0.63, a reduction of approximately 23%, and for ACS, effectiveness reduces further to 0.37, which is a reduction of approximately 55% as compared to APDU and 41% as compared to the detection effectiveness of APC. Finally, Branch Coverage has the worst average detection effectiveness across the types of seeded faults, 0.12. Compared to APDU, this is a reduction of approximately 85%, and for APC and ACS, the reduction is approximately 81% and 66%, respectively.

5.3.2 Explanation of effects

The variation in the detection effectiveness among the coupling criteria is not surprising. The weakest of the coupling criteria, ACS, 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 remaining two. poor performance. The shortcoming of ACS is that its requirements are weak in that not all locations that can contain faults due to inheritance and polymorphism must be executed. By their very nature, these faults will be located within the hierarchy associated with the objects being integrated, not in the method under test. Thus, faults at these locations will not necessarily be executed as a result of testing according to the ACS criterion.

The APC criterion performs better than ACS. This is due to the stronger testing requirements imposed by APC. As described in section 4.1, APC 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. [6]

The most effective of the three coupling-based test adequacy criteria is APDU. This too is not surprising since its requirements are stronger than ACS and APC. In particular, 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

[6] A feasible location corresponds to a statement in a method or procedure for which there exists at least one input that will cause the statement to be executed.

be bound to the coupling variable. In terms of the fault/failure model, the requirements imposed by APDU have the greatest chance of causing a fault to be executed, and this accounts for its better performance over all the experimental trials.

5.3.3 Effectiveness of the Coupling-based Criteria

Log-linear analysis permits one to analyze categorical data in much the same manner as in analysis of variance. The sampling distribution underlying table 5 is a product of independent multinomials. According to Bishop, Fienberg and Holland, the kernel of the appropriate likelihood function is the same as that for a simple multinomial or a simple Poisson [12]. 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.

We first fitted the experimental results to a model corresponding 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 u_1 (*Program*), u_2 (*Fault Type*), u_3 (*Criterion*), and u_4 (*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:

$$Log(m_{i,j,k,l}) = u_o + u_1 + u_2 + u_{1,3} + u_{1,4} + u_{2,4} + u_{1,2} + u_{3,4} + u_{1,3,4} + \dots$$

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. In Figure 8, we can see that the fitted cell counts closely match the observed cell counts.

The procedure for testing the significance of a factor is to fit the best model with that factor included and then fit the same model with that factor removed and observe the change in the chi-square goodness-of-fit statistic.

For the initial hypothesis test, we tested for an interaction between *criterion* and *fault type* by fitting the model described above with and without the fault-type/criterion term. If there is no interaction, we can simply pick the best criterion and only use it for our testing. If there is an interaction, then we will have to use two or more of the criteria to adequately test for all of the fault types. For this test, the difference in the total χ^2 that the term of $criterion\ x\ faulttype$ accounted for is negligible. Thus, we do not reject the null hypothesis (H_0), and hence conclude that there is no interaction between these two factors.

For the remaining hypothesis tests, we selected out only the data for a particular pair of criteria (indicated

Table 6: Results of hypothesis tests [5].

N	Hypothesis	χ^2	df	$\Delta\chi^2$	Δdf	Conclusion
1	H_0: APDU is no more effective than BC H_1 :APDU is more effective than BC	91.74	164	816.74	36	Reject H_0
2	H_0 :APC is no more effective than BC H_1 :APC is more effective than BC	35.93	68	175.00	12	Reject H_0
3	H_0 :ACS is no more effective than BC H_1 :ACS is more effective than BC	19.00	63	97.94	12	Reject H_0
4	H_0 :APDU is no more effective than APC H_1 :APDU is more effective than APC	51.87	68	441.47	12	Reject H_0
5	H_0 :APDU is no more effective than ACS H_1 :APDU is more effective than ACS	47.89	68	103.88	12	Reject H_0
6	H_0 :APC is no more effective than ACS H_1 :APC is more effective than ACS	69.28	68	256.97	12	Reject H_0

by the column labeled *Hypothesis* in table 6) and then tested for an interaction between these two by fitting the model described with and without the corresponding fault-type/criterion term. Table 6 summarizes the results of these tests. The column labeled *Hypothesis* states the null (H_0) and alternative hypothesis ($H1$) for each test. The columns labeled χ^2 and $\Delta\chi^2$ give the change in value of the chi-square goodness-of-fit statistic, and the columns labeled *df* and Δdf give the corresponding change in degrees of freedom. Finally, the last column gives the result of each test, indicating whether the null hypothesis is rejected or not.

