Testing the Polymorphic Relationships of Object-Oriented Components

Technical Report ISE-TR-99-05

Roger T. Alexander

Department of Information and Software Engineering

School of Information Technology and Engineering

George Mason University

Fairfax, Virginia 22030-4444

ralexand@gmu.edu

February 16, 1999

Revised February 18, 1999

Contents

List of Tables

List of Figures

1 Introduction

This research proposal outlines an investigation in the area of testing object-oriented software. Emphasis is placed on testing the ways in which abstractions (types) are defined and composed to form higher-level abstractions.

This proposal is organized as follows: the remainder of section 1 presents the problem and motivation for the proposed research; section 2 presents background material on object-oriented technology; section 3 surveys related work that has been done; section 4 presents the proposed research, including the validation approach and expected contributions; section 5 presents the proposed schedule of research activities; finally, section 6 presents the expected outline of the dissertation resulting from this research.

1.1 Motivation

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 [46]. These abstractions appear in the language as user-defined types that have both state and behavior. Unfortunately, while the use of abstract data types often results in a design of higher quality, the level of testing effort required to achieve a desired level of quality increases. This is due to the inherent complexity in the nature of the relationships found in object-oriented languages [11]. The compositional relationships of inheritance and aggregation, combined with the power of polymorphism, increase the difficulty in detecting faults that result from the integration of components to form new types. This is due to the differences in how component integration occurs in an object-oriented language [7].

In a procedure-oriented languages, such as C and Pascal, and object-based languages such as

Modula-2 and Ada 83, the unit of integration is the procedure and module, respectively.[1] The integration mechanism is simple aggregation through either procedure/function call-return, or through containment when one module includes another. This is also true for an object-oriented languages, but the key difference is the presence of another integration mechanism: *inheritance*. Inheritance permits new types to be defined in terms of the state and behavior of existing types. Such new types are said to be *descendants* of the existing type [44]. Inheritance differs from aggregation in that the encapsulation of the inherited type may not be preserved. It is possible for the newly defined type to have free access to the internal representation of the types that it is defined in terms of.

Another difference is the effect that polymorphism (dynamic binding) has on the integration of components when inheritance is a factor.[2] Polymorphism allows the actual type of an object to determine which version of a method executes [46]. Any object type T defines a new type family. Members of that family include T and any of its descendants. As a result of polymorphism, any instance of a descendant type can be freely substituted for an instance of T.[3] Thus, inheritance combined with polymorphism provides two forms of integration that must be dealt with when testing objects, neither of which has a procedure-oriented counterpart.

[1]The primary distinction between the types of languages discussed in this proposal is in the mechanisms used for abstraction. Procedure-oriented laguages employ the procedure and function as the primary abstraction mechanism. In contrast, both object-based and object-oriented languages use data abstraction as the primary mechanism.

[2]Together, inheritance and polymorphism are the key charactertics that distinguish an object-oriented language from an object-based language [46][44].

[3]An *instance* has a memory location and a value associated with it. For example, the declaration `int x = 7` in the C programming language results in a new instance of the type *int* with a memory location initialized to the integer value 7. Similarly, in a object-oriented language, creating an object of a particular class results in a new instance of that class having an associated memory location and value.

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 combined with new and overriding methods carefully 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 and polymorphism. The integration issue consists of ensuring that the aggregated type and its owner work correctly together across all forms of representation that can exist for the aggregated type. This is not just a static language issue, there is also a dynamic aspect due to the dynamic binding that results from inheritance and polymorphism. It is possible for the representation of an aggregated type to change dynamically at runtime. Because of this, all possible substitutions must be tested to ensure congruent behavior with respect to the aggregated abstraction.

1.2 Problem Statement

The problem that this thesis will address is that of finding errors in the polymorphic relationships among integrated type components in object-oriented software.

Traditional testing techniques do not work effectively for object-oriented software, at least in the sense that they are not capable of detecting the faults that programmers make in object-oriented programs [21][13]. Faults can hide in many more different types of places than in a traditional procedure-oriented program. The following sections discuss traditional approaches to software testing and their applicability to the testing of software written in object-oriented languages.

1.2.1 System level testing techniques

System level testing techniques, such as the Category-Partition technique [51], are focused primarily on the functional behavior of a system without regard for its structural characteristics. These techniques treat a system as a black-box and are aimed at determining if a given system exhibits the required functional behavior according to a specification. Test inputs are chosen without regard for the structure of the system. Consequently, it is not known what components of the system actually execute and which do not. There may be components that are not executed by the chosen test suite even though the test suite itself is adequate with respect to the system's specification. Thus, there may be faults residing at certain locations within the system, but because those locations are not executed, there is no opportunity for a failure to be observed if one occurred.

For the types of faults outlined in section 1.1, execution is a necessity if any confidence is to be obtained in the correctness of an object-oriented program. This is consistent with the first tenant of the fault/failure model, which essentially states that if a fault is to result in a failure, then the program location containing the fault must first be executed [17][48][49]. Thus, black-box techniques are not sufficient to adequately test the compositional relationships found in object-oriented programs.

1.2.2 Unit-level testing techniques

Traditional unit-level path testing techniques used to test procedure-oriented programs, such as statement and branch coverage [5], are effective at testing certain aspects of object-oriented programs[19]. They should be just as applicable to methods as they are to procedures and functions. These unit-level techniques are used to analyze various characteristics of procedures with respect to some test adequacy criterion. For example, branch coverage measures the quality of a particular test suite, and hence the quality of any testing effort, as the percentage of branches that are covered when the procedure is executed using inputs drawn from the test suite [5]. This approach to testing is well-suited for methods as well. However, its overall effectiveness for testing an object-oriented program may be far less than that of an equivalent procedure-oriented program [11]. This is due to the tendency for behavior in an object-oriented program to be distributed across a set of collaborating objects instead of being relegated to a handful of procedures and functions. The practical consequence of this is that methods tend to be much less complex and considerably shorter, many times to a trivial extreme. Consequently, the procedural complexity is shared among many different methods belonging to different classes. Unfortunately, this reduces the effectiveness of path-based unit-level techniques, and increases the necessity to test the compositional relationships among classes.

Another form of testing that is prevalent in procedure-oriented software is data flow testing [24]. These testing techniques explore the interrelationships among the data elements in a program and attempt to identify faults that are associated with well-defined patterns of definition and usage. The fault assumptions made by these techniques are that faults will be revealed if every definition is used in some way [24][5]. While there is evidence to suggest that these techniques are useful in their own right for procedure-priented programs [23], they still are not adequate for testing the complex relationships that occur among components within object-oriented programs. In particular, these techniques do not directly support the testing of state space interactions among overridden and inherited methods. Also, they are not sufficient for testing the behavior of polymorphic substitutions [9]. However, the work of Harold and Soffa [28] and also of Jin and Offutt [33] has shown data flow analysis to be an effective technique for testing the integration of procedure-oriented components. It is likely that an augmented form of this analysis technique would be useful for analyzing and testing the compositional relationships found in object-oriented programs.

1.2.3 Integration Testing Techniques

Integration testing is concerned with testing the interactions among components. Does a component that calls another do so correctly? Are the parameters of the right types, ranges, and do they observe the proper relationships? Does the called method actually return the proper type and is the value of the correct range? These questions are the focus of integration testing. Unfortunately, very little research has been conducted in this area. Work that has been done generally emphasizes inter-procedural data flow [28]. That is, determining that data passed between components is used in a consistent manner.

2 Background on Object-Oriented Programming

The following sections describe the concepts and principles associated with object-oriented programming that are relevant from a testing perspective. The reader is referred to Meyer for a thorough treatment of object-oriented concepts [46].

2.1 Classes

Object-Oriented Programming is an approach to the development of software that is based on the concepts of information hiding and encapsulation [46][54]. The fundamental building block 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. The primary role of the class in an object-oriented program is to provide a template for the creation of *objects*

[46]. Thus, a class defines a type that all of its objects share.

Each method in a class is either a procedure or a function in the traditional sense of procedure-oriented programing. Functions return values and procedures affect state changes (functions may also cause state changes, but this is considered to be bad practice by industry and academia alike [46]).

The set of methods in a class that are available for use by others constitute the *interface* of the class. This interface provides the only mechanism by which clients can interact with the class. State variables usually remain inaccessible from clients, though this practice does occasionally occur, often to the detriment of quality [46].

2.2 Compositional Relationships

There are two types of relationships that can be used to compose types (i.e. classes) to form new types. The first of these, *aggregation*, is simply the traditional notion of one type containing instances of another type as part of the its internal state representation. In a procedural language such as the C programming language, a *struct* type aggregates instances of other types as part of its definition. For example, a *struct* that describes an employee record might be composed by aggregating string instances that maintain the first, middle, and last names of an employee, and perhaps a date instance that records the date of hire. In an object-oriented language, the aggregation of instances is similar. Figure 1 provides a simple illustration of type aggregation represented in the notation of the Unified Modeling Language (UML [2]). The diamond indicates the aggregating class, A in this case. The figure illustrates a class diagram that consists of two class types, A and B, with an instance of B being aggregated into A's state space. Thus, every instance of type A will also contain an instance of type B.

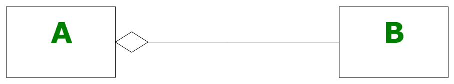

Figure 1: Sample aggregation relationship

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 (i.e. behavior and state). The definition of the ancestors becomes part of the definition of the new descendant type. An example of this is illustrated in Figure 2 where the arrow points to the ancestor class.

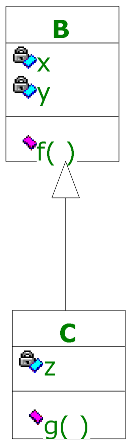

Figure 2: Sample inheritance relation

2.3 Polymorphism and Dynamic Binding

Related to both inheritance and aggregation is *polymorphism* and *dynamic binding*. Polymorphism permits instances of different types to be bound to a reference of another type 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 independent of its declared type [46]. To see this, consider the following code fragment:

```
1 void Q::p( W& w )
2 {
3     ...
4     w.m();
5     ...
6 }
```

Line 4 contains a call site in which the method p is invoked against the instance bound to the object reference w. The implementation of m that actually executes depends on the actual type of the instance. Even though the declared type of w is W, the actual type of the bound instance can be of any of the classes shown in hierarchy depicted in Figure 3. For example, if the type of the instance passed to the method shown above is of type X, then the version of m that executes will *X::m*.[4] Similarly, if the type is Z, then the version that will execute is *Z::m*. However, if the type is V, then the version of m that will execute is *W::m*, because V does not provide a definition of m.

3 Related Work

There are a number of areas that are related to the research described in this proposal. The subsections that follow discuss these areas in detail. The first subsection describes and discusses a number of issues that are peculiar to testing object-oriented software. Next, the notion of test adequacy is discussed with respect to testing object-oriented software. Following this is a discussion of class level testing that describes both state-based and method sequence-based testing approaches. Next, existing techniques for integration testing of object-oriented software is described. Finally, other testing techniques are discussed that are related to this research.

[4]The notation used here is borrowed from C++ where the scope resolution operator *::* is used to identify the namespace that a particular identifer is a member to.

