# TESTING THE POLYMORPHIC RELATIONSHIPS OF OBJECT-ORIENTED PROGRAMS:

by
Roger T. Alexander
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
the Requirements for the Degree
of
Doctor of Philosophy
Information Technology and Engineering

Committee:

| | |
|---|---|
| ______________________ | A. Jefferson Offutt, Thesis Director |
| ______________________ | Paul Ammann |
| ______________________ | James M. Bieman<br>Colorado State University |
| ______________________ | Hassan Gomaa |
| ______________________ | Elizabeth White |
| ______________________ | Stephen G. Nash, Associate Dean for Graduate Studies and Research |
| ______________________ | Lloyd J. Griffiths, Dean, School of Information Technology and Engineering |
| Date: ____________ | Spring 2001<br>George Mason University<br>Fairfax, Virginia |

**Testing the Polymorphic Relationships of Object-oriented Programs**

A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy in Information Technology and Engineering at George Mason University

By

Roger T. Alexander
Master of Science
George Mason University, 1994

Director: A. Jefferson Offutt
Department of Information and Software Engineering

Spring Semester 2001
George Mason University
Fairfax, Virginia

Copyright © 2001 Roger Thompson Alexander
All Rights Reserved

## Dedication

I’d like to dedicate this thesis to my family and friends who have supported me through the years while I toiled away.

- First, to my wife and soul mate, Suzanne. Sweetheart, you’ve been there from the beginning, and you, above all others, have had constant faith in me. You were always there.
- To my sons, Michael and Scott.
- To my Mother and my brother Walter.
- To my life-long and very best friend, Mr. Jack L. Baldwin, who over the years had faith and conviction in my intelligence and abilities long before I did myself.
- To my friend and mentor, Dr. Gene R. Lowrimore.

**Acknowledgments**

I would like to acknowledgment and thank the following individuals who have contributed to the success of this dissertation:

- Dr. Jeff Offutt, my advisor and friend. Jeff, just one more question...
- Dr. Paul Ammann, Dr. James Bieman, Dr. Hassan Gomaa, and Dr. Elizabeth White for serving on my committee
- Dr. Gene Lowrimore for assisting with the statistical analysis, being my sounding board, and for being a dear friend.
- Mr. Chuck Hutchinson, also for being my sounding board and friend, and for contributing his knowledge and experiences with OO faults.
- Dr. Ye Wu for his insightful comments on OO fault modeling
- Quansheng Xiao for providing one of the subject programs used to validate the research reported in this thesis.
- My wife, Suzanne, and my children, Michael and Scott, for putting up with me during this difficult period, and for providing encouragement, understanding, and their love.

## Table of Contents

**Abstract** . . . xi

**CHAPTER 1 Introduction and Overview** . . . 1

**1.1 Motivation** . . . 1
**1.2 Problem Statement** . . . 3
1.2.1 System level testing techniques . . . 4
1.2.2 Unit level testing techniques . . . 4
1.2.3 Integration Testing Techniques . . . 6
**1.3 Thesis Statement** . . . 6
**1.4 Object-oriented Programming** . . . 6
1.4.1 Classes . . . 6
1.4.2 Compositional Relationships . . . 7
1.4.3 Polymorphism and Dynamic Binding . . . 8
**1.5 Problems with method overriding and polymorphism** . . . 10
**1.6 Organization of Dissertation** . . . 18

**CHAPTER 2 Background and Related Work** . . . 20

**2.1 Issues in Testing Object-oriented Software** . . . 20
**2.2 Test Adequacy** . . . 22
**2.3 Class Testing** . . . 29
2.3.1 State-based Testing . . . 29
2.3.2 Method Sequence-Based Testing . . . 31
**2.4 Integration Testing of Object-Oriented Programs** . . . 36
**2.5 Other Approaches of Testing Object-Oriented Software** . . . 42
**2.6 Coupling-Based Testing** . . . 43
2.6.1 Coupling-Based Testing Definitions . . . 43
2.6.2 Coupling-Based Testing Paths . . . 46
2.6.3 Coupling-Based Testing Criteria . . . 47
2.6.4 Relationship to the Object-oriented Coupling-based Testing Criteria . . . 48

**CHAPTER 3 Inheritance and Polymorphism Faults** . . . 50

**3.1 A fault/failure model for polymorphic for object-oriented programs** . . . 53
3.1.1 Reachability . . . 54
3.1.2 Infection . . . 55
3.1.3 Propagation . . . 55
**3.2 Inheritance Faults and Anomalies** . . . 55

3.2.1 Inconsistent Type Use (ITU) . . . . 56
3.2.2 State Definition Anomaly (SDA) . . . . 58
3.2.3 State Definition Inconsistency due to State Variable Hiding (SDIH) . . . . 61
3.2.4 State Defined Incorrectly (SDI) . . . . 61
3.2.5 Indirect Inconsistent State Definition (IISD) . . . . 62
3.2.6 Anomalous Construction Behavior(1) (ACB1) . . . . 63
3.2.7 Anomalous construction behavior(2) (ACB2) . . . . 64
3.2.8 Incomplete (failed) Construction (IC) . . . . 65
3.2.9 State Visibility Anomaly (SVA) . . . . 66
**3.3 Syntactic Patterns of Inheritance** . . . . 67
3.3.1 Descendant has No Methods (DNM) . . . . 69
3.3.2 Descendant introduces extension methods . . . . 70
3.3.3 Descendant introduces refining methods . . . . 77
3.3.4 Descendant Introduces Constructors . . . . 82
3.3.5 Special cases – Complete Behavioral Redefinition . . . . 85
**3.4 Discussion** . . . . 89

**CHAPTER 4 Coupling-based Analysis of Object-Oriented Programs** . . . . 91

**4.1 Extended Coupling Definitions** . . . . 91
**4.2 Coupling Sequences** . . . . 93
4.2.1 Type I Coupling Sequences . . . . 94
4.2.2 Type II Coupling Sequences . . . . 95
4.2.3 Type III Coupling Sequences . . . . 96
4.2.4 Type IV Coupling Sequences . . . . 98
4.2.5 Other Type of Coupling Sequences . . . . 99
**4.3 Coupling Variables and Coupling Sets** . . . . 100
**4.4 Coupling Paths** . . . . 101
4.4.1 I-Def Paths . . . . 103
4.4.2 I-Use Paths . . . . 104
4.4.3 Transmission Paths . . . . 105
**4.5 The effects of inheritance and polymorphism on coupling** . . . . 106
**4.6 Polymorphic coupling sequences and coupling sets** . . . . 116
4.6.1 Polymorphic Coupling Sequences . . . . 116
4.6.2 Polymorphic Coupling Sets . . . . 119
**4.7 Coupling paths in object-oriented programs** . . . . 120
4.7.1 Non-Polymorphic Coupling Paths . . . . 120
4.7.2 Polymorphic Coupling Paths . . . . 123
4.7.3 Feasible and infeasible coupling sequences . . . . 125
**4.8 Summary** . . . . 126

**CHAPTER 5 A Set of Criteria for Testing Object-Oriented Programs** . . . . 128

**5.1 Coupling Criteria** . . . . 129
5.1.1 Definitions . . . . 129

5.1.2 All-Coupling-Sequences . . . 132
5.1.3 All-Poly-Classes . . . 133
5.1.4 All-Coupling-Defs/Some-Coupling-Uses . . . 134
5.1.5 All-Coupling-Uses/Some-Coupling-Defs . . . 135
5.1.6 All-Coupling-Defs-Uses . . . 136
5.1.7 All-Poly-Coupling-Defs-Uses . . . 137
**5.2 Generation of Test Requirements** . . . 137

**CHAPTER 6 Analyzing Coupling Properties of Object-oriented Programs** . . . 140

**6.1 Definitions** . . . 140
**6.2 Identifying Coupling Sequences** . . . 142
**6.3 Identifying Coupling Sets** . . . 143
**6.4 Instrumenting OO Programs for Coupling Analysis** . . . 144
6.4.1 Coverage Mappings . . . 144
6.4.2 Instrumentation Requirements . . . 146
**6.5 Instrumenting Java Programs** . . . 147
6.5.1 Instrumentation Instructions . . . 148
6.5.2 An example . . . 150
**6.6 Summary** . . . 154

**CHAPTER 7 CBAT - Coupling-based Analysis Tool** . . . 156

**7.1 Objectives of CBAT** . . . 156
**7.2 Representations provided by CBAT** . . . 157
7.2.1 Class Graph . . . 157
7.2.2 Abstract Syntax Tree . . . 158
**7.3 Architecture of CBAT** . . . 169
7.3.1 CBAT Core . . . 169
7.3.2 Analysis Engine . . . 171
7.3.3 Instrumentation Engine . . . 171
**7.4 Implementation** . . . 172

**CHAPTER 8 Validation** . . . 174

**8.1 Experimental design** . . . 174
8.1.1 Subject programs . . . 174
8.1.2 Test adequacy criteria . . . 175
8.1.3 Test data . . . 176
8.1.4 Injected Faults . . . 177
**8.2 Conduct of Experiments** . . . 178
8.2.1 Test oracle derivation . . . 179
8.2.2 Fault injection . . . 180
8.2.3 Test execution . . . 181
8.2.4 Result evaluation . . . 181
**8.3 Results** . . . 182

**8.4 Analysis and Discussion** . . . 183
8.4.1 Analysis of the coupling-based criteria . . . 190
8.4.2 Explanation of effects . . . 191
8.4.3 Effectiveness of the Coupling-based Criteria . . . 192
8.4.4 Discussion . . . 196
**8.5 Conclusion** . . . 197

**CHAPTER 9 Contributions and Future Work** . . . 199

**9.1 Contributions** . . . 199
**9.2 Future Work** . . . 201
9.2.1 Testing inter-method coupling sequences . . . 201
9.2.2 Specification and coupling-based testing of object-oriented programs. . . 202
9.2.3 Integration testing within class hierarchies . . . 202
9.2.4 Coupling-based testing of concurrent object-oriented programs . . . 203
9.2.5 Testing of reflective object-oriented programs . . . 203
9.2.6 Generation of test cases for coupling-based testing . . . 204
9.2.7 Metrics for coupling-based testing . . . 204
9.2.8 Mutation testing of object-oriented programs . . . 204
9.2.9 CBAT Enhancements . . . 206
9.2.10 XML-based program representations for testing and analysis . . . 207
9.2.11 Reverse engineering of software contracts . . . 208

**References** . . . 177

## List of Tables

Table Page
3-1 Faults and Anomalies due to Inheritance and Polymorphism........................56
3-2 Syntactic Patterns of Inheritance...................................................................68
3-3 Fault/anomaly types manifested by syntactic patterns..................................88
4-1 Summary of sample coupling paths ...........................................................113
4-2 Binding triples for ........................................................................................118
4-3 Sample Coupling Paths ................................................................................122
4-4 Polymorphic coupling paths for type family A...........................................124
6-1 Coverage Mappings......................................................................................145
6-2 Instrumentation instructions.........................................................................147
6-3 Java instrumentation methods ......................................................................148
8-1 Subject program characteristics....................................................................175
8-2 Number of test cases per subject program and criterion .............................176
8-3 Number of faults injected into method under test .......................................178
8-4 Experimental Results....................................................................................183
8-5 Results of hypothesis tests............................................................................195

## List of Figures


Figure Page

1-1 Sample aggregation example ........................................................................7
1-2 Sample Inheritance Hierarchy ......................................................................8
1-3 Sample Class Hierarchy ..............................................................................10
1-4 Example class hierarchy with table of definitions and uses ........................11
1-5 Def-use pairs resulting from A::h preceding A::i and A::j .........................12
1-6 Different def sets between A::h and B::h ....................................................13
1-7 Data flow anomaly due to A::h preceeding A::i .........................................14
1-8 d called through instance of A ....................................................................15
1-9 d called through instance of B ....................................................................16
1-10 Data flow anomaly for A::y with respect to A::u and A::j with respect to A::w ..16
1-11 d called through instance of C ....................................................................17
1-12 Data flow anomaly for A::i with respect to A::u and C::l with respect to A::v ....18
2-1 Chen and Kao's shape example [20] ..........................................................39
2-2 Procedural coupling criteria ........................................................................48
3-1 Class hierarchy with refining and extension methods ................................51
3-2 Example hierarchy ......................................................................................54
3-3 Descendant with no overriding methods .....................................................58
3-4 Code example showing inconsistent type usage .........................................58
3-5 State Definition Anomalies ..........................................................................60
3-6 Example of Indirect Inconsistent State Definition (IISD) ...........................63
3-7 Example of Anomalous Construction Behavior ..........................................64
3-8 Incomplete construction of state variable fd ................................................66
3-9 State Visibility Anomaly .............................................................................67
3-10 Descendant whose definitions include no methods or state variables .........70
3-11 Example showing interaction of extension methods ...................................71
3-12 Definitions and uses for extensions methods ...............................................72
3-13 Code fragment for method Submarine::evade .............................................74
3-14 Code fragment for method Submarine::blowBallast ...................................77
3-15 Code fragment for method Submarine::submerge .......................................78
3-16 Code for Submarine::accelerate illustrating RUIV, RCOM, and RDIV ......80
3-17 Complete Behavioral Redefiniditon (2) .......................................................87
3-18 Yo-yo effect resulting from extension method calling inherited method .....90
4-1 Type I Coupling Sequence ..........................................................................94
4-2 Type II Coupling Sequence .........................................................................96
4-3 Type III Coupling Sequence ........................................................................98

4-4 Type IV Coupling Sequence ....................99
4-5 Inter-method Coupling Sequences ....................100
4-6 Detailed Type I Coupling Sequence ....................104
4-7 Sample class hierarchy and def-use table ....................107
4-8 Coupling sequence when o is bound to an instance of A ....................108
4-9 Coupling sequence when o is bound to an instance of B ....................110
4-10 Coupling sequences when o is bound to an instance of C ....................111
4-11 Sample hierarchy with class D added ....................112
4-12 Sample hierarchy showing modified class C ....................114
4-13 Coupling sequence where o is bound to C and ....................115
5-1 Hierarchy of coupling-based testing criteria ....................130
6-1 Java mechanism for collecting coupling-based coverage data ....................149
6-2 Sample hierarchy ....................150
6-3 Method Client.f (without instrumentation) and corresponding CFG ....................151
6-4 Method Client.f instrumented for coupling coverage ....................154
7-1 UML Class Diagram for package ClassGraph ....................158
7-2 UML class diagram for ControlTree ....................160
7-3 UML Class Diagram for AST expression tree ....................161
7-4 Class Diagram for package MethodGraph. ....................164
7-5 CBAT architecture ....................170
7-6 CBAT Instrumenter ....................172
8-1 Class hierarchy with seeded shadow hierarchy for All-Poly-Classes ....................180
8-2 Class hierarchy with seeded shadow types for All-Coupling-Defs-Uses ....................181
8-3 Average detection effectiveness by fault type ....................184
8-4 Average effectiveness for program P1 across all subject criteria ....................185
8-6 Average effectiveness for program P3 across all subject criteria ....................186
8-5 Average effectiveness for program P2 across all subject criteria ....................186
8-8 Average Effectiveness for Program P5 across all Subject Criteria ....................187
8-7 Average effectiveness for program P4 across all subject criteria ....................187
8-10 Average effectiveness for program P7 across all subject criteria ....................188
8-9 Average effectiveness for program P6 across all subject criteria ....................188
8-12 Average effectiveness for program P9 across all subject criteria ....................189
8-11 Average effectiveness for program P8 across all subject criteria ....................189
8-13 Average effectiveness for program P10 across all subject criteria ....................190
8-14 Observed versus fitted cell frequencies ....................194

## ABSTRACT

TESTING THE POLYMORPHIC RELATIONSHIPS
OF OBJECT-ORIENTED PROGRAMS

Roger T. Alexander, MS

George Mason University, 2001

Dissertation Director: A. Jefferson Offutt

The emphasis in object-oriented programs is on defining abstractions that have both state and behavior. This emphasis causes a shift in focus from software units to the way software components are connected. Thus, we are finding that we need less emphasis on unit testing and more on integration testing. The compositional relationships of inheritance and aggregation, especially when combined with polymorphism, introduce new kinds of integration faults. The research presented in this thesis is based on the key insight that the primary mechanism for integration is coupling between components. Previous research demonstrated that coupling-based testing techniques can be used for integrating components in procedural programs. This thesis extends previous work to account for the compositional relationships found in object-oriented programs. The key contributions include a formalism for representing the state interactions and behavior that result from method calls where inheritance and polymorphism are a factor, a set of test-adequacy criteria that are effective at detecting faults that are peculiar to object-oriented programs, a technique for identifying data flow anomalies within inheritance hierarchies, a model of the faults that can occur in object-oriented programs, and a graphical model for analyzing and understanding the effects of polymorphism within a class hierarchy. This research has also produced a proof of concept tool that demonstrates the practicality and effectiveness of the coupling-based analysis and testing techniques.

# 1. Introduction and Overview

The emphasis in object-oriented languages is on defining abstractions (e.g. abstract data types) that model concepts relative to some problem and solution domain [52]. 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 can increase. This is due to the inherent complexity in the nature of the relationships found in object-oriented languages [15]. 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 object-oriented languages [11].

The research presented in this thesis is the result of an investigation in the area of testing these abstractions when inheritnace and polymorphaism are used. Emphasis is placed on testing the state space interactions between methods for a given class when invoked from another method in a different class. The following sections describe the problem and present the motivation, present the thesis statement, and provide background concepts on object-oriented programming.

## 1.1 Motivation

In 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-

---

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

return, or through containment when one module includes another. This is also true for 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 [50]. 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 based on. Together, inheritance and polymorphism are the key characteristics that distinguish an object-oriented language from an object-based language [50, 52].

Another difference is the effect that polymorphism (dynamic binding) has on the integration of components when inheritance is a factor. Any object type $T$ defines a new type family. Members of that family include $T$ and any of its descendants. Polymorphism allows the type of an object to determine which version of a method executes [52]. As a result of polymorphism, any *instance* of a descendant type can be freely substituted for an instance of $T$.[1] 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. Thus, inheritance combined with polymorphism provides two forms of integration that must be dealt with when testing objects, *integration of representation* and *integration of abstraction*. Neither of these has a procedure-oriented counterpart.

Integration of representation addresses the issues associated with combining the state space representation of existing classes to form the representation for a new classes. This is accomplished through inheritance whereby the state space of a parent class becomes part of the state space of the child. Properties and behaviors that are inherited, along with state-

---

1. The terms *instance* and *object* are synonymous, and used interchangeably throughout this thesis.

space definitions, must be carefully combined with new and overriding methods to ensure consistency in behavior and state among the related classes.

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 also is a dynamic aspect due to dynamic binding resulting 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 consistent behavior with respect to the aggregated abstraction.

As an example, an implementation of a *Stack* could use a class that is sequential data structure, such as an *Array* or *LinkedList*, to hold its contents. While these two data structures can have the same sequential behavior, they are likely to be vastly different state representation. The *Array* could be a contiguous area of memory index by a pointer, whereas the *LinkedList* would consist of a series of nodes that mutually reference one another. If both of the classes for *Array* and *LinkedList* share a common parent, say *Sequence*, then *Stack* can use *Sequence* as its state space representation. Since *Array* and *LinkedList* are children of *Sequence*, either can be used to provide the behavior required by *Stack*. To ensure that *Stack* behaves correctly, it must be tested across the extension of the type family represented by *Sequence*, *Array* and *LinkedList* in this case.

## 1.2 Problem Statement

*The problem that this thesis addresses 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 [17, 27]. Simply put, there are many more different types of places where faults can hide than in traditional procedure-oriented programs.

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 [59], usually focus on the functional behavior of a system without regard for its structural characteristics. These techniques treat a system as a black-box and try to determine if a given system exhibits the required functional behavior according to some specification. Test inputs are chosen without regard for the structure of the system. Consequently, it may not be possible to know 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 [22, 54, 55]. 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 [9], are effective at testing certain aspects of object-oriented programs [25]. They are just as applicable to methods as they are to procedures and functions, though their overall impact on quality is lessened.

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 covered when the procedure is executed using inputs drawn from the test suite [9]. 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 [15]. 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 [30]. These 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 [9, 30]. While there is evidence to suggest that these techniques are useful for procedure-oriented programs [29], 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 [12]. However, the work of Harrold and Soffa [34] and also of Jin and Offutt [38] has shown data flow analysis to be an effective technique for testing the integration of procedure-oriented components. It seems 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 and ranges, and do they observe the proper relationships? Does the called method actually return the proper type and is the value in 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 [34]. That is, determining whether or not that the data passed between components is used in a consistent manner.

## 1.3 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.*

## 1.4 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 texts such as Meyer's [52] for a thorough treatment of object-oriented concepts.

### 1.4.1 Classes

Object-Oriented Programming is an approach to the development of software that is based on the concepts of information hiding and encapsulation [52]. 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* [52]. Thus, a class defines a type that all of its objects share.

Object-Oriented Programming is an approach to the development of software that is based on the concepts of information hiding and encapsulation [52, 62]. The fundamental build-

ing 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* [52]. Thus, a class defines a type that all of its objects share.

### 1.4.2 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 1-1 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*.

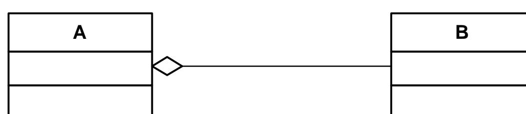



**Figure 1-1.** Sample aggregation example

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 1-2 where the arrow points to the ancestor class.

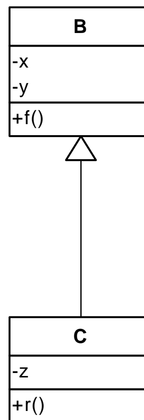



**Figure 1-2.** Sample Inheritance Hierarchy

### 1.4.3 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 [52]. As an example, consider the UML class diagram shown in Figure 1-3 and the following code fragment that provides an implementation of method *p* specified by class *Q*:

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

Line 4 contains a call site in which the method *m* is invoked against the instance bound to the object reference *w*. The implementation of *m* that actually executes depends on the type of the instance that is bound to *w*. 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 1-3. In an object-oriented language such as C++ and Java, variables that reference objects have a static type, which is the type they are declared as, and a dynamic type that is determined at runtime. The dynamic type, or actual type, is the type of the instance that is actually bound to the variable. This can be an instance of any member of the type family defined by the declared type of the variable. 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 be *X::m*.[1] 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*.

1. The notation used here is borrowed from C++ where the scope resolution operator :: is used to identify the namespace that a particular identifier is a member of.

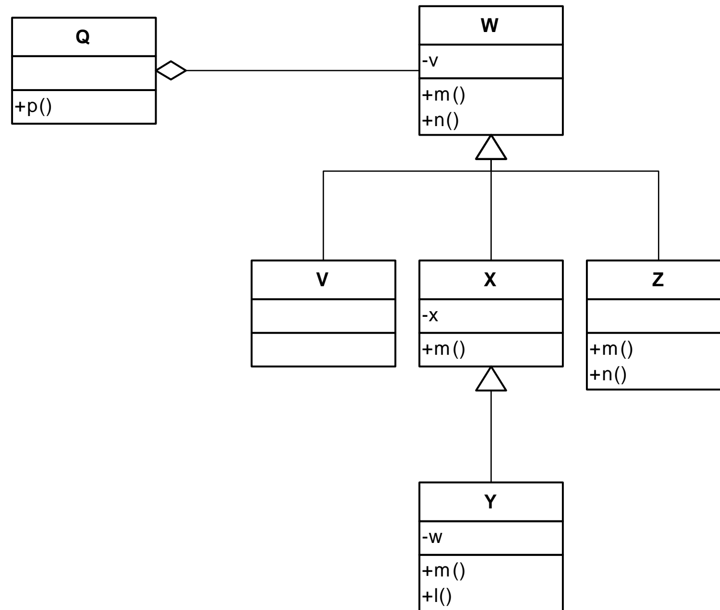
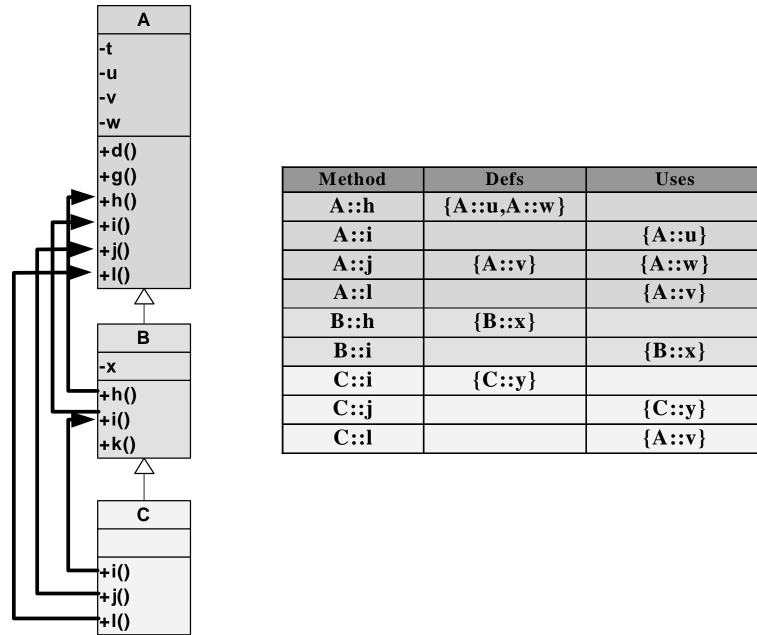



**Figure 1-3.** Sample Class Hierarchy

## 1.5 Problems with method overriding and polymorphism

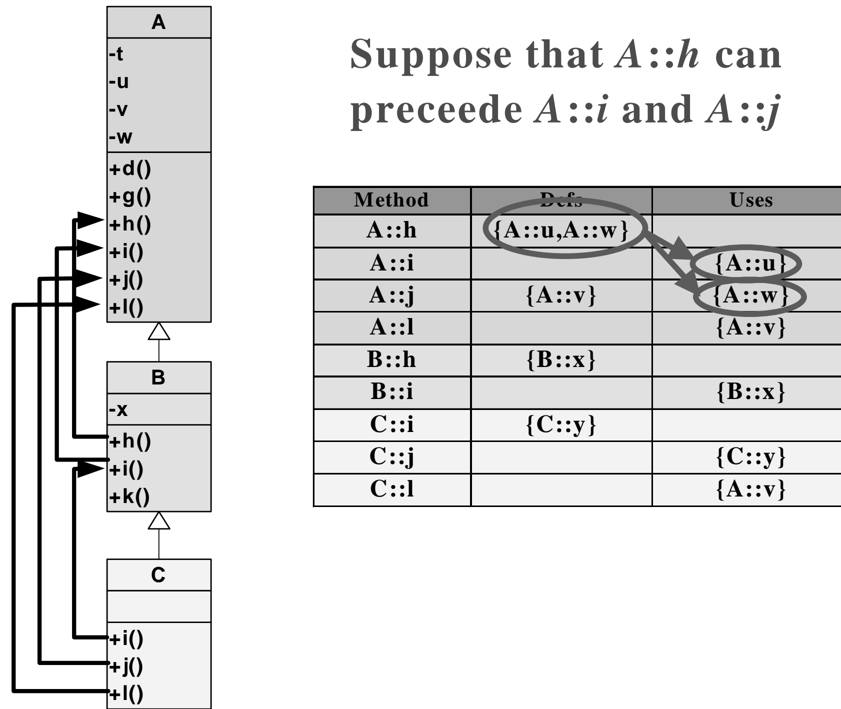
To illustrate the problems that method overriding and polymorphism, consider the simple inheritance hierarchy that is three classes deep, shows on the left of Figure 1-4. The root class *A* contains four state variables and six methods. Its direct descendant *B* specifies one state variable and three methods. Finally, class *C* specifies only three methods. You can easily see which classes have methods that override inherited methods, such as *B*::*h* overrides *A*::*h*. The table to the right of Figure 1-4 shows the state variable definitions and uses of some of the methods for each class in the hierarchy. The problem begins with the seemingly innocuous call to *A*::*d* through the instance provided by some context variable. This seemingly trivial example has some very complex interactions that potentially yield nasty problems.




| Method | Defs | Uses |
|---|---|---|
| A::h | {A::u,A::w} | |
| A::i | | {A::u} |
| A::j | {A::v} | {A::w} |
| A::l | | {A::v} |
| B::h | {B::x} | |
| B::i | | {B::x} |
| C::i | {C::y} | |
| C::j | | {C::y} |
| C::l | | {A::v} |

**Figure 1-4.** Example class hierarchy with table of definitions and uses

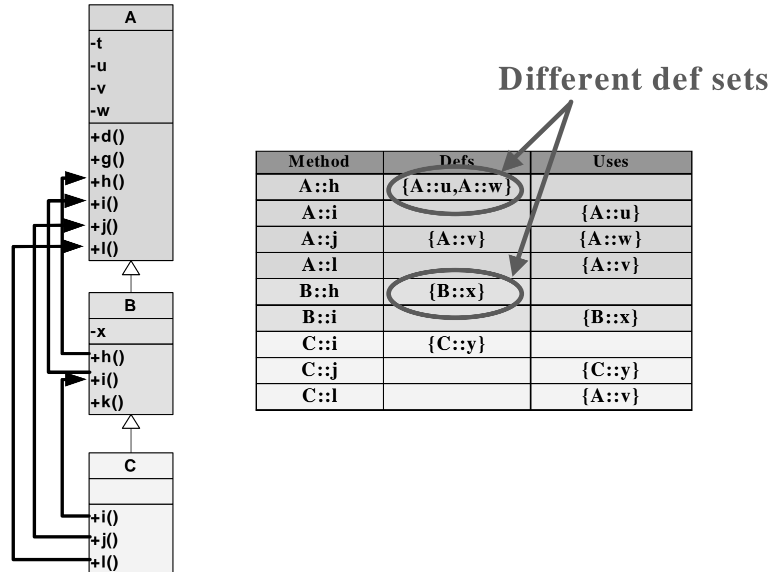
Suppose that an instance of *A* is bound to *o* and that a call to method *A*::*h* precedes calls to *A*::*i* and *A*::*j*. The def-use pairs that can occur for this sequence is highlighted in Figure 1-5. Note that the definitions of *A*::*u* and *A*::*w* by *A*::*h* are used by *A*::*i* and *A*::*j*. From a data flow perspective, this not an anomaly since *A*::*u* and *A*::*w* were defined before they were used.




| Method | Defs | Uses |
|---|---|---|
| A::h | {A::u,A::w} | |
| A::i | | {A::u} |
| A::j | {A::v} | {A::w} |
| A::l | | {A::v} |
| B::h | {B::x} | |
| B::i | | {B::x} |
| C::i | {C::y} | |
| C::j | | {C::y} |
| C::l | | {A::v} |

**Figure 1-5.** Def-use pairs resulting from *A*::*h* preceding *A*::*i* and *A*::*j*

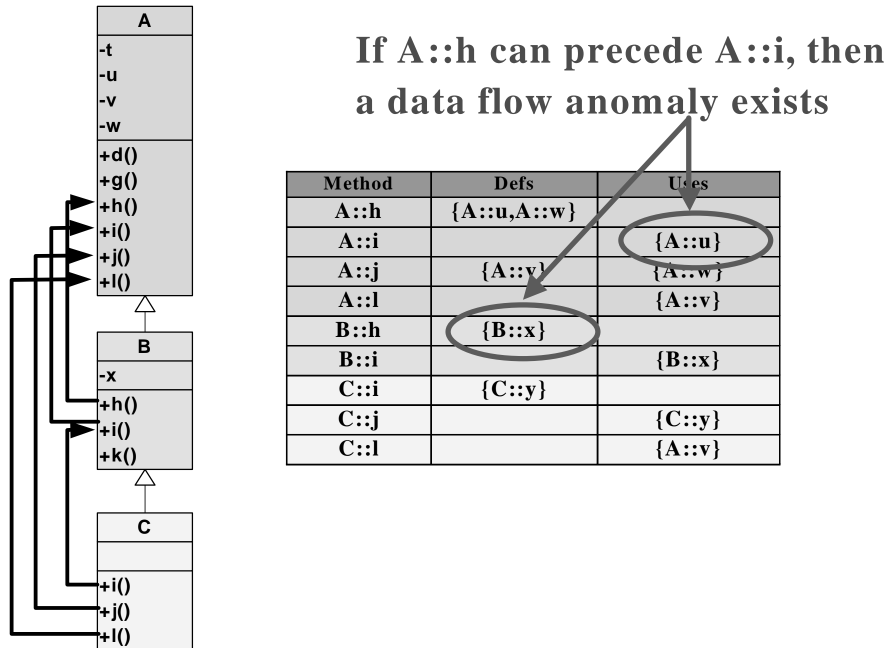
Now suppose that an instance of *B* is used in place of the instance for *A*. Examining the definition-use table in Figure 1-6 immediately reveals that overriding method *B*::*h* has a different def set than *A*::*h* does. In particular, there are state variables that *A*::*h* defines that *B*::*h* does not (i.e. *A*::*u* and *A*::*w*).




| Method | Defs | Uses |
|---|---|---|
| A::h | {A::u,A::w} | |
| A::i | | {A::u} |
| A::j | {A::v} | {A::w} |
| A::l | | {A::v} |
| B::h | {B::x} | |
| B::i | | {B::x} |
| C::i | {C::y} | |
| C::j | | {C::y} |
| C::l | | {A::v} |

**Figure 1-6.** Different def sets between *A*::*h* and *B*::*h*

The situation depicted in Figure 1-6 is not necessarily a problem. It depends on whether any method called that uses one of the variables defined by *A*::*h* not defined by *B*::*h* (e.g. *A*::*j*). Note that the sequence *B*::*h* followed by *B*::*i* is safe from a data flow perspective since the former defines *B*::*x* and the latter uses it. Thus, if this sequence of method calls were made in the context expecting an *A* but through an instance of *B*, no data flow anomally would exist. However, if the call to *A*::*h* were followed by a call to *A*::*j*, an anomaly would exist since *A*::*w* is used by *A*::*j* but was not defined because *B*::*h* was executed instead of *A*::*h*. This is indicated in Figure 1-7.




| Method | Defs | Uses |
|---|---|---|
| A::h | {A::u,A::w} | |
| A::i | | {A::u} |
| A::j | {A::v} | {A::w} |
| A::l | | {A::v} |
| B::h | {B::x} | |
| B::i | | {B::x} |
| C::i | {C::y} | |
| C::j | | {C::y} |
| C::l | | {A::v} |

**Figure 1-7.** Data flow anomaly due to *A*::*h* preceeding *A*::*i*

To see the effects that polymorphism can have on method overriding, consider the following code fragment that makes use of the same hierarchy:

```
f( A o )
{
   ...
   o.d();
   ...
}
```

Here, method *f* has a formal argument *o* whose declared type is *A*. Because the type family associted with *A* includes classes *B* and *C*, *o* can be bound to an instance of any of these. Figure 1-8 depicts a a stylized graph, called a yo-yo graph, that depicts the flow of control that results from a call to *A*::*d* when *o* is bound to an instance of *A*. As we see here, *d* calls

*g*, which in turn calls *h*, *h* calls *i*, and *i* calls *j*. Presumably, this is the intent of the programmer of *A* and is correct with respect to the specification of *A*.