As the table shows, for hypotheses one through six, there was a net change in the degrees of freedom and χ^2 goodness of fit value. In all cases, there is statistical significance at a p-value value less than 0.001. Therefore, we reject the null hypothesis (H_0) in favor of the alternative (H_1) for all six of these hypotheses. The first three hypotheses allow us to conclude that each of the three coupling-based criteria are more effective than Branch Coverage at detecting the types of faults seeded into the subject programs. The remaining three hypotheses allow us to compare the effectiveness among the coupling-based criteria. Since the null hypothesis (H_0) was rejected for each, we can conclude that there is statistical evidence to suggest that APDU is more effective than APC and ACS at detecting the subject fault types, and also that APC is more effective than ACS.

5.3.4 Discussion

The three hypotheses in table 6 that tested the effectiveness of each coupling-based criteria against Branch Coverage indicate that the coupling criteria are better at detecting the object-oriented faults used in the experiment. A remaining question is which of the three coupling criteria is the most effective. Hypotheses one, two, and three have established that each of the coupling criteria are better than Branch Coverage. Observation of the plot in figure 7 suggests that APDU is, on average, more effective that APC and ACS.

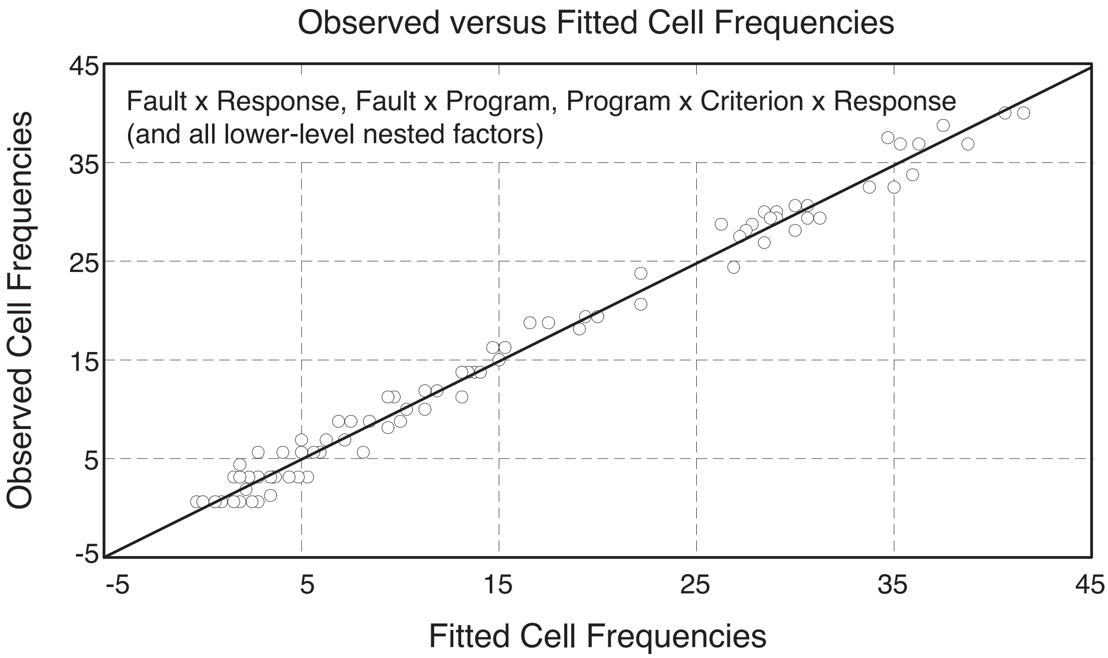


Figure 8: Observed versus fitted cell frequencies [5].

