

Criteria for Testing Polymorphic Relationships *

Roger T. Alexander and A. Jefferson Offutt

George Mason University

Department of Information and Software Engineering

Software Engineering Research Laboratory

Fairfax, Virginia 22030-4444

{ralexand, ofut}@gmu.edu

11th International Symposium on Software Reliability Engineering (ISSRE '00), pages 15–23, San Jose CA, October 2000.

Abstract

The emphasis in object-oriented programs is on defining abstractions that have both state and behavior. This emphasis causes a shift in focus from software units to the way software components are connected. Thus, we are finding that we need less emphasis on unit testing and more on integration testing. The compositional relationships of inheritance and aggregation, especially when combined with polymorphism, introduce new kinds of integration faults. This paper presents results from an ongoing research project that has the goal of improving the quality of object-oriented software. New testing criteria are introduced that take the effects of inheritance and polymorphism into account. These criteria are based on the new analysis technique of quasi-interprocedural data flow analysis. These testing criteria can improve the quality of object-oriented software by ensuring that integration tests are high quality.

1. Introduction

The emphasis in object-oriented languages is on defining abstractions (e.g. abstract data types) that model aspects of the problem [18]. These abstractions are implemented as user-defined types that have both state and behavior. Although abstract data types can help achieve a higher quality design, their use may affect how software is tested. A major factor is that shifting from procedure-oriented to object-oriented software often changes where the complexity resides. Instead of procedures that have complicated control

structures, object-oriented software often has simple procedures, with the complexity being in how the procedures and components are connected. Thus, testers are finding that less emphasis is needed on unit testing and more on integration testing.

The inherent complexity of the relationships found in object-oriented languages [7] also affects testing. The compositional relationships of inheritance and aggregation, combined with the power of polymorphism, can make it harder to detect faults in the way components are integrated. This is because component integration is different in object-oriented languages [6].

The primary distinction among the types of languages discussed in this paper is in the mechanisms used for abstraction. Procedure-oriented languages use procedures and functions as their primary abstraction mechanism, whereas object-oriented languages use data abstraction. In addition, object-oriented languages use the integration mechanism of inheritance and polymorphism (dynamic binding), both of which can strongly affect component integration. Inheritance differs from aggregation in that a new type can have access to the internal representation of the ancestor types. When a call is made to a polymorphic method, which version is executed depends on the type of the object [18]. 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.

This paper presents results from an ongoing research project that has the goal of improving the quality of object-oriented software. A previous paper [1] presented techniques to analyze the inheritance and polymorphism relationships in object-oriented software. This paper presents a solution to the problem of finding errors in the polymorphic relationships among integrated components. The general strategy for this solution is to define new coverage criteria, which will allow routine aspects of testing at the integration level to be formalized. Test adequacy criteria are important for several reasons. Often, when faced with testing a pro-

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

gram, developers do not know what to test or where to start. First, formal criteria help this situation by providing a basis for specifying test requirements. In turn, test requirements guide the testing process by providing a clear description of what must be tested. This provides a mechanism for deciding when to stop testing and a basis for adding repeatability to the testing effort. Second, formal criteria offer the tester ways to decide what test inputs to use during testing, making it more likely that the tester will find faults in the program and providing greater assurance that the software is of high quality and reliability.

1.1. Testing Object-oriented Software

A program *unit* is a procedure, function, or method. A *module* is a collection of related units, for example, a C file, an Ada package, or a Java class. *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 [4]. This is in contrast to *system testing* where the objective is test the entire integrated system as a whole. Because of the emphasis on testing interfaces, integration testing is usually a white box testing activity that requires the availability of source code. In contrast, system testing usually assumes the absence of source code, and is thus usually black box. Note that although this paper follows the standard IEEE definitions [15], these two terms are often used interchangeably in practice.

Test requirements are specific things that must be satisfied or covered, for example, reaching statements are the requirements for statement coverage. 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. A test is actually composed of several pieces. A *test case value* directly satisfies one or more test requirements. An *expected output* is the result of the test if the software behaves correctly. The rest of the test includes whatever other inputs are necessary for the software to get to the state required by the test case value and to cause the software to display or print the actual output.