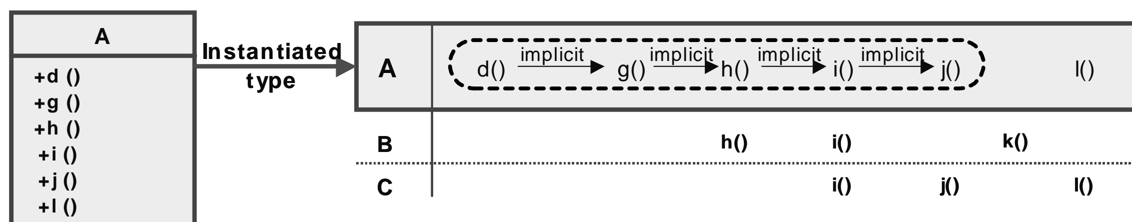



**Figure 1-8.** *d* called through instance of *A*

Now suppose that *o* is bound to an instance of *B*. The call *o.d* results in the execution sequence beginning with of *A*::*d* shown in Figure 1-9. As before. *A*::*d* then calls *A*::*g*, and *A*::*g* then apparently calls *A*::*h*. But, because *d* was called in the context of an instance of *B*, *B*::*h* executes instead. As we see, *B*::*h* calls *B*::*i*, which in turn makes an explicit call to *A*::*i*., and this the flow of control returns to purview of *A*'s implementation. On the surface, this execution path looks innocuous enough. Unfortunately, this is not the case.

As Figure 1-10 indicates, there is a data flow anomaly in method *A*::*i* with respect to state variable *A*::*u*, and method *A*::*j* with respect to state variable *A*::*w*. *A*::*i* expects that state variable *A*::*u* to have been defined prior to its invocation, and *A*::*j* has a similar expectation for state variable *A*::*w*, and indeed this is what the programmer of *A* has done. However, through means beyond the control of *A*'s programmer, the flow of control and the pattern of state definitions has been altered, with result being a violation of one of the assumptions made in the implementation of *A*.

**Figure 1-9.** *d* called through instance of *B*

• *A::i* with respect to *A::u*
• *A::j* with respect to *A::w*

Data Flow Anomaly!

**Figure 1-10.** Data flow anomaly for *A*::*y* with respect to *A*::*u* and *A*::*j* with respect to *A*::*w*

Now consider what happens when *d* is called through an instance of *C,* as depicted in Figure 1-11. As is obvious. a much more complex pattern of method calls results (and presumably state interactions as well). The first three method invocations are consisted with the implementation of *B* (though not with *A*). But the fourth method invocation, which is apparently to *B*::*i*, actually results in the execution of *C*::*i*. Now *C*::*i* calls *B*::i, which in turn

calls $A::i$ as before (the implementation of $B$ has not changed). Now, $A::i$ makes its apparent call to $A::j$. This time, control returns to $C$ with the execution of $C::j$, and so on. Clearly this is a complicated situation and not without its problems. Figure 1-12 shows that two data flow anomalies result in method $A::i$ with respect to $A::u$ and method $C::l$ with respect to $A::v$.

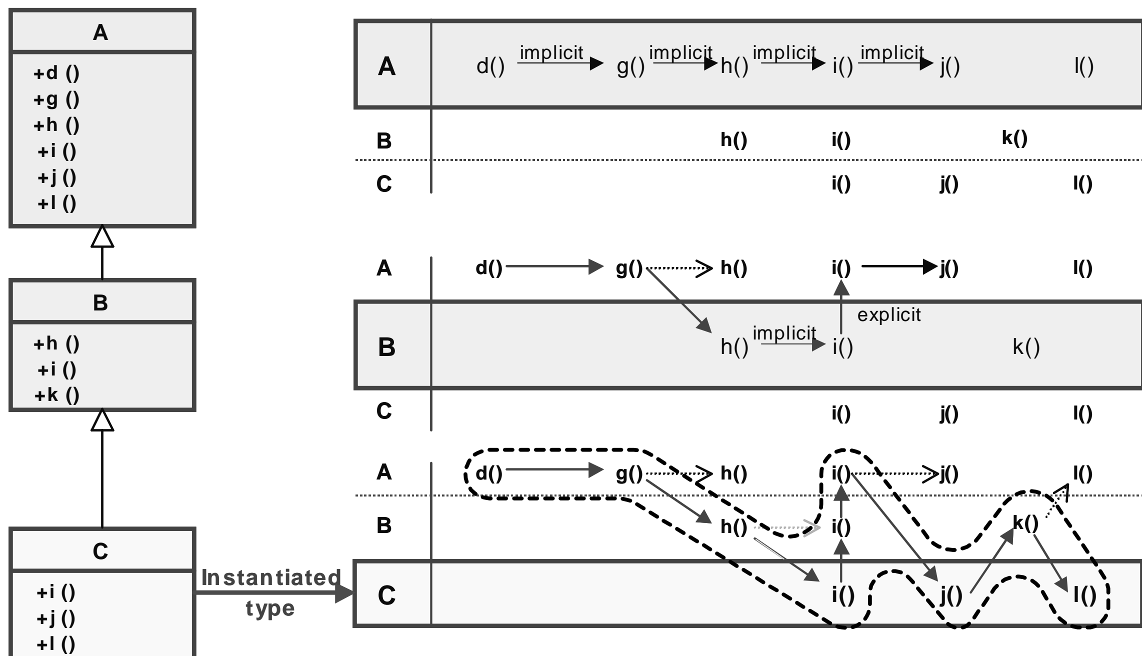



**Figure 1-11.** *d* called through instance of *C*

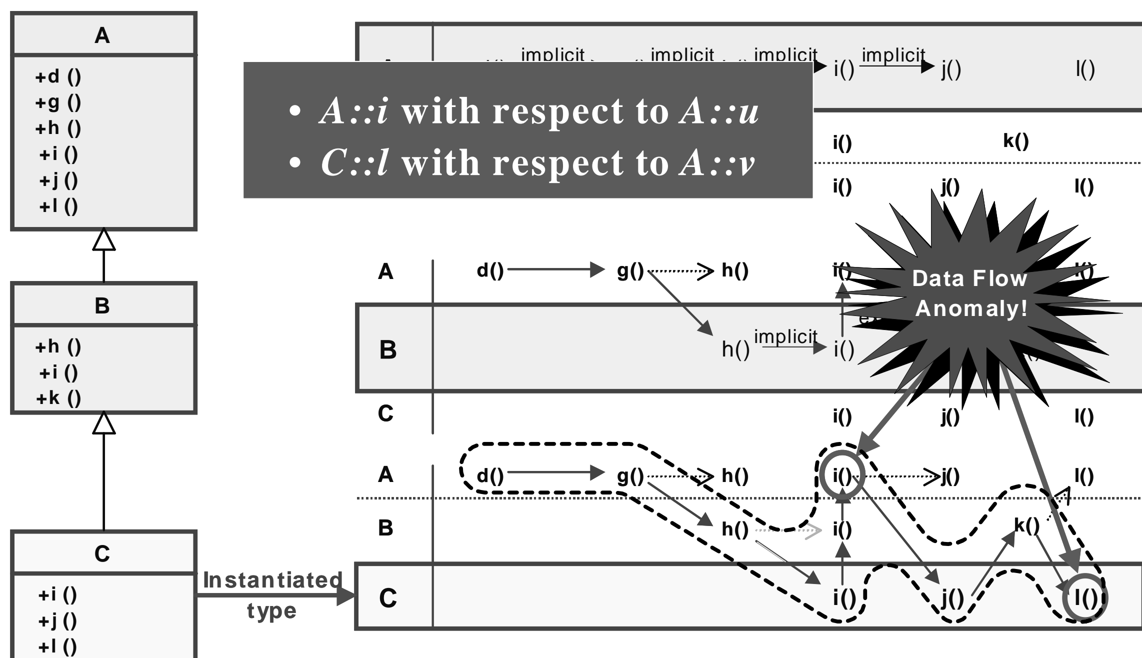



**Figure 1-12.** Data flow anomaly for *A*::*i* with respect to *A*::*u* and *C*::*l* with respect to *A*::*v*

This example has illustrated some the complexities that can result in object-oriented programs due to method overriding and polymorphism. Along with the this induced complexity comes more difficulty and required effort in testing.

# 1.6 Organization of Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 reviews background material and related work. This includes issues in testing object-oriented software, concepts of test adequacy, and other testing approaches related to object-oriented software. Chapter 3 presents a discussion of faults that are peculiar to object-oriented programs as a result of inheritance and polymorphism. Chapter 4 presents the concepts and theory of coupling-based testing applied to object-oriented programs. Chapter 5 continues with a presentation of the coupling-based test adequacy criteria for object-oriented programs. Chapter 6 discuss algorithms for analyzing object-oriented program to identify coupling sequences. Also included is a discussion on program instrumentation techniques required to collect cou-

pling-based coverage information. Chapter 7 presents CBAT, the proof of concept tool developed to validate the research presented in this thesis. Chapter 8 describes the efforts taken to validate this research. This includes a discussion of the experimental design, the raw results, and their significance. Finally, Chapter 9 summarizes the contributions of this research and presents a discussion of future work.

# 2. Background and Related Work

A number of areas are related to the research described in this dissertation. The sections that follow discuss these areas in detail. The first section 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.

## 2.1 Issues in Testing Object-oriented Software

A number of issues associated with object-oriented software that are not relevant in systems written using procedure-oriented languages. Many researchers have made the assertion that not all forms of traditional testing techniques are applicable or effective in testing object-oriented software [35] [10] [28]. The semantics of classes are embodied in their methods and in the representations chosen for their state. 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 [15]. 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 unit 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 [15].

Strong encapsulation reduces or often eliminates our ability to *observe* the state of an object [63] [12] [1] [8]. 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 [31]. This often makes it difficult to establish the necessary conditions for conducting tests (e.g. establishing initial state and determining final state), thus limiting our ability to achieve 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.[1] 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 achieve test adequacy [12]. Binder also observes that testing objects is problematic because they "...[often] exhibit sequentially dependent behavior" (i.e. the behavior and state of an object is a function of the history of its method invocation) [13]. 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 [68]. Instead, classes must be tested indirectly by testing their instances

1. In a personal communication with the author, Gail Kasier lamented the difficulty that she and DeWayne Perry had in publishing their paper on adequate testing of object-oriented programs because their claims and argument went against the commonly held beliefs of the time {Kaiser:1998:PC} {Perry:1990:ATO}.

(objects). Testing becomes a process of sampling from the class' population of instances. Unfortunately, this comes with 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 [26]. The approach was based on the use of a combination of black and white box testing techniques 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.

## 2.2 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 [64] who, drawing upon the earlier work of Weyuker [69], 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 models" [64]. The basis

for their conclusions rests upon Weyuker's axiomatic basis for determining test data adequacy [69]. 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" [64]. 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" [64]. 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" [64]. It may be the case that the component has code that was not reached during the test for the enclos-

ing 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" [64].

Intuitively, testing should be limited to just the modified class. 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" [64]. Changing a component should only require re-testing of the changed component and the other components that are dependent. Likewise, adding a new component should only require testing of the new component and re-testing of components that are dependent. 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 re-tested 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 [64].

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 [64].

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 [64].

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 [68]. 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 [18]. 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 shortsighted 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 override 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 [32]. 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 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 [64] [69]. Their decision of what to include dependends 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 (i.e. given a value) 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 [32].

- 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 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 [41]. This complexity results from the use of inheritance, aggregation and association relationships among classes. Deep inheritance hierarchies and highly nested class aggregations make it difficult to determine the optimal order in which classes should be tested. The consequences of testing in an order not optimal are that testing is not adequate because class relationships are missed, or substantial re-testing is often required. In an effort to eliminate this problem, the authors present an algorithm that generates the optimum order for unit and integration testing of classes. The objective is to minimize the amount of effort required to adequately test the classes by minimizing the number of test stubs that must be built, and to also reuse as many previously generated test cases as possible. The authors note that their work supplements that of Harrold, McGregor and Fitzpatrick [32].

The test order for a given class structure is based upon the dependencies that exist among its classes and is determined by analysis 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) [41]. 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.

## 2.3 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 [15].

### 2.3.1 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 [15]. 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 [66]. 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 intro-

ducing 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 [42]. 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 [21]. 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 $i$th element of the $k$-tuple corresponds to the state of the $i$th 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 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 [37] [36]. 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 [37] [36]. 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 and cannot be applied in general to object-oriented programs.

### 2.3.2 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 [15] [14]. 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 describe 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 con-

nected to the method's 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 [15] [14].

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 should be equivalent:

*push(1);push(4)lpush(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 [24]. 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 that given two objects $o_1$ and $o_2$ of the same class $C$, and two equivalent sequences of methods, $s_1$ and $s_2$, defined in $C$, the effect of executing $s_1$ on $o_1$ and $s_2$ on $o_2$ should leave $o_1$ and $o_2$ in the same states. If so, $C$ is deemed to be correct with respect to $s_1$ and $s_2$. The sequences $s_1$ and $s_2$ are equivalent since the resulting state of their executions are the same. Thus, $s_1$ and $s_1$ 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 [24]:

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 [19]. 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 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 $s_2$ [19]. 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 insight 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 [47]. 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 [45] [43]. 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 requirement that the inheritance relation be strict permits three implications to be inferred [46] [49]. 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 con-

current with the other substates of the inherited state. The third implication is that the child class may not delete inherited transitions. 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 [46]. 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.[1]
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 these criteria [46].

McGregor also presents an algorithm for constructing functional test cases that consist of a sequence of messages that cover a given class specification [47] [49]. The algorithm produces a set of test cases that satisfy the following requirements:

1. Test cases are constructed for all accessor methods.

---

1. McGregor does not provide a definition for a *convenience method*.

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 algorithm that McGregor describes satisfies the *all transitions* adequacy criterion [48]. 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) [47] [49].

## 2.4 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 Section 1.1 Section 1.4 of Chapter 1, 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 conducted that focuses on this area.

In his dissertation, Overbeck presents an approach that is based on testing contracts among client and server classes [60]. 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 [51]. 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 [60].

Chen and Kao present an approach for the integration testing of object-oriented programs [20]. Their approach, called *Object Flow Analysis*, utilizes data flow testing techniques to generate test requirements that require identification of all possible bindings and every definition-use of pair of every object. To do so, they collect and analyze information on both intra-procedural and inter-procedural def-use (DU) pairs. To identify inter-procedural DU pairs, they use the inter-procedural testing approach of Harrold and Soffa [34]. For identifying intra-procedural DU pairs, they introduce the *object control flow graph* (OCFG), which provides an abstraction for relating method calls and DU pairs both within and between methods.

The OCFG is a directed graph that consists of *super nodes* and edges. Super nodes represent methods in a class and are ordered pairs that consist of a set of nodes *N* and control edges *CE* from a particular method *M*. Nodes in *N* correspond to statements in *M* that either define

or use objects. A control edge ($n_i$, $n_j$) in *CE* indicates that node $n_j$ is reachable from node $n_i$ along some execution path. Edges between super nodes represent either message passing between methods, or definitions and uses between methods (a *method def-use edge*). Message passing edges indicates that method $sn_j$ is called from node $n_i$ (i.e. called from line *i*). A method def-use edge ($sn_i$, $sn_j$) represents a definition-use relationship between method $sn_i$ and $sn_j$ where $sn_j$ uses a data member that was defined by $sn_i$.

Chen and Kao use a graphic shape example, part of which is depicted by the code fragment in Figure 2-1 to demonstrate the application of their approach [20]. In the example, they report that definitions of the state variable *shape_obj* occur at lines 11, 18, and 20, and that uses occur at lines 9, 10, 11, and 12. These are indicated by labeled arrows in the figure. Note however, that Chen and Kao treat the binding of a variable to an instance and the definition of the state of the instance through use of the variable as the same. In their example, the former occurs at lines 18 and 20, whereas the former occurs at line 11. This clearly is a mistake since the binding of an instance applies only to a variable and not to the instance that the variable is bound to. Thus, the definitions of variable *shape_ob*j at lines 18 and 20 do not affect the state of any instance that is bound or that may be bound to this variable.

```
1    Class Frame {
2        Shape   shape_obj;
3    public:
4        int     draw_frame(Shape s);
5        Shape   set_obj();
6    }
7    Frame::draw_frame()
8    {
9       shape_obj.draw(); ◄──────────────────────────────── Use
10       if (shape_obj.get_param() > 10) ◄──────────────── Use
11           shape_obj.double(); ◄──────────────── Definition/Use
12       shape_obj.draw(); ◄────────────────────────────── Use
13   }
14   Shape Frame::set_obj(int shape,
15       int perimeter, int x, int y)
16   {
17       if (shape == 1)
18           shape_obj = Square(perimeter, x, y); ◄───── Definition
19       else
20           shape_obj = Circle(perimeter, x, y); ◄───── Definition
21       return shape_obj;
22   }
```

**Figure 2-1.** Chen and Kao's shape example [20]

As part of their work, Chen and Kao have defined two criteria for determining test adequacy for object-oriented programs. The first, *All-Bindings*, requires that "every possible binding of each object must be exercised at least once when the object is defined or used". The second, *All-du-Pairs*, requires that "every definition of every object to every use of that definition must be exercised under some test." Further, there must be a definition-clear path between the statements containing the definition and use of the object. They define a definition of an object as being a state initialization, the definition of a data member, or the invocation of a method that defines a state variable. They define an object use as the use of one of its data members in some computation, use of a method that uses a data member, or passing the object as an actual parameter in method call. While the first two of these make sense with respect to the traditional definition of a use, the third does not. In fact, it is more

the case that passing an object as an actual parameter to a method call results in the definition of the corresponding formal parameter. That is, the formal parameter becomes bound to the object. However, the state of the object does not change as a result of this binding, nor is its state used. It appears that Chen and Kao have overlooked this subtle distinction.

Of the related work surveyed in this chapter, the work of Chen and Kao is the closest to the research described in this thesis and their work was published at approximately the same time [4, 20]. Though their work is similar, there are significant differences. First, their criteria is coarse-grained. The criteria presented by Alexander and Offutt [4, 5], and described in detail by this thesis, are a superset of those of Chen and Kao. As Chen and Kao have defined them, their criteria require only that either all bindings or all DU pairs be covered. In particular, they do not integrate the two, though their complete example suggests that they are at least aware that this is important [20]. Secondly, not all bindings are feasible. Chen and Kao do not discriminate between feasible and infeasible bindings. Third, there likely will be DU pairs where every definition-clear path connecting them is infeasible. Thus, their criterion *All-du-Pairs* is impractical from an applied testing perspective.

In his dissertation, Orso presents a technique to the integration testing of object-oriented programs that is based on exercising polymorphic interactions among the different classes that comprise a system [58]. His technique has two steps. The first is concerned with identifying the *integration order* of the classes. A system of classes has many complex relationships that result from inheritance and aggregation. Thus, the order in which classes are integrated is important from an efficiency perspective. Orso's technique defines a total order on the set of classes and uses this to derive an integration order such that parent classes are always tested before their children, and every class is always integrated with the classes that it depends on. He derives this information by forming a graph representation of the system under test. Notes of the graph are classes, and the edges correspond to the relations among classes. Analyzing the graph results in an integration order for individual classes or clusters of classes.

After the integration order is chosen, a new data flow technique is used to select test cases that are adequate for testing combinations of polymorphic calls during the integration process. Orso extends the traditional definition and use sets with two new sets, *def-p* and *use-p*. These sets consist of information describing possible dynamic bindings resulting from polymorphism that are responsible for the definition or use of a given variable. He uses these sets to define new test adequacy criteria that are similar to the traditional data flow testing criteria of Rapps and Weyuker [65], but extended to account for these dynamic bindings.

Orso's technique is focused on the effects that polymorphism and dynamic binding have on the method under test. In particular, it focuses on testing critical combinations of bindings to variables that can affect the behavior of the method under test. While this approach does perform an integration test of a method with the possible bindings that can occur, it is only a one-way approach. In particular, it does not conduct an integration test of the effects that the method under test has on the instances that bound to the variables used by the method. This is a significant difference from the approach described in this thesis.

Jorgensen and Erickson describe an approach to integration testing that is similar in many respects to black-box testing techniques [39]. 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 Section 2.3.2) includes provisions for testing large clusters of classes by synthesizing a system level state machine (mode machine in Binder's terminology) [14]. 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.

## 2.5 Other Approaches of Testing Object-Oriented Software

Offutt and Irvine report on an experiment conducted to determine if the Category-Partition [59] [7] testing technique is effective when applied to object-oriented software [57]. In their experiment, they seeded 23 types of faults into two C++ programs. These faults were based on common programming mistakes reported by Meyer [53] 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 [33]. 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 [61] to the CCFG to compute definition-use pairs for each of the three types of analyses. Harrold and Rothermel do not describe how they apply this information in the testing procedure. Also, they only briefly discuss the application of their approach to inheritance relationships, but unfortunately, they do not provide any details.

## 2.6 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 [38].[1] 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 carred out in this dissertation, thus the are described in some detail.

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 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.

### 2.6.1 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 [38]. 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.

- ***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 $P_2$.

---

1. A call site is a location in a procedure where another procedure is invoked.

- ***Call*($m_1$,$m_2$,*call_site*,$x \rightarrow y$) : Boolean**: Evaluates to *true* if $P_1$ calls at *call_site* and actual parameter $x$ is mapped to formal parameter $y$.
- ***Return*($v$) :** The nodes that return values of $v$ to the calling unit.
- ***Start*($P$) :** The first node in $P$, $\boldsymbol{start}(\boldsymbol{P}) \in \boldsymbol{N_P}$.
- ***Def*($P$,$x$) :** The set of nodes in unit $P$ that contain a definition of $x$.
- ***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***:

$$ldbc\text{-}def(P_1, call_site, x) = \{i \in N_{P_1} \bullet x \in defs(i) \wedge def_clear_path(x, i, call_site)\}.$$

2. ***last-def-before-return*** **:**[1]

$$ldbc\text{-}def(P_2, y) = \{j \in N_{P_2} \bullet y \in defs(j) \wedge def_clear_path(y, j, Return(y))\}.$$

3. ***Shared-data-def*** :

$$shared\text{-}def(P_2, g) = \{i \in N_{P_2} \bullet i \in defs(P_3, g) \wedge nonlocal(P_3, g)\}.$$

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

- *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:

$$fac\text{-}use(P_1, call_site, x) = \{i \in N_{P_1} \bullet x \in uses(i) \wedge (def\text{-}path(P_1, call_site, x) = \emptyset)\}.$$

  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:

$$fic\text{-}use(P_2, y) = \left\{ j \in N_{P_2} \bullet (c\text{-}use(P_s, y, j) \vee i\text{-}use(P_s, y, i, j) \vee p\text{-}use(P_2, y, i, j)) \wedge (use\text{-}path(P_s, start(P_s), j, y) = \emptyset) \right\}.$$

  3. ***Shared-data-use*** **:** The set of nodes in a unit that have a use of a global variable. Formally:

$$shared\text{-}use(P_4, g) = \{i \in N_{P_4} \bullet use(P_4, g) \wedge nonlocal(P_4, g)\}.$$

### 2.6.2 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 {Jin:1999:CBC}. They define the following three types of coupling paths:

- ***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:

$$paremeter\text{-}coupling(P_1, P_2, call_site, x, y) = \left\{(i, j), i \in N_{P_1}, j \in N_{P_2} \middle| (i \in ldbc\text{-}def(P_1, call_site, x) \wedge j \in fic\text{-}use(P_2, y))\right\}.$$

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:

$$parameter\text{-}coupling(P_1, P_2, call_site, x, y) = \{(i, j), i \in N_{P_1}, j \in N_{P_2} | i \in ldbr\text{-}def(P_2, y) \wedge j \in fac\text{-}use(P_1, call_site, x)\}.$$

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

- ***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:

$$shared\text{-}data\text{-}coupling(P_1, P_2, g) = \{(i, j), i \in N_{P_1}, j \in N_{P_2} | i \in def(P_1, g) \wedge j \in use(P_2, g)\}.$$

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

### 2.6.3 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 [38]. These criteria, they claim, provide increasing amounts 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 cou-

pling 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* [65]. These criteria can be arranged in a hierarchy according to the subsumption relationships. Figure 2-2 shows the subsumption hierarchy defined by Jin and Offutt for the procedural coupling criteria [38]. As shown, *Call-coupling* is subsumed by *All-coupling-defs*, *All-coupling-defs* is subsumed by *All-coupling-uses*, and so on.

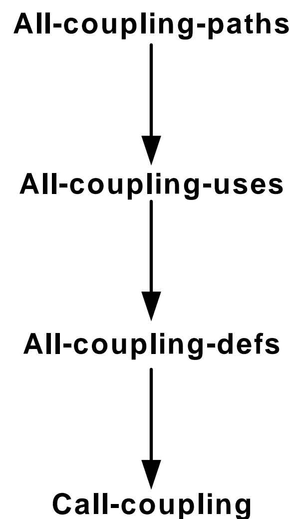



**Figure 2-2. Procedural coupling criteria**

### 2.6.4 Relationship to the Object-oriented Coupling-based Testing Criteria

Jin and Offutt's approach of coupling-based testing has proven effective to procedure-oriented programs [38]. The fundamental building block of object-oriented programs is the class. The building blocks for the class are state and behavior. State is manifested as a set

of (usually) encapsulated variables, and behavior as a collection of methods. Methods are the analogical equivalent to the procedures and functions found in traditional languages, such as C and Pascal. Within in methods, we find the same type of dataflow relationships (e.g. last definitions and first uses) that exist in their procedural counterparts. Thus, to the degree that these syntactic constructs are present, Jin and Offutt's approach still applies. However, there are additional types of data flow relationships resulting from inheritance and polymorphism that Jin and Offutt's approach is not applicable. The research described by this thesis extends the work of Jin and Offutt to account for those additional data flow relationships that result from inheritance and polymorphism.

# 3. Inheritance and Polymorphism Faults

Like their procedural counterparts, programs written in object-oriented languages have data flow anomalies and faults. Occasionally one of these faults manifests a failure, and corrective measures are then usually taken to eliminate the fault. Fortunately, many of the testing techniques and strategies for fault elimination are applicable to object-oriented programs, particularly in-so-far as the syntactic and semantic constructs found in procedure-oriented languages are also present in object-oriented languages. The power that inheritance and polymorphism brings to the expressiveness of programming languages also brings a number of new anomalies and fault types. Unfortunately, the techniques that we would use for eliminating faults in procedure-oriented programs as not applicable to those found in object-oriented programs.

In the discussion that follows, the term *refining method* is a synonym for *overriding method*. It is used here in place of the latter as it is more indicative of the view adopted by this thesis with respect to inheritance. In particular, the general view is that overriding methods *refine* the behavior of the method it overrides, as opposed to replacing its behavior with another. The phrase overriding method will be used generically when there is no need to distinguish between methods that provide a refinement in behavior as opposed to those that provide an extension to behavior.

To understand the difference between refining and extension methods, consider the class diagram shown in Figure 3-1. Class *Vehicle* defines methods *startEngine*, *stopEngine*, and *accelerate*. Derived from *Vehicle* is *Submersible*, which has methods *submerge*() and *surface*(). Both of these methods serve to extend the behavior of *Submersible* with respect to *Vehicle*, and are referred to as *extension methods*. They add behavior not present in the parent class. The figure also shows class *Submarine* as a descendant of *Submersible*, and

having methods *evade*(), *accelerate*(), and *submerge*(). Method *evade*() is also an extension method. In contrast, methods *accelerate*() and *submerge*() are refining methods in that they refine the behavior of *Submarine* by overriding methods *Vehicle*::*accelerate*() and *Submersible*::*submersible*(). The do not provide additional behavior as does *evade*(), but rather the provide a different implementation of the behavior defined and inherited from *Submarine*'s parents.

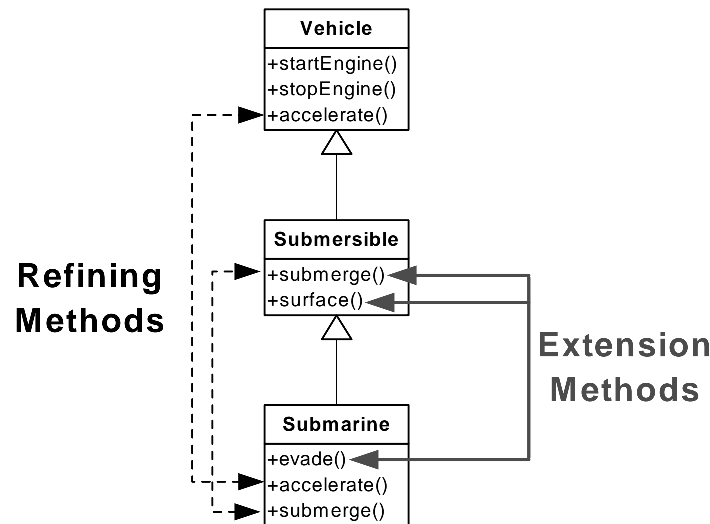



**Figure 3-1.** Class hierarchy with refining and extension methods

It is possible for a refining method to also add additional behavior (that is, to extend the behavior of the overridden method) not found in the overridden method. For example, the method *Submarine*::*accelerate*() may, in addition to increasing the speed of a submarine, retract the bow planes, perhaps to reduce drag. This extends the original behavior of *Vehicle* by adding an additional capability that is not possessed by all vehicles that accelerate.

The definition adopted in this thesis is that a refining method, regardless of what else it might do, is behaviorally compatible with the overridden method. Thus, *Submarine* also

accelerates in a manner consistent with *Vehicle* in addition to retracting its bow planes. This is consistent with the definition of inheritance that yields descendants that are sub-types of their ancestors, and thus instances of a descendant can safely be used wherever an instance of the ancestor is expected [45].

