

Analysis Techniques for Testing Polymorphic Relationships *

Roger T. Alexander

Software Productivity Consortium
2214 Rock Hill Road
Herndon, VA 20170-4117 USA
email: alexande@software.org

A. Jefferson Offutt

Information and Software Engineering
George Mason University
Fairfax, VA 22030-4444 USA
email: ofut@ise.gmu.edu

Abstract

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

KEYWORDS: Integration, Testing, Components, Inheritance, Polymorphism

1: Introduction

A number of researchers have asserted that some traditional testing techniques are not effective for object-oriented software [7, 6]. The emphasis in an object-oriented language is on defining abstractions (e.g. abstract data types) that model concepts relative to some problem and solution domain [10]. These abstractions appear in the language as user-defined types that have both state and behavior. Although abstract data types can help achieve a higher quality design, how we test software may change. A major factor is that shifting from procedure-oriented software to object-oriented software causes us to shift the complexity in the software from residing primarily in the software units, to the way in which we connect software components. Thus, we are finding that we need less emphasis on unit testing and more on integration testing.

Another factor is due to the inherent complexity in the nature of the relationships found in object-oriented languages [4]. The compositional relationships of inheritance and aggregation, combined with the power of polymorphism, makes it harder to detect faults that result from the integration of components. This is because component integration is different in object-oriented languages [3].

The primary distinction between the types of languages discussed in this paper is in the mechanisms used for abstraction. 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

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

inheritance are *descendants* of the existing type [9]. Inheritance differs from aggregation in 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 (which requires dynamic binding) can strongly affect component integration. When a call is made to a polymorphic method, which version is executed depends on the type of the object [10]. Thus inheritance and polymorphism provides new forms of integration that must be dealt with when testing objects, neither of which has a procedure-oriented counterpart.

1.1: Testing object-oriented software

This paper presents results from an ongoing research project that has the goal of improving the quality of object-oriented software. This paper presents initial results towards a solution to the problem of finding errors in the polymorphic relationships among integrated components. The general strategy for this solution is to formalize, via new coverage criteria, routine aspects of testing at the integration level. Formal coverage criteria offer the tester ways to decide what test inputs to use during testing, making it more likely that the tester will find any faults in the program and providing greater assurance that the software is of high quality and reliability. Such criteria also provide stopping rules and repeatability.

Unit and module testing (or just unit testing) is the testing of program units and modules independently from the rest of the software. *Integration testing* refers to testing interfaces between units and modules to assure that they have consistent assumptions and communicate correctly [2]. *System testing* is testing applied to an entire integrated system.

Test requirements are specific things that must be satisfied or covered, for example, reaching statements are the requirements for statement coverage, killing mutants are the requirements for mutation, and executing DU pairs are the requirements in data flow testing. A *testing criterion* is a rule or collection of rules that impose requirements on a set of test cases. Test engineers measure the extent to which a criterion is satisfied in terms of *coverage*, which is the percent of requirements that are satisfied.

2: Definitions and Background

The concepts presented in this paper are largely independent of language. However, the examples, terminology, and many of the specifics by necessity must be related to one or more languages. We choose Java, and try to point out where the rules would change for other languages.

The fundamental building block in object-oriented programming is the *class*, which is the mechanism by which new types are defined. A class encapsulates state information in a collection of variables, referred to as *state variables*, and also has a set of behaviors that are represented by a collection of methods that operate on those state variables. A class defines a type that all of its *objects* share.

There are two types of relationships that can be used to compose class types to form new types. The first of these, *aggregation*, is the traditional notion of one type containing instances of another type as part of the its internal state representation. The second form of compositional relationship is inheritance. Inheritance allows the representation of one type to be defined in terms of the representation of a set of other types. When this occurs, the type being defined is said to inherit the properties of its ancestors (that is, behavior and state). The definition of the ancestors becomes part of the definition of the new descendant type.