Similarly, APC is, also on average, more effective that ACS. This observation is, in fact, supported by the last three hypothesis tests.

Given the above conclusion, a key question that remains is *which criterion or combination of criteria should be used*? The plot in figure 7 also suggests that there is no coupling-based criterion that is particularly better for detecting one fault type versus another (i.e., the criterion do not specialize in the faults that they detect). If any criterion is good for a particular fault type, they all are. Therefore we could pick the strongest of the coupling criteria and use that for all fault types.

Realistically, other factors must be considered when choosing to use a particular test adequacy criteria *C*. Cost can be defined in many ways, including the number of test cases required to satisfy *C* and the amount of time required to analyze a program to determine if a desired level test coverage has been attained. An observation made during the course of this research is the difference between the number of tests required to achieve APDU as compared to APC and ACS was an order of magnitude. The total number of APDU test cases created for all the subject programs is 817, while it is 55 for APC and 26 for ACS. If we define cost in terms of the number of required test cases, clearly APDU is significantly more expensive than APC and ACS. From a practical perspective, is the additional cost worth the benefit received? The answer to this important question is left as future work.

5.3.5 Threats to validity

As with any experiment, there are threats to validity that affect the generality and interpretation of the results. In our experiments, two possible sources of such threats are the five specially engineered subject programs (section 5.1.1), and the injected subject faults (section 5.1.3).

While the engineered subject programs gave us some confidence in the power of the subject criteria (with respect to the subject fault types), it is possible that the results obtained are not representative of object-oriented programs that occur in the wild, thus leading to over confidence. However, as the results show for subject programs P6, P7, P8, P8, and P10, four of the seeded fault types were detected by all of the subject criteria (including Branch Coverage). From this we conclude that the subject criteria have some degree of effectiveness in detecting four of the five seeded fault types (the exception being IC).

The other possible threat to the validity are the subject fault types themselves. The subject fault types (section sec:InjectedFaults) were based on the fault model for inheritance and polymorphism developed in Alexander's dissertation [2] and presented in our previous work [38]. That fault model was developed as there was no other known fault model applicable to object-oriented programs. The model was derived by collecting and analyzing anecdotal evidence from several professional software engineers, each have ten or more years of experience developing object-oriented programs. While that model is not based on empirical evidence, the fact that the model is based on real experience lends a level of credibility. Further anecdotal evidence since our prior work was published suggests that the fault model does represent real faults that occur in object-oriented programs.

With respect to our experiments reported in this work, the lack of actual empirical evidence means that, though we are very confident in the fault model, it nonetheless is possible that the subject fault types are not representative of real faults, either completely or in part. While the experimental results are encouraging and are of value, the strength to any claim of generality must be considered conservatively. That is, while the results of the effectiveness of the subject criteria look very good, further empirical research is necessary to determine the limits of their effectiveness, and also the effectiveness of the fault model. We leave these as future work.

6 Related Work

Smith and Robson observe that object-oriented classes cannot be tested directly [45]. Instead, classes must be tested indirectly by sampling among their instances (objects). This means that the quality of the testing effort will be a function of how representative the sample is. Fiedler reports on industrial experience using a class testing approach [22]. The approach uses a combination of black and white box testing techniques

Edwards created a test set generation methodology based on specifications [21].Our approach here differs in three primary ways. First, we more carefully consider the ramifications of object oriented technologies. Second, our results do not require specifications, and finally, our approach is tailored to find faults specifically related to the polymorphism inherent in OO technologies, something Edwards does not directly consider.

Perry and Kaiser [43] concluded that single inheritance, method overriding, and multiple inheritance do not reduce the amount of testing effort, and in many cases, increase the required effort to achieve test adequacy. Among others, this claim is based on the Anticomposition axiom: "Adequately testing each individual program component in isolation does not necessarily suffice to adequately test the entire program. Composing two program components results in interactions that cannot arise in isolation". This implies that just because a class has been tested in isolation does not mean that it will work correctly with other classes. This supports the claim in this paper, that object-oriented software requires more emphasis on integration testing.