This paper presents new criteria for testing object-oriented software. First, some background definitions are given, and some new terms are introduced. Then, the testing criteria are defined. Our current status and future plans for this research project are provided last.

2. Background

The testing criteria presented in this paper are partially based on previously defined criteria for procedure-oriented programs to languages that use inheritance and polymorphism. Thus, a brief overview of inheritance and polymorphism is given, and the concepts of coupling-based testing are described. Then the concept of coupling sequences, which was fully defined in a previous paper [1], is summarized.

2.1. Inheritance and Polymorphism

The fundamental building block in object-oriented programming is the *class*, which is used to define new types. A class encapsulates state information in a collection of *state variables*, and has a collection of methods that operate on those state variables. A class defines a type that is used to create *objects* (*instances*).

Aggregation and inheritance can be used to compose class types to form new types. *Aggregation* is the traditional mechanism of one type containing instances of another type. 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* (methods and variables) of its ancestors.

Polymorphism allows variables to have different types according to the structure of the inheritance hierarchy. *Dynamic binding* allows the same statement to execute different method bodies; which one is executed depends on the current type of the object that is used in the method call. Inheritance and polymorphism complicates the testing of object-oriented software.

A *type family* is a set of classes that share a common behavior with respect to a base class *C* (*family* (*C*)). Each descendant of *C* is a member *D* of *C*'s family. If *D* is in *C*'s family, polymorphism means that any instance of *D* may be freely used wherever an instance of *C* is expected. Every class *C* defines a type family, and that type family includes at least *C*.

2.2. Coupling and Coupling-based Testing

The criteria introduced in this paper are based on previous work by Jin and Offutt [16], which was in turn based on data flow testing [11]. They present an approach to integration testing of procedure-oriented software that is based on coupling relationships among procedures.

Coupling was originally proposed to measure design [9, 19], 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 used:

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 non-local variable. External device couplings occur when two procedures access the same external storage device.

Jin and Offutt’s approach requires that programs execute from *definitions* of variables in callers to call sites, and then to *uses* of the corresponding formal arguments in called procedures. A *definition* is an assignment to a variable or other operation that results in the variable getting a new value. A *use* occurs when the value of a variable is accessed. The execution path from the definition to the use must be *definition-clear*, that is, the variable must not be re-defined 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).

The following definitions are from Jin and Offutt’s original coupling definitions [16], where they are defined more formally. V_P is the set of variables that are referenced by program component P , and N_P is the set of nodes in the control flow graph of P . P_1 and P_2 are program units, and x and y are program variables.

A path from node i to j is *definition-clear* with respect to x if there is no definition of x along the path, except possibly at node i . A *callsite* is a node $i \in N_{P_1}$ that contains a call from P_1 to P_2 . If there is a definition-clear path from i to j with respect to x , we say that the definition of x at i can *reach* j .

A node $i \in N_{P_1}$ that contains a definition that can reach a use in P_2 on some execution path is a *coupling-def*. A *coupling-use* is a node $i \in N_{P_2}$ that contains a use that can be reached by a definition in another unit on at least one execution path.

A *coupling path* between two program units is a path that begins with a definition of a variable in the calling unit and ends at a use in the called unit. This includes paths that involve parameter coupling, shared data coupling, and external coupling. Note that there must be a definition-clear path from the definition to the corresponding use through a call.

By using these definitions, four coupling-based testing criteria were defined [16]. P_1 and P_2 are program units in a system:

- **Call coupling:** The set of paths executed by a test set must cover all call sites in the software.
- **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. A coupling path set is a set of nodes that can appear on subpaths through a program unit between a coupling-def and a coupling-use. This accounts for the case where the program unit has loops. Requiring that all coupling paths be covered is impractical in general. However, covering all coupling path sets ensures that each loop body is executed at least once, but does not require all possible executions.

2.3. Coupling Sequences

Although one of the motivating goals of object-oriented design and programming was to reduce the amount of coupling between software components, the new language features also introduce new ways for components to be coupled. To handle these couplings in analysis techniques, the idea of a coupling sequence was previously introduced [1]. Intuitively, a coupling sequence represents an interaction between the method under test m and the integrated object O . The objective is not to determine if O is correct, but rather to determine if m is using O correctly. Coupling sequences represent those locations in the text of m where faults are likely to occur with respect to O .