2.1: Polymorphism and dynamic binding

Polymorphism permits variable instances to be bound to references of different types according to the structure of the inheritance hierarchy. *Dynamic binding* permits different method implementations to execute depending upon the actual type of an instance that is bound to a particular reference; this actual type is independent of its declared type [10].

2.2: Coupling-based testing

The work in this paper is based on previous work by Jin and Offutt [8]. They presented an approach to integration testing of procedure-oriented software that is based on coupling relationships among procedures.

Coupling was originally proposed to measure design [5, 11], and the original papers presented up to twelve various types of coupling in lists that were ordered in terms of severity. For testing, only three unordered types are needed: *parameter coupling*, *shared data coupling*, and *external device coupling*. Parameter couplings occur whenever one procedure passes parameters to another. Similarly, shared data couplings occur when two procedures reference the same global variable. Finally, external device couplings occur when two procedures access the same external storage device.

Jin and Offutt's approach requires that programs execute from each definition of a variable in a caller to a call site, and then to the uses of the corresponding formal arguments in the called procedure. The execution path from the definition to the use must be *definition-clear*, that is, the variable must not be redefined along the path. The underlying idea is that to have a high degree of confidence in the resulting software, all of the definitions of variables in one procedure must be correctly used in the called procedures. This approach is called *coupling-based testing* (CBT).

3: Handling Polymorphism and Inheritance

This paper extends the previous work in coupling-based testing by using coupling to detect the faults that result from the polymorphic relationships among components in an object-oriented program. This is done by first extending the CBT coupling path definitions to allow for the additional relationships in object-oriented programs. This requires the definitions for all forms of coupling-defs, coupling-uses, and external references to be modified. Next, a set of techniques and formalisms are defined to test object-oriented coupling relationships. Finally these techniques and formalisms are used to define a set of test adequacy criteria.

3.1: Coupling in the presence of polymorphism

In the following definitions, o is an identifier whose type is a *reference to an instance* of a class. A reference points to a memory location that contains an instance (value) of some type. The reference o can only refer to instances whose actual instantiated types are either the base type of o or a descendant of o 's type. The instance referenced by o is indicated by o_r .

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 also allow new types, but programmers can go further by grouping user defined types into *families* of types. All members of a given type family share a common behavior, which allows instances of any member of a type family to be freely substituted for an instance of any other member. Type families are created by utilizing inheritance and polymorphism.

Every type definition (i.e. a class definition) defines a type family. Members of the family include the base type that defines the family, and all types that are descendants of the base type. Figure 1(a) illustrates this with four type families, each defined by one of the classes in the hierarchy.

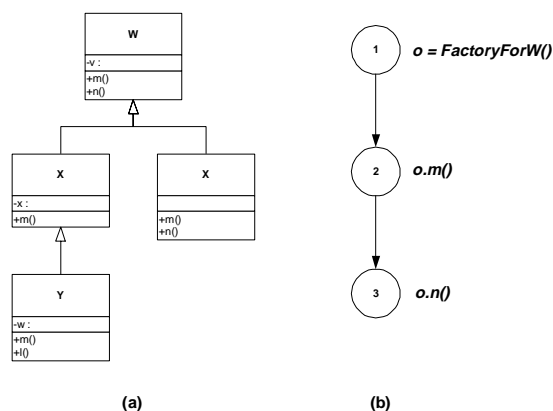


Figure 1. (a) Sample Class Diagram (b) Sample Control Flow Fragment

In OO languages, method calls can occur in two circumstances: (1) with respect to some instance, or (2) where there is no instance. Instance methods are called with respect to instance variables, and *class methods* have no instance. Instance methods can make the instance *explicit*, as in `o.m()`, or *implicit*, as in `p()`. For the call `o.m()`, `m()` executes in **the context of the instance that is bound to the reference o**. For a shorthand convenience, we say that `m()` executes in the context of `o`, or `o` is `m()`'s instance context.