One problem is that of the order in which classes should be integrated together and tested. Briand [13] recently compared several strategies for ordering class integration. The class ordering problem is independent of the problem of this paper, finding test inputs to find inheritance, dynamic binding and polymorphism failures. Solutions to the class ordering problem could be combined with the solutions in this paper.

In his dissertation, Overbeck presents an approach based on testing contracts among client and server classes [39]. A contract specifies the preconditions and postconditions of each public method in a class. Interactions among classes are tested to ensure that the client is using the class correctly, and that the results returned from the methods are understood by the client. This is done by imposing a special test filter that sits between the client and the class and that catches method invocations on the class. The filter checks to determine if the methods are of the right type and value, the method is called in the proper sequence, and if the precondition of the called method is true. If these checks pass, the call is passed on to the class for processing. Otherwise, an error is reported and an exception thrown. This technique does not help create test cases, and relies on specifications written by the programmers.

Jorgensen and Erickson describe an approach to integration testing that is similar to many black-box testing techniques [31]. They define paths through a collection of classes that are traversed through method calls. The paths end when system output has been observed. Failures are detected whenever the system output does not agree with what is expected, and faults are identified by tracing back along the path. No formal criterion for choosing paths was given.

Most research in object-oriented testing has been at the intra-class level. This includes work by Hong et al. [28], Parrish et al. [42], Turner and Robson [47], Doong and Frankl [20], and Chen et al. [14]. Intra-class testing strategies focus on one class at a time, so does not find problems that exist in the interfaces between classes, or in inheritance, dynamic binding and polymorphism among classes. In their TACCLE methodology [15] Chen et al. define class semantics algebraically as axioms and construct test cases as paths through a state-transition diagram with path selection based on non-equivalent ground terms. They extend this methodology to multiple classes by defining inter-class semantics in terms of contracts. In general, black box testing is less effective to detect commission errors while white box testing is less effective to detect omission errors. The work reported in the papers by Chen et al. [14] and [15] has the advantage of combining both black box and white box testing techniques.

Harrold and Rothermel describe an approach that applies data-flow analysis to classes [26]. In that approach, they emphasize three levels of testing: (1) intra-method testing; (2) inter-method testing; (3) intra-class testing. Intra-method testing applies traditional data flow techniques to data flow definitions and uses that occur within single methods. Inter-method testing tests method within a class that interact through procedure calls. Finally, intra-class testing tests sequences of public method class against a given class instance. To perform these analyses, Harrold and Rothermel represent a class as a Class Control Flow Graph (CCFG) consisting of a single entry and exit. The CCFG is the composite of the control flow graphs of the class' methods connected together through their call sites. They apply the data flow analysis algorithms of Pande, Landi and Ryder [41] to the CCFG to compute definition-use pairs for each of the three types of analyses. Harrold and Rothermel do not describe how they apply this information in the testing procedure. Also, they only briefly discuss the application of their approach to inheritance relationships, but unfortunately, they do not provide any details.

Only a few researchers have investigated problems with inheritance, dynamic binding and polymorphism. Chen and Kao [16] describe an approach to testing object-oriented programs called Object Flow Testing, in which testing is guided by data definitions and uses in pairs of methods that are called by the same caller, and testing should cover all possible type bindings in the presence of polymorphism. Though their work is similar, there are significant differences. First, their criteria is coarse-grained. The criteria presented by Alexander and Offutt [3, 4], and described in detail by this thesis, are a superset of those of Chen and Kao. As Chen and Kao have defined them, their criteria require only that either all bindings or all DU pairs be covered. In particular, they do not integrate the two, though their complete example suggests that they are at least aware that this is important [17]. Secondly, not all bindings are feasible. Chen and Kao do not discriminate between feasible and infeasible bindings. Third, there likely will be DU pairs where every definition-clear path connecting them is infeasible. Thus, their criterion All-du-Pairs is impractical from an applied testing perspective.