Intra-method coupling sequences are defined by pairs of method calls made within the context of a particular method, referred to as the *coupling method*. When a method m is called through an object o , we say that m executes in the *instance context* provided by o , referred to as the *context variable*. The two method calls of an intra-method coupling sequence are made through the same context variable, so they share a common *instance context*. Further, there is at least one path between the two method calls that is definition-clear with respect to the context variable and to at least one state variable that is defined by the first method and used by the second. Such a path is referred to as a *coupling path*. The intra-method coupling sequence is similar to a *def-use* pair [2], and serves to relate a definition of a state variable to a corresponding use across a procedural boundary in the context of a particular object and method. On the surface, it might appear that testing one such path is sufficient to test the class of the object. However, it is not the class that is being tested, but rather the method that makes use of the object and the corresponding methods.

An example of an intra-method coupling sequence is illustrated in Figure 1. Method f , the *coupling method*, contains a single coupling sequence, $s_{j,k}$, that starts at node j with

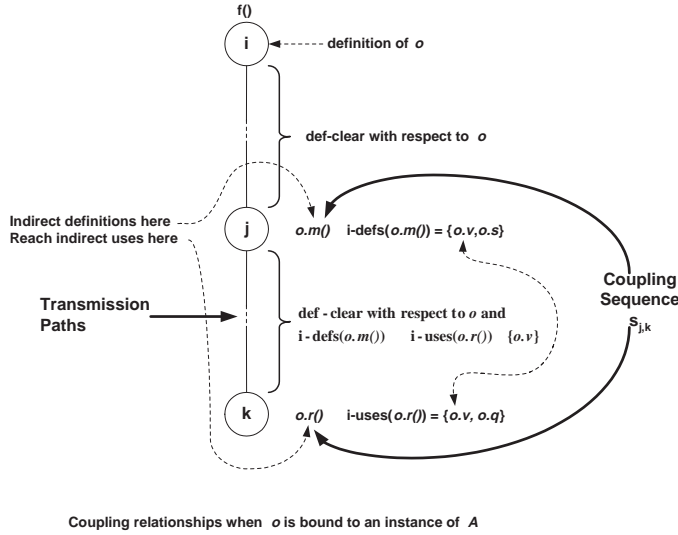


Figure 1. Example Coupling Sequence $s_{j,k}$

a call to $o.m$ (the *antecedent method*) and extends through paths that end at node k where the sequence ends with a call to $o.r$ (the *consequent method*). The nodes containing the antecedent method and consequent method are called the *antecedent node* and *consequent node* of $s_{j,k}$. Note that there must be at least one path between the call sites that is definition clear with respect to o and to the state variable *definitions* made in the antecedent method that have corresponding *uses* in the consequent method.

Another form of coupling sequence is an *inter-method coupling sequence*. Here, the calls to the antecedent and consequent methods are still made through the same instance context bound to o , but the calls are in different methods. The method g contains the call to the antecedent method, and h contains the call to the consequent method. As with the intra-method coupling sequence, the coupling sequence is formed by the call sites at the antecedent and consequent nodes. However, instead of calling the antecedent and consequent methods, the coupling method calls g , and then later h . As before, there must be at least one definition-clear path between the antecedent and consequent methods with respect to those state variables *definitions* made in the antecedent method that have corresponding *uses* in the consequent method. Further, there must be at least one definition-clear path between the call sites in g and h and their exit nodes with respect to that set of variables. Finally, there must be at least one path in the coupling method between the nodes containing the calls to g and h that are definition-clear with respect to o and the set of variables. The emphasis of this paper is on

intra-method coupling sequences (hereafter generically referred to as *coupling sequences*) and inter-method coupling sequences are left for later work.

A coupling sequence $s_{j,k}$ is defined with respect to a set of state variables that are defined by the antecedent method and used 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 1 is:

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

where $class(o)$ is the declared type of the context variable o . $\Theta_{s_{j,k}}$ contains the state variables referenced through o that are defined by the antecedent method and used by the consequent method in the coupling sequence $s_{j,k}$.