For the call to `p()`, there must be an implicit object reference. That is, `p` must appear in the program text of a method that was called through an explicit instance (e.g. `o.m()`), and `p` must be defined in the same class with `o`. Java and C++ allow the implicit object to be referred to with the keyword “*this*”.

An object (instance) o_r is defined (i.e. assigned a value) when one of the variables of the object is defined. An *indirect definition*, or *i-def*, occurs when a method `m` defines one of o_r 's variables. Similarly, an *indirect use* (*i-use*) occurs when `m` references the value of one of o_r 's variables.

Again, consider the class diagram shown in Figure 1a. Assume that `W` includes a method `FactoryForW()` that returns an instance of `W`. Figure 1b 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()`.

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 all of the methods that can potentially execute. Because of polymorphism and dynamic binding, this depends on the type of the instance that is bound to the object reference. To analyze this, we introduce the term satisfying set:

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

Thus, 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 such method, identify all state variables that are defined and used. The result is the set of *definitions* and *uses* for each satisfying method. Returning again to Figures 1a and

1b, the *i-def* set for the call at node 2 is the following set of ordered pairs:

$$i_def(2, o_r, m) = \{(W :: m, \{W_r.v\}), (X :: m, \{W_r.v, X_r.x\}), \\ (Y :: m, \{W_r.v, Y_r.w\}), (Z :: m, \{W_r.v\})\} \quad (1)$$

Each pair indicates a satisfying method s for m and the corresponding set of state variables that s defines. In this example, $X::m$ defines state variables v and x contained in classes W and X .

As Table ?? shows, the corresponding *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, 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:

$$i_use(3, o_r, n) = \{(W :: n, \{W_r.v\}), (Z :: m, \{W_r.v\})\} \quad (2)$$

3.2: Differences in coupling paths in OO programs

When polymorphism is used, 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 in the presence of inheritance and polymorphism, and what effects the calls have on the corresponding state space.

Alexander has analyzed how method calls are made in object-oriented languages and has identified twelve cases that must be considered when analyzing coupling paths [1]. The twelve cases can be partitioned into three categories. Category I cases are when the call is made in an implicit or explicit instance context. Category II cases are when the call is made in no instance context. Category III cases are when the call is made in no instance context, but there polymorphism may be involved. Category II cases are not effected by inheritance or polymorphism and are handled by the original CBT definitions. Category I and Category III cases require extensions to the definitions and additional analysis techniques. Space does not allow a detailed analysis of each of these cases, they appear in the technical report [1].

4: Object-oriented Coverage Definitions

This section describes extensions to coupling path definitions that are necessary to support object-oriented languages. Section 4.1 provide additional coupling definitions necessary to account for the structural and semantic peculiarities of object-oriented languages. Section 4.2 introduces the definition of a *coupling sequence* and associated concepts for understanding coupling paths with object-oriented programs. Section 4.3 describes coupling paths that result from indirect definitions and uses of state variables through an instance context. Section 4.4 extends the description of instance coupling paths to account for the possible presence of polymorphic method calls. Section 4.5 briefly discusses considerations for coupling-based testing criteria for object-oriented programs.

4.1: Extended coupling definitions

The original CBT definitions must be modified in a number of ways to account for the various calling contexts that occur in object-oriented programs. In the following definitions, m refers to a program unit, including methods that appear in class definitions. V_m is the set of variables that are referenced by m , and N_m the set of nodes in m .

$defs(i)$ is the set of variables that are defined at node i , and $uses(j)$ is the set of nodes that are used at node j . A *def-clear-path* (m, i, j, v) returns true if there is a definition-clear

path from node i to j with respect to v , where $i, j \in N_m$. $first(p)$ is the first node in path p and $last(p)$ is the last node. $paths(i, j, m)$ is the set of paths that start at node i and that end on node j , where $i, j \in N_m$.

$entry(m)$ is the entry node of method m , and $exit(m)$ is the exit node of m . $signature(m, n)$ returns true if the signature of method m matches that of n . $overrides(o, m, p)$ is true if method m overrides n , where $class(n) = class(o) \wedge class(m) \in family(class(o)) \wedge signature(m, n)$.