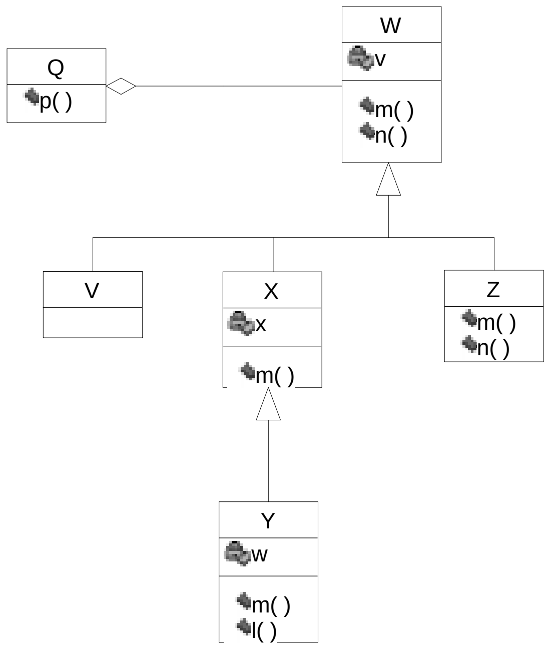

Figure 3: Sample Class Hierarchy

3.1 Issues in Testing Object-Oriented Software

There are a number of issues associated with object-oriented software that are not relevant in systems written using procedure-oriented languages. A number of researchers make the assertion that not all forms of traditional testing techniques are applicable or effective in testing object-oriented software [29][6][22]. The semantics of classes are embodied in their methods and in the representations chosen for their state. Taken in isolation, each method appears to be a function or procedure, equivalent to those found in the procedure-oriented languages. They take formal arguments and interact with state variables that act as global data. Thus, it seems reasonable to expect traditional unit testing techniques to be applicable to methods. However, these techniques are not as effective with methods as they are with procedures. This is because methods tend to be significantly smaller and less complex. It is not uncommon to find methods with one or two statements, and many, perhaps the majority, can be found to have less than ten [11]. A method of only a few statements is not as likely to have statement-level faults, or as many, as those of procedures having tens, hundreds, or even thousands of statements. Consequently, the prevailing wisdom is that the effectiveness of traditional path-oriented testing techniques is not very high. The nature of the types of faults that occur in object-oriented programs are such that path-oriented techniques are not sufficient [11].

Inheritance combined with polymorphism increases the testing by significantly adding undecidability to the testing process [4]. The actual path executed through a class is no longer a function of its static declared type, rather, it is a function of the dynamic type that occurs at run-time. Because of this, any of a number of potential different paths can taken through a class hierarchy dependent upon what type is actually used in a test. This is further complicated by polymorphism that can occur in class instances that are aggregated into the state of the class under test and those instances that are passed as parameters to methods of the class under test.

Strong encapsulation reduces, and often eliminates, the ability to *observe* the state of an object [55][9] [1][4]. If the state cannot be observed, then it is often not possible to determine if a failure has occurred. Strong encapsulation also reduces the ability to *control* the input to a test[25]. This often makes it difficult to establish the necessary conditions for conducting some tests, thus limiting our ability to conduct adequate testing of a class.

A number of researchers have observed that, contrary to many widely held beliefs, object-oriented language features actually increase the effort required to achieve adequate testing.[5] Binder observes that inheritance and polymorphism present opportunities for the commission of errors that simply do not exist in procedure-oriented programs. Furthermore, he points out that testing effort is not reduced for a descendant class simply because its parent has been thoroughly tested. This is because each new class is a different context, and perhaps even different tests are required to

[5] In a personal communication with the author, Gail Kasier lamented the difficulty that she and DeWayne Perry had in publishinh paper on adequate testing of object-oriented programs because their claims and argument went against the commonly held beliefs of the time [35][56].

achieve test adequacy [9]. Binder also observes that testing objects is problematic because they "...[often] exhibit sequentially dependent behavior" [10]. Objects can generally be viewed as state machines that transition from one state to the next as their methods are executed. Objects that exhibit modal behavior impose limits on the order in which their methods can be executed. From a testing perspective, this results in an increase in the effort required to establish initial states and to test all combinations of valid method sequences.

Smith and Robson report the observation that in an object-oriented language, a class cannot be tested directly [58]. Instead, classes must be tested indirectly by testing their instances (objects). Testing becomes a process of sampling from the class' population of instances. Unfortunately, this comes with all the limitations associated with statistical sampling. The quality of the testing effort will be limited as a function of the degree to which the sample is representative of all the class' instances.

Fiedler reports on his experiences using an approach for testing classes that was used at Hewlett-Packard's Waltham Division [20]. The approach was based on the use of a combination of black and white box testing techniques and was applied to a number of programs written in Extended C++ . He reports that the approach was applied to several generic classes that had been tested prior using traditional black-box testing techniques and that the classes where considered to be correct. However, upon application of the approach, a number of faults were detected that the prior testing effort had missed, but there are many possible reasons for this. His main conclusions are that the unit of testing in an object-oriented program must be the *class*, and that the testing activity must occur much earlier in the life-cycle. Further, he concludes that both black and white-box testing techniques must be used.

3.2 Test Adequacy

One of the early widely-held beliefs about object-oriented technology was that inheritance would reduce the amount of testing effort required. It was believed that once a parent class had been adequately tested, testing a derived class would be far simpler. All that would need to be done is to test the new methods added by the descendant, since those inherited from the parent had already been adequately tested. The conventional wisdom was that inheritance would reduce testing effort by either eliminating or reducing re-testing, or allowing the reuse of tests. This belief was dispelled by Perry and Kaiser [56] who, drawing upon the earlier work of Weyuker, analyzed the adequacy of tests for object-oriented programs with respect to single inheritance, method overriding, and multiple inheritance. Their conclusions are that these features do not reduce the amount of testing effort, and in many cases, increase the required effort to achieve test adequacy. They make the important observation that inheritance, in particular, makes the effects of changes "implicit and dependent on the various underlying, and complicated, inheritance modes" [56]. The basis for their conclusions rests upon Weyuker's axiomatic basis for determining test data adequacy [59]. They make use of four of the following of Weyuker's eleven axioms of test adequacy to show that inheritance and method overriding do not lead to a reduction of testing effort:

1. **Antiextensionality**: "If two programs compute the same function...a test set adequate for one is not necessarily adequate for the other" [56]. This is a result of the fact that program-based test adequacy is a function of the syntactic structure of source code, not its functionality. Because of this, programs that implement the same specification are quite likely to require different test sets.

2. **General Multiple Change**: "When two programs are syntactically similar (that is, they have the same shape), they usually require different test sets" [56]. Essentially, two programs have the same shape if their control structures are identical, but differ in the relational operators, constants or arithmetic operators. Two such programs would require different test sets simply because the test data for one would most likely not result in the desired coverage objectives for the other.

3. **Antidecomposition**: "Testing a program component in the context of an enclosing program

may be adequate with respect to that enclosing program but not necessarily adequate for other uses of the component" [56]. It may be the case that the component has code that was not reached during the test for the enclosing program. Thus, when the enclosing program's test passed, there still remains untested code in the enclosed component, leading to the conclusion that the enclosed component itself has not been adequately tested.

4. **Anticomposition**: "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" [56].

Intuitively, testing should be limited to just the class that is modified. However, as Perry and Kaiser point out, the anticomposition axiom states, in effect, that just because a class has been tested in isolation does not mean that it is adequately tested when it has been composed with other classes. Thus, the authors conclude that integration testing is always necessary regardless of which programming language is being used.

One side effect of object-oriented languages is that the connections between components "tend to be explicit and obvious" [56]. Changing a component should only require re-testing of the changed component and the other components that are dependent upon it. Likewise, adding a new component should only require testing of the new component and re-testing of components that are dependent upon it. Unfortunately, the antidecomposition axiom comes into play when a new subclass is added to a hierarchy. The requirement is that the methods inherited from each ancestor class must be retested since the new subclass provides a new context for these methods. However, this requirement does not apply to the case where the new subclass has no interaction with the ancestor class methods (or state). This implies that the subclass is a pure extension to its ancestors, adding its own state variables and methods [56].

Replacing an inherited method with a locally defined method (i.e. an overriding method) requires that the subclass be retested, but with a different test set. This is governed by the antiextensionality axiom: even though the overriding method and the overridden method may be close semantically, their test sets are not likely to be mutually adequate due to their syntactic differences. Another subtle, but significant, consequence is that it may be necessary to re-test every ancestor of the subclass containing the overriding method [56].

Multiple inheritance presents compounding effects that result in the applicability of both the antiextensionality axiom and general multiple change axiom. The problem occurs when a subclass inherits from multiple ancestors that define methods of the same name, which the subclass does not override. Depending upon the semantics of the particular programming language, a call to one of the multiply defined methods through an instance of the subclass will result in a specific method being executed, the choice being determined by the order of inheritance specified by the subclass. A test set that is adequate for an initial inheritance ordering may not be adequate if a change results in a new ordering [56].

Others have observed that inheritance does not necessarily reduce testing effort. Smith and Robson observed that inheritance causes problems for testing as classes undergo evolution [58]. Modifications to a class will affect all additional classes that are its descendants, and will thus require some potentially significant amount of re-testing. Such class modifications will typically require modifications to the associated test suite, and thus will have an impact on any descendant class whose test suite makes use of its parent's test suite.

Cheatham and Mellinger identify four properties that a method m in a descendant class can have [14]. First, a method can be inherited from the parent class without any alterations. They state that little re-testing is needed in this case. However, their conclusion is short-sighted since m, even though it has not been modified, may fail to behave correctly due to indirect interactions with other methods defined in the descendant through the inherited state space. The second property for m is that it can be an override for an identically named method in the parent. The third property is that m can be executed in conjunction with the parent's m by providing a wrapper that calls it. The fourth property is that m can be a new method in the descendant that is not directly related to any member of the parent. Cheatham and Mellinger state that cases two, three, and four must be treated as new methods and tested accordingly, though they do not describe what they consider to be adequate testing. They do state that in the third case, the parent's version of m can be treated as a black box for purposes of testing the new m.

Harrold, McGregor and Fitzpatrick present an approach for identifying the set of tests that are necessary to test a class adequately, particularly when inheritance is a factor [26]. Adequacy in this case means that every method is tested individually as well as its interaction with other methods in the class. Their test suites include tests that are both specification-based and program-based. They note that specification-based test cases can be constructed using existing approaches. Their approach makes use of stubs for other methods and procedures called by the *method under test* (MUT). Driver routines are also provided for executing the MUT. Each test suite is a triple consisting of the method, a set of specification-based tests, and a set of program-based tests. Also, each test set includes a flag to indicate if those tests should be run in their entirety, a subset, or none at all. This flag is used when determining the which of the parent class' attributes must be retested in a descendant.

Harrold, McGregor and Fitzpatrick base their criteria of what must be retested on the work of Perry and Kaiser's extension (for OOPs) of Weyuker's work on the axiomatization of test adequacy [56][59]. Their decision of what to include is dependent upon the effects of inheritance and the interactions that occur as the result of new and overridden attributes, and attribute redefinition and hiding. They use a graph-based representation, called a *class graph*, to determine the interactions that occur and the necessary level of re-testing required. In their approach, they classify an attribute A (methods and state variables) as follows:

New Attribute A is defined in the descendant, but not by the parent.

Recursive Attribute A is defined in the parent and inherited by the child. The child does not redefine A.

Redefined Attribute A is defined by the parent and re-defined by the child, which hides the parent's definition.

Virtual-new Attribute A is specified by the parent and may have not implementation. A may also be specified in the child but its signature differs from the parent's definition. References to A in the child refer to the local definition, but references by other attributes in the parent refer to the parent's definition.