The coupling paths of $s_{j,k}$ start at nodes in the antecedent method that have last definitions of a particular coupling variable, and end at nodes in the consequent method that have corresponding first uses of the same coupling variable.¹ Note that for a given coupling sequence, the methods that are executed as a result of a call through the antecedent or consequent nodes depend the type of the instance that the sequence's context variable is bound to. If the context variable is of type T , any instance of any class that is a member of the type family of T may be bound to the context variable.

3. Coupling Criteria

This paper defines four object-oriented coupling criteria for integration testing. 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.

When using the testing criteria, it is assumed that the antecedent and consequent methods have been tested individually before the method containing the coupling sequence. This allows the developer to assume that any discovered failures are related to the interfaces.

3.1. Quasi-interprocedural Analysis

An essential part of these criteria is that some knowledge of inter-procedural data relations must be derived from the

¹A *last-definition* of a variable v is a node n along some path of a method, where n is the last node that defines v prior to the exit node of the method. Similarly, a *first use* of v is the first node n along some path that begins with the entry node a method, such that there is no other node that uses v before the use of n .

program. However, unlike the inter-procedural data flow testing of Harrold and Soffa [12], complete information about data flows between units is not needed. For this research, it is necessary to have data flow information from definitions to call sites, from call sites to uses, from entry nodes to uses, and from definitions to exit nodes. However, the **only** information that is needed **across** the procedure calls is the information pertaining to parameter passing (which actual parameter is passed to which formal parameter). This avoids much of the expense of inter-procedural data flow analysis. Because of the limited amount of inter-procedural information that is needed, we call this *quasi-inter-procedural* analysis. It is likely that this analysis information can be used to address other problems in software engineering.

3.2. Four Coupling Criteria

The criteria are *All-Coupling-Sequences*, *All-Poly-Classes*, *All-Coupling-Defs-Uses*, and *All-Poly-Coupling-Defs-Uses*. In the following subsections, $T_{s_{j,k}}$ represents the set of test cases for coupling sequence $s_{j,k}$.

3.2.1. All-coupling-sequences. It can be argued that during integration testing, at least every coupling sequence in every method of every class should be covered. Here, coverage means that each coupling sequence is executed by at least one test case. Accordingly, the *All-Coupling-Sequences* requires that every coupling sequence be covered by at least one test case.

Definition 1 All-Coupling-Sequences: For every coupling sequence $s_{j,k}$ in method f , there is at least one test case $t \in T_{s_{j,k}}$ such that when f is executed using t , there is a path p in the coupling paths of $s_{j,k}$ that is a subpath of the execution trace of f .

3.2.2. All-coupling-sequences. The *All-Poly-Classes* criterion strengthens *All-Coupling-Sequences* by considering inheritance and polymorphism. This is achieved by ensuring there is at least one test for every class that could 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. The *All-Poly-Classes* criterion requires that for every coupling sequence $s_{j,k}$ in a method f , and for every class c in the type family defined by the context of $s_{j,k}$, there is at least one test that covers every feasible combination of c and $s_{j,k}$ for f . 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 the declared type and it defines an overriding method for the antecedent or consequent method. Thus, only classes that override the antecedent and consequent methods are considered.

Definition 2 All-Poly-Classes: For every coupling sequence $s_{j,k}$ in method f , and for every class in the family of types defined by the context of $s_{j,k}$, there is at least one test case t such that when f is executed using t , there is a path p in the coupling paths of $s_{j,k}$ that is a subpath of the trace of f .

3.2.3. All-coupling-sequences. The criterion *All-Coupling-Defs-and-Uses* takes definitions and uses of variables into account. It requires that, for every coupling sequence in a method f , and for every coupling variable v in the sequence, there must be at least one test case that executes each coupling path with respect to v . That is, every feasible coupling path between each coupling-definition and coupling-use pair for v must be executed by at least one test case.