$class(m)$ is the class that contains the definition of m , and $class(o)$ is the class that is the declared type of object reference o . $family(c)$ is the set of classes that belong to the type family specified by class c . Note that $c \in family(c)$. $state_vars(c)$ is the set of state variables that directly or indirectly comprise the state space of class c . $instance(t)$ is a function that returns an instance of type t .

$i_defs(m)$ is the set of variables in the state space of the class containing m that are indirectly *defined* by a call to m made through some instance context. Formally:

$$i_defs(m) = \{v \in state_vars(class(m)) \mid \exists j \in N_m \bullet v \in defs(j)\}$$

$i_uses(m)$ is the set of variables in the state space of the class containing m that are indirectly *used* by a call to m made through some instance context. Formally:

$$i_uses(m) = \{v \in state_vars(class(m)) \mid \exists j \in N_m \bullet v \in uses(j)\}$$

4.2: Coupling sequences

Coupling sequences are pairs of method calls made within the body of a specific method f and are made through a common instance context accessed through an object reference o . Further, there is at least one coupling path between the two methods with respect to some commonly defined and used state variable. An example is illustrated in Figure 2. As shown, method f contains a coupling sequence $s_{j,k}$ that starts at node j with the call to $o.m$ (the *antecedent method*) and extends through paths that end at node k where the sequence ends with the call to $o.r$ (the *consequent method*). The nodes containing the antecedent method and consequent method are referred to as the *antecedent node* and *consequent node*, respectively. Note that there is at least one path between the call sites that is definition clear with respect to o and to those indirect definitions made in the antecedent method that have corresponding indirect uses in the consequent method. Such paths are referred to as *transmission paths*.

A particular coupling sequence $s_{j,k}$ is with respect to a set of state variables that are defined by the antecedent method and subsequently use by the consequent method. This set of variables is referred to as the *coupling set* $\Theta_{s_{j,k}}$ of $s_{j,k}$, and each member of this set is a *coupling variable*. The coupling set for the sequence $s_{j,k}$ shown in Figure 2 is:

$$\Theta_{s_{j,k}} = \{class(o)::v\}$$

That is, $\Theta_{s_{j,k}}$ contains exactly those state variables referenced through o that are defined by the antecedent method and used by the consequent method in the coupling sequence $s_{j,k}$.

A coupling sequence consists of three parts, each consisting of a distinct set of path segments defined with respect to the elements of $\Theta_{s_{j,k}}$. These segments are used to generate the set of coupling paths for $s_{j,k}$. The following sections describe each of these sets in detail.

4.2.1: I-def paths

For a given coupling path in the coupling sequence $s_{j,k}$, there are a set of paths in the antecedent method that begin at nodes that have *last-definitions-before-return* of the

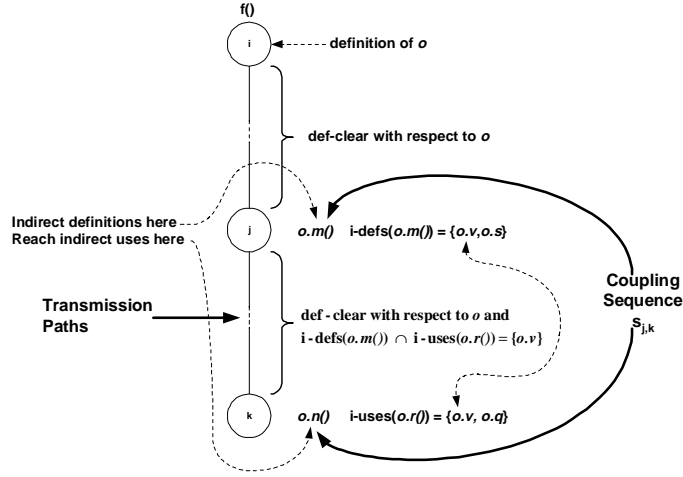


Figure 2. Example Coupling Sequence $s_{j,k}$

variables contained in the coupling set $\Theta_{s_{j,k}}$. For each such node l , the path to $exit(m)$ is definition clear with respect to the corresponding coupling variable defined at l . These paths constitute the *indirect-def path set* (or *i-def-path-set*) of the coupling sequence. Each of these paths is referred to as an *indirect definition path* (or *i-def-path*). Formally:

$$\begin{aligned}
 i_def_paths(c, m, V) &= \{(i, exit(c::m)) \mid i \in N_{c::m} \bullet \\
 &(\exists v \in V \bullet v \in defs(i)) \wedge \\
 &v \in state_vars(c) \wedge \\
 &def_clear_path(c::m, i, exit(c::m), v)\}
 \end{aligned} \tag{3}$$

where c is the class, m is either defined or inherited by c , and $V = i_defs(o, m) \cap i_uses(o, n)$.

4.2.2: I-use paths

The *indirect-use path set* (or *i-use-path-set*) of the coupling sequence $s_{j,k}$ is the set of paths that are definition clear with respect to the particular coupling variable used at j . That is, the set of paths in the consequent method r that begin at $entry(r)$ and end at a node $j \in N_r$ such that j has a *first-use-in-callee* of a variable in the coupling set $\Theta_{s_{j,k}}$

$$\begin{aligned}
 i_use_paths(c, r, V) &= \{(entry(c::r), j) \mid j \in N_{c::r} \bullet \\
 &(\exists v \in V \bullet v \in uses(j)) \wedge \\
 &v \in state_vars(c) \wedge \\
 &def_clear_path(c : : r, entry(c::r), j, v)\}
 \end{aligned}$$

where c and V are as defined in equation (3), and r is a method that is defined or inherited by c .

4.2.3: Transmission paths

For a given coupling sequence $s_{j,k}$, there is some set of paths T that connect the antecedent node m and the consequent node r , such that each $t \in T$ is definition clear with respect to $i_def_path_set(m) \cap i_use_path_set(r)$. This set of paths is referred to as the

transmission path set (or *t-path-set*) of $s_{j,k}$, and each t is a transmission path (*t-path*) with respect to a specific coupling variable in $\Theta_{s_{j,k}}$. These paths transmit the value of a coupling variable from the defining method to the using method. Formally:

$$\begin{aligned} t_paths(f, j, k, o, V) = & \{p \in paths(j, k, f) \mid j, k \in N_f \bullet \\ & def_clear_path(f, first(p), last(p), o) \wedge \\ & \exists v \in V \bullet def_clear_path(f, first(p), last(p), v)\} \end{aligned}$$

where f is the calling method, j and k are the antecedent and consequent nodes, respectively, o is the object reference that defines the instance context of coupling sequence $s_{j,k}$ appearing in f , and V is as defined in equation (3).

4.3: Instance coupling paths

From a testing perspective, 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 *instance coupling paths* that extend from a node containing a *last-def-before-return* in an antecedent method to a node in a consequent method that contains a *first-use-in-callee* with respect to the coupling variable of interest. We form these instance coupling paths by taking the cross product of the *i-def path set*, *t-path set*, and *i-use path set* for a particular coupling sequence. Formally,

$$\begin{aligned} InstanceCouplingPaths(f, j, k, o, m, n) = & \{(d, t, u) \bullet \\ & d \in i_def_paths(class(o), m, V) \wedge \\ & t \in t_paths(f, j, k, o, V) \wedge \\ & u \in i_use_paths(class(o), m, V)\} \end{aligned} \quad (4)$$

where $V = i_defs(o, m) \cap i_uses(o, n)$, f is the calling method, $s_1, s_2 \in N_f$, o is the object reference that defines the instance context, and $m, n \in class(o)$.