The following assumptions are made with respect to the discussion contained in this chapter:

- Unless otherwise noted, inherited state has sufficient visibility to allow direct reference by methods defined in descendant classes. Thus, in languages such as Java and C++, the access specifier associated with each state variable is not private.
- Inheritance results in classes that are subtypes of their parents, not sub-classing. The use of inheritance to create sub-classes easily results in classes that cannot safely be used where instances of parents are expected. Unfortunately, not all object-oriented languages have mechanisms to prevent a programmer from inadvertently using a subclass as subtype of its parent, which can easily lead to faults that are difficult to detect and diagnose. Further, a number of researchers consider this to be a bad programming practice and contrary to the engineering of high quality software [3, 44, 45, 52].
- The faults that we are concerned with in this thesis are dependent upon the syntactic constructs used to represent the semantics of classes (e.g. overriding methods directly defining inherited state variables, extension methods calling inherited methods, etc.).

- Methods may be overridden in a descendant class. That is, all the methods in the ancestor are polymorphic. Lessening this restriction simply means that some of the faults cannot occur.

- When considering the state effects of a particular method, the transitive closure of state definitions is assumed over called methods that are locally defined in a descendant class. Without loss of generality, we can ignore those non-public methods in a descendant that affect state and that are only called by other methods also defined in the descendant. This is safe to do since the state definitions made by those methods cannot be called by any method defined outside of the descendant, and considering them would add nothing to the result of the analysis presented in this thesis. Further, the effects of these methods is captured in the transitive closure mentioned above.

This chapter explores the anomalies and faults peculiar to inheritance and polymorphism and analyzes how they are manifested as a result of polymorphic behavior. The first section introduces an interpretation of the fault-failure model for object-oriented programs. The second then discusses the nature of the anomalies and faults found in object-oriented programs that result specifically from inheritance and polymorphism. The third section then proceeds to present and analyze the syntactic patterns used in defining descendant classes in terms of their ancestors. Finally, the fourth section discusses the added complexity that polymorphism brings to the process of software development and provides examples of the complex behavior patterns that can result.

## 3.1 A fault/failure model for polymorphic for object-oriented programs

The fault/failure model states that there are three conditions necessary for a failure to be observed [23, 54]. First, the location in the program containing the fault manifesting the

failure must be reached (*Reachability*). Second, after executing the location, an infection in the state of the program must occur (*Infection*). Third, the infected state must be propagated to the output of the program (*Propagation*).[1] Faults that result from polymorphic behavior must conform to this model, and a general failure model can be formulated in terms of this model. Figure 3-2 depicts a UML class diagram showing the inheritance hierarchy and client relationships that are described in the following subsections to describe the fault model for failures that result from the use of polymorphism.

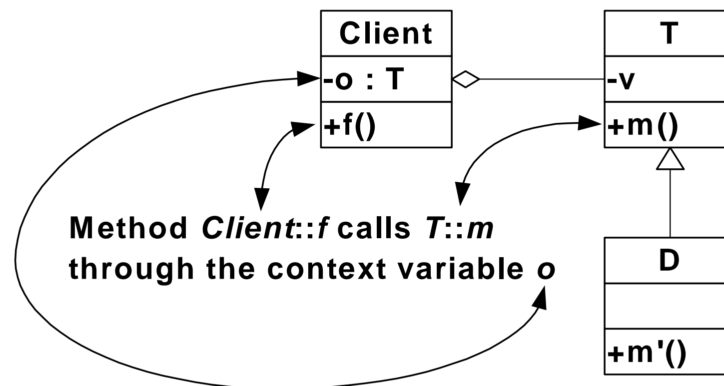



**Figure 3-2.** Example hierarchy

### 3.1.1 Reachability

1. There exists an inheritance hierarchy rooted at class *T* with a descendant *D*.
2. There is a variable *o* in a client *C* with a declared type *T*, and methods $m' \in D$ and $m \in T$ such that $m'$ overrides *m*.
3. The actual type of the instance bound to *o* is *D*.
4. *C* invokes *m* through the instance context provided by *o* (e.g. *o*.*m*()).

1. Morell used *Execution*, *Infection*, and *Propagation* [54]. Offutt used *Reachability*, *Sufficency*, and *Necessity* [23]. We choose to combine the two disparate sets of terms by using what we consider to be the most descriptive.

### 3.1.2 Infection

For a polymorphic fault to exist, $m$ and $m'$ must modify different portions of the state space of $T$. Note that $m'$ may define a variable with an incorrect value, but we do not consider this to be a polymorphic fault, but a "traditional" fault.[1] To model this situation, we want to compare the portion of the state that is declared by $T$. Any state added by $D$ is not relevant. Thus: $defs(m') \cup state(T) \neq defs(m)$. That is, there is some state variable $v$ such that $m'$ defines $v$ but $m$ does not, or $m$ defines $v$ but $m'$ does not.

### 3.1.3 Propagation

One of the variables defined by $m'$ or by $m$ (and not both) must be used. That is: $\exists\, n \in methods(T) \mid C\ calls\ o.n() \wedge \exists\, w \in state(T) \mid uses(n,w) \wedge ((w \in defs(m') \wedge w \notin defs(m)) \vee (w \notin defs(m') \wedge w \in defs(m))$. $C$ need not be the same client that called $o.m()$ earlier. The only requirement is that at some future point in time, $n$ is called in the context of the same instance that $m$ was called in.

## 3.2 Inheritance Faults and Anomalies

Inheritance affords creativity, efficiency, and reuse. Unfortunately, it also allows for a number of anomalies and potential faults that anecdotal evidence has shown to be some of the most difficult problems to detect, diagnose, and correct. This section examines several fault types manifested by polymorphism. Table 3-1 summarizes the set of fault types that result from inheritance and polymorphism. The goal is a complete list of faults, though we do not make this claim. Most of the cases are language-independent. In all cases, we are concerned with how each anomaly or fault is manifested through polymorphism in a context that uses an instance of the ancestor. Thus, we assume that instances of descendant classes can be substituted for instances of the ancestor.

---

1. A definition may be a direct through an assignment, as in $x = y$, or indirect through a method call whose effect is to change the state of the instance bound to the variable. Without loss of generality, the (conservative) view adopted in this chapter is that any such method call always results in a state change of the instance. However, by using static analysis techniques it is generally possible to identify those calls that actually have a definitional effect on state.

The following subsections explore a number of the anomalies that can lead to problems, and in some cases, to faults.

**Table 3-1.** Faults and Anomalies due to Inheritance and Polymorphism

| Acronynm | Fault/Anomaly | Section |
|---|---|---|
| ITU | Inconsistent Type Use (context swapping) | 3.2.1 |
| SDA | State Definition Anomaly (possible post-condition violation) | 3.2.2 |
| SDIH | State Definition Inconsistency (due to state variable hiding) | 3.2.3 |
| SDI | State Defined Incorrectly (possible post-condition violation) | 3.2.4 |
| IISD | Indirect Inconsistent State Definition | 3.2.5 |
| ACB1 | Anomalous Construction Behavior (1) | 3.2.6 |
| ACB2 | Anomalous Construction Behavior (2) | 3.2.7 |
| IC | Incomplete Construction | 3.2.8 |
| SVA | State Visibility Anomaly | 3.2.9 |

### 3.2.1 Inconsistent Type Use (ITU)

A descendant class does not override any inherited method. Thus, there can be no polymorphic behavior. Every instance of a descendant class *C* used where an instance of *T* is expected can only behave exactly like an instance of *T*. That is, only methods of *T* can be used. Any additional methods specified in C are hidden since the instance of *C* is being used as if it is an instance of *T*. However, anomalous behavior is still a possibility. If an instance of *C* is used in multiple contexts (i.e. through coercion, say first as a *T*, then as a *C*, then a *T* again), anomalous behavior can occur if *C* has extension methods. In this case, one or more of the extension methods can call a method of *T* or directly define a state variable inherited from *T*. Anomalous behavior will occur if either of these actions results in an inconsistent inherited state.

As an example, consider the class hierarchy shown in Figure 3-3.[1] Class *Vector* encapsulates a sequential data structure supporting direct access of its elements. Class *Stack* also

encapsulates a sequential data structure that has a "last-in/first-out" access policy. As shown, *Stack* uses methods inherited from *Vector* to implement its behavior. The top table summarizes the calls made by each method, and the bottom table summarizes the definitions and uses (represented as "d" and "u", respectively) of the state space of *Vector*.

The extension method *Stack::pop*() calls *Vector::removeElementAt*(), and extension method *Stack::push*() calls *Vector::insertElementAt*(). Clearly these two classes have different semantics. As long as an instance of *Stack* is used solely as an instance of *Stack*, there will be no behavioral problems. Alternatively, the *Stack* instance could be used solely as a instance of *Vector*, again without experiencing behavioral problems. However, if the usage of the instance is mixed between *Stack* and *Vector*, behavioral problems can occur.

The code fragment in Figure 3-4 illustrates how behavioral anomalies can occur when the type system is used to manipulate the manner in which instances of classes are used. For the method *f*, the instance bound to the formal argument *s* is used solely as a *Stack* in lines 3 through 10. However, at line 12, *s* is passed as an actual argument to method *g*, which expects an instance of *Vector*. There is no problem here in so far as the type system is concerned since an instance of *Stack* is also an instance of *Vector*. There is a potential behavioral problem that begins at line 23 where the last element of *s* is removed. The fault is manifested when control returns and reaches the first call to *Stack::pop*() at line 14. Here, the element removed from the stack is not the last element added.

---

1. This example is based on the library provided with the Java Development Kit version 1.2.

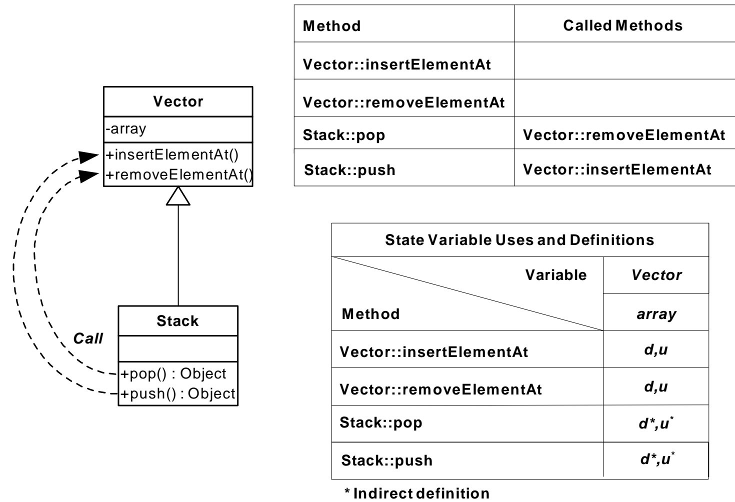



**Figure 3-3.** Descendant with no overriding methods

```
 1 public void f( Stack s )
 2 {
 3     String s1 = "s1";
 4     String s1 = "s2";
 5     String s1 = "s3";
 6      ...
 7
 8      s.push( s1 );
 9     s.push( s2 );
10     s.push( s3 );
11
12     g( s );
13
14     s.pop();
15     s.pop();
16     s.pop();   // Oops! The stack is empty!
17
18     ...
19 }

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

**Figure 3-4.** Code example showing inconsistent type usage

### 3.2.2 State Definition Anomaly (SDA)

In general, for a descendant and ancestor class to be behaviorally compatible, the state interactions of the descendant must be consistent with those of its ancestor. That is, the

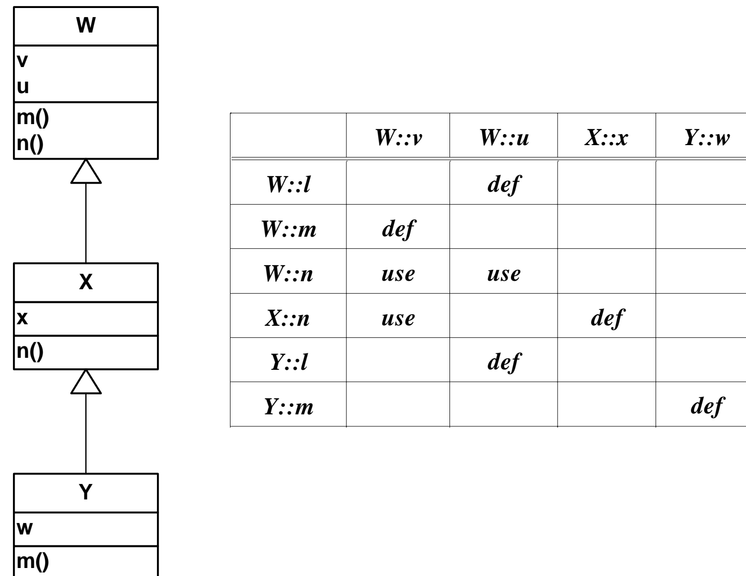
refining methods implemented in the descendant must leave the ancestor in the same (or equivalent) state as the ancestor's methods that are overridden. For this to be true, the refining methods provided by the descendant must yield the same net state interactions as each public method that is overridden. From a data flow perspective, this means that the net effect of the definitions made by a refining method against the set of inherited state variables from an ancestor class must *at least* provide the same definitions of the corresponding overridden method.[1,2] If this is not the case. then a potential data flow anomaly exists. Whether or not an anomaly actually occurs depends upon the sequences of methods that are valid with respect to the ancestor.

As an example, consider the class hierarchy and tables of definitions and uses shown in Figure 3-5. The parent of the hierarchy is class *W*, having descendants *X*, and *Y*. *W* defines methods *m*, and *n*, each having the definitions and uses shown in the table. Assume that a valid method sequence is *W::m*() followed by *W::n*(). As the table of definitions and uses shows, *W::m*() defines state variable *W::v* and *W::n*() uses it. Now consider class *X* and its refining method *X::n*(). As the table shows, it too uses state variable *W::v*, which is consistent with the overridden method and with the method sequence given above. Thus far, there is no inconsistency in how *X* interacts with the state of *W*. In fact, because a use can never affect future state-dependent behavior, *X::n*() could just as easily have used a different variable.[3]

1. This assumes that only a subset of the ancestor's methods are overridden by a descendant. If all of the methods are overridden, then the descendant has more flexibility in what is or is not defined, subject to the restriction that the externally observed behavior remains consistent with the ancestor's behavior.

2. *Net effect* refers to all of the state interactions that occur as a result of execution of an overriding method *m*, including other methods called by *m*.

3. There are cases where this is not true, such as when the definition received by a different variable is a function of the variable that was used.




| | *W::v* | *W::u* | *X::x* | *Y::w* |
|---|---|---|---|---|
| ***W::l*** | | ***def*** | | |
| ***W::m*** | ***def*** | | | |
| ***W::n*** | ***use*** | ***use*** | | |
| ***X::n*** | ***use*** | | ***def*** | |
| ***Y::l*** | | ***def*** | | |
| ***Y::m*** | | | | ***def*** |

**Figure 3-5.** State Definition Anomalies

Now consider class *Y* and the method *Y::m*(), which overrides *W*::*n*() through refinement. Observe that *Y*::*m*() does not define *W*::*v*, as *W*:*m*() does; but defines *Y*::*w* instead,. Now, a data flow anomaly exists with respect to the method sequence *m;n* for the state variable *W::v*. When an instance of *Y* is subjected to this sequence, *Y::w* is defined first (because *Y*::*m*() executes), but then *W::v* is used by method *X::n*. The assumption made in the implementation of *X*::*n* that *W::v* is defined by a call to *m* prior to a call to *n* no longer holds, and a data flow anomaly has occurred. In this particular example, a fault has occurred since there is no prior definition of *W::v* when *Y* is the type of an instance being used. Note that this will not be true in the general case since the controlling factor in whether a fault has occurred will be a function of prior method invocations, any default initializations that were applied, and how individual state variables are handled during instance construction.

Any extension method that is called by a refining method must also interact with the inherited variables of the ancestor in a manner consistent with the ancestor's current state. Since

the extension method provides a portion of the refining method's effects, to avoid a data flow anomaly the extension method must avoid defining inherited state variables in a manner that would be inconsistent with the method being refined by the calling method. The net effect of the extension method cannot leave the ancestor in a state that is logically different from when it was invoked. For example, if the logical state of an instance of a stack is currently *not-empty/not-full*, then execution of an extension method cannot result in the logical state spontaneously being changed to either *empty* or *full*. Doing so would preclude the execution of *pop* or *push* as the next methods in the sequence, respectively.

### 3.2.3 State Definition Inconsistency due to State Variable Hiding (SDIH)

The introduction of an indiscriminately named local state variable can easily result in a data flow anomaly where none would otherwise exist. If a local variable is introduced to a class definition where the name of the variable is the same as an inherited variable *v*, the effect is the inherited variable is hidden from the scope of the descendant (unless explicitly qualified, as in *super.v*). A reference to *v* by an extension or overriding method will refer to the local (i.e. the descendant's) *v*. This is not a problem if all inherited methods are overridden since no other method would be able to implicitly reference the inherited *v*. However, this pattern of inheritance is the exception rather than the rule. There will typically be one or more inherited methods that are not overridden. There is a possibly for a data flow anomaly to exist if a method that normally defines the inherited *v* is overridden in a descendant when a inherited state variable is hidden by a local definition.

As an example, again consider the class hierarchy shown in Figure 3-5. Suppose the specification of class *Y* has the local state variable *v* that hides the inherited variable *W::v*. Further suppose method *Y::m* defines *v*, just as *W::m* defines *W::v*. Given the method sequence *m;n*, a data flow anomaly exists between *W* and *Y* with respect to *W::v*.

### 3.2.4 State Defined Incorrectly (SDI)

Suppose an overriding method defines the same state variable (or variables) *v* as the overridden method. If the computation performed by the overriding method is not semantically

equivalent to the computation of the overridden method with respect to *v*, then subsequent state dependent behavior in the ancestor will likely be affected, and the externally observed behavior of the descendant will be different from the ancestor. While this problem is not a data flow anomaly, it is a potential behavior anomaly.

### 3.2.5 Indirect Inconsistent State Definition (IISD)

An inconsistent state definition can occur when a descendant adds an extension method that defines an inherited state variable. For example, consider the class hierarchy shown in Figure 3-6a, where *Y* specifies a state variable *x* and method *m*(), and the descendant *D* specifies method *e*. Since *e* is an extension method, it cannot be directly called from an inherited method, in this case *T*::*m*(), because *e* is not visible to the inherited method.[1] However, if an inherited method is overridden. the overriding method (such as *D::m*() as depicted in Figure 3-6b) can call *e* and in so doing, introduce a data flow anomaly by having an effect on the state of the ancestor that is not semantically equivalent to the overridden method (e.g. with respect to *T::y* in the example). Whether an anomaly results is a function

1. Strictly speaking, in some object-oriented languages (e.g. Java, C++), this can be circumvented through the use of type coercion. The implementation of the inherited method casts the instance whose context it is executing in to the type of the descendant, and then makes the call to the descendant's extension method. For the example in Figure 3-6, this would could be accomplished with the Java statement ((*D*)*this*).*e*(). However, does not appear to happen often in practice, and should be frowned upon from the perspective of software engineering and object-oriented design principles.

of which state variable *e* defines, where *e* executes in the sequence of calls made by a client, and what state dependent behavior the ancestor has on the variable defined by *e*.

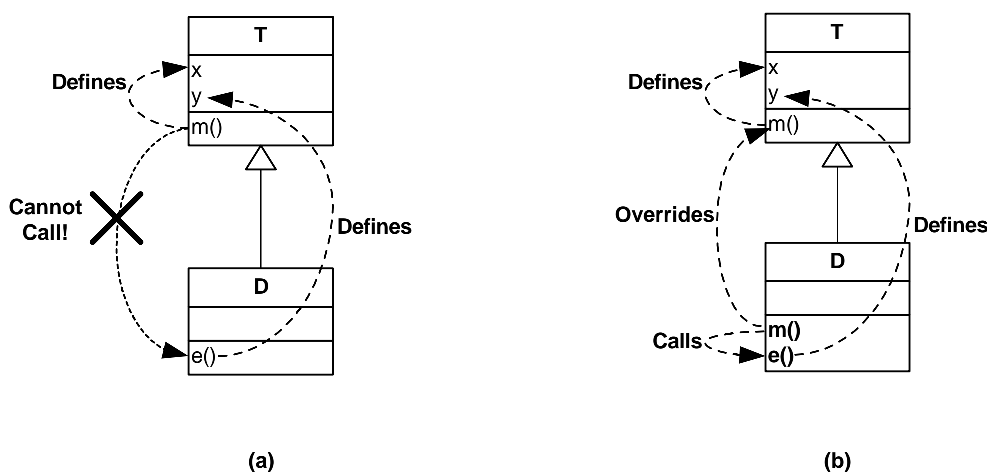



**Figure 3-6.** Example of Indirect Inconsistent State Definition (IISD)

### 3.2.6 Anomalous Construction Behavior(1) (ACB1)

The constructor of an ancestor class *C* calls a locally defined polymorphic function *f*. Because *f* is polymorphic, a descendant class *D* can provide an overriding definition of *f*. If so, then the *D*'s version of *f* will execute when the constructor of *C* calls *f*, not the version defined by *C*. To see this, consider the class hierarchy shown in the left half of Figure 3-7. Class *C*'s constructor calls *C*::*f*(). Class *D* contains the overriding method *D*::*f*() that defines the local state variable *D*::*x*. There is no apparent interaction between *D* and *C* since *D*::*f*() does not interact with the state of *C*. However, *C* interacts with *D*'s state by virtue of the apparent call that *C*'s constructor makes to *C*::*f*(). In some object-oriented languages (e.g. Java and C#), constructor calls to polymorphic methods execute the method that is closest to the instance that is being created. The class *C* in the hierarchy in Figure 3-7, the closest version of *f*() to *C* is specified by *C* itself, and thus executes when an instance of *C* is constructed. For *D*, the closest version is *D*::*f*(), which means that when an instance of *D* is being constructed, the call made to *f*() in *C*'s constructor actually executes *D*::*f*() instead of

its own locally specified *f*(). This is illustrated by the yo-yo graph in the right half of Figure 3-7.

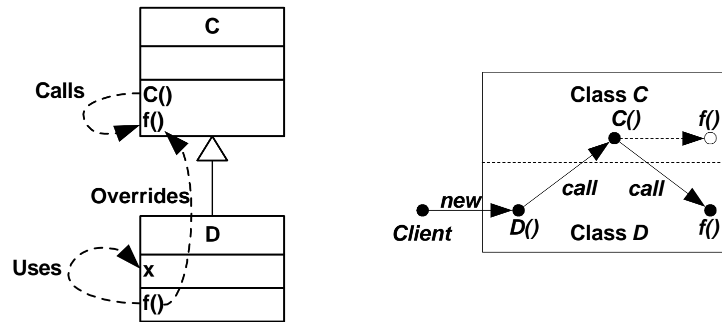



**Figure 3-7.** Example of Anomalous Construction Behavior

The result of the behavior shown in Figure 3-7 can easily result in a data flow anomaly if *D::f*() uses variables defined in the state space of *D*. Because of the order of construction, *D*'s state space will not have been constructed. Whether or not an anomaly exists depends on if default initializations have been specified for the variables used by *f*(). Furthermore, a fault will likely occur if the assumptions or preconditions of *D::f*() have are not satisfied prior to construction [3].

### 3.2.7 Anomalous construction behavior(2) (ACB2)

Similar to ACB1 (Section 3.2.6), the constructor of an ancestor class *C* calls a locally defined polymorphic function *f*. A data flow anomaly can occur if *f* is overridden in a descendant class *D* and if that overriding method uses state variables inherited from *C*. The anomaly occurs if the state variables used by *D::f* have not been properly constructed by *C::f*. This is dependent upon the set of variables used by *D::f* and the order the variables in the state of *C* are constructed, and the order in which *f* is called by *C*'s constructor. Note that it is not generally possible for the programmer of class *C* to know in advance which version of *f* will actually execute, and which state variables that the executing version

depends on. Thus, the invocation of polymorphic method calls from constructors is unsafe and introduces non-determinism into the construction process. This is true of both ACB2 and ABC1.

### 3.2.8 Incomplete (failed) Construction (IC)

In some programming languages, the value of the variables in the state space of a class prior to construction is undefined. This is true, for example, in C++ but not in Java. The role of the constructor is to establish the initial state conditions and the state invariant for new instances of the class. To do so, the constructor will generally have statements that define every state variable. In some circumstances, again depending upon the programming language, default or other explicit initializations may be sufficient. In either case, by the time the constructor has finished, the state of the instance should be well defined. There are two possibility for faults here. First, the construction process may have assigned an initial value to a particular state variable, but it is the wrong value. That is, the computation used to determine the initial value is in error. Second, the initialization of a particular state variable may have been overlooked. In this case, there is a data flow anomaly between the constructor and each of the methods that will first use the variable after construction (and any other uses until a definition occurs).

An example of incomplete construction is shown by the code fragment in Figure 3-8. Class *AbstractFile* contains the state variable *fd* that is not initialized by a constructor. The intent of the designer of *AbstractFile* is that a descendant class provide the definition of *fd* prior to its use, which is done by method *open* in the descendant class *SocketFile*. If any descendant that can be instantiated defines *fd*, and no method is called that uses *fd* prior to the definition, there is no problem. However, a fault will occur if either of these conditions is not satisfied.[1]

Observe that while the designer's intent is for a descendant to provide the necessary definition, a data flow anomaly exists within *AbstractFile* with respect to *fd* for methods *read* and

---

1. This example was contributed by Charles D. Hutchinson.

*write*. Both of these methods use *fd*, and if either are called immediately after construction, then a fault will occur. Note that this design introduces an element of non-determinism into *AbstractFile* since it is not known at design time what type of instance *fd* will be bound to, or if it will be bound (i.e. defined) at all. Suppose that the designer of *AbstractFile* also designed and implemented *SocketFile*, as also shown in Figure 3-8. By doing so, the designer has ensured that the data flow anomaly that exists in *AbstractFile* is abated by the design of *SocketFile*. However, this still does not eliminate the problem of non-determinism and the introduction of faults since, at some point in time in the future, a new descendant can be added that fails to provide the necessary definition.