Virtual-Recursive Attribute A is specified in the parent, and its implementation may be deferred. The child does not define A.

Virtual-Redefined Attribute A is specified in the parent and its definition may be deferred. Further, A is defined in the child and has the same signature as the version of A specified in the parent.

Harrold, McGregor and Fitzpatrick's approach requires the following types of testing to occur in a subclass [26].

- A New or Virtual-New attribute A requires individual testing since it was not included in the parent's test suite. Due to the antiextensional axiom, A must also be integration tested with other attributes that it interacts with.
- Recursive or Virtual-Recursive attributes require "very limited" re-testing since they were individually tested in the parent. The authors claim that the specification-based and program-based test suites need not be re-run. However, A's interaction with new or redefined attributes will need to be re-tested.
- Virtual or Virtual-Redefined attributes require extensive re-testing, but the specification-based tests defined for the parent may be reused since on the implementation of A changes.

The preceding three items form the basis for which Harrold, McGregor and Fitzpatrick use to determine which tests can be reused and what re-testing is necessary.

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 [36]. 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 that is 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. The authors note that their work supplements that of Harrold, McGregor and Fitzpatrick [26].

The test order for a given class structure is based upon the dependencies that exist among its classes and is determined by analyzing of a graph-based formalism known as an Object Relation Diagram (ORD). An ORD contains an explicit representation for the relationships that can occur in an object-oriented program, including inheritance, aggregation, association, using, and instantiation. The ORD is a multigraph that consists of vertices corresponding to classes, and edges that model the relationships among the classes. The idea behind Kung et al.'s algorithm is to traverse the ORD, modifying it as necessary to remove cycles. Once this is accomplished, the test order is produced by applying a topological sort of the nodes in the graph (that is, the classes) [36]. 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.

3.3 Class Testing

Just as the procedure and function are the basic units of abstraction in procedure-oriented languages, the class is the basic unit of abstraction in object-oriented languages (and object-based). Naturally, it makes sense that testing applied to these types of languages should focus on their primary abstraction mechanisms. This view is reflected by the proportion of literature on testing object-oriented software that is devoted to the testing of classes [11].

3.3.1 State-based Testing

The testing of classes is largely an integration testing issue. Since a class consists of a number of methods and a collection of variables that define its state, the testing effort must focus on the interaction of these methods with respect to each other and with respect to their indirect interactions through the state space. As reported in the scientific and industrial literature, the majority of class testing approaches adopt one of two perspectives. The first is that of a class viewed as a state machine [11]. In this perspective, each class to be tested has its behavior modeled as a finite state machine. The typical approach is to derive a collection of modes that are based on the logical behavior of the class rather than its representation, and a set of transitions that correspond to each of the public methods. Each mode corresponds to a disjoint set of states in the underlying state space representation [57]. This has the advantage of avoiding the explosion of states that would result if the behavior were modeled directly on the representation. For example, a class that abstracts the notion of stack logically has three states: *full*, *empty*, and *not full and not empty*. If the internal representation chosen for the class consisted of an array and two integer index variables, the resulting physical state space would be the cross product of all the possible values that could occur for the array and index variables. The resulting size is too large to be of practical use from a testing perspective. However, folding the physical states into modes does increase the tractability of the testing problem tremendously, but at the expense of introducing non-determininsm. From a state-based class testing perspective, this results in a significant reduction in the amount of effort testing effort since fewer tests are required to cover all of the logical states than would be if the states based on the representation were tested.

Kung, Suchak, Gao, Hsia, Toyoshima, and Chen (KSGHTC) describe an approach for modeling an object using a formalism known as a Composite Object State Diagram (COSD), and a procedure for reverse engineering a COSD from the implementation of a class [37]. The resulting state machine model is used to test the state dependent behavior of an object rather than using the control or data structure of the implementation. The COSD is comprised of other COSDs (recursively) or Atomic Object State Diagrams (AOSDs). An AOSD is comprised solely of states, transitions, and actions. Each AOSD corresponds to an attribute of a class' state space. Each such attribute that corresponds to an AOSD must have an effect on the behavior of the class when the attribute changes values. That is, to be considered a state defining attribute, there must be at least two distinct sets of values for an attribute that will result in different observed behaviors for an object. For this to be true, there must be conditions involving the attribute in one or more of the class' methods.

The KSGHTC approach makes use of Chow's method for generating test cases from finite state machines [16]. The basic idea is to create a test tree for a COSD where the nodes represent the composite states in the COSD. Edges between nodes represent transitions between states. The composite states are represented as k-tuples, where k is the number of AOSDs in the COSD. the ith element of the k-tuple corresponds to the state of the ith AOSD. There may be several COSD test trees due to the fact that each AOSD can have more than one initial state. Test sequences are generated from the tree by walking its root to its leaves. Each such path corresponds to a single test sequence.

Hong, Kwon, and Cha present a technique for testing classes based on representing behavior as a finite state machine (which they refer to as a Class State Machine - CSM) and using data flow testing techniques to generate test cases [31][30]. The CSM is used as the basis for forming a Class Flow Graph (CFG) that integrates the state machine with the methods of the class. Each node in the graph corresponds to either a state, a guard, or a transition. State nodes in the CFG correspond to state nodes in the CSM. Guards represent predicates that determine when a particular transition is valid. Transition nodes correspond to method calls and have associated actions (i.e. state transitions). The CSM is transformed into a CFG using an algorithm designed by Hong, Kwon, and Cha [31][30]. Test cases are generated based on the definition and usage patterns of class state variables within the class' methods and within the guards on transitions. These are used to produce a set of definition-use associations that form the basis of the test requirements for a class. Hong, Kwon, and Cha's technique does not account for inheritance or aggregation relationships. Thus, their approach is object-based at best and cannot be applied in general to object-oriented programs.

3.3.2 Method Sequence-Based Testing

The second perspective on class testing focuses on the externally observable behavior of the class when it is subjected to a sequence of method invocations. This, of course, is related to state-based testing since the state of an object is a function of the method invocations that have occurred. However, with this testing perspective, the emphasis is not on testing individual states and transitions directly. Rather, it is based on idea of subjecting different instances of a particular class to different method sequences that leave the objects in correct states.

Binder presents the Flattened Regular Expression (FREE) approach to testing object-oriented software [11][8]. In this approach, classes and clusters of classes are modeled as state machines. Inheritance is accounted for by flattening the class hierarchy so that each class to be tested is self contained. Tests are regular expressions that described method sequences that cover all of the states and transitions. The implementation of a class is tested by constructing a graph that connects all of the methods within a class along all of the intra-method and inter-method dataflow paths. The regular expression that describes the state behavior of the class is also incorporated into this graph. Each transition edge in the state machine is replaced by the corresponding method flow graph. The transition edge is connected to the methods entry node, and the method's exit node is connected to the state resulting from the transition. This resulting graph represents all of the possible paths that can occur among the methods within the class. Binder states that this graph can be used to support path-based test suites, though he does not give an example or describe a procedure [11] [8].

Another approach to testing using method sequences is to subject two instances of the same class to equivalent message sequences and observe if they both end in the same logical state. For example, for a class implementing an abstraction of a stack, the resulting states of the following two method sequences are equivalent:

$$push(1); push(4); push(2); push(3); pop(); pop()$$

$$push(1); push(4)$$

Two different instances of the stack class subjected to each of these method sequences, respectively, should, if stack is implemented correctly, end with final states where 1 is at the bottom of the stack, 4 is at the top, and there are no other elements on the stack.

The equivalent message sequence approach is taken by Doong and Frankl in their ASTOOT approach to testing object-oriented software [18]. ASTOOT is based upon algebraic specifications and the notion of observational equivalence between sequences of methods. They use algebraic specifications, expressed in language called LOBAS, to define such sequences that, when applied to objects of the same class, result in the same final state. The basis of their approach is given two objects o_1 and o_1 of the same class C, and two equivalent sequences of methods, s_1 and $s2$, defined in C, the effect of executing s_1 on o_1 and s_2 on $o2$ should leave o_1 and o_2 in the same states. If so, the C is deemed to be correct with respect to s_1 and s_2. The sequences s_1 and s_2 are equivalent because the resulting state of their executions are the same. Thus, s_1 and s_2 are observationaly equivalent sequences of methods.

Method sequences are derived from LOBAS specifications of abstract data types. To derive these sequences, Doong and Frankl's approach places the following restrictions on the methods of a class [18]:

1. Methods must have no side-effects on their parameters.

2. Methods whose purpose is to return state information must be side-effect free.

3. Observers can only appear as the last method in a sequence.

4. Sequences passed as parameters to methods must not contain any observer methods.

The above restrictions are required to make the test case generation process tractable. Unfortunately, these restrictions also limit the applicability of the approach. Doong and Frankl's criterion for correctness is that an implementation class C of an abstract data type T is correct if it has the same set of signatures as T, and all states of T that can be reached by any pair of methods sequences have a corresponding state in C that can be reached by the same pair. A class is deemed to be correct if all test cases pass.

Another approach based on method sequences is that of Chen, Tse, Chan, and Chen [15]. In their approach, a class specification consists of a set of equational algebraic axioms. Each axiom, consisting of one or more terms, states a rule that specifies a permissible ordering of message sequences. A term is simply a series of operations consisting of variables and constants. Terms that have no variables are referred to as *ground terms*. A term is said to be in *normal form* if it cannot be transformed any further by application of any of the axioms contained in the specification. Two terms are equivalent if they can be transformed into the same normal form.

A test case consists of two ground terms u_1 and u_2 and their corresponding implementation of method sequences s_1 and s_2 [15]. An error is said to occur if u_1 and u_2 are equivalent but the application of s_1 and s_2 to different instances of the class under test result in observationally different objects. The test case $\{u_1, u_2\}$, and others like it, are produced by first partitioning the input domain of each method into subdomains, where each subdomain corresponds to a particular path in the method. Test points are then selected from each subdomain. Term rewriting is applied to the equational axioms using the test points to produce corresponding method sequences.

A major limitation of Chen, Tse, Chan, and Chen's approach is a lack of support for inheritance and polymorphism. The authors state that this is not covered by their approach. Unfortunately, they do not offer any insights regarding how the lack of support for these object-oriented features can be overcome.

McGregor presents another approach to testing classes according to their functional behavior, and combines the two perspectives of state-based and method sequence-based testing [41]. Like the approaches of Doong and Frankl, his approach tests a class by subjecting it to a sequence of method invocations. However, unlike these other two approaches, McGregor's approach does account for a strict form of inheritance where instances of subclass must be substitutable for instance of their superclasses [39][38]. This view restricts the types of changes/extensions that a derived class can make with respect to its parents. In turn, this has the effect of placing constraints on the test cases for the functional behavior of new (derived) classes. Specifically, all of the test cases defined for the parent should continue to function correctly for the child. Further, additional test cases must be added to account for any new substates introduced by the derived class.

McGregor's restriction that the inheritance relation be strict permits three implications to be inferred [40][43]. First, all states present in the parent must also be present in the child. The child class cannot remove a state. This is fundamental to preserving the observable behavior of the parent. Second, any new state that is introduced by the child is contained as a substate of one of the states inherited from the parent. This includes the case where additional attributes are added. In this situation, the resulting substates are considered to be concurrent with the other substates of the inherited state. The third implication is that the child class may not delete any inherited transition. This too is fundamental to the preservation of the parent's observable behavior.