4.4: Polymorphic coupling paths

The instance coupling paths described in section 4.3 do not take into account the possibility of polymorphic behavior resulting from dynamic variation of types that can be bound to an object reference. With the possibility of polymorphic behavior, a given instance coupling results in one path set for each member of the associated type family. The size of these sets is determined by the number of overridden methods within a given type, either defined directly or inherited from another type. Pragmatically, this has the potential to result in a combinatorial explosion of path sets. The number of path sets is a function of the depth and breadth of the inheritance relations. However, as an optimization to reduce the number of sets, those types that do not have overriding methods will have an empty type set. This is possible since any coupling path that could be executed through the type will necessarily appear in the path sets of other ancestor types that are members of the same type family.

To see an example of polymorphic coupling paths, consider the class hierarchy shown in Figure 3a. This hierarchy forms a type family with respect to class A , the root of the hierarchy. Instances of every class in this family can be used anywhere that an instance of A is required. Because of this, the actual execution path resulting from a given method call can vary depending upon the actual type of the instance context, which results in different indirect coupling paths.

To see this, consider the method control flow fragment shown in Figure 3b that shows method $F::a$ containing call sites at nodes j , k , and l where method calls are made in the context of the instance bound to o . The methods that actually execute for each of these calls, shown in Figure 4, are dependent upon the actual type of the instance that is bound to o through the assignment at node i . Since the declared type of o is A , any instance whose type is a member of the type family defined by A may be bound to o . Thus, instances of any of the classes shown in Figure 3a may be used. When o is bound to an instance of A , the method calls at nodes j , k and l result in the execution of $A::m$, $A::p$, and $A::r$, respectively. When the instance is of type B , the call at node k results in the execution of overriding method p defined in a class. Since B does not override any other method defined by A , the calls at j and l result in the execution of $A::m$ and $A::r$, respectively. Similar reasoning applies when the instance bound to o is an instance of class C or D , but with overriding methods for m and r , respectively.

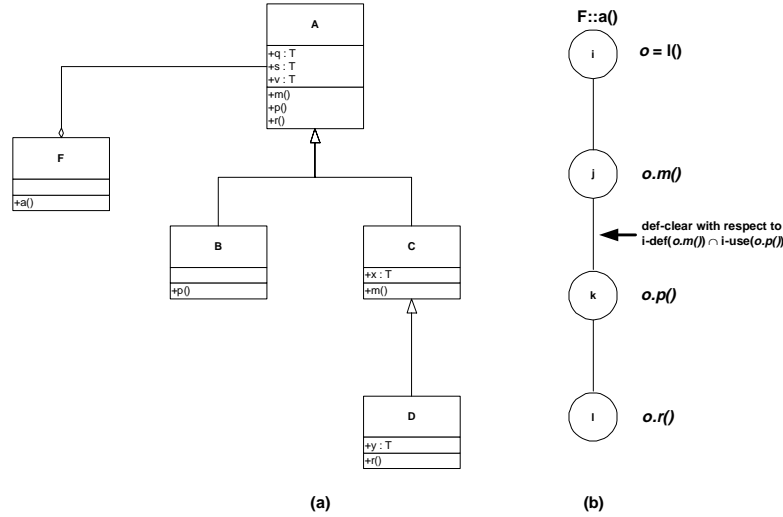


Figure 3. (a) Sample Class Diagram (b) Control Flow Fragment for Method $F::a$

To determine the instance coupling paths for a polymorphic call site, it is necessary to consider all of the coupling paths that could possibly result. This is accomplished by considering all of the possible types that the object reference providing the instance context of a call can take on. Determining this is simply a matter of examining the inheritance hierarchy whose root is the declared type of the object reference, and then identifying the set of descendant types D . Sets of instance coupling paths are then computed for each element of D . Formally, using equation (4), the maximal set of instance coupling paths is:

$$PolyCouplingPath(f, j, k, o, m, n) = \bigcup_c InstanceCouplingPaths(f, j, k, instance(c), m, n)$$

where $c \in family(class(o))$.