Kung, Gao, Hsia, Toyoshima, and Chen observe that one of the key difficulties in testing object-oriented software is understanding the relationships that exist among the components [32]. This complexity results from the use of inheritance, aggregation and association relationships among classes. Deep inheritance hierarchies and highly nested class aggregations make it difficult to determine the optimal order in which classes should be tested. The consequences of testing in an order not optimal are that testing is not adequate because class relationships are missed, or substantial re-testing is often required. In an effort to eliminate this problem, the authors present an algorithm that generates the optimum order for unit and integration testing of classes. The objective is to minimize the amount of effort required to adequately test the classes by minimizing the number of test stubs that must be built, and to also reuse as many previously generated test cases as possible. As an example of the effectiveness of their algorithm, they report its use with the InterViews library in an experiment against a randomly generated test order. There they found that the total number of stubs required for that test order was 400, where if the optimal test order is used, only 8 test stubs are required.

7 Future Work

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 3, 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 3.1 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 at the cost of a more expensive analysis.

Another key area of related research is the generation of test cases that satisfy a particular coupling-based criterion. During the research reported in this paper, tests were generated using hand crafted custom test case generators. While this is acceptable for a scientific investigation, it is of limited applicability in practical settings. Thus, there is a need to enhance the test case generation process through automation.

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 criterion *C*, etc. The coupling-based testing approach naturally yields a number of artifacts (coupling sequences and coupling sets), and object-oriented programs also have a distinct set of artifacts (classes, methods, and inheritance hierarchies). There is the potential to combine these and use them as the basis of a measurement theory for the approach. For example, there likely is a strong positive correlation between the depth of an inheritance hierarchy and 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.

8 Conclusion

This paper has introduced new data flow analysis techniques for object-oriented software, new testing criteria to address problems that can arise from using inheritance, dynamic binding and polymorphism, and results from an experimental validation of the techniques. The techniques are based on previous work for procedure-oriented software called coupling-based testing (CBT). The traditional notion of software coupling has been updated to apply to object-oriented software, handling the relationships of aggregation, inheritance, dynamic binding and polymorphism. This allows the introduction of a new integration analysis and testing technique for data flow interactions within object-oriented software, called object-oriented coupling-based testing (OOCBT). OOCBT can help practitioners who are performing integration testing on object-oriented software.

A key contribution is a technique for analyzing and testing polymorphic behavior. 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 object-oriented programs can now analyze and better understand the interactions within their software. Though the coupling sequence has been cast for testing problems involving inheritance, dynamic binding and polymorphism, is generally applicable to any program that makes uses of encapsulated data types (e.g. Modula-2, Ada83, etc.).

This paper also presented a set of test-adequacy criteria that take inheritance, dynamic binding 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 level of required to achieved 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.

References

[1] Roger T. Alexander. Testing the polymorphic relationships of object-oriented components. Technical Report ISE-TR-99-02, Department of Information and Software Engineering, George Mason University, February 1999.

[2] Roger T. Alexander. *Testing the Polymorphic Relationships of Object-oriented Programs*. Dissertation, George Mason University, 2001.

[3] Roger T. Alexander and Jeff Offutt. Analysis techniques for testing polymorphic relationships. In *Thirtieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS30)*, pages 104–114, Santa Barbara, CA, 1999.

[4] Roger T. Alexander and Jeff Offutt. Criteria for testing polymorphic relationships. In *Proceedings of the International Symposium on Software Reliability and Engineering (ISSRE00)*, San Jose, California, 2000. IEEE Computer Society.

[5] Roger T. Alexander, Jeff Offutt, and James M. Bieman. Fault detection capabilities of coupling-based oo testing,. In *Thirteenth International Symposium on Software Reliability Engineering (ISSRE '02)*, Annapolis, Maryland, 2002. IEEE Computer Society.