In another work, McGregor provides guidelines for the generation of functional test cases based on a class' state machine [40]. In his approach, every method that can change the state is considered to be a transition from all states. Those situations where such a transition is illegal result in the generation of an exception. The following outline summarizes the process of test case construction:

1. Construct a test case for every method that results in a state change.
2. Construct a test case for every initial state.
3. Construct a test case for every transition in the state representation.
4. Construct a test case for every "convenience" method in the class.[6]
5. Construct test cases for every destructor.

McGregor claims that tests generated using the above outline subsumes the "all methods" and "all states" test adequacy criteria, though he offers no proof of this nor a definition of this criteria [40].

McGregor also presents an algorithm for constructing functional test cases that consist of a sequence of messages that cover a given class specification [41][43]. The algorithm produces a set of test cases the satisfy the following requirements:

1. Test cases are constructed for all accessor methods.
2. Test cases are constructed that produce all of the post-conditions for each method. This includes all possible outcomes: normal conditions, exceptions, etc.
3. Each test case includes a check to ensure that the class invariant holds.
4. Each test case begins with a valid initial state for the class.
5. Test cases are generated that test every transition in the state model for a class.

Each state-based test case is a triple consisting of the initial state of the class under test, the sequence of messages that are to be sent to the class (including whatever messages are necessary to place the class in the require state for the test), and the expected outcome of the test. The

[6]McGregor does not provide a definition for a *convenience method.*

algorithm that McGregor describes satisfies the *all transitions* adequacy criterion [42]. McGregor points out that the algorithm can be easily modified to provide an n-way switch (i.e. all possible combinations of states and transitions)[41] [43].

3.4 Integration Testing of Object-Oriented Programs

While unit level testing and class testing are important approaches for testing object-oriented programs, a perhaps more important form is that of integration testing. As discussed in Sections 1.1 and 2, object-oriented programs consist of a number of separate units (classes) that are built in isolation and then composed using inheritance and aggregation relationships to form higher level units. Unfortunately, the techniques for class testing described in the previous section are not sufficient to test these relationships. Surprisingly, very little research has been to date that focuses on this area. In his dissertation, Overbeck presents an approach that is based on testing contracts among client and server classes [52]. A contract is a constraint contained in the specification of a class and specifies the preconditions and postconditions of each public method that the class defines. Further, a contract imposes certain relationships among classes that are related by inheritance. In particular, preconditions can only be weakened and postconditions strengthened in overriding methods [45]. The basic idea is to test the interactions among classes to ensure that the client is using the server correctly, and that the results returned from the server are understood by the client. This is done by imposing a special test filter that sits between the client and the server and that catches method invocations on the server. 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 server for processing. Otherwise, an error is reported and an exception thrown.

The client's ability to understand the results returned from the server is tested by varying the conditions of each test to achieve as much diversity in the results returned from the server as possible. Conventional unit testing techniques are used to assess the correctness of the results.

Overbeck also applies his approach to inheritance in a similar manner, but includes tests that ensure that the correct precondition and postcondition relationships among classes in an inheritance hierarchy are preserved [52].

Jorgensen and Erickson describe an approach to integration testing that is similar in many respects to black-box testing techniques [34]. In their approach, they define paths through a collection of classes that form a system. Each of these paths is associated with a particular input event and traverses those 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 agree with what is expected. Faults are identified by tracing back along the path to each of the participants.

Binder's FREE approach (described in the preceding section) includes provisions for testing large clusters of classes by synthesizing a system level state machine (mode machine in Binder's terminology) [8]. The boundary of the system under test is established, with clients calling into the system, and the system making calls into the environment (operating system). States are used to identify stimulus-response pairs. These interactions are used to define the scope of the testing effort. Similar to Jorgensen and Erickson's approach, Binder's FREE approach identifies the inputs to the system, identifies the classes that are the recipients of these inputs, and then traces the execution through the system.

3.5 Other Approaches of Testing Object-Oriented Software

Offutt and Irvine report on an experiment conducted to determine if the Category-Partition [51][3] testing technique is effective when applied to object-oriented software [50]. In their experiment, they seeded 23 types of faults into two C++ programs. These faults were based on common programming mistakes reported by Meyer [47] and professional experience of one of the authors. Their results showed that the Category-Partition technique is effective at detecting faults that involve implicit functions, inheritance, initialization and encapsulation, but not at detecting memory related faults.

Harrold and Rothermel describe an approach that applies data-flow analysis to classes [27]. 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) graph 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 [53] to the CCFG to compute definition-use pairs for each of the three types of analyses. Interestingly enough, 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.

3.6 Coupling-Based Testing

Jin and Offutt present an approach to integration testing that is based upon coupling relationships that exist among variables across call sites in procedures [33][7]. In their work they define three types of coupling relationships that must be tested: *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 one procedure references global variables that are referenced by another. Finally, external device couplings occur when a procedure accesses the same external storage medium that another does. These concepts are cruical to, and form the basis of, the research that will carred out in this dissertation.

Jin and Offutt's approach requires that programs under test 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

[7]A call site is a location in a procedure where another procedure is invoked.

called procedure. 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.

3.6.1 Coupling-Based Testing Definitions

A number of definitions are necessary to discuss the concepts of Coupling-Based Testing. The definitions below are from Jin and Offutt's original coupling definitions [33]. In the following, for program P, V_P is the set of variables that are referenced by P, and N_P is the set of nodes in P. P_1 and P_2 are specific program units, and x and y are program variables.

- $\mathbf{def\_clear\_path(P, x, i, j) : Boolean}$: Evaluates to *true* if there is a definition-clear path from i to j with respect to x, where $x \in V_P$, and $i, j \in N_P$.
- **call_site** A node $i \in N_{P_1}$ such that there is a call at i from P_1 to P_2.
- $\mathbf{Call(m_1, m_2, call\_site, x \rightarrow y)}$: *Boolean* Evaluates to *true* if P_1 calls P_2 at *call_site* and actual parameter x is mapped to formal parameter y.
- $\mathbf{Return(v)}$: The nodes that return values of v to the calling unit.
- $\mathbf{Start(P)}$: The first node in P, *Start*$(P) \in N_P$.
- $\mathbf{Def(P, x)}$: The set of nodes in unit P that contain a definition of x.
- $\mathbf{Use(P, x)}$: The set of nodes in unit P that contain a use of x.
- **Coupling-def**: A node $i \in N_{P_1}$ that contains a definition that can reach a use in P_2 on some execution path. The following list formally defines the three types of coupling definitions that occur:

  1. **Last-def-before-call**: *ldbc-def*$(P_1, call\_site\_x) =$
     $\{i \in N_{P_1} \bullet x \in defs(i) \wedge def\_clear\_path(x, i, call\_site)\}$

2. **Last-def-before-return**:[8] $\textit{ldbr-def}(P_2, y) =$

   $\{j \in N_{P_2} \bullet y \in \mathit{defs}(j) \wedge \mathit{def\_clear\_path}(y, j, \mathit{Return}(y)\}$

3. **Shared-data-def**: $\textit{shared-def}(P_2, g) = \{i \in N_{P_2} \bullet i \in \mathit{def}(P_3, g) \wedge \mathit{nonlocal}(P_3, g)\}$

- **Coupling-use**: A node $i \in N_m$ that contains a use that can be reached by a definition in another unit on at least one execution path.

  1. **First-use-after-call**: For call-by-reference parameters, the set of nodes after *call_site* that have a use of a formal parameter x such that there is a def-clear path from *call_site* to that node with respect to x. Formally: $\textit{fac-use}(P_1, \mathit{call\_site}, x) = \{i \in N_{P_1} \bullet x \in \mathit{uses}(i) \wedge \mathit{def\_path}(P_1, \mathit{call\_site}, x) = \varnothing\}$.

  2. **First-use-in-callee**: The set of nodes in the callee that contain a use of a formal parameter such that there is a def-clear path from the start node of the unit to the node containing the use. Formally: $\textit{fic-use}(P_2, y) = \{j \in N_{P_2} \mid \mathit{c\_use}(P_2, y, j) \vee$
  $\mathit{i\_use}(P_2, y, i, j) \vee \mathit{p\_use}(P_2, y, i, j)) \wedge \mathit{use\_path}(P_2, \mathit{start}(P_2), j, y) = \varnothing\}$

  3. **Shared-data-use**: The set of nodes in a unit that have a use of a global variable. Formally: $\mathit{shared\_use}(P_4, g) = \{i \in N_{P_4} \mid i \in \mathit{use}(P_4, g) \wedge \mathit{nonlocal}(P_4, g)\}$.

3.6.2 Coupling-Based Testing Paths

Jin and Offutt define a coupling path between two program units to be a path that begins with a definition of a variable in the calling unit that extends to a corresponding use in the called unit [33]. They define the following three types of coupling paths:

1. **Parameter Coupling Path**: The ordered pair (i, j) consisting of the node i that contains the last definition of a variable x prior to a call site j, and to every first use in the called unit. Formally,:

[8] Jin and Offutt restrict this definition to parameters that are passed by reference, where y is a formal argument to P_2.

- *parameter-coupling*$(P_1, P_2, call\_site, x, y) =$

  $\{(i,j), i \in N_{P_1}, j \in N_{P_2} \mid i \in ldbc\_def(P_1, call\_site, x) \wedge j \in fic\_use(P_2, y)\}$

  If x is passed by reference, then a parameter coupling path also exists from each last definition of the corresponding formal parameter prior to a return. This is defined as:

- *parameter-coupling*$(P_1, P_2, call\_site, x, y) =$

  $\{(i,j), i \in N_{P_1}, j \in N_{P_2} \mid i \in ldbr\_def(P_2, y) \wedge j \in fac\_use(P_1, call\_site, x)\}$

  Note that in both of these definitions, there must be definition-clear paths from the definition to the corresponding use across unit boundaries.

2. **Shared Data Coupling Path**: A shared data coupling path exists for each global variable g that is defined in P_1 and used in P_2. The path extending from the definition to the use must be definition-clear with respect to g. Formally:

   - *Shared-data-coupling*$(P_1, P_2, g) =$

     $\{(i,j), i \in N_{P_1}, j \in N_{P_2} \mid i \in def(P_1, g) \wedge j \in use(P_2, g)\}$

3. **External Device Coupling Path**: Each pair of references *(i,j)* to a common device along an execution path is an external device coupling.

3.6.3 Coupling-Based Testing Criteria

Jin and Offutt use their coupling-based testing formalisms to extend traditional data flow testing criteria by defining four coupling test criteria [33]. These criteria, they claim, provide an increasing amount of coverage, but at an additional cost in terms of effort. In the following definitions, 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 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 paths 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 does ensure that each loop body is executed at least once, but does not require all possible executions.

A subsumption relationship exists between two test adequacy criteria A and B if and only if for every possible program, any test set that satisfies A also satisfies B [**?**]. These criteria can be arranged in a hierarchy according to the subsumption relationships. Figure 4 shows the subsumption hierarchy defined by Jin and Offutt for the coupling criteria [33]. As shown, *Call-coupling* is subsumed by *All-coupling-defs*, *All-coupling-defs* is subsumed by *All-coupling-uses*, and so on.