```
 1 Class abstract AbstractFile
 2 {
 3     FileHandle fd;
 4
 5     abstract public open();
 6
 7     public read() { fd.read( ... ); }
 8
 9     public write() { fd.write( ... ); }
10
11     abstract public close();
12 }
13
14 Class SocketFile extends AbstractFile
15 {
16     public open()
17     {
18         fd = new Socket( ... );
19     }
20
21     public close()
22     {
23         fd.flush();
24         fd.close();
25     }
26 }
```

**Figure 3-8.** Incomplete construction of state variable *fd*

### 3.2.9 State Visibility Anomaly (SVA)

The state variables in an ancestor class *A* are declared *private*, and a polymorphic method *A::m* defines *A::v*. Suppose that *B* is a descendant of *A*, and *C* of *B*, as depicted in Figure 3-9a. Further, *C* provides an overriding definition of *A::m* but *B* does not. Since *A::v* has private visibility, it is not possible for *C::m* to properly interact with the state of *A* by directly

defining *A::v*. Instead, *C::m* must call *A::m* to affect the proper interaction. Now suppose that *B* also overrides *m* (Figure 3-9b) Then for *C::m* to properly define *A::v*, *C::m* must call *B::m* which in turn must call *A::m*. Thus, *C::m* has no direct control on whether the data flow anomaly is resolved due to *B*'s overriding *m*. In general, when private state variables are present, the only way that a data flow anomaly can be avoided is for every overriding method in a descendant to call the overridden method in its ancestor class. Failure to do so will likely result in the manifestation of a fault in the state and behavior of *A*.

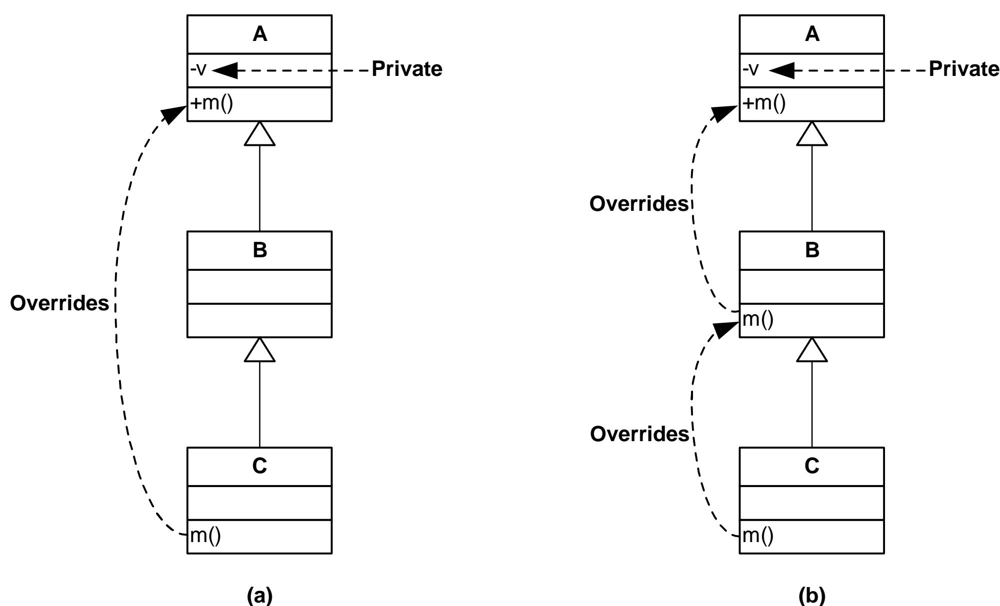



**Figure 3-9.** State Visibility Anomaly

## 3.3 Syntactic Patterns of Inheritance

There a number of basic syntactic patterns that can be used to extend a class through inheritance. The use of individual or combinations of these patterns in part determines the semantics of a descendant class and its behavioral compatibility with its ancestor. It is this behavioral compatibility that determines whether or not instances of the descendant can be safely substituted for instances of the ancestor. A preliminary list of syntactic patterns is

summarized in Table 3-2. Each entry gives an acronynm, a short description, and a reference to the subsection where the pattern is described.

**Table 3-2.** Syntactic Patterns of Inheritance

| | Acronym | Syntactic Inheritance Pattern | Section |
|---|---|---|---|
| | DNM | Descendant has no methods | 3.3.1 |
| | DNEM | Descendant introduces non-interacting extension methods | |
| Extension | ECE | Extension method calls another extension method | 3.3.2.2 |
| | ECI | Extension method calls inherited methods | 3.3.2.3 |
| | ECR | Extension method calls refining method | 3.3.2.4 |
| | EDIV | Extension method defines inherited state variable | 3.3.2.5 |
| | EDLV | Extension method defines local state variable | 3.3.2.6 |
| | EUIV | Extension method uses inherited state variable | 3.3.2.5 |
| | EULV | Extension method uses local state variable | 3.3.2.6 |
| Refinement | RCE | Refining method calls extension method | 3.3.3.1 |
| | RCI | Refining method calls other inherited method | 1-10 |
| | RCR | Refining method calls another refining method | 3.3.3.4 |
| | RCOM | Refining method calls overridden method | 3.3.3.5 |
| | RDIV | Refining method defines inherited state variable | 3.3.3.5 |
| | RDLV | Refining method defines local state variable | 3.3.3.6 |
| | RUIV | Refining method uses inherited state variable | 3.3.3.5 |
| | RULV | Refining method uses local state variable | 3.3.3.6 |
| Construction | CCIM | Constructor calls inherited method | 3.3.4.1 |
| | CCRM | Constructor calls refining method | 3.3.4.2 |
| | CCEM | Constructor calls extension method | 3.3.4.3 |
| | CDIV | Constructor defines inherited state variable | 3.3.4.4 |
| | CDLV | Constructor defines local state variable | 3.3.4.5 |
| | CUIV | Constructor uses inherited state variable | 3.3.4.4 |
| | CULV | Constructor uses local state variable | 3.3.4.5 |
| Special Cases | CBR1 | Complete behavioral redefinition(1) | 3.3.5.1 |
| | CBR2 | Complete behavioral redefinition(2) | 3.3.5.2 |

Whether or not a descendant is compatible with its ancestor is a function of the effects that the descendant has on the state of its ancestor. These effects are manifested through methods contained in the definition of the descendant. Each of these methods may either refine (through overriding) a method specified by the ancestor, or reflect behavioral extensions provided by the descendant. In either case, it is the *definitional interactions* of these methods with the ancestor's state that determines the substitutability of the descendant. A direct definition interaction occurs when a state variable is used in an expression, such as an assignment. An indirect interaction occurs when an expression calls a method that contains an expression that has a direct interaction. A state interaction may either be a definition or use of a state variable. In some cases, compatibility is guaranteed by virtue of the fact that no definitional interactions are possible. This occurs when the descendant either does not define new methods and does not override inherited methods, or when the descendant defines new methods that do not interact directly or indirectly with the inherited state. That is, for the latter case, the new methods at most use inherited state either by direct reference or by calling inherited methods that return a value but do not change the state of the ancestor.

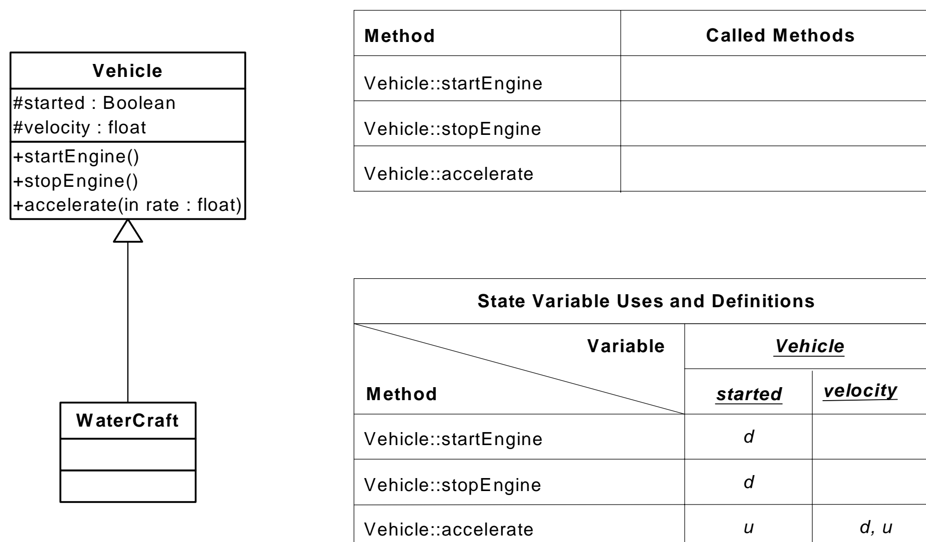
The following subsections discuss each of these cases and additional syntactic inheritance patterns that affect the behavioral compatibility of a descendant class with its ancestor.

### 3.3.1 Descendant has No Methods (DNM)

This is the most trivial case of inheritance: the definition of the descendant contains no methods. Its behavior is defined by the methods that it inherits. The descendant could have its own set of state variables, but there would be little point since these variables could not be changed.[1] An example is shown in the UML class diagram depicted in Figure 3-10, along with tables that summarize the state variable definitions and uses of each method, and the methods that are called. Class *Vehicle* defines state and behavior in terms of its methods *startEngine*, *stopEngine*, *accelerate*, and its state variables *started* and *velocity*. Methods

1. It could be the case that the specification of a descendant includes state variables whose default initialization results in some global state interaction, such as opening a database or network connection.

*startEngine* and *stopEngine* both define variable *started*; method *accelerate* uses *started* and both uses and defines *velocity*.




| Method | Called Methods |
|---|---|
| Vehicle::startEngine | |
| Vehicle::stopEngine | |
| Vehicle::accelerate | |

| State Variable Uses and Definitions | | |
|---|---|---|
| Variable / Method | *Vehicle* | |
| | *started* | *velocity* |
| Vehicle::startEngine | *d* | |
| Vehicle::stopEngine | *d* | |
| Vehicle::accelerate | *u* | *d, u* |

**Figure 3-10.** Descendant whose definitions include no methods or state variables

As Figure 3-10 shows, the specification of class *WaterCraft* does not introduce methods or state variables. Its behavior is determined entirely by *Vehicle*, thus it is not possible for a client that uses an instance of *WaterCraft* where an instance of *Vehicle* is expected to discern any difference in behavior. *WaterCraft*, however, does serve to partition the set of all instances of *Vehicle* into those that are instances of *WaterCraft* and those that are not. This is useful in cases where knowing that a particular *Vehicle* instance is really an instance of *WaterCraft*.

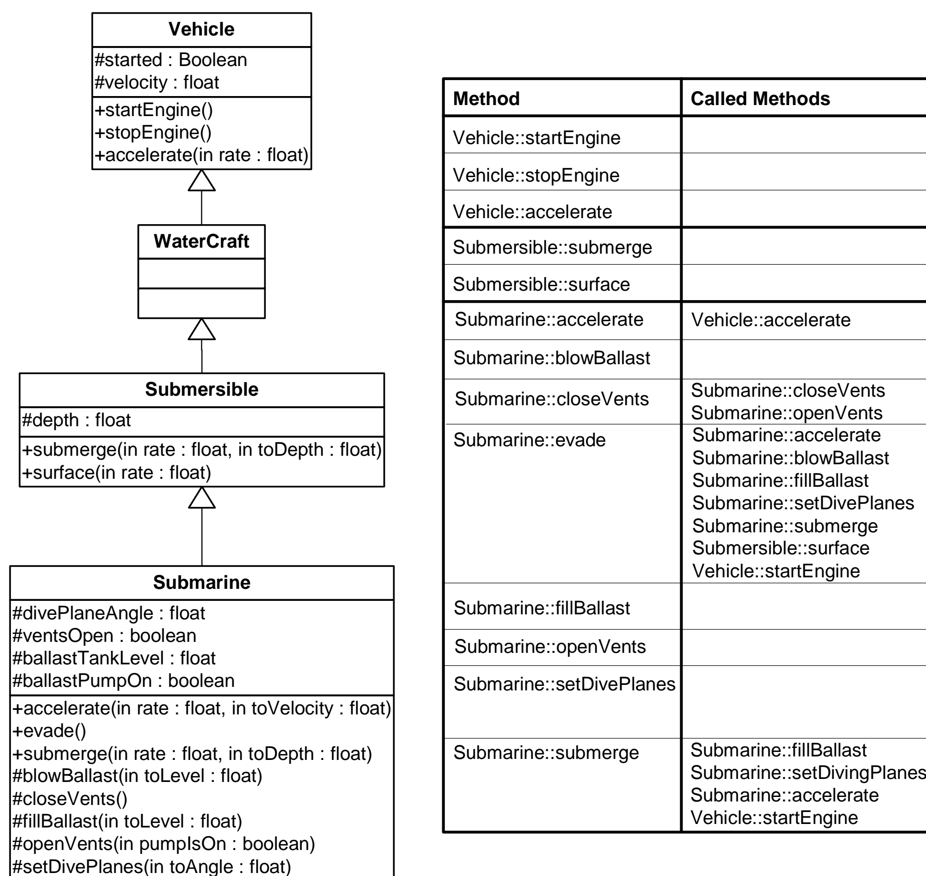
**Faults/anomalies manifested by DNM**. Since the descendant class has no methods, there can be no faults or anomalies due to polymorphism. The only methods that could possibly execute through an instance of the descendant are those belonging to an ancestor.

## 3.3.2 Descendant introduces extension methods

A descendant class can extend the behavior it inherits by defining *extension methods*. Extension methods are methods contained in the specification of a descendant class. They

do not override inherited methods, rather, they add additional behavior not already present in ancestor classes. In so doing, extension methods may or may not have an affect on inherited state.

Figure 3-11 shows the class diagram and table of called methods for the example used in the remainder of Section 3.3. The corresponding definitions and uses are shown in Figure 3-12. This example extends the *Vehicle* class hierarchy by adding *Submarine* as a direct descendant of *Submersible*. This class is an abstraction of a hypothetical submarine that has the additional capability of taking evasive action. Supporting this are behaviors for filling and emptying ballast tanks and setting the angle of diving planes. It also refines the inherited behaviors for submerging and accelerating.




| Method | Called Methods |
| --- | --- |
| Vehicle::startEngine | |
| Vehicle::stopEngine | |
| Vehicle::accelerate | |
| Submersible::submerge | |
| Submersible::surface | |
| Submarine::accelerate | Vehicle::accelerate |
| Submarine::blowBallast | |
| Submarine::closeVents | Submarine::closeVents<br>Submarine::openVents |
| Submarine::evade | Submarine::accelerate<br>Submarine::blowBallast<br>Submarine::fillBallast<br>Submarine::setDivePlanes<br>Submarine::submerge<br>Submersible::surface<br>Vehicle::startEngine |
| Submarine::fillBallast | |
| Submarine::openVents | |
| Submarine::setDivePlanes | |
| Submarine::submerge | Submarine::fillBallast<br>Submarine::setDivingPlanes<br>Submarine::accelerate<br>Vehicle::startEngine |

**Figure 3-11.** Example showing interaction of extension methods

| State Variable Uses and Definitions | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Variable / Method | *Vehicle* | | *Submersible* | *Submarine* | | | | |
| | *started* | *velocity* | *depth* | *divePlaneAngle* | *ventsOpen* | *tankLevel* | *ballastPumpOn* | *ballastTankLevel* |
| Vehicle::startEngine | *d* | | | | | | | |
| Vehicle::stopEngine | *d* | | | | | | | |
| Vehicle::accelerate | *u* | *d, u* | | | | | | |
| Submersible::submerge | | | *d, u* | | | | | |
| Submersible::surface | | | *d, u* | | | | | |
| Submarine::accelerate | | *d, u* | | | | | | |
| Submarine::blowBallast | | | | | *d* | *u,d* | | *u,d* |
| Submarine::closeVents | | | | | | | *d* | |
| Submarine::evade | *u* | | *u,d* | | | | | |
| Submarine::fillBallast | | | | | | *u,d* | | *u,d* |
| Submarine::openVents | | | | | | | *d* | |
| Submarine::setDivePlanes | | | | *d* | | | | |
| Submarine::submerge | *u* | | *u,d* | | | | | |

**Figure 3-12.** Definitions and uses for extensions methods

#### 3.3.2.1 Descendant introduces Non-interacting Extension Methods (DNEM)

The descendant may introduce extension methods that do not interact with inherited state. This form of extension method does not define inherited state variables, nor does it call inherited methods that do. As part of the extending behavior, the descendant may introduce local state variables to support the behavior provided by the extension methods, or it may use variables inherited from an ancestor. The latter may be achieved through either direct reference of a state variable, or by calling some other method that uses the inherited variable.

Figure 3-11 extends the example shown in Figure 3-10 to include class *Submersible,* a direct descendant of *WaterCraft*. *Submersible* adds the two extension methods *submerge* and *surface* and supporting state variable *depth*. As the definition/use table in the Figure 3-12 shows, these methods do not interact with the state of *Vehicle* or *WaterCraft* (which has no state), nor do these methods call inherited methods that alter state.

As part of a behavioral extension, a descendant class will often have its own local set of state variables (as in the variable *float* that is a member of class *Submersible*). Collectively, these variables serve to record the state of the descendant with respect to its set of extension

methods. To affect a local state change, one or more of the extension methods must define each variable in the local state space. In so doing, the behavioral extension of the descendant must either introduce additional states not present in the ancestor (such as when the descendant is capable of doing things that the ancestor is not), or it must ensure that any additional states are logically substates of the ancestor. That is, for the latter case, the stateful behavior of the descendant must be consistent with that of the ancestor.

Any state represented by the descendant's state space must partition each of the ancestor's states for those cases where an extension method can change the state of the descendant. Put another way, the stateful behavior of the descendant must fit within the state machine of the ancestor. No transitions may be removed by the descendant, nor new transitions added that would cause the ancestor to transition to a different state.

**Faults/anomalies manifested by DNEM**. Since a descendant *D* only has extension methods that do not interact with inherited state, there can be no faults due to polymorphism when *D* is used solely in the context of its ancestor. The only methods that can execute in this situation are those available through the ancestor's context. There is, however, still the possibility of inconsistent behavior due to inconsistent type usage (Section 3.2.1). This would occur when an instance of *D* is used in the context of the ancestor as well as that of the descendant. Thus, DNEM can manifest the fault type ITU.

#### 3.3.2.2 Extension method Calls another Extension method (ECE)

Quite often, as part of a descendant's implementation, one extension method *e* will call another to achieve some desired effect. It may be that *e* implements a high-level algorithm (e.g. sorting) and delegates subproblems to other methods (e.g. comparison). Regardless of the number of methods called and the level of nested calls involved, from a client's perspective, the net effect of calling *e* is the result of the computation performed by *e* directly or through that of any methods called (directly or indirectly) by *e*. In terms of state space interactions, the net effect is the set of state variables used or defined by *e*, or by a method

called by *e*, and so forth. Method *e* is said to *absorb* the effects of the methods it calls or causes to be called.

ECE is illustrated in Figure 3-13, which presents an annotated code fragment of a hypothetical implementation of method *Submarine::evade* in Java. As shown, an example of ECE occurs at line 12 where the method *blowBallast* is called. This is ECE because both *evade* and *blowBallast* are extension methods of *Submarine*. Though not annotated, other examples of ECE occur at lines 18 (*setDivePlanes*), 20 (*fillBallast*), and 26 (*setDivePlanes*).

```
 1 public void evade()
 2 {
 3     // Prepare for emergency dive/surface
 4     if ( !started )
 5         startEngine();
 6                                                    ECI
 7     accelerate( MAX_ACCEL, MAX_VELOCITY );
 8                                                    ECR
 9     if ( depth < 0 ) // Are we already submerged?
10     {                                              EUIV
11         setDivePlanes( -MAX_PLANE_ANGLE ); // Max rate of ascent
12         blowBallast( 0 ); // Emergency blow!
13     }                                              ECE
14
15     else
16     {
17         // No, so dive, dive, dive!
18         setDivePlanes( +MAX_PLANE_ANGLE );
19
20         fillBallast( 100.0 ); // Take her down ASAP!
21
22         while ( depth < MAX_DEPTH )
23             depth = ...;                           EUIV
24                                                    EDIV
25         // Now level off.
26         setDivePlanes( 0.0 );
27     }
28 }
```

**Figure 3-13.** Code fragment for method *Submarine::evade*

**Faults/anomalies manifested by ECE**. Descendant classes that use ECE have the possibility to manifest SDA anomalies if the called extension method *c* defines inherited state variables or calls inherited methods that do. This will possibly result in a fault if a method that is subsequently called depends in some way on the state defined by *c*.

The possibility of a local anomaly exists if *c* has public visibility. In this case, *c* provides part of the interface of the descendant and a component of its behavior. The anomaly occurs

if *c* uses state variables that have not yet been defined. Alternatively, *c* can cause an anomaly by defining a set of state variables that are different from those that would be defined by the next method invocation that should occur given the current state of ancestor.

#### 3.3.2.3 Extension method Calls Inherited methods (ECI)

Part of the behavior of a descendant class *C* is defined by an extension method *e* that calls one or more inherited methods that directly define the state inherited from the ancestor class *A*. The called methods may be specified in the same ancestor class that provides the state variable that is defined, or they may be in another class that is also descendant of *A* but is an ancestor of *C*. In either case, execution of the *e* has an effect on the inherited state space received by *C*. An example of this is shown by the call to *Vehicle::startEngine* at line 5 in the code fragment depicted in Figure 3-13.

**Faults/anomalies manifested by ECI**. If the inherited method *i* called by an extension method defines state variables (in the ancestor's context), an SDA anomaly can occur if a subsequently called method depends upon the ancestor's state in some way that has been affected by *i*, and possibly will lead to a fault. Alternatively, an SDA anomaly will exist if *i* uses state variables and is called out of sequence with respect to the current state, particularly if those state variables have not be defined by a prior method invocation.

#### 3.3.2.4 Extension method Calls Refining method (ECR)

An extension method *e* in the specification of a descendant *D* may call a refining method *r* that is also contained in *D*'s specification. In doing so, *e* interacts with the inherited state through the computation carried out by *r*. Whatever state variables *r* defines, *e* effectively defines as well (though not directly) and likewise for used variables. The nature of this interaction is completely out of the control and influence of *e*; it is determined solely by the implementation of *r*. However, *e* does have the choice of when (or if) *r* is called during its execution. To preserve behavioral compatibility between the descendant and the ancestor, the designer of *e* can take measures to ensure that *r* is called in a manner that is consistent with the current state of the ancestor. However, there is no obligation or guarentee on *e*'s designer to do so. An example of ECR is shown at line 7 of Figure 3-13.

**Faults/anomalies manifested by ECR**. The problems for ECR are similar to ECI (Section 3.3.2.4). But in this case, a refining method *r* is called. If *r* defines inherited state variables, an SDA anomally can occur if a subsequently called method depends upon the ancestor's state in some way that has been affected by *r*, and possibly will lead to a fault. A fault will also occur if the state variables are defined incorrectly even though a definition is appropriate for the current state of the ancestor. Similar to ECI, an SDA anomaly will also exist if *r* uses state variables and is called out of sequence with respect to the current state of the ancestor.

#### 3.3.2.5 Extension method Uses/Defines Inherited state Variable (EUIV/EDIV)

The specification of a descendant class includes an extension method that directly uses (EUIV) or defines (EDIV) one or more inherited state variables. By defining the inherited variable, the extension method directly affects the behavior of the ancestor class. An example is shown in Figure 3-13 at line 23 where *Vehicle::depth* is defined (EDIV), and at line 22 where it is used (EUIV).

**Faults/anomalies manifested by EUIV and EDIV.** From the perspective of polymorphic behavior, EUIV does not yield any of the faults or anomalies described in Section 3.2. This is not to say that there is no possibility of any type of fault or anomaly, merely that those that can occur do not manifest themselves as the result of polymorphism. With EDIV, however, the situation is different. Since the extension method is defining inherited state variables, there is the possibility of SDA anomalies that have the potential to yielding a failure in the context of the ancestor.

Assuming that an extension method defines an inherited state variable *v* at a time that is consistent with the current state of the ancestor, an SDI fault can result if the definition given to *v* is not consistent with how the variable is defined by ancestor methods.

#### 3.3.2.6 Extension method Uses/Defines Local state Variable (EULV/EDLV)

An extension method within a descendant class *C* either uses (EULV) or defines (UDLV) one or more state variables that are contained in the specification of *C*. As an example of

EDLV, again consider class *Submarine* (Figure 3-13). The specification of *Submarine* includes the state variable *ballastTankLevel*. This variable is defined (EDLV) in the code fragment shown in Figure 3-14 by the extension method *Submarine::blowBallast* at line 9, and used at line 8 (EULV).

```
 1 protected void blowBallast( float toLevel )
 2 {
 3     if ( ballastTankLevel > toLevel )
 4     {
 5         openVents( true ); // Force water out of tanks.
 6
 7         // Wait for tank to empty.
 8         while ( ballastTankLevel > toLevel )                EULV
 9             ballastTankLevel = ...;                         EDLV
10
11         closeVents();
12     }
13 }
```

**Figure 3-14.** Code fragment for method *Submarine::blowBallast*

**Faults/anomalies manifested by EULV and EDLV.** There are no anomalies or faults manifested directly as a result of polymorphism for EULV and EDLV. Extension methods can only be invoked by a client outside of the descendant's inheritance hierarchy through an instance in the context of the descendant. That is, there must be an object reference whose declared type is in the type family defined by the descendant. However, it is possible for a definition of a local state variable made by an extension method to indirectly manifest a fault through polymorphism. This can occur if the extension method is called by a refining method (see Section 3.3.3.1). Thus, EDLV can result in an SDA anomaly. It can also result in an SDI anomally if the state variable is defined incorrectly.

### 3.3.3 Descendant introduces refining methods

Method refinement allows a descendant class to modify an ancestors's behavior by providing overriding definitions of inherited methods. When an overridden method is called, the overriding definition is invoked instead of the original inherited definition. This allows the descendant to directly refine the behavior exhibited by the ancestor. This refinement is manifested using any of three syntactic mechanisms: directly calling the refined (overrid-

den) method, replacing the refined method, or directly defining inherited state variables. Note that the last mechanism, out of necessity, must be used in combination with only the first two.

#### 3.3.3.1 Refining method Calls Extension method (RCE)

As part of its behavior, a refining method can call extension methods defined by the descendant. The latter can effect local state change, or simply participate in the refinement of the overridden method. In either case, the behavior of the extension method becomes part of the net effect of the refining method's behavior. Thus, the gross behavior of the combination of methods must be consistent with the behavior of the overridden method.

Syntactically, RCE looks just like any other free-standing method call (i.e. not through an instance context). If an instance context is used to qualify the call, out of necessity it must be through the context provided by the self-referencing variable used to denote the current instance (e.g. *this* in Java and C++, and *current* in Eiffel).

An example of RCE is illustrated in Figure 3-15. At line 9, the extension method *Submarine::setDivePlanes* is called by the refining method *Submarine::submerge*.

```
 1 public void submerge( float rate, float toDepth )
 2 {
 3     // Prepare to dive.
 4     if ( !started )
 5         startEngine();                                   RCI
 6
 7     accelerate( NORMAL_ACCEL, DIVING_VELOCITY );
 8                                                          RCR
 9     setDivePlanes( rate );
10                                                          RCE
11     fillBallast( 50.0 ); // Take her down slowly.
12
13     while ( depth < toDepth )
14         depth = ...;
15
16     // Now level off.
17     setDivePlanes( 0.0 );
18 }
```

**Figure 3-15.** Code fragment for method *Submarine::submerge*

**Faults/anomalies manifested by RCE.** Anomalies and faults manifested by RCE include SDA, SDI, and IISD. A refining method manifests SDA by failing to define the same set of the ancestor's state variables as the overridden method does. Similarly, if it does define the right state variables, it could define them incorrectly (an SDI fault). Finally, the refining method can exhibit an IISD anomaly (a composite of SDA and SDI) if it calls one of the descendant's extension methods (see Section 3.3.2.6).

#### 3.3.3.2 Refining method Calls other Inherited method (RCI)

A refining method *r* calls another method *m* inherited from the ancestor, and *m* is not overridden by the descendant. This has the effect of replacing the method *o* overridden by *r* with *m* in terms of the state effects on the ancestor, or possibly combining with those of *o* should *r* call it (see RCOM, Section 3.3.3.4). An example of RCI is shown in Figure 3-15 at line 5.

**Faults/anomalies manifested by RCI.** A refining method that calls an inherited method (other than the overridden method *o*) can manifest both an SDA anomaly and an SDI fault. This is dependent upon the state effects of the inherited method *i* that is called. It could be that *i* defines the same set of state variables as *o* does, or a different set. The latter results in the SDA anomaly. If *i* does define the same set of state variables as *o* (or a proper subset), but the semantics of the resulting definition are different, then an SDI fault occurs.

#### 3.3.3.3 Refining method Calls other another Refining method (RCR)

As part of its implementation, a refining method can call other refining methods. Since both the caller and the called method are members of the descendant, the call will generally be unqualified. However, if it is qualified, it must be through a reference to the current instance (e.g. *this* in *Java* and C++).

**Faults/anomalies manifested by RCR.** From an anomaly and fault perspective, the effects of a refining method calling another refining method are similar to a refining method calling an extension (Section 3.3.3.2). Both SDA anomalies and SDI faults are possibilities.

### 3.3.3.4 Refining method Calls Overridden Method (RCOM)

Perhaps the simplest form of behavioral modification is where the refining method directly calls the refined (overridden) method in addition to providing additional behavior. This form of modification takes advantage of existing behavior rather than replicating or replacing it completely. The result of calling the inherited method is that the refining method interacts with the ancestor's state indirectly by virtue of having called the overridden methods. Method *Submarine::accelerate* shown in Figure 3-16 provides an example of RCOM. The overridden method *Vehicle::accelerate* is called at line 6 through the instance context provided by the explicit ancestor reference *super*.