Definition 3 All-Coupling-Defs-and-Uses: For every coupling sequence $s_{j,k}$ in method f , and for every coupling variable v of $s_{j,k}$ and every node d in the antecedent method of $s_{j,k}$ that contains a last definition of v , there is at least one test case t such that when f is executed using t , there is a coupling path p in the trace of f that begins at d and that reaches a node in the consequent method of $s_{j,k}$ that has a first-use of v .

3.2.4. All-coupling-sequences. In addition to inheritance and polymorphism, the criterion *All-Poly-Coupling-Defs-and-Uses* takes the effects of definitions and uses into account. *All-Poly-Coupling-Defs-and-Uses* requires that all coupling paths be executed for every member of the type family defined by the context of a coupling sequence.

Definition 4 All-Poly-Coupling-Defs-and-Uses: For every coupling sequence $s_{j,k}$ in method f , and for every class in the family of types defined by the context of $s_{j,k}$, and for every coupling variable v of $s_{j,k}$ and every node m having a last definition of v and every node n having 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 subpath of the trace of f .

3.3. 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 [11, 16]. Criterion A *subsumes* criterion B if and only if every test set that satisfies A also satisfies B . Although there are certainly exceptions, the common assumption is that criteria at higher levels in a subsumption hierarchy have more testing power, at higher cost. The issue for testers, then, is to determine which criterion to select for a particular situation. Figure 2 depicts the subsumption hierarchy for the criteria presented in this paper.

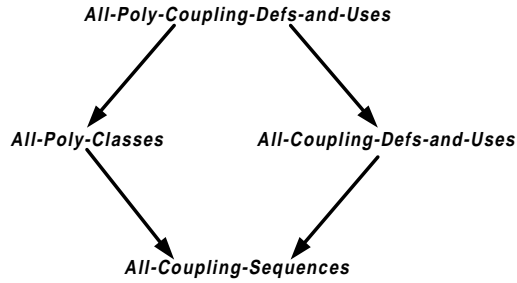


Figure 2. OO Coupling Testing Subsumption Hierarchy

3.4. Quasi-interprocedural Analysis Notes

One difficulty that arises with this kind of analysis is brought on by the ability to have variables dynamically bind to instances of different types, which in turn allows for polymorphism. Without dynamic binding, the data flow relationships between methods are static and can be known before execution. But when dynamic binding is possible, polymorphic effects can occur in which different methods are executed through the same call site, depending on the actual type of the instance currently bound to the variable. To understand this, consider the UML class diagram shown in Figure 3(a). As depicted, class *A* includes methods *m* and *n* along with state variables *u* and *v*. Similarly, class *B* includes methods *n* and *l*, and state variable *w*. Finally, class *C* includes method *m* and state variable *v*. Note that *B::n* overrides method *A::n*, and *C::m* overrides method *A::m*.² Now consider the definition of method *f* and the coupling sequence $s_{3,5}$ depicted by the source code fragment shown in Figure 3(b). First observe that *f* takes a formal argument *o* that is declared to be of type *A*. Inheritance allows *o* to be bound to an object whose actual type is *A*, *B*, or *C*. Now consider the call sites at statements three and five. Here, *apparent* calls are made to methods *A::m* and *A::n*. However, because of dynamic binding and polymorphism, the methods that are actually called depend on the type, *T*, of the instance bound to *o*. When *T* is *A*, then the *actual* calls are to *A::m* and *A::n*. However, when *T* is *B*, then the actual calls are to *A::m* and, instead of *A::n*, *B::n*. Likewise, when *T* is *C*, the actual calls are to *C::m* and *B::n*.

To understand the effects that dynamic binding and polymorphism can have on the data flow relationships between methods, consider the table in Figure 3(c), which summarizes the definitions and uses of the methods in Figure 3(a). Method *A::m* defines state variables *A::u* and *A::v* and method *A::n* uses *A::v*. From a data flow perspective, there

²The C++ scope resolution operator `::` is used to indicate which methods and state variables are being referred to.