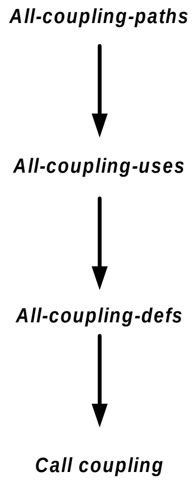

Figure 4: Coupling Testing Subsumption Hierarchy

4 Proposed Research

Thesis Statement: *Coupling based testing techniques can be extended to detect the faults that result from the polymorphic relationships among components in an object-oriented program.*

4.1 Approach

The approach undertaken in this research will involve extension of the work done by Jin and Offutt to accomodate the object-oriented features of inheritance and polymorphism. Validation of the resulting research will be established by experimentation and empirical analysis. The following list summarizes the research activities that will carried out in support of this approach:

1. Extend the coupling path definitions of Jin and Offutt [33] to accommodate the effects of the additional relationships found in object-oriented programs. Specifically, this includes amending the definitions for all forms of coupling-defs, coupling-uses, and external references.

2. Define a set of techniques and formalisms that test the additional types of coupling relationships found in object-oriented components.

3. Define a set of test adequacy criteria based on the techniques and formalisms defined in activity 2 above.

The following sections describe each of these activities in detail.

4.1.1 Extension of Coupling Path Definitions

This section presents the definitions and concepts necessary to support the extension of the coupling path definitions of Jin and Offutt [33].

Definitions for OO-Enhanced Coupling Paths. In the following definitions, o is an identifier whose type is *reference to an instance* of some declared object type (i.e. o's declared base type – a class). A reference is a pointer to some memory location that contains an instance of some type (i.e. a value of the type). The reference o is restricted to referring only to instances whose actual types (i.e. their instantiated types) is the declared base type of o, or one of the descendants of o's declared type. The instance referenced by o is indicated by o_r.

Type Families. In procedural languages such as C and Pascal, and object-based languages such as Ada 83 and Modula-2, programmers can define new types. Similarly, strongly typed object-oriented languages such as Java, C++, and Ada 95 provide such mechanisms as well. However, unlike their procedural and object-based counterparts, user defined types in object-oriented languages can be grouped into families of types. All the members of a given type family share a common behavior. Because of this, an instance of any member of a given type family can be freely substituted for an instance of any other member. The mechanisms that make this possible are inheritance and polymorphism. Inheritance allows type families to be defined, and polymorphism allows instances in a family to be freely substituted for instances of the base for the family. 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 5 illustrates this with four type families, each defined by one of the classes in the hierarchy. Table 1 summarizes each of these families.

Instance Contexts. In procedure-oriented languages, all procedures and functions are not associated with any type or data object. This is in contrast to object-oriented and object-based languages where many or all methods are associated with a particular type definition. In this situation, these methods have special access privileges to the underlying state representation of the type. For example, the state representation of a stack might be defined using a linked list, and the methods defined in the stack would directly manipulate the linked list in providing the

Base type	**Members**
W	$\{W, X, Y, Z\}$
X	$\{X, Y\}$
Y	$\{Y\}$
Z	$\{Z\}$

Table 1: Sample type families

corresponding semantics of the type.

Method calls can occur in two circumstances: (1) with respect to some instance, or (2) where there is no instance. The former are called *instance methods* and the later *class methods*.[9] Calls to class methods are syntactically identical to calls made to a procedure or function typically found in procedure-oriented programs.

For a call to an instance method, there are two syntactic possibilities. The first occurs when the instance is explicit, as in *o.m()*. Here, method m is made with respect to the type (class) instance that is bound to the reference o. In the second situation, the call is made with respect to an implicit instance, as in *p()*. This can only occur when p appears in the program text of another method that was called through an explicit instance (e.g. *o.m()*). Furthermore, p must be an instance method appearing in the same type definition that contains the method that was called through the explicit instance (e.g. the declared type of o). In this case, the implicit instance for *p()* is the same as the explicit instance. In object-oriented language such as Java and C++, this implicit instance is referred to syntactically as *this*. However, its use is generally optional. Other object-oriented languages such as Eiffel and Ada-95 have similar syntactic constructs.

[9]In hybrid object-oriented languages, such as C++, method calls can also be to *free-subprograms*. These are essentially equialent in every repect to procedures and functions found in procedure-oriented langauges, and the original coupling definitions of Jin and Offutt apply in their unmodified form. Consequently, free-subprograms will not be considered further.

As an example illustrating explicit instance contexts, consider the following code fragment where there are two instances of `Stack`, one each associated with the identifiers `s1` and `s2`, respectively.

```
1 Stack s1;
2 Stack s2;

3 s1.push(7);
4 s2.push(1);
```

At line 3, the method *push* is invoked in the context of the `Stack` instance associated with `s1`. Similarly, at line 4 *push* is also invoked in the context of a `Stack` instance. However, in this case, the instance is that associated with `s2`. These two calls to *push* are made in different *instance contexts*. Any effects on the instance associated with `s1` caused by the first call to *push* are completly independent of the effects in the instance associated with `s2` that the second call has. Further, the first call has no effects on the instance associated with `s2`.

Stating that a method is called *in the context of an instance* implies that the call is made through either an implicit or explicit class instance, as described above.

Indirect definitions and uses. A class has a state space that is made up of one or more variables. An object (instance) o_r of that class is defined (i.e. assigned a value) when one or more of the variables in the object's state space is defined. An *indirect definition*, abbreviated *i-def*, occurs when a call is made to a method m that is bound to an object reference o, and m defines one or more of the state variables in o_r's state space. Similarly, an *indirect use* (abbreviated *i-use*) occurs when m references the value of one or more of o_r's state variables. Note that an *i-def* results in a state change for the instance o_r, but not to the reference o. In all cases, an *i-use* is state-preserving for both o and o_r.

To see an example of these concepts, consider the class diagram shown in Figure 5. Assume that class W includes a method *FactoryForW()* that returns an instance of W. At node 1 in the control flow fragment shown in Figure 6, this instance is bound to o. This constitutes a local definition of the object reference o that results from the call to the method *FactoryForW()*.

Table 2 shows that *W::m* defines v *(*the notation W_r represents the instance of W that the definition/use is with respect to) , so an indirect definition occurs at node 2 through *o.m().* Thus, any call to m with respect an instance of W bound to o results in an indirect definition of the state of o_r. Note that there are no indirect uses by method m. Table 2 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.

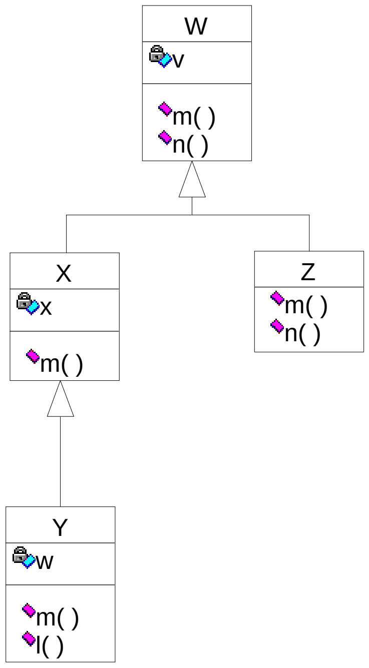

Figure 5: Sample Class Diagram

Satisfying methods. When discussing the indirect definitions and uses that can occur at call sites through object references, it becomes necessary to consider not just the syntactic (i.e. apparent) call that is made, but all of the methods that can potentially execute. This depends

Method	**Definitions**	**Uses**
$W{::}m$	$\{W_r.v\}$	
$W{::}n$		$\{W_r.v\}$
$X{::}m$	$\{W_r.v, X_r.x\}$	
$Y{::}m$	$\{W_r.v, Y_r.w\}$	
$Y{::}l$		$\{W_r.w\}$
$Z{::}m$	$\{W_r.v\}$	
$Z{::}n$		$\{W_r.v\}$

Table 2: Indirect Definitions and Uses for type family W

upon the type of the instance that is bound to the object reference (assuming dynamic binding). For example, again referring to Figures 5 and 6, a call to method m at node 2 can refer to any of the elements in S:

$$S = \{\, W{::}m, X{::}m, Y{::}m, Z{::}m\} \tag{1}$$

Based on the preceding discussion, we define a 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. Collectively, this set determines the extent of the state changing effect that can potentially result from the call. Returning again to Figures 5 and 6, the *i-def* set for the

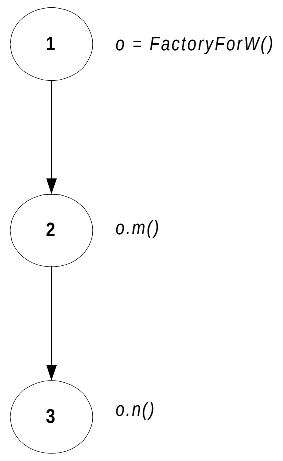

Figure 6: CFG Fragment

call at node 2 is the following set of ordered pairs:

$$i\_def(2, o_r, m) = \{(W{::}m, \{W_r.v\}), (X{::}m, \{W_r.v, X_r.x\}), (Y{::}m, \{W_r.v, Y_r.w), Z{::}m, \{W_r.v\})\} \tag{2}$$

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

As Table 2 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 2 shows that there are two methods that satisfy the call to *o.n()* that do have non-empty *i-use* sets (but their *i-def* sets are empty), which yields the following *i-use* set:

$$i\_use(3, o_r, n) = \{(W{::}n, \{W_r.v\}), (Z{::}m, \{W_r.v\})\} \tag{3}$$

Extending the Coupling Definitions. The original definitions of Jin and Offutt consider couplings among procedures in three different contexts: (1) procedure parameters, (2) shared global variables, and (3) external devices [33]. These coupling contexts exist in object-oriented programs as well, and Jin and Offutt's technique are applicable in object-oriented programs in many situations where these coupling contexts occur. However, there are several cases that occur in which their techniques are not applicable.

In object-oriented programs, the third context (*coupling through external devices*) is a special case of the first two. To see this, consider that the primary abstraction mechanism in an object-oriented language is the class. These languages represent external devices (files, printers, etc.) by one or more classes that encapsulate the behavior and state of these devices. Instances of these classes are either passed as parameters between methods (procedures), exist in a global state defined by a class, or are variables local to a method. All manipulations of the external device are through the abstraction. Indeed, a program that uses the abstraction is completely unaware of the existence of any external device. Coupling occurs through either of the first two cases, though the third case can be involved in a parameter coupling. Regardless, all coupling occurs through instances of the abstraction that is used to represent the external device. In object-oriented languages (and some procedure-oriented languages), the fact that a device is external is hidden behind the abstraction. Because of this, coupling through external devices will not be considered further in this research since it is subsumed by the first two cases.

In analyzing the coupling paths in an object-oriented program, twelve cases must be considered. Many of these are due to the effects of inheritance and polymorphism, and require additional definitions and techniques beyond Jin and Offutt's. Others occur in procedure-oriented programs and require no special consideration. The following list enumerates each of these twelve and discusses the coupling path implications for each that arise due to the differences between object-oriented and procedure-oriented programs:

1. Calls made through an explicit object reference.

2. Calls to locally defined non-polymorphic methods.

3. Explicit calls to inherited methods.

4. Implicit calls to inherited non-polymorphic methods.
5. Calls to inherited class methods.
6. Calls to external class methods.
7. Calls to methods through external polymorphic class variables.
8. Calls to methods through locally defined class variables.
9. Calls to polymorphic methods through inherited class variables.
10. Implicit calls to inherited polymorphic methods.
11. Calls to locally defined polymorphic methods.
12. Calls to polymorphic methods through local object references.