```
 1 public void accelerate( float rate, float toVelocity )
 2 {
 3     if ( velocity < toVelocity )
 4     {                                         RUIV
 5         // Accelerate to desired velocity.
 6         super.accelerate( rate );
 7                                               RCOM
 8         // Continue to accelerate.
 9         while ( velocity < toVelocity )
10             velocity = ...;
11                                               RDIV
12         // Stop accelerating.
13         super.accelerate( 0.0 );
14     }
15 }
```

**Figure 3-16.** Code for *Submarine*::*accelerate* illustrating RUIV, RCOM, and RDIV

**Faults/anomalies manifested by RCOM.** When a refining method *r* calls the overridden method *o*, the net effect of *o* is included in *r*. If *r* does nothing but call *o*, then there can be no anomalies or faults that will be manifested as a result of polymorphism. However, if *r* does more, in particular, if it defines additional state variables not defined by *o* or if it redefines those defined by *o*, then SDA anomalies and SDI faults are a possibility (see Section 3.3.3.5).

### 3.3.3.5 Refining method Defines/Uses Inherited state Variable (RDIV/RUIV)

The refining method can interact with the state of an ancestor simply by defining or using state variables. Definition is accomplished through direct reference, such as in an assign-

ment statement, or indirectly by calling state defining methods (if the variable is a reference to an object).[1] Similarly, a state variable can be used on the right-hand side of an assignment and as part of a conditional expression. If the variable is a reference to an object, then calling a method through the instance context provided by the variable is also an example of a use.

Both RUIV and RDIV are illustrated in Figure 3-16. At line 16, variable *Vehicle::velocity* is defined (RDIV) by method *Submarine::accelerate*. The method also uses (RUIV) *Vehicle::velocity* at line 3 (and also at line 8 though this is not annotated).

**Faults/anomalies manifested by RDIV and RUIV.** Both SDA anomalies and STI faults are possibilities for RDIV. An SDA anomaly will occur if the refining method does not define the same state variables as the overridden method. An SDI fault will occur if the refining method defines an inherited variable in a manner inconsistent with how the overridden method defines the same variable.

An SDIH anomaly occurs in conjunction with RDIV if the specification of the descendant includes a local state variable *v* whose name is identical to one that is inherited and that is defined by the refining method. An SDIH anomaly also occurs with RUIV if *v* is used to define an inherited state variable.

#### 3.3.3.6 Refining method Defines/Uses Local state Variable (RDLV/RULV)

Instead of interacting with the state inherited from the ancestor, the refining method can have an effect on the local state of the descendant. As with RDIV, definition is accomplished through direct reference, such as in an assignment statement, or indirectly by calling state defining methods (if the variable is a reference to an object). Likewise, as with

1. In some object-oriented languages, such as C++, it is possible to specify that a given method does not change the state of an object (through the use of *const* methods). In other languages, this is not possible. Thus, without the availability of knowledge to contrary, we take the conservative view that all method calls result in a state change of the object referred to by the variable that provides the instance context of the call.

RUIV, a state variable can be used on the right-hand side of an assignment, or as part of a conditional expression.

Syntactically, RDIV and RUIV are similar to the syntax for RDLV and RULV, respectively. The difference is that the variables referenced are specified locally in the descendant, and any qualification present must reference the current instance.

**Faults/anomalies manifested by RDLV and RULV.** There are no faults for RDLV or RULV that manifest themselves as a result of polymorphism.

### 3.3.4 Descendant Introduces Constructors

Classes usually have special methods, called constructors, whose job is to initialize the state of a newly created instance. At the end of the construction process, the state of the instance should be well-defined and ready to suffer the effects of the classes's methods.

There are a number of syntactic patterns that can be used to define the behavior required for construction. A number of the patterns involve calls to other methods. Depending upon the language (e.g. Java), there is inherent danger in calling polymorphic methods from a constructor. The problem is that the designer of the constructor *c* can never know for sure that the called method *e* will be the one executed. This is due to method overriding and polymorphism. If *e* is polymorphic and is overridden by some descendant class, then when an instance of that child class is being constructed, the overriding method will be the one executed from the constructor call instead of *e*. This yields two further complications. First, there is no guarentee that the overriding method will have the same effect on the instance being constructed by *c*. Second, when the overriding method executes, it will be in the context of the child class, which will not have been constructed yet. Thus, there is a strong likelihood that a data flow anomaly or fault will occur. Even though this is an unwise practice, it is possible and people do it

A constructor can introduce an IC anomaly if it fails to properly initialize all state variables defined locally to the class. This may result from the failure to assign a value to a variable,

assigning it the wrong value, calling the wrong method if the variable refers to an object. Either way, the likely result will be anomalous behavior when the newly constructed instance is used. Note that this applies to all of the syntactic patterns that involve construction.

The following sections describe in detail each of the syntactic patterns that involve construction.

#### 3.3.4.1 Constructor Calls Inherited Method (CCIM)

During the construction process, a descendant's constructor can call a method *m* inherited from an ancestor. Unless overridden by the descendant, *m* will execute in the context of the ancestor, having an effect on the ancestor's state inherited determined by its implementation. By the time that *m* executes, the ancestor's construction process will have completed. Any effects *m* has will place the ancestor in a state different from that provided by the constructor.

**Faults/anomalies manifested by CCIM.** A constructor can introduce an SDA anomaly by defining a state variable *v* inherited from the descendant's ancestor. This can be accomplished either by direct definition of *v* (Section 3.3.4.4), or by calling an inherited method that defines *v*. Either way, an anomaly will occur if the resulting definition is not consistent with the current state of the ancestor. Observe that by calling an inherited method, the descendant's constructor is effectively changing the construction process that the ancestor has carried out. Note that if the inherited method called by the constructor is polymorphic, then the anomalous behavior described in the introduction to Section 3.3.4 is possible.

#### 3.3.4.2 Constructor Calls Refining Method (CCRM)

Similar to CCIM, during the construction process, a refining method *r* may be called. The act of calling *r* might have an effect on the local state of the descendant. Presumably, this effect will be part of the intended construction process and will contribute to the initialization of a locally well-defined state for the descendant. Note that the refining method may

call the overridden method (or another non-overridden inherited method). The result of such a call will be equivalent to CCIM (Section 3.3.4.1).

**Faults/anomalies manifested by CCRM.** As with CCIM (Section 3.3.4.1), a data flow anomaly will occur if the result of the called refining method *r* is that the state of the ancestor is defined in some manner that is inconsistent with its state, or if *r* uses portions of the ancestor's state that are not consistent with the assumptions made in the implementation of *r*. Note that if the refining method called by the constructor is polymorphic, then the anomalous behavior described in the introduction to Section 3.3.4 is possible.

#### 3.3.4.3 Constructor Calls Extension Method (CCEM)

A constructor can call an extension method *e* as part of the construction process. Similar to CCRM (Section 3.3.4.2), calling *e* might result in an affect on the local state of the descendant. Likewise, *e* could also call other methods (extension, refining, or inherited) that affect either the local or inherited state.

**Faults/anomalies manifested by CCEM.** The fault model for CCEM is the same as for CCRM: a data flow anomaly will occur if the result of the called extension method *e* is that the state of the ancestor is defined in some manner that is inconsistent with its state, or if *e* uses portions of the ancestor's state that are not consistent with the assumptions made in the implementation of *e*. Note that if the extension method called by the constructor is polymorphic, then the anomalous behavior described in the introduction to Section 3.3.4 is possible.

#### 3.3.4.4 Constructor Defines/Uses Inherited state Variable (CDIV/CULV)

During the construction process, out of necessity a constructor will define one or more state variables. Usually these are local to the class being constructed. However, it is possible for a constructor to define an inherited state variable, either directly through assignment or indirectly through method call (if the variable refers to an object).

**Faults/anomalies manifested by CCRM.** Both SDA anomalies and STI faults are possibilities for CDIV. An SDA anomaly will occur if the refining method does not define the same state variables as the overridden method. An SDI fault will occur if the refining method defines an inherited variable in a manner inconsistent with how the overridden method defines the same variable.

#### 3.3.4.5 Constructor Defines/Uses Local state Variable (CDLV/CULV)

A constructor can as part of its implementation use both local and inherited state variables. The key distinction between the two is that the ancestor's construction process has completed, and the inherited state variables should be properly initialized. For local state variables, proper initializations will only have occurred prior to use if the constructor has defined their values, or if there are suitable default initializations provided (as in Java).

**Faults/anomalies manifested by CCRM.** An SDIH anomaly occurs in conjunction with CDLV if the specification of the descendant includes a local state variable *v* whose name is identical to one that is inherited and that is defined by the refining method. An SDIH anomaly also occurs with CULV if *v* is used to define an inherited state variable.

### 3.3.5 Special cases – Complete Behavioral Redefinition

There are two situations that warrant consideration in this discussion. Instead of describing distinct patterns, both are combinations of those patterns previously described. It is entirely possible for the behavior provided by an ancestor class to be completely replaced by refined behavior.[1] This can occur in two different ways. First, a descendant overrides all methods inherited from the ancestor, thereby directly nullifying the behavior of the ancestor all together. Second, a sequence of descendants incrementally overrides proper subsets the ancestor's methods until ultimately the behavior of the ancestor is nullified. These two scenarios are discussed in the following subsections.

---

1. This is only possible if all of the ancestor's methods that are visible to the descendants are polymorphic (i.e. they can be overridden).

#### 3.3.5.1 Complete Behavioral Redefinition(1) (CBR1)

A descendant may override all methods inherited from the ancestor. In so doing, the descendant has assumed full behavioral responsibility from the ancestor, and the state inherited from the ancestor either becomes irrelevant or the constraints on the inherited state change. The inherited state will become irrelevant if none of the overriding methods references any of the state variables, either through direct or indirect use. This effectively relegates the ancestor class to the role of providing only an interface definition to the descendant. Clients will at least see the descendant as being an instance of the ancestor from a syntactic perspective. However, the descendant's behavior may turn out to be inconsistent with the ancestor's.

It may be that the descendant makes use of the inherited state. Depending upon how this is done, the constraints on how overriding and extension methods use the inherited state may change. If none of the overridden methods are called, then the overriding methods are free to use the inherited state at will.[1] If any of the overridden methods are called, then a data flow anomaly may result.

#### 3.3.5.2 Complete Behavioral Redefinition(2) (CBR2)

A sequence of descendant classes (rooted at a particular ancestor) in combination may override the complete behavior inherited from the ancestor. Similar to CBR1 (Section 3.3.5.1), the combination has assumed full behavioral responsibility from the ancestor. For example, consider the hierarchy shown in Figure 3-17. Class *A* defines a set of methods that operate on its state. Classes *B*, *C*, *D* and *E* are descendants of *A*, with *B*, *C* and *D* overriding disjoint subsets of *A*'s methods, such *B*, *C* and *D* partition *A*'s methods. The further down the hierarchy we traverse, the fewer the number of *A*'s methods that reach

1. This is subject to the constraint that the descendant must still exhibit external behavior that is logically equivalent to the ancestor.

a particular descendant. Ultimately, none of *A*'s methods are inherited by *E*. By the time *D* is reached, a complete behavioral redefinition of *A* has occurred.

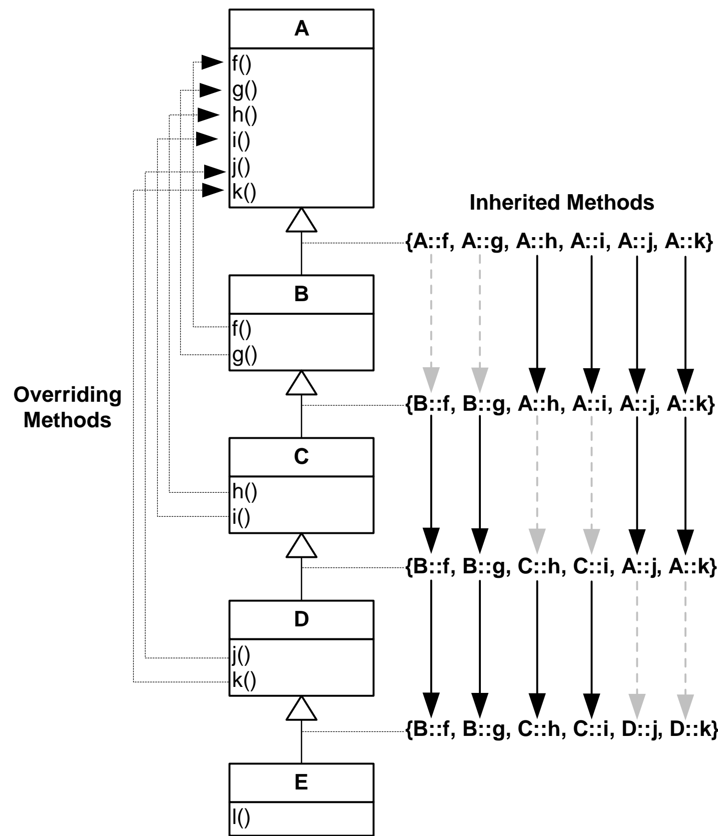



**Figure 3-17. Complete Behavioral Redefiniditon (2)**

Unlike CBR1, there generally is not the same flexibility in how the state inherited from the ancestor is treated. The descendants higher in the hierarchy than *E* (*B*, *C* and *D*) only override a portion of the behavior that is inherited from *A*. To preserve behavioral compatibility with *A*, classes *B*, *C* and *D* must interact with the state of *A* in a manner consistent with the state interactions of the methods that each inherits. For *B*, this means that the implementation of its overriding methods must ensure consistent state interactions with respect to the methods they override. The same applies for *C*, though it must also consider the methods

that it inherits from *B* as well as those that reach it from *A*. *D* only has to be concerned with the state interactions of the methods that it inherits from *B* and *C*. This is because *D* is overriding the remaining methods inherited from *A* that have not been overridden by *B* and *C*.

**Table 3-3.** Fault/anomaly types manifested by syntactic patterns

| | ITU | SDA | SDIH | SDI | IISD | ACB1 | ACB2 | IC | SVA |
|---|---|---|---|---|---|---|---|---|---|
| **DNM** | | | | | | | | | |
| **DNEM** | ✓ | | | | | | | | |
| **ECE** | | ✓ | | | | | | | |
| **ECI** | | ✓ | | | | | | | |
| **ECR** | | ✓ | | | | | | | |
| **EDIV** | | ✓ | | ✓ | | | | | |
| **EDLV** | | | | | | | | | |
| **EUIV** | | | | | | | | | |
| **EULV** | | | | | | | | | |
| **RCE** | | ✓ | | ✓ | ✓ | | | | |
| **RCI** | | ✓ | | ✓ | | | | | |
| **RCR** | | ✓ | | ✓ | | | | | |
| **RCOM** | | ✓ | | ✓ | | | | | |
| **RDIV** | | ✓ | ✓ | ✓ | | | | | |
| **RDLV** | | | | | | | | | |
| **RUIV** | | | ✓ | | | | | | |
| **RULV** | | | | | | | | | |
| **CCIM** | | ✓ | | | | | | ✓ | |
| **CCRM** | | ✓ | | | | ✓ | ✓ | ✓ | |
| **CCRM** | | ✓ | | | | ✓ | ✓ | ✓ | |
| **CDIV** | | ✓ | ✓ | | | | | | |
| **CDLV** | | | | | | | | | |

**Table 3-3.** Fault/anomaly types manifested by syntactic patterns

| | ITU | SDA | SDIH | SDI | IISD | ACB1 | ACB2 | IC | SVA |
|---|---|---|---|---|---|---|---|---|---|
| **CUIV** | | | ✓ | | | | | | |
| **CULV** | | | | | | | | | |

# 3.4 Discussion

For expository purposes, the discussion of the anomalies and fault types summarized in Table 3-3 and described in Section 3.3 has primarily focused on single instances of syntactic patterns of inheritance. In reality and out of necessity, the patterns are often combined to form complex aggregates of control and data flow. Naturally, this combination of patterns can result in combinations of faults.

As the examples have shown, the control flow that results from inheritance and polymorphism can be quire complex, and can yield very complicated faults and anomalies. In fact, the use of polymorphism induces non-determinism to the actual flow of control [15]. Sadly, the situation in reality can be far more complicated than the examples have indicated. If inheritance hierarchies are deep, visibility is unrestricted, and polymorphic methods are abundant, the flow of control resulting from a single method invocation can be inordinately complex, depicted by the simple yo-yo graph shown in Figure 3-18. Likewise, it can be

expected that the effort required to detect, diagnose, and correct the resulting faults to increase significantly in complexity.

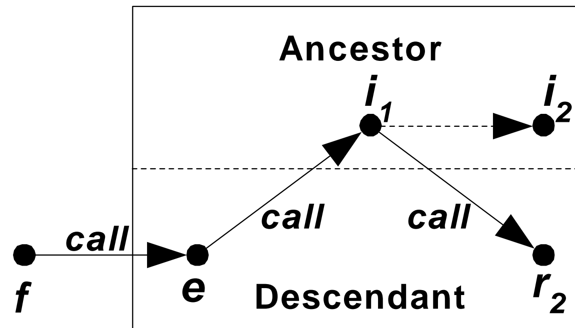



**Figure 3-18.** Yo-yo effect resulting from extension method calling inherited method

# 4. Coupling-based Analysis of Object-Oriented Programs

This Chapter provides the foundation for coupling-based testing of object-oriented programs. This work extends the original coupling-based testing approach of Jin and Offutt [38] to account for the effects of inheritance and polymorphism. The key contribution is the *coupling sequence*, which is an abstraction for representing the interaction between called methods that result from inheritance and polymorphism.

## 4.1 Extended Coupling Definitions

The o*riginal coupling-based* testing definitions of Jin and Offutt require a number of modifications to account for the various calling contexts that occur in object-oriented programs [38]. In the following definitions, *m* refers to a program unit, including methods that appear in class specifications. $V_m$ is the set of variables that are referenced by *m*, and $N_m$ the set of nodes in *m*. A node of a method is either the method's entry node, it's exit node, or corresponds to one of the method's statements. Each definition is expressed as a function whose domain is given by a possibly empty set of formal arguments and a range given as a return type. Note that use of the notation $\mathbb{P}$***T*** represents a set of elements each of which is an instance of *T*, where *T* is the name of some type (e.g. integer, class, variable, etc.).

- ***def-clear-path (i, j, v) : Boolean*** : True if there is a definition-clear path from node *i* to *j* with respect to *v*, except possibly at node *i*.
- ***def-clear-path (p, v) : Boolean*** : True if path *p* is definition-clear with respect to *v*, except possibly at *first(p)*.
- ***defs*(*i*) :** $\mathbb{P}$***Variable*** : The set of variables that are defined at node *i*.

- ***uses(j)* : ℙ*Variable*** : The set of variables that are used at *j*.
- ***entry(m)* : *Node*** : The entry node of method *m*.
- ***exit(m)* : *Node*** : The exit node of method *m*.
- ***family(c)* : ℙ*Class*** : The set of classes that belong to the type family specified by class *c*. Note that *c* itself is a member of *family(c)*.
- ***first(p)* : *Node*** : The first node in path *p*.
- ***state(c)* : ℙ*Variable*** : The set of variables that directly or indirectly comprise the state space of class *c*.
- ***type(m)* : *Class*** : The class whose specification contains member *m*.
- ***type(o)* : *Class*** : The class *C* that is the declared type of the variable *o*, where *o* is a reference to an instance of some class that is a member of the type family induced by *C*.
- ***i-defs(m)* : ℙ*Variable*** : The set of variables in the state space of the class containing *m* that are indirectly *defined* by a call to *m* made through some instance context. Formally:

  $$i\text{-}defs(m) = \{ v \in state\text{-}vars(type(m)) \mid \exists j \in N_m \bullet v \in defs(j) \}$$

- ***i-uses (m)* : ℙ*Variable*** : The set of variables in the state space of the class containing *m* that are indirectly *used* by a call to *m* made through some instance context. Formally:

$$i\text{-}uses(m) = \{v \in state(type(m)) \mid \exists j \in N_m \bullet v \in uses(j)\}$$

- ***instance*(*t*) : *object*** : Returns an instance (object) of type *t*.
- ***method*($s_{j,k}$) : *Method*** : Returns the method that contains coupling sequence $s_{j,k}$.
- ***signature*(*m, n*) : *Boolean*** : Evaluates to true if the signature of methods *m* and *n* match.
- ***overrides*(*m, p*) : *Boolean*** : Evaluates to true if method *m* is an overriding method of *p*. That is,

  $overrides(m, p) \Leftrightarrow type(m) \in family(type(p)) \wedge signature(m, p)$.
- ***paths*(*i, j, m*) : $\mathbb{P}$*Path*** : The set of paths that emanate from node *i* and that are incident upon node *j*, where $i, j \in N_m$.

## 4.2 Coupling Sequences

Coupling sequences are pairs of nodes within the body of a specific method *f* that correspond to an indirect coupling of state variables through a common instance context that is accessed through an object reference. There are four structural types of coupling sequence that are of interest in coupling-based testing of object-oriented programs. Each is illustrated by a *control flow schematic*, such as the one depicted in Figure 4-1. The schematic abstracts away the details of control flow graph and shows only those nodes that are of interest from a coupling analysis perspective. Individual methods are represented as shaded rectangles that enclose their respective control flow. Method entry and exit nodes are depicted as non-solid and solid ellipses, and individual statements as solid circles. Method call sites are depicted as a smaller solid circle within a larger non-solid circle, and their corresponding return sites are non-solid circles. Control flow is represented as undirected thin line segments connecting two nodes. Thicker undirected line segments indicate control flow that is part of a particular coupling path. Each line segment is considered to represent one or more

sub-paths that connect two nodes. A path may be annotated with a *transmission set*, which consists of variables that the path is definition-clear with respect to. These variables are listed inside a set of brackets, such as [*o*,*o.v*], which indicates that the particular path bearing this annotation is definition-clear with respect to the variables *o* and *o.v*.

The following subsections discuss each of the four types of coupling sequence in detail.

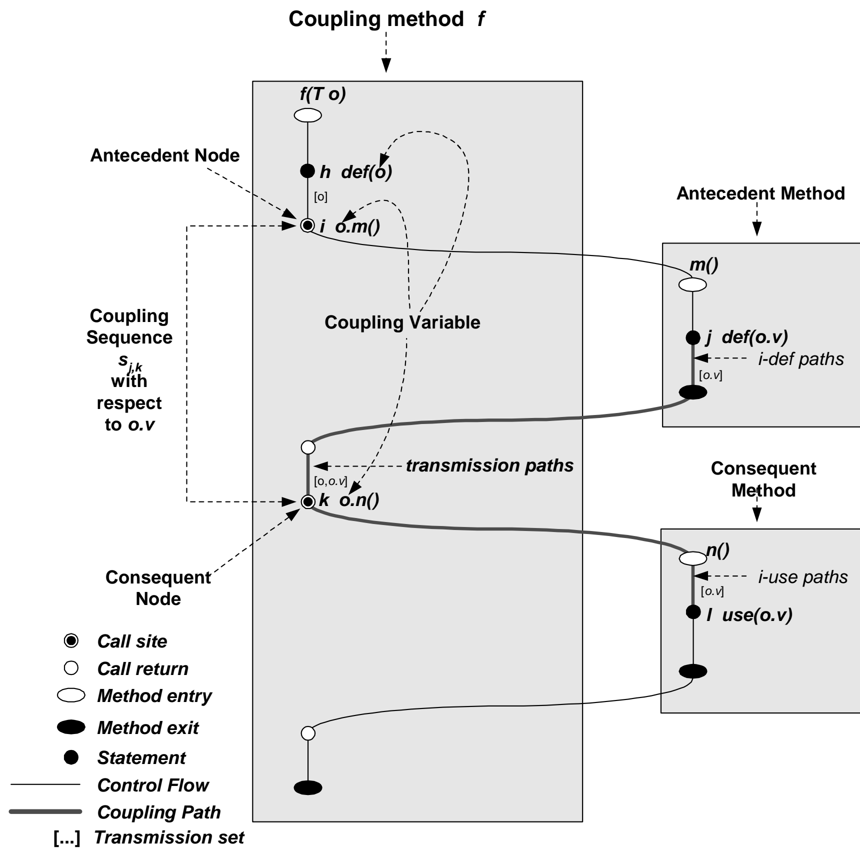



**Figure 4-1.** Type I Coupling Sequence

### 4.2.1 Type I Coupling Sequences

The first coupling type is illustrated by the *control flow schematic* depicted in Figure 4-1. Structurally, this type of coupling sequence is represented by calls to two different methods through the same instance context. As the schematic shows, method *f*, referred to as the *coupling method*, contains a coupling sequence $s_{j,k}$ that starts at node *j* with the call to *o.m*

(the *antecedent method*) and extends through paths that end at node $k$ where the sequence ends with the call to $o.n$ (the *consequent method*). The nodes containing the antecedent method and consequent method are referred to as the *antecedent node* and *consequent node*. Note that there is at least one path between the call sites that is definition-clear with respect to $o$. There is also one sub-path in which $o$ is definition-clear with respect to the indirect definitions made in the antecedent method that have corresponding indirect uses in the consequent method. The identifier $o$ is referred to as the *context variable* and such paths as these are referred to as *transmission paths* (*t-paths*).

Formally, a Type I coupling sequence $s_{j,k}$ is given by the 9-tuple in Equation 4-1, where $f$ is the coupling method that contains the coupling sequence; $o$ is the context variable of the sequence and $T$ is the coupling type (i.e. the type of instance bound to $o$ – the declared type of $o$ in this case). $j, k \in N_f$ are the antecedent and consequent nodes, respectively; $m$ and $n$ are the antecedent and consequent methods used at the call sites at $j$ and $k$, $\Theta_{s_{j,k}}$ is the set of variables defined by $m$ and used by $n$, and $\Pi_{s_{j,k}}$ is the set of transmission paths between $j$ and $k$ with respect to $\Theta_{s_{j,k}}$.

**Equation 4-1.** Type I coupling sequence

$$s_{j,k} = (f, o, T, j, k, m, n, \Theta_{s_{j,k}}, \Pi_{s_{j,k}})$$

### 4.2.2 Type II Coupling Sequences

A Type II coupling sequence has the structure depicted in Figure 4-2. As shown, the antecedent node $i$ contains an indirect definition through the object reference $o$ and the corresponding indirect use occurs at the consequent node $j$ through the call to the consequent method $m$. The coupling set for this sequence is $\Theta_{s_{j,k}} = \{t::v\}$ and is given formally by $\Theta_{s_{j,k}} = i\text{-}defs(i) \cap i\text{-}uses(m)$, where $i$ is the antecedent node and $m$ is the consequent method called at the consequent node $j$.

Coupling paths are formed by combining elements of the *t-path* set from the coupling method *f* with elements of the *i-use* set from the consequent method. Thus, the coupling sequence $s_{j,k}$ extends from the antecedent node *i* through the call site at the consequent node *j*, and through the entry node of *m* to node *l* that contains the first-use of *v* in *m*. Note that a Type II coupling sequence does not have an antecedent method, and hence has no *i-def* paths.

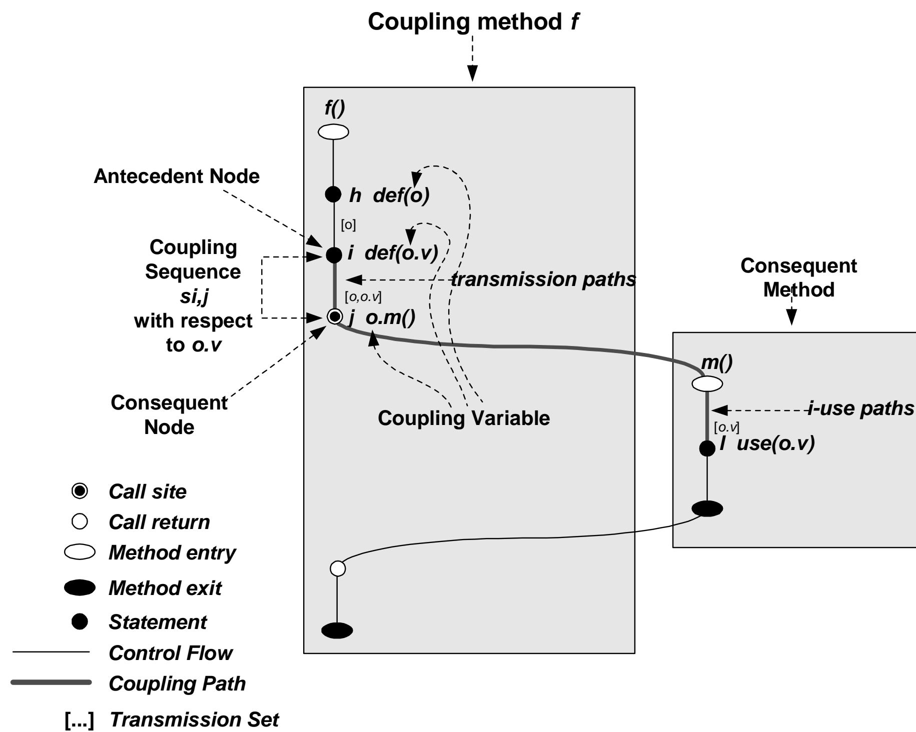



**Figure 4-2.** Type II Coupling Sequence

### 4.2.3 Type III Coupling Sequences

Type III coupling sequences are similar to Type II sequences in that the reference to a coupling variable occurs through a single method call and object reference. However, the difference is that the roles are reversed. That is, the indirect definition is made by the method and the indirect use through the object reference. This is depicted in Figure 4-3 which illus-

trates the structure of the Type III coupling sequence $s_{j,k}$. As shown, the antecedent node *j* in the coupling method *f* contains a call to the antecedent method *m*. In *m*, node *k* contains a definition of the coupling variable *v*. The corresponding indirect use occurs at the consequent node *l* back in *f*. Thus, the set of coupling paths extend from node *k* in *m* to node *l*. Each coupling path is formed by combining elements of the *i-def* set of *m* with the *t-path* set of *f*. Note that a Type III coupling sequence does not have a consequent method, and hence has no *i-use* paths.

The coupling set for a type III coupling sequence is given formally by $\Theta_{s_{j,l}} = i\text{-}defs(m) \cap i\text{-}uses(l)$ where *m* is the antecedent method called at the antecedent node *j* and *l* is the consequent node *l*.

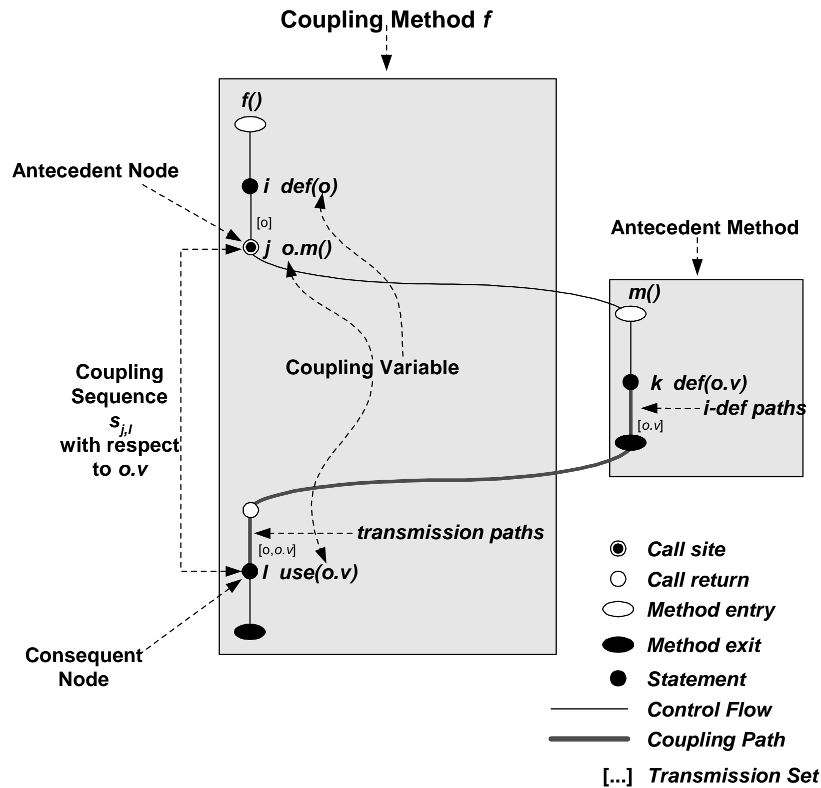



**Figure 4-3.** Type III Coupling Sequence

### 4.2.4 Type IV Coupling Sequences

The fourth type of coupling sequence is shown in Figure 4-4. Here, the coupling sequence occurs between two nodes in the coupling method in which both the indirect definition and use occurs through the instance context provided by *o*. Every coupling path in the sequence is identical to the set of transmission paths between the antecedent and consequent nodes. Thus, there are no *i-def* or *i-use* paths in a Type IV coupling sequence. Also, there are no call sites at either node in the sequence.

The coupling set for a type IV coupling sequence is given formally by $\Theta_{s_{j,k}} = i\text{-}defs(j) \cap i\text{-}uses(k)$, where *j* and *k* are the antecedent and consequent nodes, respectively. Type IV coupling sequences are not discussed further in this thesis as they are

covered by traditional data flow testing criteria [30, 65]. They are included here merely for the sake of completeness.

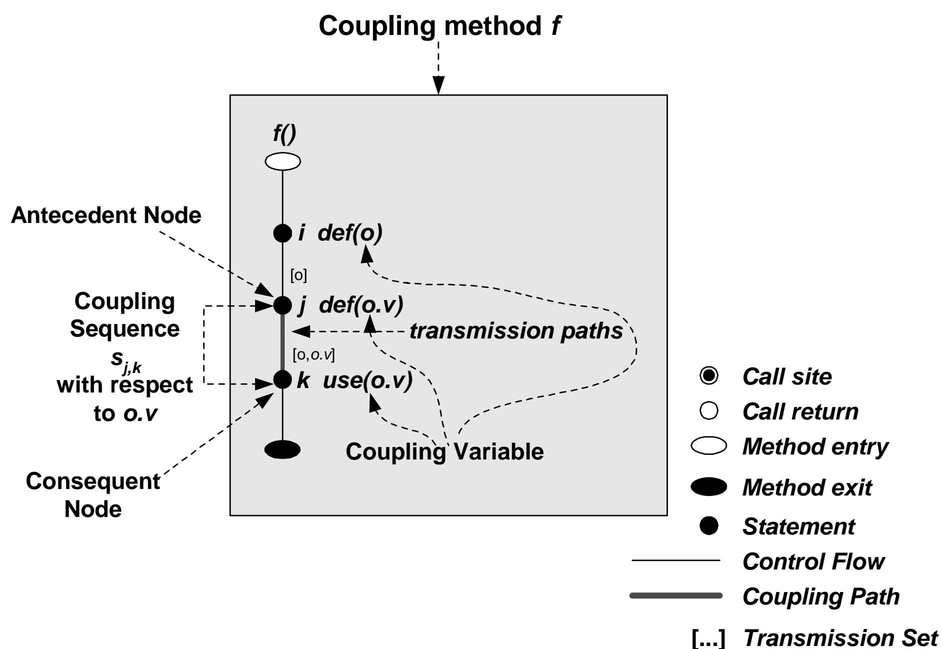



**Figure 4-4.** Type IV Coupling Sequence

### 4.2.5 Other Type of Coupling Sequences

The research presented in this thesis focuses on coupling sequences where the calls to the antecedent and consequent methods are both in the coupling method. There are other more complicated situations where coupling sequences occur in object-oriented programs. In essense, all of these situations can be characterized as the calls to the antecedent and consequent methods do not both occur in the coupling method. An example of this is illustrated in Figure 4-5 where the coupling method *f* does not contain either call. Instead, the call sites *j* and *k* invoke methods that contain the calls to the antecedent and consequent methods. Though the coupling between the antecedent and consequent methods does not occur directly from *f*, the coupling sequence $s_{j,k}$ exists in *f* between nodes *j* and *k*, but is referred to as an *indirect coupling sequence* (alternatively an *inter-method coupling sequence*) of *f*.

Indirect coupling sequences are not discussed further in this thesis, but are instead are left for future research.

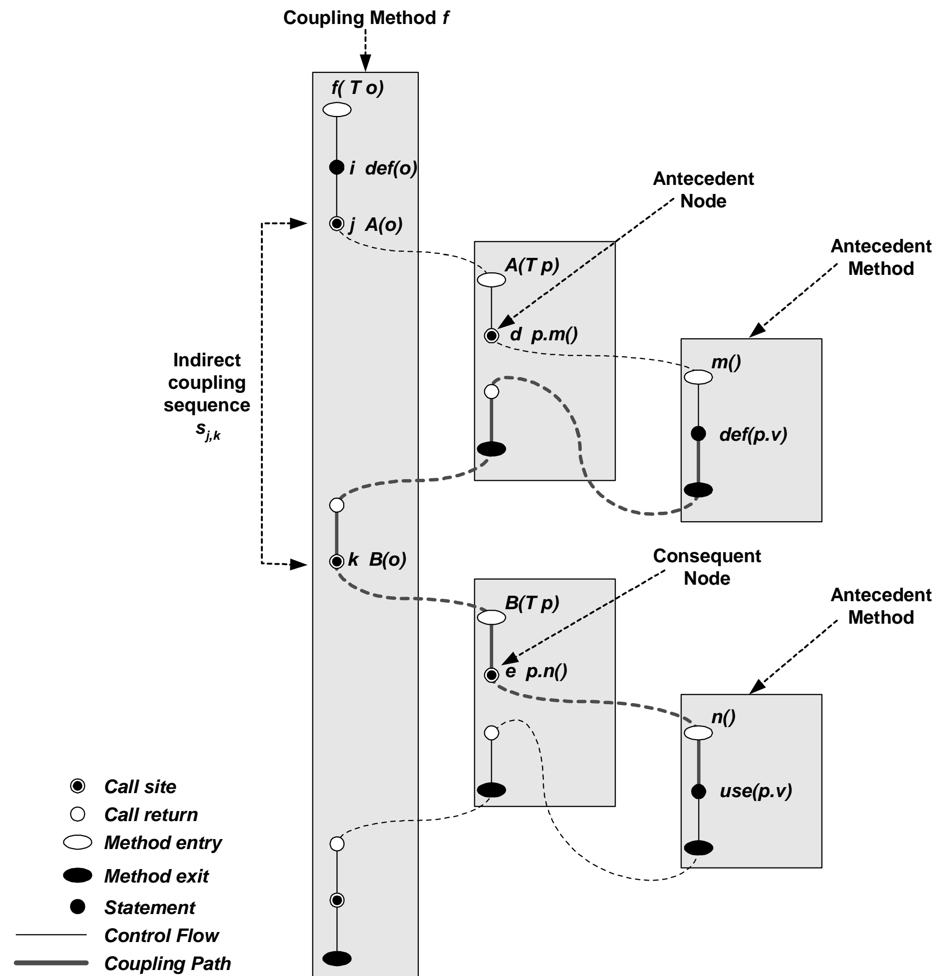



**Figure 4-5.** Inter-method Coupling Sequences

## 4.3 Coupling Variables and Coupling Sets

Every coupling sequence $s_{j,k}$ has an associated set of state variables that are defined by the antecedent method and subsequently used by the consequent method with respect to the

coupling type *t*. This set of variables is referred to generically as the *coupling set* $\Theta^{t}_{s_{j,k}}$ of $s_{j,k}$ and is defined as the intersection of those variables defined by *m* (an *indirect-def*, or *i-def*) and used by *n* (an *indirect use*, or *i-use*) through the instance context provided by a context variable *o* that is bound to an instance of *t*. Note that the particular *m* and *n* that execute are determined by the actual type *t* of the instance bound to *o*. Each member of this set is called a *coupling variable*. Coupling sets are formally defined by Equation 4-2.

**Equation 4-2.** Coupling Sets

$$\Theta^{t}_{s_{j,k}} = i\text{-}defs(o.m) \cap i\text{-}uses(o.n)$$

For a Type I coupling sequence, the indirect definition occurs in the antecedent method called at node *j* and the indirect use occurs in the consequent method called at node *k*. For example, the coupling set for the sequence $s_{j,k}$ shown in Figure 4-1 is: $\Theta_{s_{j,k}} = \{t::v\}$ and the definition of the coupling variable *v* occurs in the antecedent method *m*, and the corresponding indirect use occurs in the consequent method *n*.

# 4.4 Coupling Paths

Each coupling sequence has an associated set of paths, called *coupling paths*, in which an indirect definition of a variable *v* is *transmitted* to the corresponding indirect use. That is, the path between the nodes having the indirect definition and indirect use is definition-clear with respect to *v*, thus the definition of *v* is transmitted by the path.

Each path consists of up to three sub-paths, or segments: indirect-def sub-paths, indirect-use sub-paths, and transmission sub-paths. The *indirect-def sub-path* is the portion of the coupling path that occurs in the antecedent method *a*, extending from the last (indirect) definition of a coupling variable to the exit node of *a*. Similarly, the *indirect-use sub-path* is the portion of the consequent method *c* that extends from the entry node of *c* to the first (indirect) use of a coupling variable. Finally, the *transmission sub-path* is the portion of the coupling path that extends from the antecedent node to the consequent node, such that the

value of the path's coupling variable and the coupling sequence's context variable is transmitted without redefinition.

The type of sub-paths that each coupling path has is determined by the mechanism used to affect the indirect definitions and uses of the coupling sequence. A Type I coupling sequence has coupling paths that contain all three types of sub-path since the indirect definition occurs in the antecedent method and the indirect use occurs in the consequent method. In contrast, coupling paths of Type IV coupling sequences only have a transmission sub-path since the indirect definition and use occurring in the coupling method itself. The coupling paths of Type II sequences have a transmission sub-path and an indirect-use sub-path. There is no indirect-def sub-path since the indirect definition occurs in the coupling method. A Type III coupling sequence is just the opposite of a Type II sequence. The coupling paths of a Type III sequence includes an indirect-def sub-path and a transmission sub-paths, but not a indirect-use sub-path.

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

The sets of coupling sub-path segments form the foundation of the source code analysis used to identify coupling sequences in object-oriented programs. Each of these sets is described in detail in the sub-sections below.

The following definitions are used to provide access to the individual components of the coupling sequence $s_{j,k}$:

- ***a-node*(*$s_{j,k}$*) : *Node***: The antecedent node $j$ of $s_{j,k}$.

- ***c-node*($s_{j,k}$) : *Node***: The consequent node $k$ of $s_{j,k}$.
- ***a-method*($s_{j,k}$) : *Method***: The antecedent method $m$ of $s_{j,k}$.
- ***c-method*($s_{j,k}$) : *Method*** : The consequenct method $n$ of $s_{j,k}$.
- ***context*($s_{j,k}$) : *Variable*** : The variable that contains an object reference that refers to an instance providing context at the call sites of $s_{j,k}$ (e.g. *o.m*()).

### 4.4.1 I-Def Paths

For a given coupling sequence $s_{j,k}$, there are a set of paths in the antecedent method $m$ that begin at nodes that have *last-definitions-before-return* (see Section 2.6.1 on page 43) of the variables contained in the coupling set $\Theta_{s_{j,k}}$. For each such node $l$, the path to *exit*($m$) is definition-clear with respect to the corresponding coupling variable defined at $l$. These paths constitute the *indirect-def path set* (or *i-def-path-set*) of the coupling sequence. Each of these paths is referred to as an *indirect definition path* (or *i-def-path*). Equation 4-3 gives the formal definition for the set *i-def-paths*, where $m$ is the consequent method of $s_{j,k}$ and $V$ is a subset of the coupling variables for $s_{j,k}$.

**Equation 4-3.** *Indirect def path set*

$$\begin{aligned} i\text{-}def\text{-}paths(m, V) = \{(p, v) \mid\ & first(p), last(p) \in N_m \wedge \\ & p \in paths(first(p), last(p), m) \wedge \\ & last(p) = exit(m) \wedge \\ & v \in V \subseteq state(class(m)) \wedge \\ & v \in defs(first(p)) \wedge \\ & def\text{-}clear\text{-}path(first(p), last(p), v)\} \end{aligned}$$

Figure 4-6 shows an example of the i-def-paths for the coupling sequence $s_{j,k}$. As the figure shows, there are three *i-def-paths* (labeled $a$, $b$, and $c$) that emanate from nodes in *o.m* such that each *i-def-path* has a *last-definition-before-return* of the coupling variable *o.v*. As indicated, each of these paths is definition-clear with respect to the coupling variable *o.v*. Collectively, these paths constitute the *indirect-def path set* for the coupling sequence $s_{j,k}$.

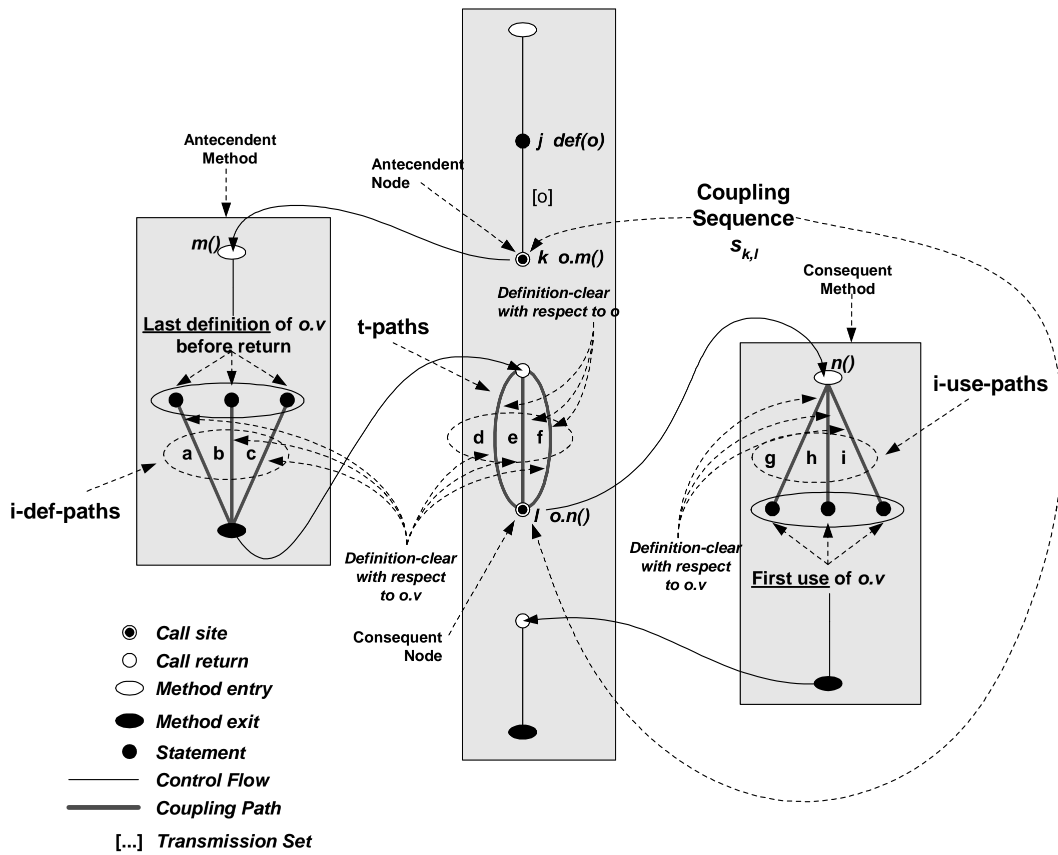



**Figure 4-6.** Detailed Type I Coupling Sequence

## 4.4.2 I-Use Paths

Referring again to Figure 4-6, the set of paths in the consequent method $n$ that begin at *entry*($n$) and end at a node $l \in N_n$ such that $l$ has a *first-use-in-callee* of a coupling variable contained in the state of the class that specifies $n$ is referred to as the *indirect-use path set* (or *i-use-path-set*) of the antecedent method $n$ for coupling sequence $s_{j,k}$. Each path in this set is definition-clear with respect to the particular coupling variable used in the consequent method. Equation 4-4 formally defines *i-use-path*s for a given pair of node and state variable. In the equation, $n$ is the consequent method of $s_{j,k}$ and $V$ is some subset of the coupling variables for $s_{j,k}$.

**Equation 4-4.** *i-use-paths*

$$\begin{aligned} i\text{-}use\text{-}paths(n, V) = \{(p, v) \mid & (first(p), last(p) \in N_p \\ & \wedge p \in paths(first(p), last(p), n) \wedge \\ & first(p) = entry(n) \wedge \\ & v \in V \subseteq state(class(n)) \bullet \\ & v \in uses(last(p)) \wedge \\ & def\text{-}clear\text{-}path(first(p), last((p), v)))\} \end{aligned}$$

In Figure 4-6, the elements of the *i-use-set* for $s_{j,k}$ are $g$, $h$, and $i$. Each of these nodes ends at a node having a first use of $o.v$, and each is definition-clear with respect to $o.v$.

### 4.4.3 Transmission Paths

For every coupling sequence $s_{j,k}$, there is a set of paths $\Pi_{j,k}$ that connect the antecedent node $j$ and the consequent node $k$, such that for each $t \in \Pi_{j,k}$, $t$ is definition-clear with respect to some subset $\theta_{s_{j,k}} \subseteq \Theta_{s_{j,k}}$. These paths transmit the value of each $v \in \theta_{s_{j,k}}$ from where it is indirectly defined by the execution of the antecedent node, to where it is indirectly used by execution of the consequent node. Equation 4-5 presents the formal definition for *t-paths*.

**Equation 4-5.** *T-Paths*($s_{j,k}$)

$$\begin{aligned} t\text{-}paths(s_{j,k}) = \{(p, v) \mid & p \in paths(a\text{-}node(s_{j,k}), \\ & c\text{-}node(s_{j,k}), method(s_{j,k})) \wedge \\ & v \in \Theta_{s_{j,k}} \bullet \\ & def\text{-}clear\text{-}path(first(p), last(p), context(s_{j,k})) \wedge \\ & def\text{-}clear\text{-}path(first(p), last(p), v)\} \end{aligned}$$

The penultimate clause of the set expression in Equation 4-5 requires that the transmission path $p$ be definition-clear with respect to the object reference that defines the context of the coupling sequence, and the last clause requires $p$ to be definition-clear with respect to the coupling variable $v$.

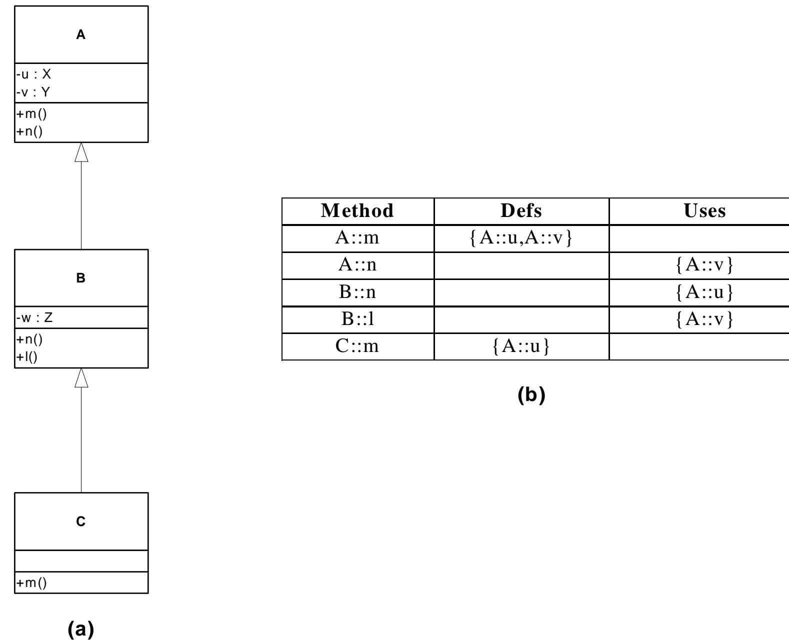
If there is no transmission path between the antecedent node $j$ and the consequent node $k$, then $t\text{-}paths(s_{j,k}) = \varnothing$. In this case, the coupling sequence $s_{j,k}$ is a *vacuous coupling*

*sequence*, and hence the coupling path set of $s_{j,k}$ is empty. Note that this can only occur if there is not at least one definition-clear path from $j$ to $k$ for some $v \in \Theta_{s_{j,k}}$.

In Figure 4-6, the set of paths labeled *T-Paths* consisting of the elements $d$, $e$, and $f$, constitute the transmission path set for the sequence $s_{j,k}$. Note that each path in the transmission set must be definition-clear with respect to the coupling variable that it transmits *and* with respect to the object reference that defines the context of the coupling sequence ($o$ in the example).

## 4.5 The effects of inheritance and polymorphism on coupling

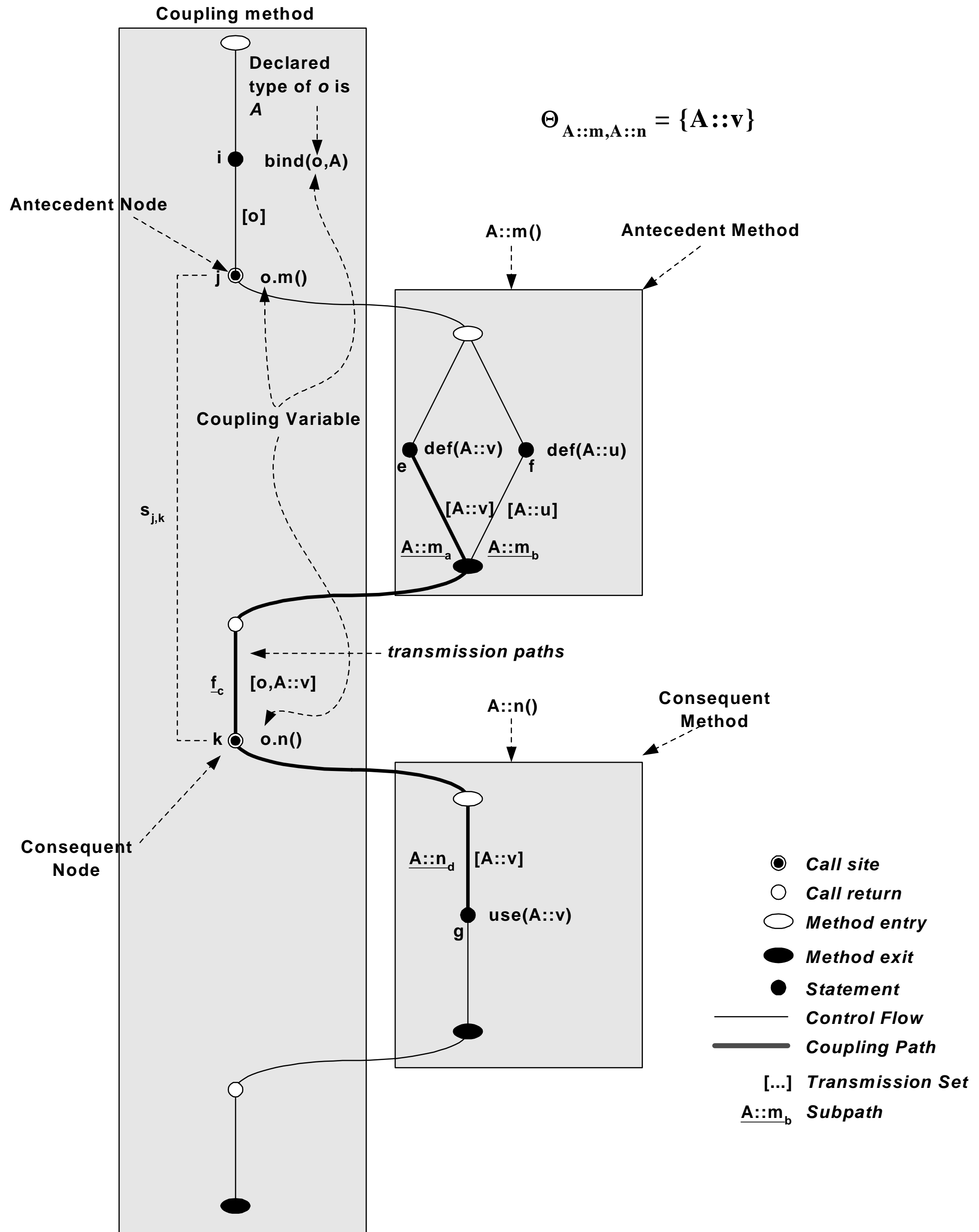
To see the effects of inheritance and polymorphism on path sets, consider the class diagram shown in Figure 4-7a. The type family that corresponds to this hierarchy includes the three classes $A$, $B$, and $C$, with each having one ore more methods or state-variables. Class $A$ defines methods $m$ and $n$ and state variables $u$ and $v$. Class $B$ defines methods $n$ and $l$, where $n$ is an overriding method of $A$'s $n$ (*A::n*). Likewise, class $C$ defines method $m$, which overrides *A::m*. The corresponding definitions and uses for each of these methods is shown in Figure 4-7b.




| Method | Defs | Uses |
|---|---|---|
| A::m | {A::u,A::v} | |
| A::n | | {A::v} |
| B::n | | {A::u} |
| B::l | | {A::v} |
| C::m | {A::u} | |

**Figure 4-7.** Sample class hierarchy and *def-use* table

Figure 4-8 shows the coupling paths that result from the hierarchy in Figure 4-7a. Observe that the declared type of the coupling variable provided by $o$ is $A$. The coupling sequence $s_{j,k}$ extends from the node $j$ where the antecedent method $m$ is called, to the call site of the consequent method at node $k$. As shown, the corresponding coupling set for $s_{j,k}$ when $o$ is bound to an instance of $A$ is $\Theta^{A}_{s_{j,k}} = \{A::v\}$. Thus, the set consists of the coupling paths for $s_{j,k}$ that extend from node $e$ in $A::m$ to the exit node of in $A::m$, back to the consequent node $k$ in the coupling method, and through the entry node of $A::n$ to node $g$. There is no coupling path with respect to $A::u$ because $A::u$ does not appear in the coupling set for $A::m$ and $A::n$.




$$\Theta_{A::m,A::n} = \{A::v\}$$

**Figure 4-8.** Coupling sequence when *o* is bound to an instance of *A*

Now, consider the affect on the elements that comprise the set of coupling paths when *o* is bound to an instance of *B*, as shown in Figure 4-9. The coupling set for this case is different from when *o* was bound to an instance of *A*. This is because *B* provides an overriding method *B::n* that has a different use set than the overridden method *A::n* has. Thus, the coupling set is different with respect to the antecedent method *A::m* and the consequent method *B::n* yielding $\Theta^{B}_{s_{j,k}} = \{A::u\}$. In turn, this results in a different set of coupling paths as depicted by the Figure 4-9. The set of coupling paths now extend from node *f* in *A::m* back through call site at node *k* in the coupling method and through the entry node of *B::n* to node *g* of *B::n*.

Figure 4-10 depicts the coupling sequence that results when *o* is bound to an instance of *C*. First, observe that execution of the node *j* in the coupling method results in the invocation of the antecedent method, which is now *C::m*. Likewise, execution of node *k* results in the invocation of the consequent method *n*. Since *C* does not provide an override for *m*. and because *C* is a descendant of *B*, the version of *n* that is invoked is actually *B::n*. Thus, the coupling set for $s_{j,k}$ is taken with respect to the antecedent method *C::m* and the consequent method *B::n*, which yields $\Theta^{C}_{s_{j,k}} = \{A::u\}$. The corresponding coupling path set includes those paths that begin at node *e* in *C::m* and extend to the exit node of *C::m*, then back node *j* of the coupling method, and through the entry node of *B::n* to node *g*, also in *B::n*.

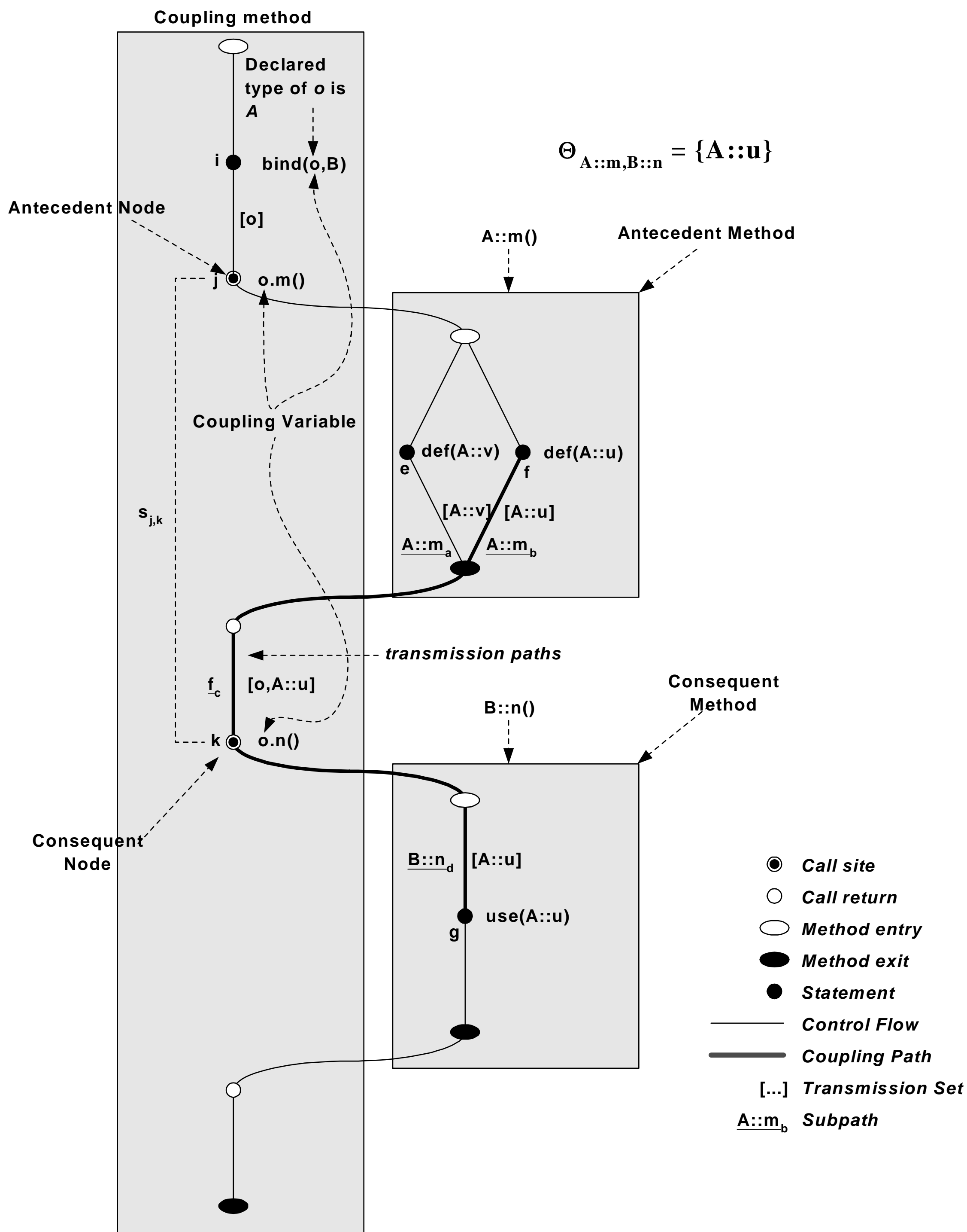



**Figure 4-9.** Coupling sequence when *o* is bound to an instance of *B*

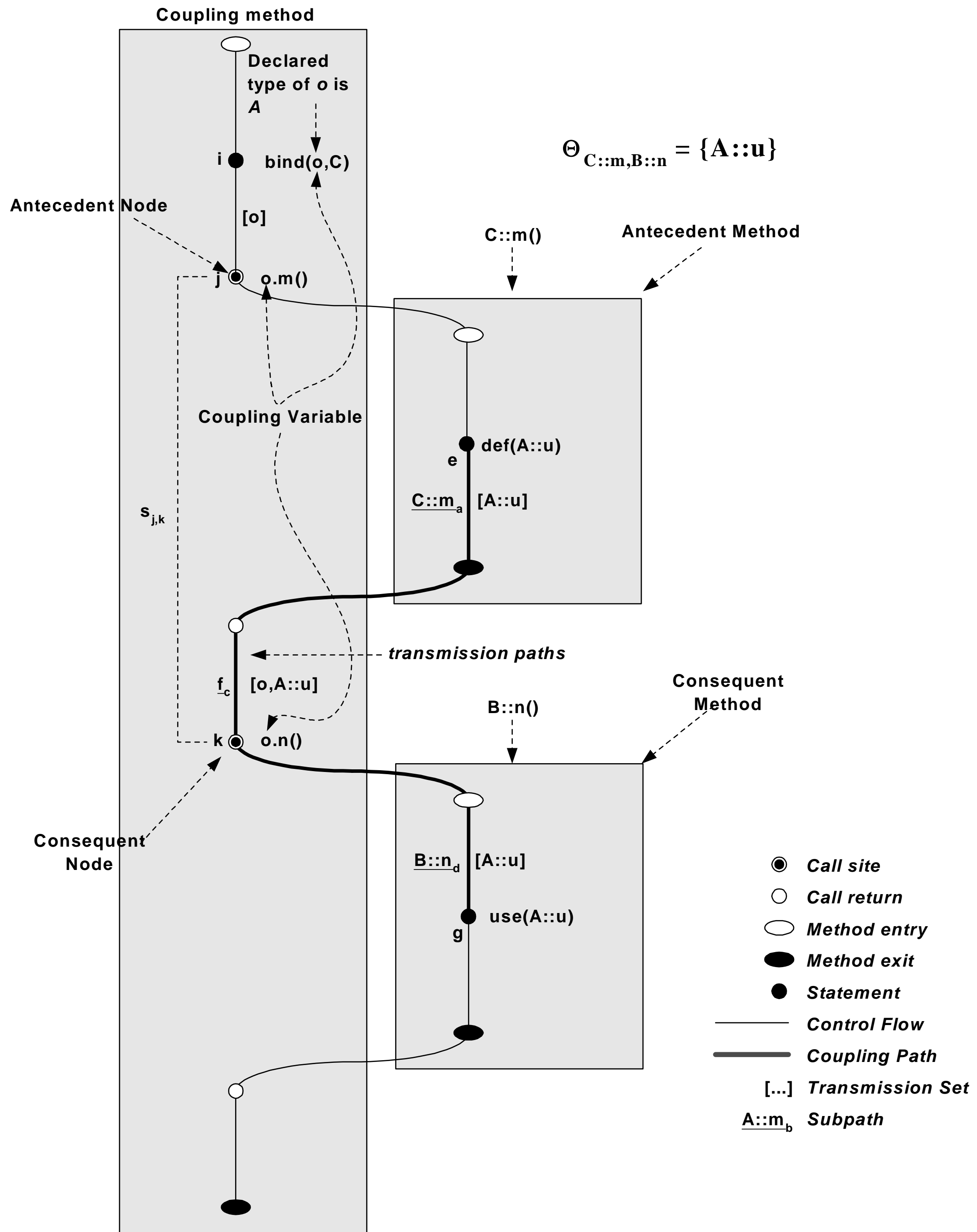



**Figure 4-10.** Coupling sequences when *o* is bound to an instance of *C*

Not every class *c* in a particular type family will have methods that participate in a coupling sequence as a result of inheritance and polymorphism for a particular coupling method. This will be true whenever *c* does not provide overriding definitions for methods that are invoked by the antecedent or consequent nodes. For example, suppose a new class *D* that has no overriding methods used in the coupling sequence shown in Figure 4-8 is added to the sample class hierarchy given above, as illustrated in Figure 4-11. In this situation, whenever *o* is bound to an instance of *D*, the coupling sequence that results will be the same as when *o* is bound to an instance of *A*. This is because the antecedent and consequent methods that execute (*m* and *n*, respectively) are those defined by *A*. Because of this, we can safely ignore consideration of *D* from the coupling analysis.

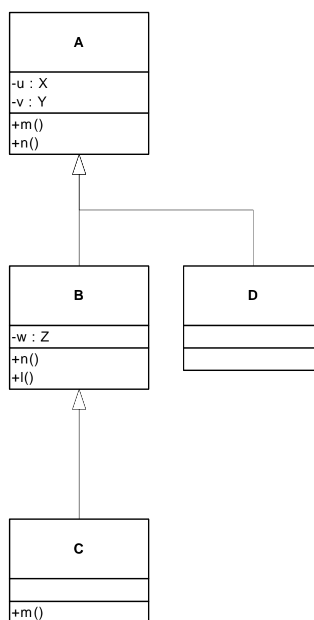



**Figure 4-11.** Sample hierarchy with class *D* added

Table 4-1 summarizes the coupling paths for the example shown in Figure 4-6 on page 104 and the corresponding control flow schemata depicted in Figure 4-7, Figure 4-8, and Figure 4-9. Paths are represented as a sequence of comma delimited nodes. Each node is of the form *method(node)*, where *method* is the name of the method that contains the node,

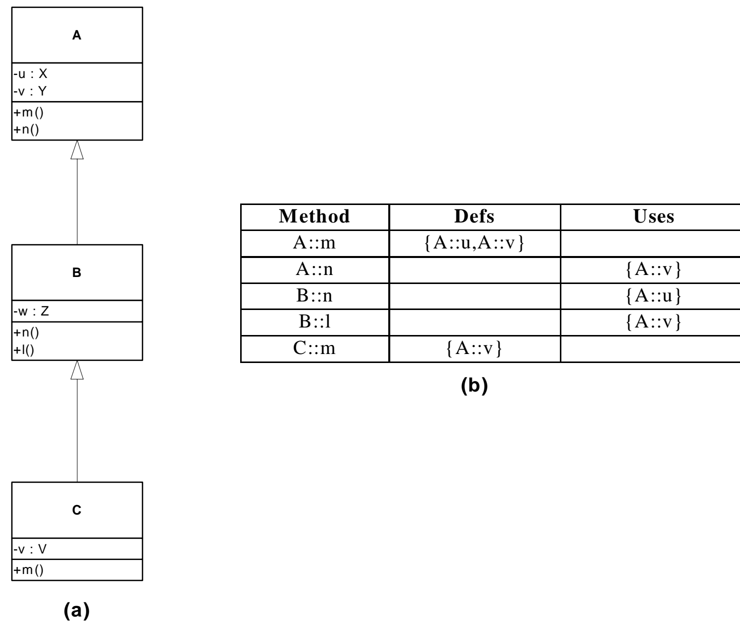
and *node* is the node identifier within the method. Note that the prefixes "*call*" or "*return*" are appended to the names of nodes that correspond to call or return sites.

**Table 4-1.** Summary of sample coupling paths

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

Another situation can exist in which a class *c* in a particular type family will not have methods participating in a coupling sequence. This will occur when *c* provides a definition of an overriding method that does not define the same set of coupling variables as the overridden. For example, suppose that class *C* has been changed such that its overriding definition for *m* no longer defines the same set of coupling variables as does the overridden method *A::m*, as shown in Figure 4-12a and Figure 4-12b. The on the coupling sequence $s_{j,k}$ is shown in Figure 4-13. As the figure shows, there are no coupling paths for $s_{j,k}$ with respect to *C::m* and *B::n*, and thus, $s_{j,k}$ is a vacuous coupling sequence whenever *o* is bound to an instance of *C*. This is because the coupling set is empty with respect to *C::m* and *B::n* is empty,

$\Theta^{C}_{s_{j,k}} = \varnothing$. That is, there are no coupling variables defined by *C::m* that are subsequently used by *B::n*.




| Method | Defs | Uses |
|---|---|---|
| A::m | {A::u,A::v} | |
| A::n | | {A::v} |
| B::n | | {A::u} |
| B::l | | {A::v} |
| C::m | {A::v} | |

**Figure 4-12.** Sample hierarchy showing modified class *C*

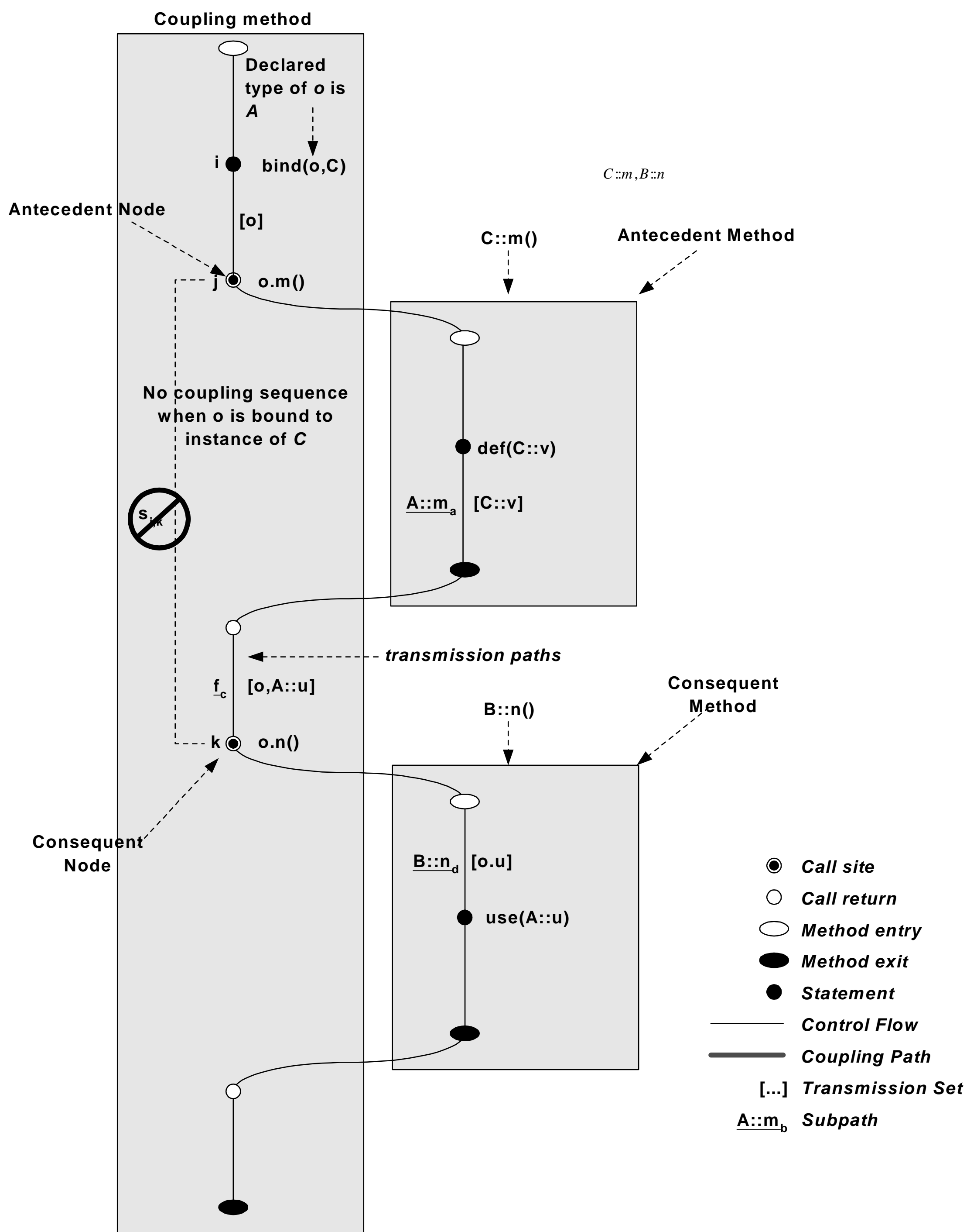



**Figure 4-13.** Coupling sequence where o is bound to C and $defs(C::m) \neq defs(A::m)$

Pragmatically, the effects of inheritance and polymorphism have the potential to result in a combinatorial explosion of path sets. The number of path sets is a function of the depth and breadth of the inheritance hierarchy. However, as an optimization to reduce the number, as noted earlier, those types that do not have overriding methods will have an empty type set. This is possible since any coupling path that could be executed through the type will necessarily appear in the path sets of other ancestor types that are members of the same type family.

## 4.6 Polymorphic coupling sequences and coupling sets

Inheritance increases the number of possible bindings that can occur for a given coupling sequence. As a result of this, the actual methods that execute and the variables indirectly defined and used can vary at runtime. Since the depth and breadth of inheritance is always finite, there is an upper limit on the amount of variation that can occur at runtime. In the worst case, every member of the type family that corresponds to the coupling type can provide an instance context for the coupling sequence.