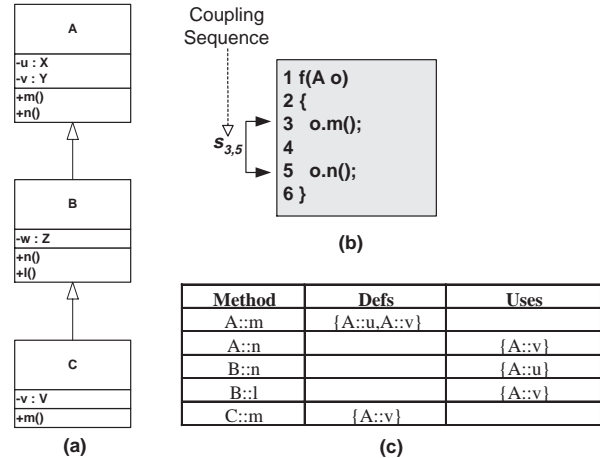


Figure 3. Example Class Hierarchy with Overriding Methods

is no problem when *o*'s type is *A* and these methods are executed as shown in Figure 3(b). 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 seems 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 result of dynamic binding and polymorphism, as the preceding discussions illustrates, is that the data flow graph of an object-oriented program can change **dynamically**. Although this greatly complicates the analysis, 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 in-

feasible 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.

The problem of infeasible coupling sequences is a specific instance of a more general problem, commonly called the *feasible path problem*, which says that for certain structural testing criteria some of the test requirements are infeasible in the sense that the semantics of the program imply that no test case satisfies the test requirements. Equivalent mutants, unreachable statements in path testing techniques, and infeasible DU-pairs in data flow testing are all instances of the feasible path problem. Although techniques have been developed that can identify infeasible paths in most cases for mutation testing, traditional data flow testing, and path testing techniques [20], the problem is generally undecidable. We have not yet addressed this problem in the context of infeasible coupling sequences, although it is hoped that the techniques developed by Offutt and Pan [20] can be applied to this case as well.

Another issue that arises is with which types are feasible for a particular coupling sequence. A type T is considered feasible if the context variable of the coupling sequence can be bound to an instance of T , where T is either the declared type of the context variable, or a descendant of the declared type. In some situations this is trivial. For example, in the coupling method f (shown in Figure 3(b)), the context variable o of the coupling sequence $s_{3,5}$ is passed as a formal argument of f . To vary o , a test driver need only call f passing an instance of a different type as the actual argument. However, if o were instead a local variable that was initialized by binding it to the return value of a method call g , varying the type of the instance would be extremely difficult, and is currently beyond the scope of our present research.

4. Related Work

There are a number of testing issues that are unique to object-oriented software. A number of researchers have asserted that some traditional testing techniques are not effective for object-oriented software [5, 10, 14]. The way methods are commonly used in object-oriented software means that many of the traditional software testing techniques test the wrong things. Specifically, methods tend to be smaller and less complex, so path-based testing techniques are less applicable. Additionally, inheritance and polymorphism introduce undecidability to the software [3]. 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.

In his dissertation, Overbeck presents an approach based on testing contracts among client and server classes [22]. 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 formal specifications written by the programmers.

Jorgensen and Erickson [17] describe an approach to integration testing that is similar to many black-box testing techniques. They define paths through a collection of classes that form a system. Each path is associated with a particular input event and traverses the classes that participate in the system response. The path includes all classes that are traversed through method calls, and ends when the system output has been observed. Failures are detected whenever the system output does not match the expected output. Faults are identified by tracing back along the path to each of the participants. Again, there is no help in creating test cases.

Harrold and Rothermel describe an approach that applies data-flow analysis to classes [13]. Their approach emphasizes three levels of testing: (1) intra-method testing; (2) inter-method testing; (3) intra-class testing. 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 paper does not describe how this information is used during testing.

Chen and Kao describe an approach to testing object-oriented programs called Object Flow Testing [8]. In their approach, they seek to identify and test possible object bindings that can occur within a method. The idea is to identify the def-use pairs that occur within a method, and between pairs of methods that are invoked from the same caller. They have defined two criteria that impose testing requirements on a method. The first, *All-bindings*, requires that every possible binding of each object be executed at least once, and further that all possible combinations of binding be tested when an expression involves multiple objects. This criterion is similar to our *All-Poly-Classes* criterion. Their second criterion, *All-du-pairs*, requires that every definition-clear path between every definition of an object and every use of that object be tested at least once. This is similar to *All-Coupling-Defs-and-Uses*. The criteria defined here are more stratified and take into account both finer and coarse grained interactions among the integrated objects and methods, which should allow for more stringent testing.