[6] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[7] Stephane Barbey and Alfred Strohmeier. The problematics of testing object-oriented software. In *SQM'94 Second Conference on Software Quality Management*, volume 2, pages 411–426, Edinburgh, Scotland, UK, 1994.

[8] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, New York, 2nd edition, 1990.

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

[10] Edward V. Berard. *Essays on Object-Oriented Software Engineering*, volume 1. Prentice Hall, 1993.

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

[12] Yvonne M. M. Bishop, Stephen E. Fienberg, and Paul W. Holland. *Discrete Multivariate Analysis: Theory and Practice*. MIT Press, Cambridge, Massachusetts, 1975.

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

[14] Huo Yan Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: An integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 7(3):250–295, 1998.

[15] Huo Yan Chen, T. H. Tse, and T. Y. Chen. TACCLE: A methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering Methodology*, 10(1):56–109, 2001.

[16] Mei-Hwa Chen and Ming-Hung Kao. Testing object-oriented programs - an integrated approach. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, pages 73–83, Boca Raton, FL, November 1999. IEEE Computer Society Press.

[17] Mei-Hwa Chen and Ming-Hung Kao. Testing object-oriented programs - an integrated approach. In *10th International Symposium on Software Reliability Engineering (ISSRE'99)*, pages 73–83, Boca Raton, FL, November 1999. IEEE Computer Society.

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

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

[20] Roong-Ko Doong and Phyllis Frankl. Case studies on testing object-oriented programs. In *Fourth Symposium on Software Testing, Analysis and Verification*, pages 165–177. ACM Press, 1991.

[21] Stephen H. Edwards. Black-box testing using flowgraphs: an experimental assessment of effectiveness and automation potential. *Software Testing, Verification & Reliability*, 10(4):249–262, 2000.

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

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

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

[25] Leonard Gallagher, Jeff Offutt, and Tony Cincotta. Integration testing of object-oriented components using finite state machines. *Software Testing, Verification, and Reliability*, 17(1):215–266, January 2007.

[26] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. In *Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 154–163. ACM Press, New York, New York, 1994.

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

[28] Hyoung Seok Hong, Yong Rae Kwon, and Sung Deok Cha. Testing of object-oriented programs based on finite state machines. In *1995 Asia Pacific Software Engineering Conference*, pages 234–241. IEEE Computer Society Press, Los Alamitos,California, 1995.

[29] JGuru. ANTLR complete language translation solutions, 2003. http://www.antlr.org/, last accessed Jan 24, 2003.

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

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

[32] David Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. A test strategy for object-oriented systems. In *Nineteenth Annual International Computer Software and Applications Conference*, pages 239–244. IEEE Computer Society Press, Los Alamitos, CA, 1995.

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

[34] Bertrand Meyer. *Introduction to the Theory of Programming Languages*. Prentice Hall International Series In Computer Science. Prentice Hall, 1990.

[35] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, New Jersey, 2nd edition, 1997.

[36] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005.

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

[38] Jeff Offutt, Roger Alexander, Ye Wu, Quansheng Xiao, and Chuck Hutchinson. A fault model for subtype inheritance and polymorphism. In *Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE '01)*, pages 84–95, Hong Kong, PRC, 2001. IEEE Computer Society.

[39] Jan Overbeck. *Integration Testing for Object-Oriented Software*. Ph.D. Dissertation, Vienna University of Technology, 1994.

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

[41] H. D. Pande, W. A. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, 1994.

[42] Allen S. Parrish, Richard B. Borie, and David W. Cordes. Automated flow graph-based testing of object-oriented software modules. *Journal of Systems and Software v 23*, 23(2):95–109, 1993.

[43] Dewayne E. Perry and Gail E. Kaiser. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming*, 2(5):13–19, 1990.

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

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

[46] SourceForge.net. JMK - Make in Java: Summary, 2003. http://sourceforge.net/projects/jmk, last accessed Jan 24, 2003.

[47] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *Conference on Software Maintenance*, pages 302–310. IEEE Computer Society Press, Los Alamitos, CA, 1993.

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

[49] E. J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, SE-12(12):1128–38, 1986.

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