The following subsections present modified definitions of coupling sequences and coupling sets that take polymorphism into account.

### 4.6.1 Polymorphic Coupling Sequences

To account for the possibility of polymorphic behavior at a call site, the definition of a coupling sequence given by Equation 4-1 must be amended to handle all methods that can possibly execute. To accomplish this, we introduce the notion of a *binding triple*. A binding triple for a coupling sequence consists of the antecedent node $m$, the consequent node $n$, and the set of coupling variables that result from the binding of the context variable to an instance of a particular type. The triple matches together a pair of methods $p$ and $q$ that can potentially execute as the result of an invocation at the call sites of the antecedent and consequent nodes $j$ and $k$. Neither of these methods is required to be from the class $c$ that provides the instance context for the coupling sequence. Each may be from different classes that are members of the type family defined by $c$, and provided that $p$ is an overriding

method for $m$ or $q$ is an overriding method for $n$. Note that there will be exactly one binding triple for each class $d \in family(c)$ that defines an overriding method for either $m$ or $n$. Classes that do not define such overriding methods are excluded.

To modify the coupling sequence definition, we add a set of binding triples $\Phi_{j,k}$ into the definition of a Type I coupling sequence. This set is the extension of all possible binding triples for a given coupling sequence. Thus, the new definition for a coupling sequence becomes:

**Equation 4-6.** Polymorphic coupling sequence

$$s_{j,k} = (f, m, n, o, \Phi^t_{j,k}, \Pi_{j,k}, \Theta^t_{j,k})$$

where $f$, $m$, $n$, $o$, and $\Pi_{j,k}$ are as defined for Equation 4-1, $t$ is the type of instance bound to $o$, $\Theta^t_{j,k}$ is the set of coupling variables for the sequence, and $\Phi^t_{j,k}$ is the set of binding triples for $s_{j,k}$.

The definition of the set of binding triples for a Type I coupling sequence is given in Equation 4-7.

**Equation 4-7.** Type I binding triple sets

$$\begin{aligned}\Phi(o, m, n) = \{(p, q, \Theta^t_{p,q}) \mid\, & type(p) \in family(type(o)) \\ & \wedge type(q) \in family(type(o)) \bullet \\ & (p = m \vee overrides(p, m)) \wedge \\ & ((q = n \vee overrides(q, n)) \wedge \Theta^t_{p,q} = i\text{-}defs(p) \cap i\text{-}uses(q))\}\end{aligned}$$

where $t$ is the actual type of the instance bound to $o$ and $t \in family(type(o))$.

Equation 4-7 accounts for the possibility of polymorphic behavior in a coupling sequence, and is interpreted as follows:

1. The classes that define methods $p$ and $q$ are members of the type family defined by the declared type of $o$.

2. Method $p$ is either the antecedent method $m$, or is a method that overrides $m$. Likewise, method $q$ is either the consequent method $n$, or is a method that overrides $n$.

3. The coupling set $\Theta^t_{p,q}$ is the intersection of the set of state variables that are indirectly defined by $p$ and subsequently indirectly used by $q$.

Note that for a Type I coupling sequence the set induced by $\Phi$ will never be empty. There will always be at least one binding triple that corresponds to the antecedent and consequent methods. This occurs when the type family defined by the context of the coupling sequence does not contain any members that provide overriding methods for both the antecedent and consequent methods. Thus, the only member of the binding triple set will correspond to the declared type of the context variable, assuming that the type is not abstract. If the type is abstract, then an instance of the nearest concrete descendant to the declared type is used.

As an example, the set of binding triples $\Phi(o, A{::}m, A{::}n)$ for the coupling sequence $s_{j,k}$ shown in Figure 4-8 on page 108, Figure 4-9 on page 110, and Figure 4-10 on page 111 are given in Table 4-2. The type hierarchy corresponding to the coupling type is shown in Figure 4-7 on page 107.

**Table 4-2.** Binding triples for $\Phi(o, A{::}m, A{::}n)$

| $t$ | $p$ | $q$ | $\Theta^t_{p,q}$ |
|---|---|---|---|
| $A$ | $A{::}m$ | $A{::}n$ | $\{A{::}v\}$ |
| $B$ | $A{::}m$ | $B{::}n$ | $\{A{::}u\}$ |
| $C$ | $C{::}m$ | $B{::}n$ | $\{A{::}u\}$ |

Type II and III coupling sequences have definitions that are slightly different, as given by equations Equation 4-8 and Equation 4-9, respectively. In Equation 4-8, $j$ is the antecedent

node and $n$ the consequent method, and in Equation 4-9 $m$ is the antecedent method and $k$ the consequent node. As with Type I coupling sequences, there will also be at least one binding triple that corresponds to the antecedent node and consequent method, and vice-versa for Types II and III, respectively.

**Equation 4-8.** Type II binding triple sets

$$\Phi(o, j, n) = \{(j, q, \Theta^{t}_{j,q}) \,|\, type(q) \in family(type(o)) \bullet ((q = n \vee overrides(q, n)) \wedge \Theta^{t}_{j,q} = i\text{-}defs(j) \cap i + i\text{-}uses(q))\}$$

**Equation 4-9.** Type III binding triple sets

$$\Phi(o, m, k) = \{(p, k, \Theta^{t}_{p,k}) \,|\, type(p) \in family(type(o)) \bullet ((p = m \vee overrides(p, m)) \wedge \Theta^{t}_{p,k} = i\text{-}defs(p) \cap i\text{-}uses(k))\}$$

### 4.6.2 Polymorphic Coupling Sets

The original definition of a coupling set for a coupling sequence only considers the antecedent and consequent methods (for a Type I sequence) in the context of the declared type of the coupling variable. This is no longer sufficient since inheritance and polymorphism can result in different methods being executed. Instead, the coupling sets of all permissible method combinations must be combined to form an aggregate coupling set for the sequence. Thus, the coupling set $\Theta_{s_{j,k}}$ is defined as the union of all the coupling sets for each binding triple in $\Phi_{s_{j,k}}$. Formally, $\Theta_{s_{j,k}}$ is given by Equation 4-10, where $\Theta_t$ is the coupling set $\Theta^{t}_{p,q}$ for binding triple $t$. The coupling set for a sequence is the union of all the coupling sets for the individual pairs of methods that could potentially execute through the call sites at the antecedent and consequent nodes.

**Equation 4-10.** Polymorphic coupling set

$$\Theta_{s_{j,k}} = \bigcup_{t \in \Phi_{j,k}} \Theta_t$$

## 4.7 Coupling paths in object-oriented programs

In their original work, Jin and Offutt were concerned with couplings that occur between procedures in terms of parameters explicitly passed as arguments or through shared global data [38]. In an object-oriented program, other cases exist that are also of interest. In particular, we care about those paths that have couplings originating at last definitions in an antecedent method and that terminate at first uses in a consequent method.

There are two general cases in which coupling paths can occur. The most basic is where there is no possibility of polymorphic behavior at the call sites of a coupling sequence. In this case, the methods that execute are specified by the declared type of the context variable. The most complex case is where there is a possibility of polymorphic behavior at the call sites. As a consequence, it is not possible to determine statically which methods will execute. At best, an approximation can be obtained by considering all possible types of instances that can be bound to the context variable. The following subsections discuss the coupling paths that result from each of these cases.

### 4.7.1 Non-Polymorphic Coupling Paths

Consider again the Type I coupling sequence shown in Figure 4-1 on page 94 where in the body of method *f* there is an object reference *o* of declared type *T*. Assume that *o* is bound to an instance whose actual type is *T*. In this scenario, there is no possibility of polymorphic behavior. Within *f*, there are two instance couplings at call sites where methods *m* and *r* (specified by *T*), respectively, are called successively through the instance context provided by *o*.[1] These two call sites are members of the coupling sequence $s_{j,k}$, as described in Section 4.2.

Ignoring polymorphism for the moment, we are interested in all of the indirect definitions that can reach indirect uses with respect to a particular instance context. Thus, we desire to identify all *non-polymorphic coupling paths* that extend from a node containing a *last-def-*

---

1. An *instance coupling* occurs wherever an object reference is used to access methods or state variables of an instance.

*before-return* in an antecedent method to a node in a consequent method that contains a *first-use-in-callee* with respect to the coupling variable of interest. Collectively, this set of paths is the *coupling path set* for the coupling sequence $s_{j,k}$. We form these paths by taking the cross product of the *i-def path set*, *t-path set*, and *i-use path set* for a particular Type I coupling sequence.

Formally, using the 9-tuple definition for a Type I coupling sequence given in Section 4.2.1, the set of *instance coupling paths* for the coupling sequence $s_{j,k}$ is a set of pairs consisting of a single non-polymorphic coupling path and coupling variable as expressed by Equation 4-11, where $\theta^t_{j,k}$ is a subset of $\Theta^t_{s_{j,k}}$ and $t$ is the actual type of the instance bound to the context variable of $s_{j,k}$. The coupling path is represented as the sequence of coupling path segments $\langle d, t, u \rangle$, where $d$, $t$, and $u$ are the coupling sub-path segments described in Section 4.4. For every $v \in \theta^t_{j,k}$, there is at least one transmission path that is definition-clear with respect to $v$, expressed formally as:

$$\forall v \in \theta^t_{j,k} \bullet \exists t \in \textit{t-paths}(s_{j,k}) \bullet \textit{def-clear-path}(first(t), last(t), v).$$

**Equation 4-11.** *Instance coupling paths for Type I coupling sequences*

$$\begin{aligned} InstanceCouplingPaths(s_{j,k}) = \Big\{ & (\langle d, t, u \rangle, v) \mid v \in \theta^t_{j,k} \subseteq \Theta^t_{s_{j,k}} \wedge \\ & \textit{def-clear-path}(d, v) \wedge \textit{def-clear-path}(t, v) \wedge \\ & \textit{def-clear-path}(u, v) \bullet d \in \textit{i-def-path}(\textit{a-method}(s_{j,k}), \theta^t_{j,k}) \wedge \\ & t \in \textit{t-paths}(s_{j,k}) \wedge u \in \textit{i-use-paths}(\textit{c-method}(s_{s_{j,k}}), \theta^t_{j,k}) \Big\} \end{aligned}$$

Each non-polymorphic coupling path is formed by concatenating a single path $p$ from each of the coupling path segments (*i-def-paths*, *t-paths*, and *i-use-paths*), subject to the constraint that $p$ be definition-clear with respect to a particular coupling variable $v$.

The diagram in Figure 4-6 on page 104 presents an abstract example of non-polymorphic Type I coupling paths. Within method $m$, there are three nodes that contain last definitions

of *o.v*, and associated with each are the definition-clear paths *a*, *b* and *c*, respectively, giving the indirect-defs path set. On return to method *m*, there are three paths (*d*, *e*, and *f*) that are definition-clear with respect to *o* and *o.v* that reach the node that contains the call to *r*. These paths yield the set transmission path set. Within method *r*, there are three definition-clear paths, (*g*, *h*, and *i*), to nodes that contain first uses of *o.v*. These last three paths give the indirect uses path set. Taking the cross product of these three sets yields a set of 27 coupling paths, which are shown in Table 4-3.

**Table 4-3.** Sample Coupling Paths

| Number | Path | Number | Path | Number | Path |
|---|---|---|---|---|---|
| 1 | a;d;g | 10 | b;d;g | 19 | c;d;g |
| 2 | a;d;h | 11 | b;d;h | 20 | c;d;h |
| 3 | a;d;i | 12 | b;d;i | 21 | c;d;i |
| 4 | a;e;g | 13 | b;e;g | 22 | c;e;g |
| 5 | a;e;h | 14 | b;e;h | 23 | c;e;h |
| 6 | a;e;i | 15 | b;e;i | 24 | c;e;i |
| 7 | a;f;g | 16 | b;f;g | 25 | c;f;g |
| 8 | a;f;h | 17 | b;f;h | 26 | c;f;h |
| 9 | a;f;i | 18 | b;f;i | 27 | c;f;i |

Type II coupling sequences are formed by taking the cross product of the *t-path set* with the *i-use path set* for a particular type coupling sequence $s_{j,k}$. This is because the indirect definition that occurs when the antecedent node *j* is executed does not involve a method call. Thus, the coupling path begins with the antecedent node. Formally, the coupling paths for a Type II coupling sequence $s_{j,k}$ are defined in Equation 4-12, where $\Theta^{t}_{s_{j,k}}$ is given by: $\Theta^{t}_{s_{j,k}} = i\text{-}defs(j) \cap i\text{-}uses(c\text{-}method(s_{j,k}))$.

**Equation 4-12.** *Instance coupling paths for Type II coupling sequence*

$$InstanceCouplingPaths(s_{j,k}) = \left\{ (\langle t, u \rangle, v) \mid v \in \theta^t_{j,k} \subseteq \Theta^t_{s_{j,k}} \wedge \right.$$

$$\textit{def-clear-path}(t, v) \wedge \textit{def-clear-path}(u, v) \bullet$$

$$\left. t \in \textit{t-paths}(s_{j,k}) \wedge u \in \textit{i-use-paths}(\textit{c-method}(s_{j,k}), \theta^t_{j,k}) \right\}$$

Similar to Type II coupling sequences, Type III coupling sequences are formed by taking the cross product of the *i-def-path* set with the *t-path set* for a particular type coupling sequence $s_{j,k}$. However, the antecedent node contains an indirect definition and the consequent node contains a call to consequent method where the indirect uses occur. Formally, the coupling paths for a Type III coupling sequence $s_{j,k}$ are defined by the expression in Equation 4-13.

**Equation 4-13.** *Instance coupling paths for Type III coupling sequence*

$$InstanceCouplingPaths(s_{j,k}) = \left\{ (\langle d, t \rangle, v) \mid v \in \theta^t_{j,k} \subseteq \Theta^t_{s_{j,k}} \wedge \right.$$

$$\textit{def-clear-path}(d, v) \wedge \textit{def-clear-path}(t, v) \bullet$$

$$\left. d \in \textit{i-def-paths}(\textit{a-method}(s_{j,k}), \theta^t_{j,k}) \wedge t \in \textit{t-paths}(s_{j,k}) \right\}$$

where $\Theta^t_{s_{j,k}} = \textit{i-defs}(\textit{a-method}(s_{j,k})) \cap \textit{i-uses}(k)$.

### 4.7.2 Polymorphic Coupling Paths

The instance coupling paths described in the previous section do not take into account the possibility of polymorphic behavior that results from dynamic variation of types that can be bound to an object reference. With this possibility, a given instance coupling results in one path set for each member of the associated type family. The size of these sets is determined by the number of overriding methods within a given type, either defined directly or inherited from another type. The following subsections discuss the effects of inheritance and polymorphism on coupling and the set of coupling paths that result.

### 4.7.2.1 Type 1 Polymorphic Coupling Paths

For a Type I coupling sequence $s_{j,k}$, coupling paths are formed by taking each binding triple $\phi \in \Phi^t_{j,k}$ as expressed by Equation 4-14. As an example, Table 4-3 presents the set of polymorphic coupling paths for the class hierarchy shown in Figure 4-7 on page 107, and for the corresponding control flow schematics shown in Figure 4-8 on page 108, Figure 4-9 on page 110, and Figure 4-10 on page 111.

**Equation 4-14.** *Type I polymorphic coupling paths*

$$
\begin{aligned}
CouplingPaths(s_{j,k}) = \{ (\langle d, t, u \rangle, v) \mid\ & v \in \Theta^t_{j,k} \wedge t \in \Pi_{j,k} \\
& \wedge \textit{def-clear-path}(t, v) \bullet \\
& (\exists \phi \in \Phi^t_{j,k} \bullet d \in \textit{i-def-paths}(antecedent(\phi), \{v\}) \\
& \wedge u \in \textit{i-use-paths}(consequent(\phi), \{v\})) \}
\end{aligned}
$$

**Table 4-4.** Polymorphic coupling paths for type family *A*

| ***A*** | ***B*** | ***C*** |
|---|---|---|
| $\langle A::m_a, f_c, A::n_d \rangle$ | $\langle A::m_a, f_c, B::n_d \rangle$ | $\langle C::m_a, f_c, B::n_d \rangle$ |
| $\langle A::m_b, f_c, A::n_d \rangle$ | $\langle A::m_b, f_c, B::n_d \rangle$ | |

### 4.7.2.2 Type II Polymorphic Coupling Paths

For a Type II coupling sequence $s_{j,k}$, coupling paths are formed by taking each binding triple $\phi \in \Phi^t_{j,k}$ as expressed by Equation 4-15. The first element of each tripe in this set, $\varepsilon$, is an empty set of i-def paths, recording the fact that the indirect definition of the coupling variable takes place in the coupling method instead of an antecedent method.

**Equation 4-15.** *Type II polymorphic coupling paths*

$$
\begin{aligned}
CouplingPaths(s_{j,k}) = \{ (\langle \varepsilon, t, u \rangle, v) \mid\ & v \in \Theta^t_{j,k} \wedge t \in \Pi_{j,k} \\
& \wedge \textit{def-clear-path}(t, v) \\
& \bullet (\exists \phi \in \Phi^t_{j,k} \bullet u \in \textit{i-use-paths}(consequent(\phi), \{v\})) \}
\end{aligned}
$$

**4.7.2.3 Type III Polymorphic Coupling Paths**

For a Type III coupling sequence $s_{j,k}$, coupling paths are formed by taking each binding triple $\phi \in \Phi^{t}_{j,k}$ as expressed by Equation 4-16. In this situation, the third element of the binding triple is the empty sub-path set ε, corresponding to the fact that there are no i-use paths for a Type III coupling sequence.

**Equation 4-16.** *Type III polymorphic coupling paths*

$$CouplingPaths(s_{j,k}) = \{(\langle d, t, \varepsilon\rangle, v) \mid v \in \Theta^{t}_{j,k} \wedge t \in \Pi_{j,k} \wedge def\text{-}clear\text{-}path(t, v) \bullet (\exists \phi \in \Phi^{t}_{j,k} \bullet d \in i\text{-}def\text{-}paths(antecedent(\phi), \{v\}))\}$$

## 4.7.3 Feasible and infeasible coupling sequences

For a coupling sequence to exist, the context variable must be bound to some instance on a path that reaches the antecedent node. The location in a method where a context variable *o* is defined (i.e. made to refer to a particular object) is called a *binding site*. For a coupling sequence to exist, there must be at least one definition-clear path with respect to *o* from the binding site to the antecedent node.

For a binding site *s*, not all bindings to *o* are possible. A binding to *o* must be an instance of some member of the type family induced by *o*'s declared type. The set of types that can be bound to *o* is determined by the *binding mechanism*. The binding mechanism is how an instance is bound to a variable, which can be by one of the following:

1. *Parameter passing*. *o* is passed as an actual parameter to a method.
2. *Explicit instance creation*. For example, *o* = *new T*().
3. *Assignment of another variable to the context variable*. For example, *o* = *p*, where *p*'s declared type is a member of the type family induced by *o*'s declared type.

4. *Assignment of the return value of a method call to o*. For example, $o = f()$, where the return type of *f* is a member of the type family induced by *o*'s declared type.

For binding mechanisms 2, 3, and 4, the type of the instance bound to *o* is restricted by the declared type used in the mechanism. For type 2, the type of the binding is that of the type specified in the instance creation operator (e.g. *new* in Java and C++). Thus, any feasible coupling sequences involving this binding must use the same type.

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

For type 4, the resulting type is limited to the set *S* of possible types that can be returned by *f*(). If *f*'s declared return type is not the same as o's declared type *T*, then the set of types *R* that can be returned by *f* will be a proper subset of the type family defined by *T*, thus $T \notin R$. Therefore, there is no feasible coupling sequence whose context is *T*. In the general case, there will be no feasible coupling sequence whose context variable has a declared type in $family(T) - family(R)$.

## 4.8 Summary

This chapter has presented the extensions for handling inheritance and polymorphism to the coupling-based testing approach of Jin and Offutt [38]. The key insight to the extensions is in recognizing that the majority of coupling occurring in object-oriented programs resides in the state space interactions among the methods of a class. These couplings are dependent upon the context in which instances of the class are used, and are determined by the mechanisms (i.e. direct or indirect *definition* and *use*) used to alter the state of an instance. To model this context, the *coupling sequence* has been defined, which captures information about methods that are called in the context of a particular instance, the state variables for which the interaction occurs, and the paths between the locations where the interactions

take place. This information as the basis for a code-based static and dynamic analysis, and also as the basis for the set of test-adequacy criteria that form the coupling-based testing of object-oriented programs. Subsequent chapters discuss these criteria in detail and how source code is analyzed to identify coupling sequences.

# 5. A Set of Criteria for Testing Object-Oriented Programs

This chapter presents a new set of integration-level testing criteria that are based on the coupling theory presented in Chapter 4**.** These criteria are based on the data flow characteristics of coupling sequences, and are similar in nature to the original definitions of Jin and Offutt [38]. However, they differ in that the effects of both inheritance and polymorphism are explicitly accounted for. The handeling of inheritance and polymorphism is the most novel aspect of this thesis.

The definitions for coupling sequences, binding triples, and coupling paths presented in Chapter 4 lay a foundation with which to derive a set of new test adequacy criteria. These criteria can be used to guide the testing process and provide both requirements for testing and decision criteria for when to stop testing.

A fundamental issue with testing is *how much is enough*? In a perfect world no testing would be required, but in reality, a considerable amount is usually necessary. Typically, experience shows that the criterion most often applied is that of *date coverage*: testing stops when the amount of time allocated to the testing effort has been exhausted or eliminated.[1] Unfortunately, the basis for this criterion has virtually nothing to do with the intrinsic quality of the software. The greater complexity among the connections of software components found in object-oriented programs introduce new types of faults, thus new testing criteria are needed.

1. Date coverage was first theorized and described by Alexander, and later presented by Offutt [56].

# 5.1 Coupling Criteria

The new criteria are shown in Figure 5-1 and are similar to the original data flow testing criteria of Rapps and Weyuker [65]. The subsumptive relations of the criteria are shown in Figure 5-1. Examining the hierarchy from the bottom up reveals a basic structuring of the criteria along two distinct paths. At the bottom of the hierarchy is *All-Coupling-Sequences*, which is the most basic criterion and does not handle inheritance, polymorphism. or state space interactions. Moving up the left branch we find the criterion *All-Poly-Classes*, which takes into account the effects of inheritance and polymorphism. Moving up the right branch reveals a grouping of criteria that place emphasis on coupling paths based on the interactions of *definitions* and *uses* of coupling variables. Just below the top of the hierarchy is the criterion *All-Poly-Coupling-Defs-Uses*, which serves to unify those criteria based on definitions and uses of coupling variables, with the criterion that considers the effects of inheritance and polymorphism. Finally, at the top of the hierarchy is *All-Coupling-Paths*, which is the most comprehensive of all the criteria, and also serves as the point of unification between the new OO-based criteria and the original criteria of Jin and Offutt [38]. Each of these criteria are described and discussed in detail in the sub-sections that follow.

## 5.1.1 Definitions

The following definitions are used in the subsections below:

- ***context***$(s_{j,k})$ **:** ***Type*** : Returns the type of the instance bound to the context variable of the coupling sequence $s_{j,k}$.
- ***first***$(p)$ **:** ***Node*** : The first node in path $p$.
- ***last***$(p)$ **:** ***Node*** : The last node in path $p$.
- ***family***$(c)$ **:** $\mathbb{P}$***Class*** : The set of classes that belong to the type family specified by class $c$. Note that $c$ itself is a member of *family(c)*.

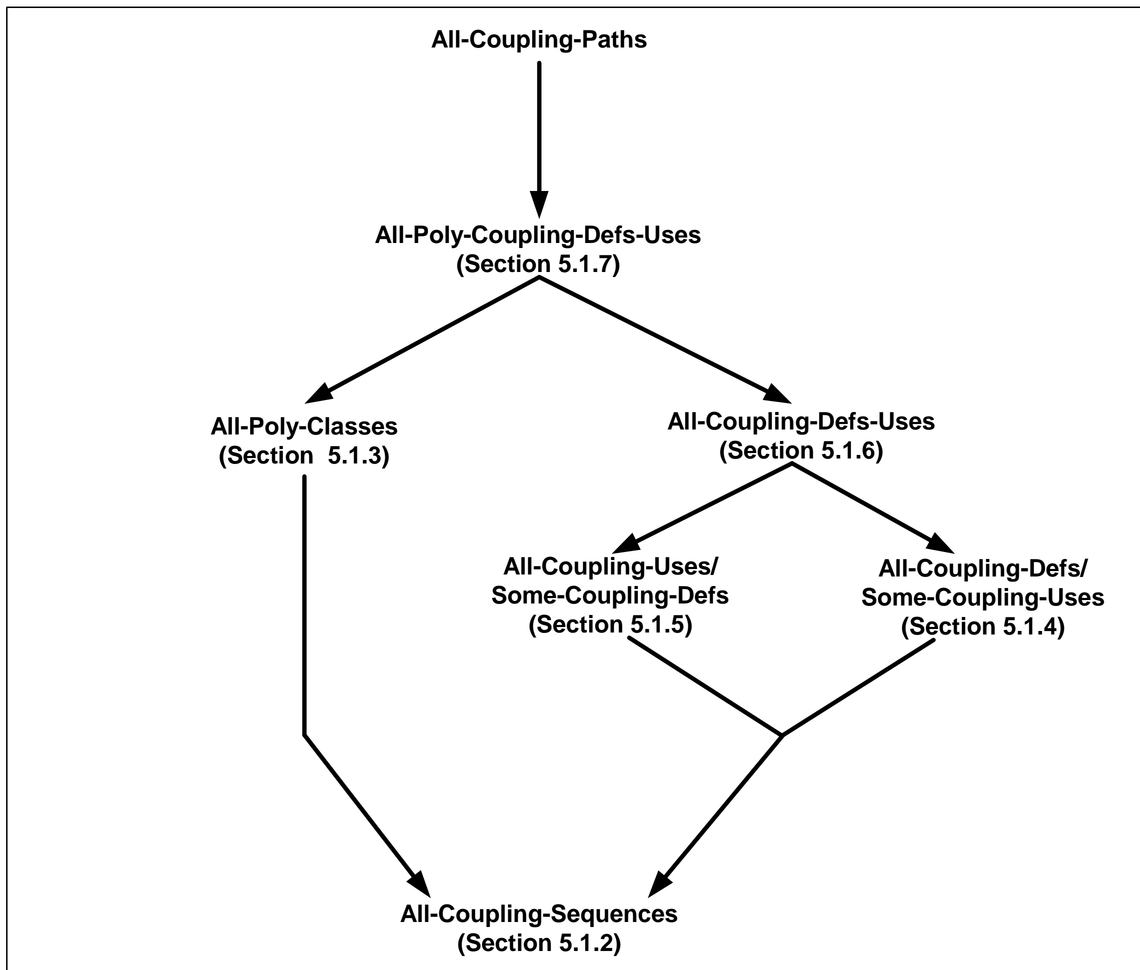



**Figure 5-1.** Hierarchy of coupling-based testing criteria

- ***FU*($s_{j,k}$,*c*,*v*)** : The set of nodes in the consequent method of $s_{j,k}$ with instance context *c* that have first-uses of *v*.

- ***i-defs*(*m*) : ℙ*Variable*** : The set of variables in the state space of the class containing *m* that are indirectly *defined* by a call to *m* made through some instance context. Formally:

$$i\text{-}defs(m) = \{v \in state\text{-}vars(class(m)) \mid \exists j \in N_m \bullet v \in defs(j)\}$$

- ***i-uses* (*m*) : ℙ*Variable*** : The set of variables in the state space of the class containing *m* that are indirectly *used* by a call to *m* made through some instance context. Formally:

$$i\text{-}uses(m) = \{v \in state\text{-}vars(class(m)) \mid \exists j \in N_m \bullet v \in uses(j)\}$$

- ***LD*($s_{j,k}$,*c*,*v*)** : The set of nodes in the antecedent method of $s_{j,k}$ with instance context *c* that have last-definitions of *v*.

- ***paths*($s_{j,k}$) : ℙ*Path*** : The set of coupling paths for coupling sequence $s_{j,k}$.

- ***sequences*(*f*) : ℙ*CouplingSequence*** : The set of coupling sequences contained in method *f*.

- ***trace*(*p*,*i*,$s_{j,k}$,*c*)** : The coupling path (i.e. execution trace) that results from the execution of method *f* using input *i* for coupling sequence $s_{j,k}$ in an instance context of the type family of *c*. For a Type I coupling sequence, the coupling path begins at a node in the antecedent method of $s_{j,k}$ that contains a last-definition of a coupling variable in $s_{j,k}$, and ends at a node in the consequent method that has a first-use of the same coupling variable. For a Type II sequence, the coupling path begins at a node in the coupling method that defines a coupling variable directly through an object reference. Like the Type I sequence, the coupling path ends at a node in the consequence method that has a first use of the same coupling variable. Finally, a Type III sequence, like a Type I sequence, begins at a node in the antecedent method that defines a coupling variable. However, the coupling path ends at a node in the coupling method that has the first-use of the coupling vari-

able. In all three cases, when the coupling method *f* is executed using input *i*, a path is traced from the entry node of *f* to its exit node. The coupling path is a sub-path of this path.

- $\boldsymbol{T_{S_{j,k}}}$ : The set of test cases that satisfy coupling sequence $s_{j,k}$. These are test cases that have been verified to test $s_{j,k}$.

The criteria described in the sections that follow are each presented with two definitions. The first is a formal definition that is based on the coupling theory presented in Chapter 4. The other is an intuitive definition stated in terms that make the criteria practical for testing purposes, whereas the formal definition is more suited for the static and dynamic analysis of the coupling properties of object-oriented programs.

### 5.1.2 All-Coupling-Sequences

Arguably, the minimum acceptable level of integration testing for an object-oriented program should cover every coupling sequence in every method of every class. Here, coverage means that each coupling sequence is executed by at least one test case. Accordingly, the *All-Coupling-Sequences* requires that every coupling sequence in a method be covered by at least one test case. Note that this criterion is coarse-grained in that it does not consider the *definition* and *use* interactions that can occur for the coupling variables that participate in a particular coupling sequence.

Observe that there is a similarity to Jin and Offutt's definition for *call coupling*. In their definition, every call site is required to be executed by at least one test. The key difference is that *All-Coupling-Sequences* considers those calls made through an instance context. Jin and Offutt's call coupling criterion is a degenerate case of *All-Coupling-Sequences*. Each procedure-oriented language program can be considered to have a single class that corresponds to the system itself. In the running system, there is a single instance of that class. Every procedure and function in the system are methods in that class. Thus, *All-Coupling-Sequences* is equivalent to Jin and Offutt's *call coupling* criterion where every call to a pro-

cedure or function is considered to be through an implicit *self* instance context (similar to *this* in C++ and Java). Note, however, that the converse, in general, is not true.

DEFINITION ***All-Coupling-Sequences***: *For every coupling sequence* $s_{j,k}$ *in coupling method f, there is at least one test case* $t \in T_{s_{j,k}}$ *such that when f is executed using t, there is a path p in the coupling paths of* $s_{j,k}$ *that is a sub-path of the trace of f.* The *All-Coupling-Sequences* criterion is stated formally in Equation 5-1, where $c \in family(context(s_{j,k}))$. The symbol *m* indicates that the path *p* is a sub-path of another path. In Equation 5-1, it is used to denote that *p* is a sub-path of an execution trace of *f*.

**Equation 5-1.** Criterion *All-Coupling-sequences*

$$\forall s_{j,k} \in sequences(f) \bullet (\exists t \in T_{s_{j,k}} \bullet (\exists p \in paths(s_{j,k}) \bullet p \text{ m } trace(f, t, s_{j,k}, context(s_{j,k}))))$$

### 5.1.3 All-Poly-Classes

The *All-Poly-Classes* criterion extends *All-Coupling-Sequences* to consider the effects of inheritance and polymorphism on coupling sequences. This is achieved by requiring at least one test for every class that could provide an instance context for each coupling sequence. The underlying idea is to know that a particular integration is successful with respect to every possible type substitution that can occur in a given coupling context. Each such substitution must be tested. Like *All-Coupling-Sequences*, *All-Poly-Classes* does not consider the possible *definition* and *use* interactions that can occur for particular coupling variables.

The *All-Poly-Classes* criterion requires that for every coupling sequence $s_{j,k}$ in a method *f*, and for every class *c* in the type family defined by the context of $s_{j,k}$, there is at least one test that covers every feasible combination of *c* and $s_{j,k}$ for *f*. The combination $(c, s_{j,k})$ is *feasible* if and only if *c* is the declared type *T* of the context variable for $s_{j,k}$, or *c* is a proper descendant of *T* and *c* defines an overriding method for the antecedent or consequent method. Thus, we do not consider classes that do **not** override the antecedent and consequent methods. We say that such a class is *oblivious* with respect to $s_{j,k}$, because without an overriding definition, regardless of which methods actually execute for the antecedent and

consequent, the thread of control will not enter such a class. This assumes that for a class *c*, the methods that will execute against the instance context when called in that context do not call other polymorphic methods.[1] For a class *c* that is a member of some type family *F*, where $m = antecedents(s_{j,k})$, $n = consequent(s_{j,k})$, and $n \in methods(F)$ the effective method for *m* or *n* when the context is an instance of *c* will be the nearest definition of *m* or *n*, respectively, when traversing up the inheritance hierarchy from *c* to *F*. When *m* is not polymorphic, the corresponding effective method will always be in the specification of the type that is the base of the family, *F* in this example.

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

The *All-Coupling-Sequences* criterion is stated formally in Equation 5-2.

**Equation 5-2.** Criterion *All-Poly-Classes*

$$\forall s_{j,k} \in sequences(f) \bullet \forall c \in family(context(s_{j,k})) \bullet (\exists t \in T_{s_{j,k}} \bullet (\exists p \in paths(s_{j,k}) \bullet p \in trace(f, t, s_{j,k}, c)))$$

### 5.1.4 All-Coupling-Defs/Some-Coupling-Uses

The criterion *All-Coupling-Defs/Some-Coupling-Uses* requires that for every coupling sequence in a method *f*, and every coupling variable *v* in the sequence and corresponding coupling-use *d* of *v*, there must be at least one test that executes a coupling path that begins at *d* and ends at a coupling-use of *v*.

DEFINITION ***All-Coupling-Defs/Some-Coupling-Uses***: *For every coupling sequence $s_{j,k}$ in coupling method f, and for every coupling variable v of $s_{j,k}$ and every node d in the antecedent method of $s_{j,k}$ that contains a last definition of v, there is at least one test case t such*

1. If the assumption does not hold, then it is possible that the thread of control can enter a class that is apparently oblivious.

*that when f is executed using t, there is a coupling path p in the trace of f that begins at d and that reaches some use of v in the consequent method of* $s_{j,k}$.

The *All-Coupling-Sequences* criterion is stated formally in Equation 5-3.

**Equation 5-3.** Criterion *All-Coupling-Defs/Some-Coupling-Uses*

$$\begin{aligned} \textit{All-coupling-sequences} \equiv\ & \forall s_{j,k} \in sequences(f) \\ & \bullet \forall c \in family(context(s_{j,k})) \bullet (\forall v \in \Theta_{s_{j,k}} \\ & \bullet (\forall m \in LD(s_{j,k}, c, v), n \in FU(s_{j,k}, c, v) \bullet v \\ & \in i\text{-}defs(m) \wedge v \in i\text{-}uses(n) \bullet (\exists t \in T_{s_{j,k}} \bullet (\exists p \\ & \in CouplingPaths(s_{j,k}) \\ & \bullet p.first \text{ m } trace(f, t, s_{j,k}, c) \wedge first(p.first) = m \\ & \wedge last(p.first) = n)))) \end{aligned}$$

## 5.1.5 All-Coupling-Uses/Some-Coupling-Defs

The *All-Coupling-Uses/Some-Coupling-Defs* criterion is the converse of *All-Coupling-Defs/Some-Coupling-Uses*. For every coupling sequence in a method *f*, and every coupling variable *v* in the sequence and corresponding coupling-use *u* of *v*, there must be at least one test that executes a coupling path that begins at some coupling-definition of *v* and ends at *u*.

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

The *All-Coupling-Uses/Some-Coupling-Defs* criterion is stated formally in Equation 5-4.

**Equation 5-4.** Criterion *All-Coupling-Uses/Some-Coupling-Defs*

$$\forall s_{j,k} \in sequences(j) \bullet (\forall v \in \Theta_{s_{j,k}} \bullet$$
$$(\forall n \in FU(s_{j,k}, context(s_{j,k}), v) \bullet v \in i\text{-}uses(n) \bullet$$
$$(\exists t \in T_{s_{j,k}} \bullet (\exists p \in CouplingPaths(s_{j,k})$$
$$\bullet\ p.first\ \mathrm{m}\ trace(f, t, s_{j,k}, context(s_{j,k})) \wedge last(p.first) = n))))$$

### 5.1.6 All-Coupling-Defs-Uses

The criterion *All-Coupling-Defs-Uses* is the combination of A*ll-Coupling-Uses/Some-Coupling-Defs* and *All-Coupling-Defs/Some-Coupling-Uses*. It requires that, for every coupling sequence in a method *f*, and for every coupling variable *v* in the sequence, there must be at least one test case that executes each coupling path with respect to *v*. That is, every feasible coupling path between each coupling-definition and coupling-use pair for *v* must be executed by at least one test case.

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

The *All-Coupling-Defs-Uses* criterion is stated formally in Equation 5-5.

**Equation 5-5.** Criterion *All-Coupling-Defs-Uses*

$$\forall s_{j,k} \bullet sequences(f) \bullet \forall v \in \Theta_{s_{j,k}} \bullet (\forall m$$
$$\in LD(s_{j,k}, context(s_{j,k}), v), n \in FU(s_{j,k}, context(s_{s,j}), v)$$
$$\mid v \in i\text{-}defs(m) \wedge v \in i\text{-}uses(n) \bullet (\exists t \in T_{s_{j,k}}$$
$$\bullet (\exists p \in CouplingPaths(s_{j,k}) \mid p.first\ \mathrm{m}\ trace(f, t, s_{j,k}, context(s_{j,k}))$$
$$\wedge (first(p.first) = m \wedge last(p.first) = n))))$$

### 5.1.7 All-Poly-Coupling-Defs-Uses

The criterion *All-Poly-Coupling-Defs-Uses* takes into account the effects of inheritance and polymorphism and serves to unify the two branches of the criteria shown in Figure Figure 5-1. *All-Poly-Coupling-Defs-Uses* requires that all coupling paths be executed for every member of the type family defined by the context of a coupling sequence.

DEFINITION ***All-Poly-Coupling-Defs-Uses***: For every coupling sequence $s_{j,k}$ in coupling method *f*, every class in the family of types defined by the context of $s_{j,k}$, every coupling variable *v* of $s_{j,k}$, every node *m* having a last definition of *v* and every node *n* having a first-use of *v* there is at least one test case *t* such that when *f* is executed using *t*. Further, there is a path *p* in the coupling paths of $s_{j,k}$ that is a sub-path of the trace of *f*.

The *All-Poly-Coupling-Defs-Uses* criterion is stated formally in Equation 5-6

**Equation 5-6.** .Criterion *All-Poly-Coupling-Defs-Uses*

$$\forall s_{j,k} \in sequences(f) \bullet \forall c \bullet family(context(s_{j,k})) \bullet (\forall v \in \Theta_{s_{j,k}}$$
$$\bullet (\forall m \in LD(s_{j,k}, c, v), n \in FU(s_{j,k}, c, v)$$
$$| v \in i\text{-}defs(m) \wedge v \in i\text{-}uses(n) \bullet (\exists t \in T_{s_{j,k}}$$
$$\bullet (\exists p \in CouplingPaths(s_{j,k})$$
$$| p.first \text{ m } trace(f, t, s_{j,k}, c) \wedge first(p.first) = m \wedge last(p.first) = n$$
$$))))$$

## 5.2 Generation of Test Requirements

A question that must be considered for the criteria presented in section Section 5.1, with respect to inheritance and polymorphism, is *which classes should be selected for consideration* when generating test requirements according to a particular criterion. Both *All-Poly-Classes* and *All-Poly-Classes-Defs-Uses* require coverage of all members of the type family for a particular coupling sequence. However, in the general case, this will only be feasible in those cases where the context variables are passed as formal arguments to the method under test, and the binding occurs at the site where the method is called. In this situation, the coupling variables passed as formal arguments can potentially be bound to an

instance of any type that is a member of the type family defined by the variable's declared type. Unfortunately, there are other cases where a coupling variable is bound to an instance, but where the type of the instance cannot so easily be generalized. The first occurs when the type of an instance bound to a context variable is explicitly named. This occurs in statements where an instance creation operator appears (e.g. *new* in Java and C++). In this circumstance, the type of the instance bound to the coupling variable must be from a member of the coupling variable's type family. However, since the type is explicitly named, there can be no variation in the type bound at a particular statement (referred to as a *binding site*).

The next case occurs when the actual type of the instance bound to a coupling variable cannot be controlled. This happens when the context variable is part of the state of an object and the binding results from some state dependent behavior caused by some prior sequence of method calls. In this case, varying the type of the instance bound to the context variable means varying the sequence of method calls, which may not be feasible. Further, even if the sequence can be varied, it is not guaranteed that it can be varied in such a way as to result in the desired variation of the types of the instance bound to the context variable.

The final difficulty in varying the type of the instance bound to a coupling variable occurs when the binding is to the result of a method call. Since the called method has the freedom to determine which instance is created, it may not be possible to achieve the desired variation of types. Thus, without the benefit of knowing what the called method does, it is not possible to assume anything stronger other than that some instance of the context variable's type family will be returned.[1] When the binding is a function of some state dependent behavior, the sequence of methods called prior to the method under test determines which binding is in effect, and cannot be determined as part of the input parameters to the method under test. From a testing perspective, this means that it is not possible to control all of the

1. Note that under this scenario, it is possible for the called method to return no instance at all. However, this would be considered a data flow anomally where a use of the context variable occurred after the context variable had been killed.

inputs to the method under test in every case desired [31]. Thus, there is an element of non-determinism in the testing process where inheritance and polymorphism are a factor.

From the perspective of generating test requirements according to a particular criterion, it is not reasonable, or practical, to require that the context variable of every coupling sequence be tested using an instance of every possible type. Instead, these criteria must be relaxed.

In the case where the type of the instance bound to a context variable cannot be controlled, the *All-Poly-Classes* criterion is relaxed to require only that only at least one test case that results in the binding of the context variable to some member of the corresponding type family be used. Similarly, *All-Poly-Classes-Defs-Uses* is relaxed to require that only the definitions and uses be tested that result from a binding of the context variable to some member of the corresponding type family.

For the case where the type of instance bound to a context variable is explicit, the requirement for *All-Poly-Classes* is that there be at least one test case that causes the explicit binding to occur. For *All-Poly-Classes-Defs-Uses*, the requirement is relaxed to require that all definitions and uses of coupling variables be tested through an instance of the explicit type bound to the context variable.

# 6. Analyzing Coupling Properties of Object-oriented Programs

This chapter discusses the static analysis of object-oriented programs to determine their coupling properties that are relevant to coupling-based testing. It presents algorithms for identifying coupling sequences for a method under test, and the corresponding coupling sets that result from the various types of instances that can be bound to a context variable.

This chapter also presents a discussion of the instrumentation mechanism for object-oriented programs that is used to collect coupling information and to support the coupling-based testing criteria presented in Chapter 5. The details of the instrumentation are out of necessity specific to the Java programming language. However, the techniques themselves are generally applicable to many strongly typed object-oriented language, including C++, Eiffel, C#, and Ada.

## 6.1 Definitions

The following definitions are used in this chapter:

- ***aliases*($v_1$,$v_2$) : *boolean*** : Returns *true* if $v_1$ aliases $v_2$, and vice versa.
- ***antecedent*( $s_{j,k}$ ) : *Node*** : Returns the antecedent node associated with coupling sequence $s_{j,k}$.
- ***antecedent_method*( $s_{j,k}$ ) : *Method*** : Returns the antecedent method associated with coupling sequence $s_{j,k}$.
- ***callee*( *c* ) : *method*** : Returns the method that is called at callsite *c*.

- ***calls*( *n* ) : $\mathbb{P}$*Call*** : Returns the set of calls that appear in the statement that corresponds to node *n*. Each *Call* is the pair (*o,m*), where *o* is the context variable for the call, and *m* is the called method.

- ***consequent*( $s_{j,k}$ ) : *Node*** : Returns the consequent node associated with coupling sequence $s_{j,k}$.

- ***consequent_method*( $s_{j,k}$ ) : *Method*** : Returns the consequent method associated with coupling sequence $s_{j,k}$.

- ***context*( $s_{j,k}$ ) : *Variable*** : Returns the context variable associated with the coupling sequence $s_{j,k}$.

- ***context-var*( *c* ) : *Variable*** : Returns the context variable used in a method call. For example, in the call *o.m*(), *o* is the context variable.

- ***control-successor( $n_2$, $n_1$ ) : boolean*** : Returns *true* if $n_2$ is a control successor of $n_1$.

- ***family*( *o* ) : $\mathbb{P}$*Type*** : Returns the set of types in the type family defined by *o*'s declared type.

- ***i-defs*( *m* ) $\mathbb{P}$*Variable*** : Returns the set of state variables indirectly defined by *m* in the class that contains *m* in its specification.

- ***i-uses*( *m* ) $\mathbb{P}$*Variable*** : Returns the set of state variables indirectly used by *m* in the class that contains *m* in its specification.

- ***is-instance-method*( *m* ) : *boolean*** : Returns *true* if *m* is specified as an instance method by some class. If *m* is not an instance method, then it is a class method.
- ***transmitted*( *o*, *v*, $n_1$, $n_2$ ) : *boolean*** : Returns *true* if there is a definition-clear path from $n_1$ to $n_2$ with respect to state variable *v* in the instance referenced by *o*.
- ***type*( *v* ) : *Type*** : Returns the type of variable *v*.

## 6.2 Identifying Coupling Sequences

The algorithm for identifying coupling sequences within a method *f* under test is presented in Algorithm 6-1. It begins by initializing the set of coupling sequences $S_f$ to the empty set. It then iterates over pairs of distinct nodes $n_1$, $n_2$ (represented by the two outer loops) that have call sites containing calls to methods made through an instance context. For each such pair, where there is at least one control flow path from $n_1$ to $n_2$, the algorithm iterates over the pairs of calls $c_1$ and $c_2$ to instance methods made at each site such that the context variable of each call is the same (or aliases one another). For each pair, a coupling sequence ($f$, $n_1$, $n_2$, *callee*($c_1$), *callee*($c_2$)) is added to the set of coupling sequences for *f*. The running time of this algorithm is $O(n^2)$, where *n* is the number of nodes having call sites that involve instance contexts and $n \leq N_f$, where $N_f$ is the number of nodes in *f*. Note that $c_1$ and $c_2$ represent the number of individual calls on individual statements. Their product contributes only a small constant to the running time of the algorithm, and thus $n^2$ is the dominating term. This is justified by observing that most statements, the number of calls will be one (e.g. $a = f() + g$), thus the term of the complexity contributed by the number of calls will also be one. However, suppose that the number of calls at both call sites were greater than one, say four. In this case, the term contributes a factor of 16 to the overall expression. Still, in the majority of cases this is insignificant when compared to the number of nodes (i.e. statements) in a method.

**Algorithm 6-1**: Identifying coupling sequences