Orso and Silva present a technique that focuses on testing polymorphic definitions and uses by identifying paths that contain invocation of polymorphic methods within a method under test (MUT) [21]. This work also considers the possible bindings that can occur for a given object reference. However, this approach is focused on the effects of the MUT that result from polymorphic invocations. Our approach considers this as well, but also places considerable emphasis on testing the interactions that occur between methods that result from calls made within the MUT. This considers not only what the effects are on the MUT, but also the effect that the execution sequence of the MUT has on the objects that it uses through the methods that it invokes.

5. Conclusion and Future Work

This paper has introduced a new integration testing technique for object-oriented software. Four distinct criteria were presented that can help testers evaluate connections among software components that are based on inheritance and polymorphism. These criteria depend on a new type of program analysis, quasi-interprocedural analysis. This level of integration testing represents an important and difficult problem area for object-oriented developers, because the use of object-oriented designs often mean that crucial design decisions are encoded in the interconnections among components, and the abstraction mechanisms of inheritance and polymorphism can result in very complicated and potentially error-prone relationships.

This research addresses the problem of developing formalizable, measurable criteria for generating tests from object-oriented software. These criteria depend on detailed analysis of the source code, and can be used to generate integration tests for object-oriented software that are effective at detecting design and programming mistakes in the connections among software components.

We are currently building a proof-of-concept coverage analyzer tool to support this technique. This tool is being built in Java to test Java programs, and is based on the general Java parser generator JavaCC and Java Tree Builder (JTB) by Tao and Palsberg at Purdue University³. The tool currently parses Java programs and produces several kinds of graphs for analysis. We are currently designing techniques to instrument test programs to determine whether testing has satisfied the test criteria presented in this paper. The tool will be used to provide evidence of the effectiveness of this technique.

This paper is part of an ongoing project to provide object-oriented software developers with better tools and techniques for integration testing. This will eventually lead to software that can be built cheaper and more reliably.

³JTB can be downloaded from the web at <http://www.cs.purdue.edu/jtb/index.html>.

References

- [1] Roger T. Alexander and A. Jefferson 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.
- [2] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–146, March 1976.
- [3] 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.
- [4] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, New York, 2nd edition, 1990.
- [5] Edward Berard. Issues in the testing of object-oriented software. In *Electro'94 International*, pages 211–219. IEEE Computer Society Press, 1994.
- [6] Edward V. Berard. *Essays on Object-Oriented Software Engineering*, volume 1. Prentice Hall, 1993.
- [7] Robert V. Binder. Testing object-oriented software: A survey. *Journal of Software Testing, Verification & Reliability*, 6(3/4):125–252, September/December 1996.
- [8] 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.
- [9] L. L. Constantine and E. Yourdon. *Structured Design*. Prentice-Hall, Englewood Cliffs, NJ, 1979.
- [10] 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.
- [11] 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.
- [12] M. J. Harrold and M. L. Soffa. Selecting and using data for integration testing. *IEEE Software*, 8(2):58–65, March 1991.
- [13] 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.
- [14] 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.
- [15] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. ANSI/IEEE Std 729-1983, 1983.
- [16] 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.
- [17] Paul C. Jorgenson and Carl Erickson. Object-oriented integration testing. *Communications of the ACM*, 37(9):30–38, 1994.
- [18] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, New Jersey, 2nd edition, 1997.
- [19] 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.
- [20] A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. *The Journal of Software Testing, Verification, and Reliability*, 7(3):165–192, September 1997.

- [21] A. Orso and S. Silva. Integration testing of procedural object-oriented languages with polymorphism. In *16th International Conference on Testing Computer Software (ICTCS'99)*, Washington, DC, 1999.
- [22] Jan Overbeck. *Integration Testing for Object-Oriented Software*. Ph.D. Dissertation, Vienna University of Technology, 1994.