Each path set generated from a class c represents those coupling paths that would exist if the dynamic type of the instance bound to the object is an instance of c .

4.5: Coupling criteria

Each coupling path represents a semantic dependency between two methods for a specific instance of a type with respect to a specific state variable. The significance of these paths

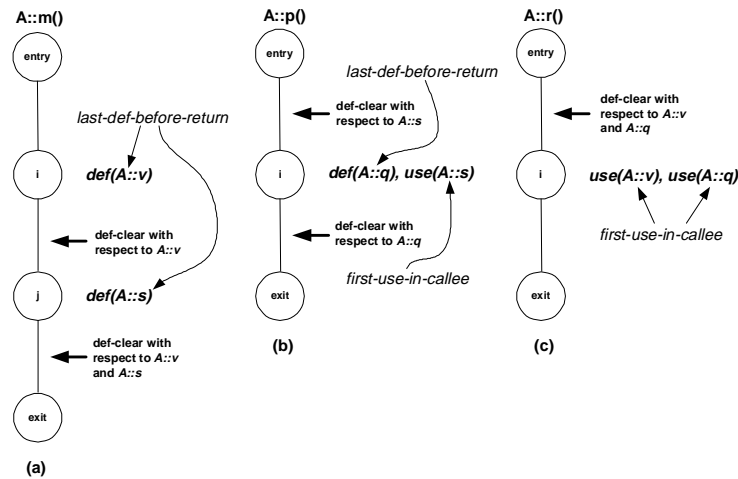


Figure 4. Control Flow Fragments for Methods $A::m$, $A::p$, and $A::r$

is that they represent connections among components that define behavior and state and a way for methods to interact. From a testing perspective, we want to identify faults that hide within these connections. The next focus of our research is to take the extended coupling definitions and derive test adequacy criteria that are practical and effective for object-oriented programs. We envision that these criteria will exist in a subsumptive hierarchy similar to the traditional data flow test adequacy criteria, but structured differently to account for the effects of inheritance and polymorphism.

5: Conclusions

This paper has introduced a new integration analysis and testing technique for object-oriented software. This technique is based on previous work for procedure-oriented software called coupling-based testing (CBT). Analysis was presented that showed how object-oriented software differs from procedure-oriented software, and the specific ways in which the CBT technique does not suffice for object-oriented software. This analysis was used to construct new definitions to analyze coupling relationships among OO software components. Specifically, the traditional notion of software coupling has been updated to apply to object-oriented software, handling the relationships of aggregation, inheritance and polymorphism. This allows the introduction of a new integration analysis and testing technique for object-oriented software, object-oriented coupling-based testing (OOCBT). OOCBT can benefit practitioners who are performing integration testing on object-oriented software.

Future plans are to develop algorithms for analyzing the relationships among object-oriented software components, build a proof-of-concept coverage analysis tool for this OOCBT, and to use the tool to empirically evaluate the method.

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. <http://www.ise.gmu.edu/techrep>.
- [2] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, New York, 2nd edition, 1990.

- [3] Edward V. Berard. *Essays on Object-Oriented Software Engineering*, volume 1. Prentice Hall, 1993.
- [4] Robert V. Binder. Testing object-oriented software: A survey. *Journal of Software Testing, Verification & Reliability*, 6(3/4):125–252, September/December 1996.
- [5] L. L. Constantine and E. Yourdon. *Structured Design*. Prentice-Hall, Englewood Cliffs, NJ, 1979.
- [6] 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, NJ, 1993.
- [7] 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.
- [8] Zhenyi Jin and A. Jefferson Offutt. Coupling-based criteria for integration testing. *The Journal of Software Testing, Verification, and Reliability*, 8(3):133–154, September 1998.
- [9] Bertrand Meyer. *Introduction to the Theory of Programming Languages*. Prentice Hall International Series In Computer Science. Prentice Hall, 1990.
- [10] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, New Jersey, 2nd edition, 1997.
- [11] 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.