```
Sf = ∅
for n1 ∈ call-sites(f):
    for n2 ∈ call-sites(f)|n2 ≠ n1 ∧ control-successor(n2, n1):
        for c1 ∈ calls(n1)|is-instance-method(callee(c1)):
            for c2 ∈ calls(n2)|c1 ≠ c2 ∧ is-instance-method(callee(c2)):
                if (context-var(c1) = context-var(c2) ∨
                    aliases(context-var(c1),context-var(c2))) ⇒
                        Sf ∪= { (f, context-var(c1),type(context(c1)),n1,
                                  n2, callee(c1), callee(c2)) }
                end if
            end for
        end for
    end for
end for
```

# 6.3 Identifying Coupling Sets

The algorithm for identifying the coupling set associated with each coupling sequence for a method under test *f* is presented in Algorithm 6-2. The algorithm iterates over each coupling sequence $s_{j,k}$ in *f*, and each *t* in the type family defined by the declared type of $s_{j,k}$'s context variable. For each such *t*, the coupling set is formed by taking the intersection of the state variables indirectly defined by the antecedent method with those that are indirectly used by the consequent method. The coupling set is further restricted to only those state variables that are *transmitted* between antecedent and consequent nodes of *f*. As defined in Section 4.4 on page 101, transmitted means for each state variable *v* in the coupling set: (1) there is no other method called between the antecedent and consequent nodes that use the same instance bound to the context variable *o* that results in the definition of *v*; and (2) there is no direct assignment of *v* through the same instance bound to *o*.

**Algorithm 6-2**: Identifying coupling sets