The following discussion examines each of these differences in detail.

1. *Calls made through an explicit object reference.* This is the most common calling context for method calls in object-oriented languages. Consider the call *o.m()*. There are three entities that participate in this call: the called method m, the object reference o, and the instance (object) bound to o, represented as o_r. There may be several versions of m that can be executed and each can have a different effect on the state of o_r. Which version is determined by whether or not the static (i.e. declared) type of *o, C,* declares m to be polymorphic, and if so, what the actual type of o_r is (i.e. C or one of its descendants). However, since the actual type is a dynamic property, the best that can be done is to recognize that the method that is actually executed will be an element of the set of methods that can satisfy the call *o.m()*. Consequently, the effect of the call on the state of o_r is the union of the effects of all the methods in the satisfying set of *C.m()*. From a coupling perspective, what must be considered is how definitions of state variables by one method can reach corresponding uses in other methods. Jin and Offutt's original definitions do not handle such cases.

2. *Calls to locally defined non-polymorphic methods.* A local method is defined within the caller's class. Non-polymorphic methods cannot be overridden by a descendant of the method's class. These methods have the same call syntax as calls to their procedural counterparts. For example, suppose that a class C contains a definition of a polymorphic method m and a non-polymorphic method *f*. Further suppose that m calls f directly. This will appear in the body of m as the statement *f()*. Since f is not a polymorphic method, the satisfying set for the call in m will be the set containing only the element $\{f\}$, which is equivalent to a call to a procedure in a procedure-oriented language (ignoring parameter types). Consequently, the definitions of Jin and Offutt are applicable in their original form.

3. *Explicit calls to inherited methods.* In some languages, a method in a class C can make an explicit call to a method that is defined in an ancestor class. For example, in Java, a call to a method defined in the parent of C is expressed as *super.g()*, where g is a method visible in C that is defined by C's parent. Similarly, in C++, a call can be made to any visible method defined by any ancestor of C using the scope resolution operator, as in *A::g()*. In this case, g is defined in class A, and A is an ancestor of C. The key observation to note for both of these languages is that explicit calls to a method defined in an ancestor does not involve an explicit instance context. Thus, there is no possibility of polymorphic behavior. From a coupling perspective, this is identical to a call to any procedure in a procedure-oriented languages though there are additional syntactic mechanisms present to disambiguate the call. Thus, the definitions of Jin and Offutt apply to this case as well.

4. *Implicit calls to inherited non-polymorphic methods.* In this case, a method m is called that is defined by an ancestor class, and m cannot be overridden by a method in a descendant class. In Java, such methods are declared as *final*; in C++, they are methods that are not declared to be *virtual*. This characteristic of m precludes the possibility of any polymorphic behavior whenever it is called. A call to m is syntactically equivalent to a call to a non-polymorphic

local method because there is no explicit instance context specified. The original definitions of Jin and Offutt still apply.

5. *Calls to inherited class methods.* These are calls to methods that are in classes that are ancestors of the class containing the calling method. They are declared using the *static* keyword in Java and C++ and indicate that defined method does not execute with respect to an instance context, and hence are not polymorphic. Syntactically, calls to such methods are structurally equivalent to calls to procedures and functions in procedure-oriented languages. The original definitions of Jin and Offutt apply to these types of method calls.

6. *Calls to external class methods.* These are public class methods defined in a class P that is not an ancestor. Syntactically, calls to external class methods are usually qualified by the name of the method's class. For example, in Java, if P contains a static class method q, then a call would look like *P.q()*. Since this call is made independent of any instance context, it is not polymorphic. Thus, Jin and Offutt's original definitions still apply.

7. *Calls to methods through external polymorphic class variables.* Like external class methods, these calls are made through the context of a class that is not an ancestor. However, they differ in that the call is made in the context of a class variable that is declared *public* in the defining class. An example taken from the standard Java class library is *System.out.println("some string")*, where *out* is a public class variable that is an instance of the class *PrintStream*, and *println* is a public instance method contained in the definition of *PrintStream*. The call to *println* is made in the instance context specified by the class variable *out*, and the actual method that is executed depends upon the declared type of *out* (i.e. *PrintStream*) and the actual type of the instance bound to it (i.e. *PrintStream* or one of its descendants that override *println*). From a coupling perspective, the issue is what are the effects of the call to *println* on the instance bound to *out*, and also to any other methods that have a corresponding use of that instance. Note that this is a special case of 1 above. As with that case, Jin and Offutt's

definitions do not apply.

8. *Calls to methods through locally defined class variables.* These are similar to case 7 except that the calling method's class also defines the class variable through which the method is called. However, syntactically the example would most likely appear as *out.println(...)*, which is structurally equivalent to case 1. The only difference is that the instance context is through a class variable instead of an instance variable. Jin and Offutt's original definitions do not apply for the same reasons cited in cases 1 and 7.

9. *Calls to polymorphic methods through inherited class variables.* These are similar to case 8 except that the calling method is contained in a descendant class. Again, the possibility of a polymorphic call exists, and the definitions of Jin and Offutt do not apply for the reasons given in cases 1 and 7.

10. *Implicit calls to inherited polymorphic methods.* These are calls to methods that are defined by an ancestor of the caller's class, and which are not declared as *final* methods in Java, and *virtual* methods in C++. They are syntactically equivalent to a call to a locally defined instance method. Since polymorphic behavior is a possibility, the definitions of Jin and Offutt do not apply to this case.

11. *Calls to locally defined polymorphic methods.* These calls are the same as those in case 10 except that they are defined in the caller's class. For similar reasons, Jin and Offutt's definitions do not apply to this case.

12. *Calls to polymorphic methods through local object references.* Methods can be called either through object references that are passed as parameters to the caller, or through references that are declared as variables local to a method. These calls are a special case of the prototypical calling context described in case 1.

The effects of method calls for cases 10 and 11 can be particularly complex when the actual type of the current instance is a descendant of the class that contains the caller. To understand this, consider the class hierarchy shown in Figure 7. Class *A* contains definitions for methods *d, g, h, i, j*, and *l*, class *B* contains definitions for methods that override *h* and *j*, and a definition for a new method *k,* and class *C* contains overriding definitions for methods *i*, *j*, and *l.* Assume that all of these methods are polymorphic, and hence can be overridden by a descendant class. Figure 8 shows a *Polypmorphic Call Graph* that depicts the sequence of methods that would be called from method *d* executing in the context of an instance of *A*. The solid directed line segments indicate which methods are actually executed in the presence of polymorphic behavior (there is no polymorphic behavior in this example). As the diagram shows, *d* calls *g*, which in turn calls *h*, *h* calls *i*, and *i* calls *j.* Note that there is no polymorphic behavior present in this method call sequence since all of the methods are defined by *A,* and the current instance context that *d* is executing in is also of type *A*. The arrows marked as implicit indicate that the particular call is not made with an explicit instance context. Note that each of these calls is an example of case 11.

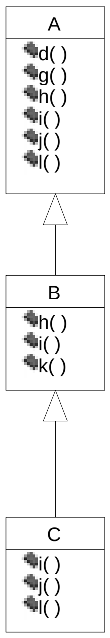

Figure 7: Sample class hierarchy illustrating polymorphic behavior

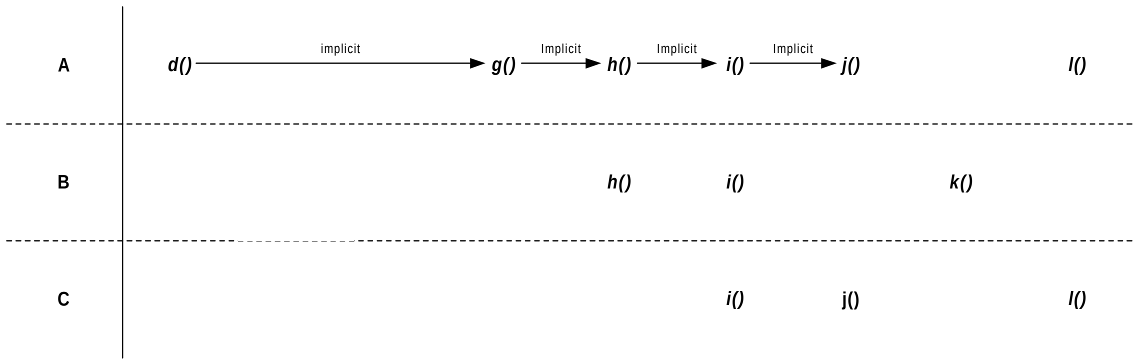

Figure 8: Method *d* executing in the context of an instance of *A*

Now consider the method sequence that results when *d* executes in the instance context of type *B*. This is shown in Figure 9. As before, the solid directed line segments indicate the calls that are executed. In contrast, the dashed directed line segments indicate the *apparent call* that is made from the text of the calling method. As depicted, the call that *d* makes to method *g* is followed by *g* making a call to *A::h*. However, this call does not execute *A::h* (as indicated by the dashed arrow), but instead the overriding method *B::h* is executed. In turn, *B::h* calls the locally defined method *B::i*. In response, *B::i* makes an explicit call to *A::i* (e.g. *super.i()* in Java). Finally, *A::i* calls *A::j* as in Figure 8. The directed dashed line indicates the call path that would be taken if the current instance were of type *A*. This example illustrates how inheritance and polymorphism combine to affect the execution path of simple method calls within a type family. Note that the calls from *d* to *g*, *h* to *i*, and *i* to *j* are examples of case 11. The explicit call from *B::i* to *A::i* is an example of case 3.

Finally, consider the sequence of method calls shown in Figure 10. This example depicts the situation where *A::d* is executing in the context of an instance of *C*. Because of this, the interaction of methods is much more complex due to the presence of overriding methods in *C* and in *B*. The method calls *A::d* to *A::g*, *A::g* to *A::h*, *B::h* to *B::i*, and *A::i* to *A::j* are examples of case 11. Likewise, the explicit class to ancestor methods *C::i* to *B::i*, and *B::i* to *A::i* are examples of case

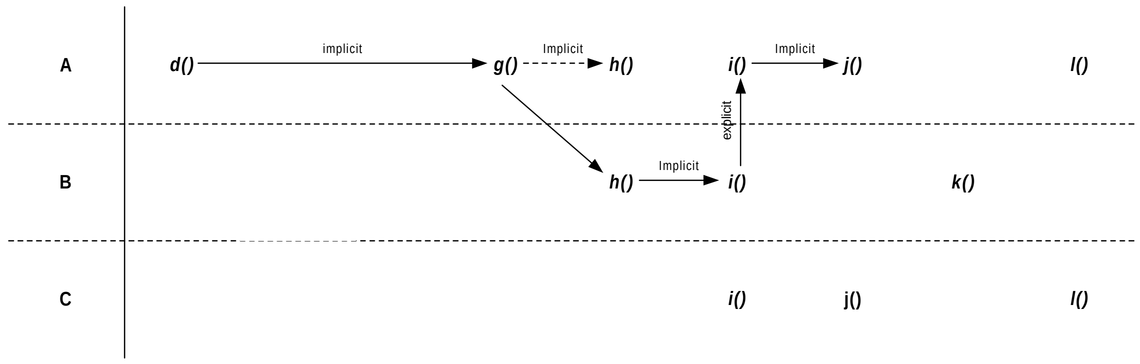

Figure 9: Method d executing in the context of an instance of B

3. Finally, the calls *C::j* to *B::k*, and *B::k* to *A::k* are examples of case 10.

Observe that a potential data flow anomaly occurs as a result of this method overriding in both A and C. Figure 8 shows the method sequence where the call to *A::h* is part of the execution of *A::g*, and *A::i* is called from *A::h*. However, when *A::d* is executing in the context of a B or C instance, *A::h* does not execute at all since B provides an overriding definition of h. Further, regardless of the type of the current instance context, *A::i* will execute. The potential anomaly results from the fact that *A::i* may have a dependency on the state effects produced by *A::h*. For example, it may be that *A::h* defined a state variable v defined in A for which *A::h* has a subsequent use. If the methods overriding *A::h* do have at least the same state effects, i.e. a definition of v, then a data flow anomaly will occur. Clearly, the presence of inheritance and polymorphism increases the complexity of the method interaction substantially, particularly with respect to state space effects.

These preceding three examples serve to illustrate the complexities that can occur when inheritance and polymorphism factor in to the composition of components in an object-oriented language. From a coupling and integration perspective, there are two issues determining what calls can be executed in the presence of inheritance and polymorphism, and what effects the calls have on the corresponding state space. Careful analysis of the preceding twelve cases with respect to these issues results in the three categories shown in Table 3. Those cases that are assigned to Category

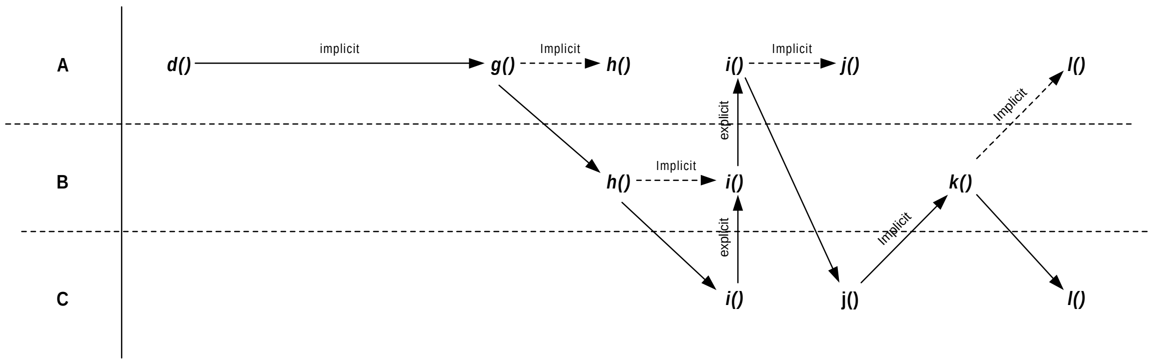

Figure 10: Method d executing in the context of an instance of C

Category	**Cases**	**Instance context characteristics**
I	1,7,8,9,12	Implicit or explicit instance context
II	2,3,4,5,6	No instance context
III	10,11	No instance context, possible polymorphism

Table 3: Categorized call types

II are handled by the original definitions of Jin and Offutt. In contrast, those cases assigned to Category I and to Category III will require extension to their definitions and additional analysis techniques.

4.1.2 Definition of techniques and formalisms

Once the coupling definitions have been extended, it will be necessary to develop new techniques for the instrumentation of programs in order to collect and analyze coupling coverage data. A new representation must be developed that is rich enough to model the diverse structural relationships that occur in object-oriented programs. In particular, this requires an extension to the System Dependence Graph (SDG) developed by Horwitz and others to accommodate these relationships [32]. The Coupling-Based Coverage Tool will be based upon an extended SDG known as the

Program Graph Representation (PGR) (Section 4.2.2). The generation of instrumented code itself should be straightforward given a suitable representation.

4.1.3 Definition of test adequacy criteria

Offutt and Jin define several criteria based on software couplings [33]. Clearly, the differences between the coverage that their original definitions provide and those required to support object-oriented programs will require modifications to their criteria. At this point it is not known precisely what form these modifications and additions will take. At the very least support for inheritance and polymorphic behavior will need to be added.

4.2 Validation Method

Validation of this research will be conducted through the application of a carefully and well designed experiment. The goal of this experiment will be to determine if there is any statistically significant difference among two existing subject test adequacy criteria (see section 4.2.1) and the test adequacy criteria developed as part of this research. To facilitate this experiment, a tool will be developed that is capable of evaluating coverage for the test adequacy criteria discussed in section 4.1. The following subsections provide an outline of the experimental design and the architecture of the coverage tool.

4.2.1 Experiment Design

This section describes the design of the experiment that will be conducted to validate the results of this research.

Subject Programs. The programs used to validate this research will be drawn from programs written by professional programmers at the Software Productivity Consortium. The programs selected will be written in Java and have moderate lengths ranging between 100 to 1000 lines.

Further, these programs will be restricted to console-based Java applications. Java applets and those applications having graphical user interfaces (GUIs) will be excluded from the experiment. The number of programs actually selected will be determined at a later time.

Test Adequacy Criteria. The following three test adequacy criteria will be used in the experiment: Category-Partition, Random Testing, and Coupling-Based Testing Criteria (CBTC). The first two will be used as controls for comparing the effectiveness of the third criteria. The decision to accept or reject the experimental hypotheses of this research rests upon the outcome of statistical tests between the first two criteria and the third.

The Category-Partition method is used for the specification and generation of functional tests for a program [51][3]. The technique is based on the idea of partitioning the input domain of the program function being tested, and then selecting test data from each partition. All elements from the same partition are considered to be equivalent for purposes of testing. For purposes of this experiment, each subject program's specification will be analyzed to define the appropriate partition according to its behavior. Test data will be generated for each partition in a random fashion. Finally, adequate test cases will be created by choosing values randomly from each partition to assemble a single test case.

Random testing is an approach to testing whereby values are selected at random from the input domain of the program or system under test. The test cases are selected according to some distribution (usually uniform) and the program is executed against each test case.

The development of a set of criterion using the Coupling-Based testing techniques extended for object-oriented programs is one of the objectives for this research. As such, the details of each of these criterion are not yet known. However, it is expected that a set of criteria will be developed that are capable of guiding the testing of object-oriented programs from the perspective of the couplings that exist among components. Further, it is expected that these criteria will exist in a subsumptive hierarchical relationship such that the criteria lower in the hierarchy provide less coverage than those criteria in higher in the hierarchy, and the coverage provided by a lower hierarchy is included in that provided by a higher coverage.

Test Data. A set of test data will be developed for each subject program and test adequacy criterion pair. The test cases will be generated in a random fashion using custom generators written by hand in the Perl programming language. Each set will satisfy the corresponding coverage criterion. The procedures required to generate a given test set are specific to the program being tested.

Fault Sets. This research will be validated by determining the effectiveness of the Coupling-Based Testing Criteria (CBTC) at detecting faults that are associated with the compositional relationships found in object-oriented programs. It is necessary to start with a set of programs that contain faults known to be of this type. Accordingly, each subject program will be injected with a number of faults of the appropriate type to simulate the faults that programmers make with respect to the compositional relationships. However, this is problematic since the types of faults that programmers actually make in object-oriented programs are generally not well known, though Offutt and Irvine have done some research in this area [51]. This work will form the initial basis for the categories of faults that will be injected. However, one of the subordinate objectives for this research will be the development of a taxonomy of fault types for the compositional relationships found in object-oriented programs. This taxonomy will then serve as a guide for the types of faults that will be injected into the subject programs.

Response Measurements. The organization of this experiment is that of an $n \times 3$ non-randomized block design. Each block consists of a single subject program crossed with each of the three test adequacy criteria described earlier. This design is justified since any variation between the subject programs is not relevant, only the variation between the test adequacy criterion is of interest. Accordingly, an observation will consist of a measurement of the effectiveness of the three test adequacy criteria. The fault detecting effectiveness of a given test adequacy criteria c for a given program p with respect to a specific fault set f is defined as the ratio of the number of faults detected to the number of faults seeded (i.e. the cardinality of the fault set). This measurement will be made for each combination of subject program and test adequacy criterion.

Experimental Procedure. The conduct of the experiment will consist of several steps. In the following, let P be the set of subject programs, C the set of test adequacy criteria described above, and T be the set of test data sets generated for each combination of program and test adequacy criterion.

For each $p \in P$ and $c \in C$:

1. Generate the c-adequate test data set T_p^c.
2. Define the fault set F_p for p.
3. For each $f \in F_p$ define the set of fault-seeded programs S_p by injecting p with the corresponding fault, yielding the seeded program p_f, where each $p_f \in S_p$.
4. For each $t \in T_p^c$, determine if $p(t) = p_f(t)$ for at least one $p_f \in S_p$. If there is at least one such p_f, increment N_p^c, the number of faults detected by the test data set T_p^c.
5. Determine the fault detection effectiveness E_p^c, for test adequacy criterion c with respect to subject program p, as follows:

$$E_p^c = \frac{N_p^c}{|T_p^c|}$$

Data Analysis. The data resulting from the experiment will be done using a statistical test, such as a paired t-test [12], to determine if there is in fact any difference between the effectiveness of the two test adequacy criteria used for controls and the Coupling-Based Testing Criteria. The actual test used will depend upon the characteristics of the collected data.

For a given criteria pair, the null hypothesis (that CBTC is not more effective) will be rejected in favor of the alternate hypothesis (that CBTC is more effective) at significance level of 0.05.

4.2.2 Coupling-Based Coverage Tool

To aid in the validation of this research, a tool will be developed that is capable of analyzing arbitrary single-threaded Java application programs. This tool, the Coupling-Based Coverage Tool (CBCT), will be based on the architecture shown in Figure 11. There are two principal components of this architecture that represent separate concerns with respect to the program analysis. The first consists of the abstract syntax tree (AST) and symbol table (ST) that are produced by the parser. The AST utilizes a *visitor-based* architecture that allows for arbitrary analyses of a program's syntax tree. In this type of architecture, traversal of a tree and the processing of its nodes are done separately. This allows for varying combinations of traversal policy and semantic handling of nodes (the responsibility of the visitor). In the architecture shown in Figure 11, the *PGR Generator* plays the role of a visitor.

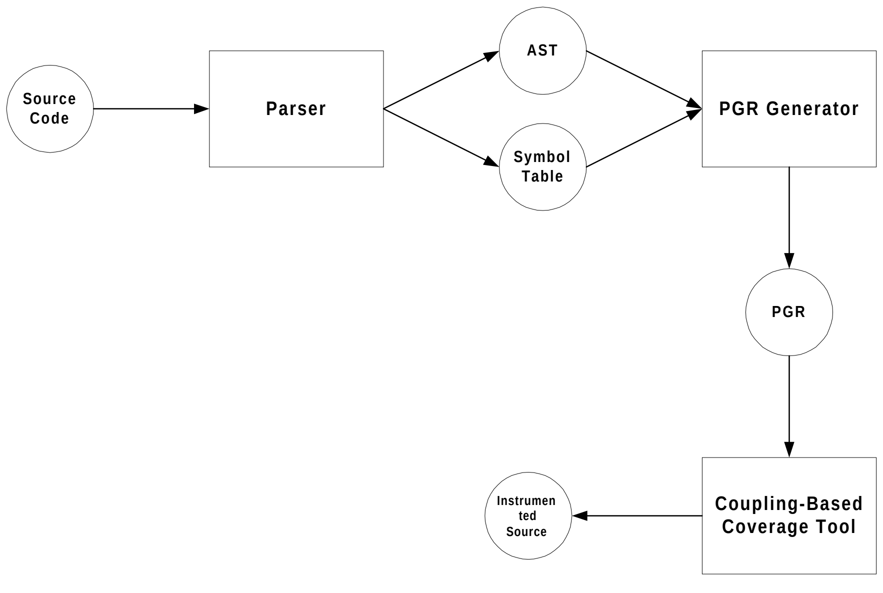

Figure 11: Architecture of the Coupling-Based Coverage Tool

The second principal component of the architecture is the *Program Graph Representation* (PGR). The PGR provides a low-level structural representation for Java programs, and includes graph representations for control flow, control dependence, and data dependence. It also includes the appropriate structure necessary to represent inheritance. Like the AST, the PGR is also a visitor-based architecture. For the architecture shown in Figure 11, the Component-Based Coverage Tool is a visitor that uses the PGR to produce a version of the original program instrumented to collect coupling coverage metrics. At present, the PGR has been completely defined by the author and Jim Bieman[10], but only partially implemented.

Figure 12 shows the resulting instrumented program in relation to its inputs, outputs and test driver. Also shown is the results analyzer that interprets the collected coverage metrics and produces a corresponding report.

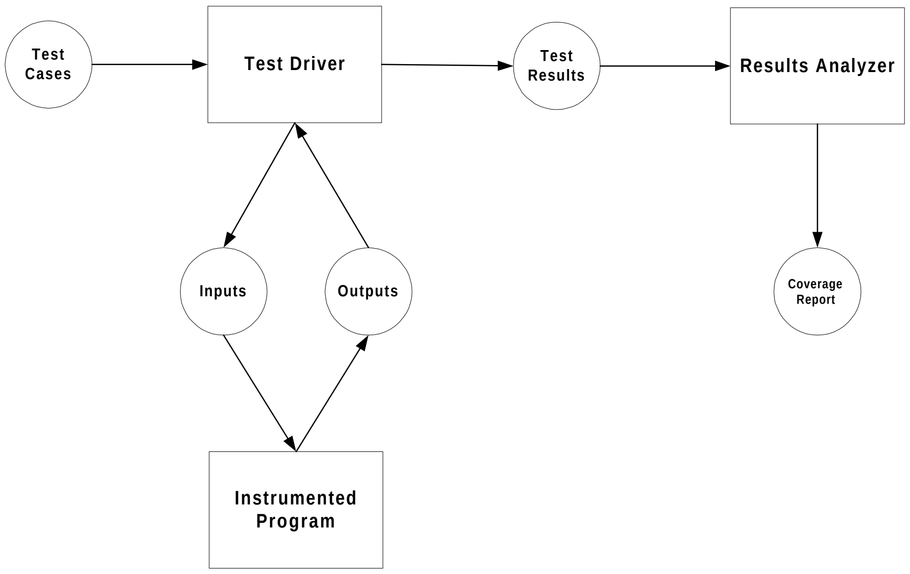

Figure 12: Test execution framework of instrumented program

[10]Department of Computer Science, Colorado State University

4.3 Unique Contributions

The unique contributions of this thesis will be:

1. An effective way to do integration testing of object-oriented programs.
2. An extended set of coupling test definitions sufficient for testing object-oriented programs.
3. A set of test adequacy criteria based on the extended coupling definitions.
4. An automated tool showing proof of concept and demonstrating practicality of approach.
5. A taxonomy of faults that result from the use of inheritance and polymorphic behavior in object-oriented programs.
6. A set of metrics for measuring software based on couplings.

5 Research Schedule

Table 4 outlines the expected plan of action and associated schedule.

6 Approximate Dissertation Outline

- Introduction
- Background and Related Material
- Extended Coupling-Based Testing
- Coupling-Based Testing Tool (CBTC)
- Experimentation
- Discussion
- Future Research and Conclusions
- Appendices
- List of References

Task	Description	End Date
1	Extension of coupling definitions	March 15, 1999
2	Development of CBCT	August 1, 1999
3	Development of OO fault classification	September 1, 1999
4	Conduct of experiment	September 15, 1999
5	Analysis of data	October 15, 1999
6	Write-up of dissertation	January 1, 2000
7	Defense of dissertation	Febuary 15, 2000

Table 4: Schedule of research activities

References

[1] Testability of object-oriented systems. Technical Report 95-01, Reliable Software Technologies, Dec. 31, 1994 1994.

[2] Unified Modeling Language, version 1.1, 1997.

[3] Balcer, M. J., Hasling, W. M., and Ostrand, T. J. Automatic generation of test scripts from formal test specifications. In *ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification (TAV3)* (Key West, FL, USA).

[4] Barbey, S., and Strohmeier, A. The problematics of testing object-oriented software. In *SQM'94 Second Conference on Software Quality Management* (Edinburgh, Scotland, UK, 1994), vol. 2, pp. 411–426.

[5] Beizer, B. *Software Testing Techniques*, 2 ed. Van Nostrand Reinhold, New York, New York, 1990.

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

[7] BERARD, E. V. *Essays on Object-Oriented Software Engineering*, vol. 1. Prentice Hall, 1993.

[8] BINDER, R. V. The FREE approach for system testing: Use-cases, threads, and relations. *Object Magazine 6*, 2.

[9] BINDER, R. V. Testing objects: Myth and reality. *Object Magazine 5*, 2 (1995), 73–75.

[10] BINDER, R. V. Trends in testing object-oriented software. *Computer 28*, 10 (1995), 68–69.

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

[12] BOX, G. E., HUNTER, W. G., AND HUNTER, J. S. *Statistics for Experimenters.* John Wiley & Sons, New York, New York, 1978.

[13] CAPPER, N. P., COLGATE, R. J., HUNTER, J. C., AND JAMES, M. F. The impact of object-oriented technology on software quality: Three case histories. *IBM Systems Journal 33*, 1 (1994), 131–157.

[14] CHEATHAM, T. J., AND MELLINGER, L. Testing object-oriented software systems. In *ACM 18th Annual Computer Science Conference* (1990), ACM Press, pp. 161–165.

[15] CHEN, H. Y., TSE, T., CHAN, F., AND CHEN, T. In black and white: An integrated approacht to class-level testing of object of object-oriented programming. 250–295.

[16] CHOW, T. S. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering SE-4*, 3 (1978), 178–87.

[17] DEMILLO, R. A., GUINDI, D. S., KING, K. N., MCCRACKEN, W. M., AND OFFUTT, A. J. An extended overview of the Mothra software testing environment. In *Second Workshop*

on Software Testing, Analysis, and Verification (Banff Alberta, July 1988), IEEE Computer Society Press, pp. 142–151.

[18] Doong, R.-K., and Frankl, P. G. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology 3*, 4 (1994), 101–130.

[19] Dorman, M. Unit testing of C++ objects. pp. 71–101.

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

[21] Firesmith, D. G. Testing object-oriented software. Technical report, Advanced Technology Specialists, 1992.

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

[23] Frankl, P. G., and Weiss, S. N. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering 19*, 8 (1993), 774–87.

[24] Frankl, P. G., and Weyuker, E. J. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering 14*, 10 (1988), 1483–98.

[25] Freedman, R. S. Testability of software components. *IEEE Transactions on Software Engineering 17*, 6 (1991), 553–64.

[26] Harrold, M. J., McGregor , J., and Fitzpatrick, K. Incremental testing of object-oriented class structures. In *14th International Conference on Software Engineering* (1992), IEEE Computer Society.

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

[28] Harrold, M. J., and Soffa, M. L. Selecting and using data for integration testing. *IEEE Software 8*, 2 (1991), 58–65.

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

[30] Hong, H. S., Kwon, Y. R., and Cha, S. D. Testing of object-oriented programs based on finite state machines. In *1995 Asia Pacific Software Engineering Conference* (1995), IEEE Computer Society Press, Los Alamitos, California, pp. 234–241.

[31] Hong, H. S., Kwon, Y. R., and Cha, S. D. A state-based testing method for classes. *Journal of Korea Information Science Society(B 23*, 11 (1996), 1145–1154.

[32] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12*, 1 (1990), 26–60.

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

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

[35] Kaiser, G. Personal communication, October 1998.

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

[37] Kung, D., Suchak, N., Gao, J., Hsia, P., Toyoshima, Y., and Chen, C. On object state testing. In *Eighteenth Annual International Computer Software & Applications Conference* (1993), IEEE Computer Society Press, Los Alamitos, Calif., pp. 222–227.

[38] Leavens, G. T. Modular specification and verification of object-oriented programs. *IEEE Software 8*, 4 (1991), 72–80.

[39] Liskov, B., and Wing, J. M. Specifications and their use in defining sub-types. In *OOP-SLA'93* (New York, 1993), ACM Press.

[40] McGregor, J. D. Constructing functional test cases using incrementally derived state machines. In *11th International Conference on Testing Computer Software* (1994), USPDI, Washington, DC, pp. June 13–16.

[41] McGregor, J. D. Functional testing of classes. In *7th International Software Quality Week* (1994), Software Research Institute, San Francisco, p. May 1994.

[42] McGregor, J. D., and Dyer, D. M. A note on inheritance and state machines. *Software Engineering Notes 18*, 4 (1993), 61–69.

[43] McGregor, J. D., and Dyer, D. M. Selecting functional test cases for a class. In *11th Annual Pacific Northwest Software Quality Conference* (1993), PNSQC, Portland, Oregon, pp. 109–121.

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

[45] Meyer, B. Design by contract. In *Advances in Object-Oriented Software Engineering*, D. Mandrioli and B. Meyer, Eds. Prentice Hall, Englewood Cliffs, N.J., 1991, pp. 1–50.

[46] Meyer, B. *Object-Oriented Software Construction*, 2nd ed. Prentice Hall, Englewood Cliffs, New Jersey, 1997.

[47] MEYER, S. *Effective C++.* Addison-Wesley, Reading, Massachusetts, 1992.

[48] MORELL, L. J. Theoretical insights into fault-based testing. In *ACM SIGSOFT '89 2nd Symposium on Software Testing Analysis and Verification (TAV2)* (Banff Alberta, 1988), pp. 45–62.

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

[50] OFFUTT, A. J., AND IRVINE, A. Testing object-oriented software using the category-partition method. In *TOOLS USA'95* (Santa Barbara, California, 1995), Prentice Hall.

[51] OSTRAND, T. J., AND BALCER, M. J. The category-partition method for specifying and generating functional tests. *Communications of the ACM 31*, 6 (1988), 676–86.

[52] OVERBECK, J. *Integration Testing for Object-Oriented Software.* Ph.d., Vienna University of Technology, 1994.

[53] PANDE, H. D., LANDI, W. A., AND RYDER, B. G. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering 20*, 5 (1994), 385–403.

[54] PARNAS, D. L., SHORE, J. E., AND WEISS, D. Abstract types defined as classes of variables. In *Proceedings of Conference on Data: Abstraction, Definition and Structure* (Salt Lake City, UT, USA, 1976), pp. 22–24.

[55] PAYNE, J. E., ALEXANDER, R. T., AND HUTCHINSON, C. D. Design-for-testability for object-oriented software. *Object Magazine 7*, 5 (July 1997), 34–43.

[56] PERRY, D. E., AND KAISER, G. E. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming 2*, 5 (1990), 13–19.

[57] SANDEN, B. *Software Systems Construction with Applications in Ada.* Prentice Hall, Englewood Cliffs, New Jersey, 1993.

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

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