```
for s_{j,k} ∈ S_f:
    for t ∈ family(context(s_{j,k})):
        Θ^t_{s_{j,k}} = ∅
        for v ∈ (i-defs(antecedent_method(s_{j,k})) ∩ i-uss(consequent_method(s_{j,k})):
            if transmitted( v, antecedent(s_{j,k}), consequent(s_{j,k})) ⇒
                Θ^t_{s_{j,k}} ∪= {v}
            end if
        end for
    end for
end for
```

# 6.4 Instrumenting OO Programs for Coupling Analysis

To test object-oriented programs, we need to know which elements of interest are covered during the execution of a method under test for a particular input. From a coupling-based testing perspective, this means that for a given input (i.e. test case), we need to determine which coupling sequences are executed, which definitions and uses occur, and which calls are made. To this end, a number of *coverage mappings* are defined that formally specify the information that must be collected. Each mapping is a requirement for instrumentation necessary to collect coverage data.

## 6.4.1 Coverage Mappings

Each coverage mapping relates a particular event of interest to corresponding coverage information. For example, the execution of a statement that contains a call to a method made through an instance context is important if we are concerned with the coupling-based testing criteria. For the same reason, binding sites are also of interest since they are where the identify of the instances bound to object references are determined.

There are twelve coverage mappings relevant to collecting coverage information for coupling-based testing, as summarized in Table 6-1. The first column of the table gives the name of the mapping, while the second gives the corresponding form. The third column gives the interpretation of the mapping. Except for the last mapping (*TracedPath*), the form

of each mapping is that of a function that relates a mapping tuple to an integer. The tuple contains the parameters that characterize the mapping, and the integer gives the count of the times that the corresponding event of interest occurs in the method under test a runtime (e.g. the number of times that a particular variable has been defined).

**Table 6-1.** Coverage Mappings

| Map | Form | Interpretation |
|---|---|---|
| Coupling Sequences | (*context variable, context type, object id, antecedent node, consequent node, f*) ⟶ (*integer, integer*) | Execution counts for antecedent and consequent nodes, respectively. |
| Definitions | (*identifier*, *node*, *line*, *f*) ⟶ *integer* | Number of definitions of *identifier* at location *node* in *line* of *mut*. |
| Indirect Definitions | (*identifier*, *node*, *line*, *f*) ⟶ *integer* | Number of definitions of state variable *identifier* at location *node* in *line* of *mut*. |
| Last Definitions | (*identifier*, *node*, *line*, *f*) ⟶ *integer* | Number of last definitions of *identifier* at location *node* in *line* of *mut*. |
| Last Indirect Definitions | (*identifier*, *node*, *line*, *f*) ⟶ *integer* | Number of last definitions of state variable *identifier* at location *node* in *line* of *mut*. |
| Uses | (*identifier*, *node*, *line*, *f*) ⟶ *integer* | Number of uses of *identifier* at location *node* in *line* of *mut*. |
| Indirect Uses | (*identifier*, *node*, *line*, *f*) ⟶ *integer* | Number of uses of state variable *identifier* at location *node* in *line* of *mut*. |
| First Uses | (*identifier*, *node*, *line*, *f*) ⟶ *integer* | Number of first uses of *identifier* at location *node* in *line* of *mut*. |
| First Indirect Uses | (*identifier*, *node*, *line*, *f*) ⟶ *integer* | Number of first uses of state variable *identifier* at location *node* in *line* of *mut*. |
| Calls | (*identifier*, *method*, *node*, *line*, *bound type*, *object id*, *f*) ⟶ *integer* | *method* is called through *identifier* (*identifier.method*()), where *identifier* is bound to an instance of *bound type*. |
| | (*method*, *node*, *line*, *f*) ⟶ *integer* | *method* is called independent of any instance context (*method*()). |
| Traced Nodes | *Node* ⟶ *integer* | Number of times *Node* has been executed. |
| Traced Path | Sequence of *Node* | Sequence of traced nodes as encountered during an execution of the method under test. |

As an example of a mapping tuple, the first coverage mapping, *CouplingSequences* consists of the parameters *context variable*, *context type*, *object id*, *antecedent node*, *consequence node*, and the name of the method under test (given as *f* in the table). These mapping param-

eters are based on the definition of a coupling sequence given in Section 4.2 on page 93. Both the antecedent node and consequent node (and all other references to node in the table) are the names of nodes in the control flow graph that corresponds to the method under test *f*. The parameter *object id* is a value that corresponds to the identify of the instance bound to the context variable. This ensures uniqueness across coupling sequences that would otherwise not be if identity is ignored. The ability to discriminate coupling sequences by instance identity is important for purposes of determining coverage adequacy.

The *Taced Path* mapping is a sequence of nodes. For a given execution of the *f*, it records the nodes that were executed and captured by the *Traced Nodes* mapping.

### 6.4.2 Instrumentation Requirements

The coverage mappings described in Section 6.4.1 are used to determine the set of instrumentation requirements necessary to collect coupling-based coverage data for a given programming language *L*. For a given *L*, each instrumentation requirement yields an instruction set $I_L$ that contains one or more instrumentation instructions expressed as statements of *L*. Each $I_L$ is injected into the method under test (MUT) *m* at locations where the corresponding coverage data can be collected as *m* executes. Table 6-2 presents the instrumentation instructions that correspond to the coverage mappings given in Table 6-1. The first column gives the instrumentation category, the second the name of the instruction, which maps one-to-one to the column labeled *Map* in Table 6-1. The third column gives the placement for the instrumentation instruction in relation to the statement of the MUT

that yields the event of interest. The fourth column provides commentary that further clarifies the placement of the corresponding instrumentation instruction.

**Table 6-2.** Instrumentation instructions

| Category | Instruction | Placement | Comments |
|---|---|---|---|
| **Coupling Sequences** | *registerCouplingSequence* | Immediately after each binding site. | One per reachable coupling sequence. |
| | *AntecedentCall* | Immediately before each call site where the antecedent method is called. | One per coupling sequence that has the called method as its antecedent. |
| | *ConsequentCall* | Immediately before each call site where the consequent method is called. | One per coupling sequence that has the called method as its consequent. |
| **Definitions** | *def* | Immediately before each node where definition occurs. | |
| | *iDef* | | |
| | *lastDef* | | |
| | *lastIDef* | | |
| **Uses** | *Use* | Immediately before each node where use occurs. | |
| | *iUse* | | |
| | *firstUse* | | |
| | *firstIUse* | | |
| **Miscellaneous** | *call* | Immediately before each node where call is made. | |
| | *trace* | Immediately before execution of traced node. | |

# 6.5 Instrumenting Java Programs

The research in this thesis is validated using programs written in Java. Accordingly, a set of instrumentation instructions and data collection mechanism is required that will collect the necessary information within the constraints imposed by the syntax and semantics of Java.

### 6.5.1 Instrumentation Instructions

Table 6-3 summarizes the methods that implement the coverage mappings and instruction requirements given in Table 6-1 and Table 6-2. Note that the signatures of each method are isomorphic to the tuples of the corresponding coverage mapping.

**Table 6-3. Java instrumentation methods**

| Category | Java Method Signature |
|---|---|
| **Coupling Sequences** | `registerCouplingSeq( String contextVar,`<br>`String couplingType,`<br>`int objectId,`<br>`String antecedentNode,`<br>`String consequentNode,`<br>`String mut )` |
| | `antecedentCall( String contextVar,`<br>`String couplingType,`<br>`int objectId,`<br>`String antecedentNode,`<br>`String consequentNode,`<br>`int line,`<br>`String mut )` |
| | `ConsequentCall( String contextVar,`<br>`String couplingType,`<br>`int objectId,`<br>`String antecedentNode,`<br>`String consequentNode,`<br>`int line,`<br>`String mut )` |
| **Defini-tions** | `def( String identifier, String nodeId, int line, String mut )` |
| | `iDef( String identifier, String nodeId, int line, String mut )` |
| | `lastDef( String identifier, String nodeId, int line, String mut )` |
| | `lastIDef( String identifier, String nodeId, int line, String mut )` |
| **Uses** | `use( String identifier, String nodeId, int line, boolean puse, String mut )` |
| | `iUse( String identifier, String nodeId, int line, boolean puse, String mut )` |
| | `firstUse( String identifier, String nodeId, int line, boolean puse, String mut )` |
| | `firstIUse( String identifier, String nodeId, int line, boolean puse, String mut )` |
| **Miscella-neous** | `call( String method, String nodeId, int line, String mut )` |
| | `call( String identifier, String method, String nodeId, int line, String bound-Type, int objectId, String mut )` |
| | `trace( String nodeId )` |

The instructions for Java are defined as methods in a special class called *DataCollector* whose purpose is to provide the implementation of a suitable mechanism for collecting coupling-based coverage data. This class is depicted in the UML Class Diagram shown in Figure 6-1 along with supporting classes *CouplingSequence*, *Definition*, *Use*, and *Call*. The supporting classes are used to hold information specific to the events of interest (e.g. variable definitions, method calls, etc.) that occur at runtime. The MUT is instrumented by adding calls to these methods at the appropriate points in its execution. An instance of the *DataCollector* itself is instantiated upon entry to the MUT before any of its statements are executed. A detailed example is presented in the next section.

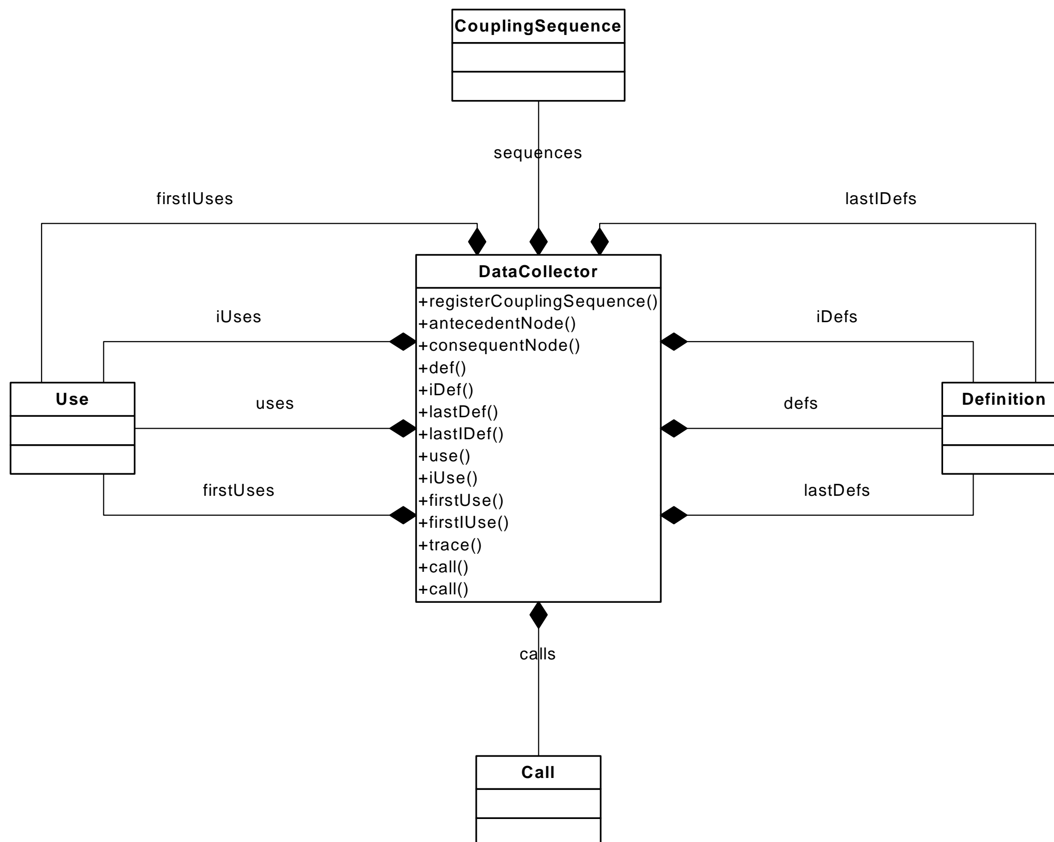



**Figure 6-1.** Java mechanism for collecting coupling-based coverage data

### 6.5.2 An example

The class diagram shown in Figure 6-2 depicts a class *Client* that has a method *f* that takes an instance of *A* as an argument, or an instance a descendant of *A*. Figure 6-3 shows the specification of *Client* with the body of *f*. The control flow graph is depicted in the right portion of the figure. Observe that *f*'s control flow graph shows two coupling sequences, labeled $s_{4,6}$ and $s_{5,8}$. These correspond to the calls in *f* at lines 16 and 21, and 19 and 25, respectively.

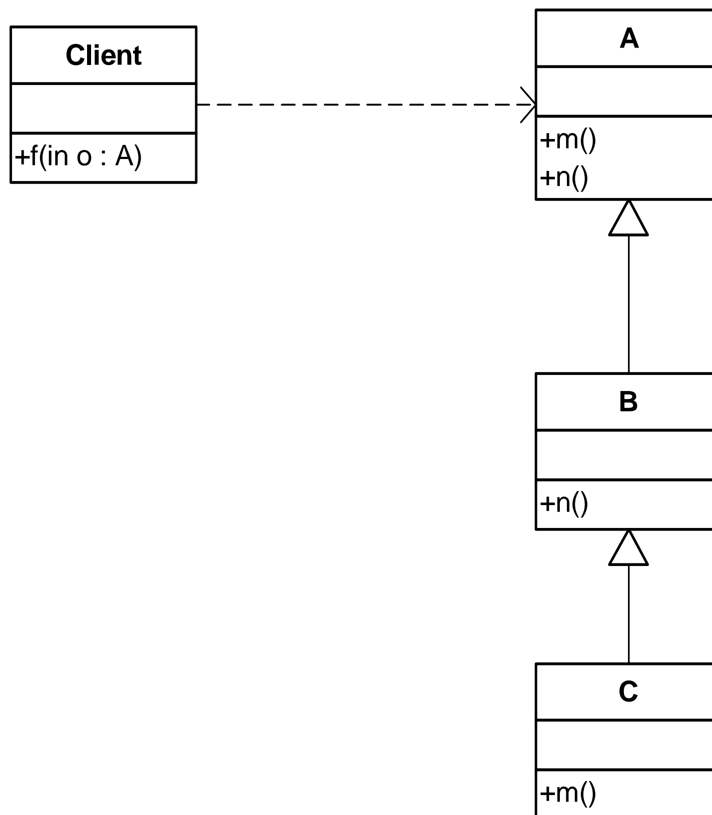



**Figure 6-2.** Sample hierarchy

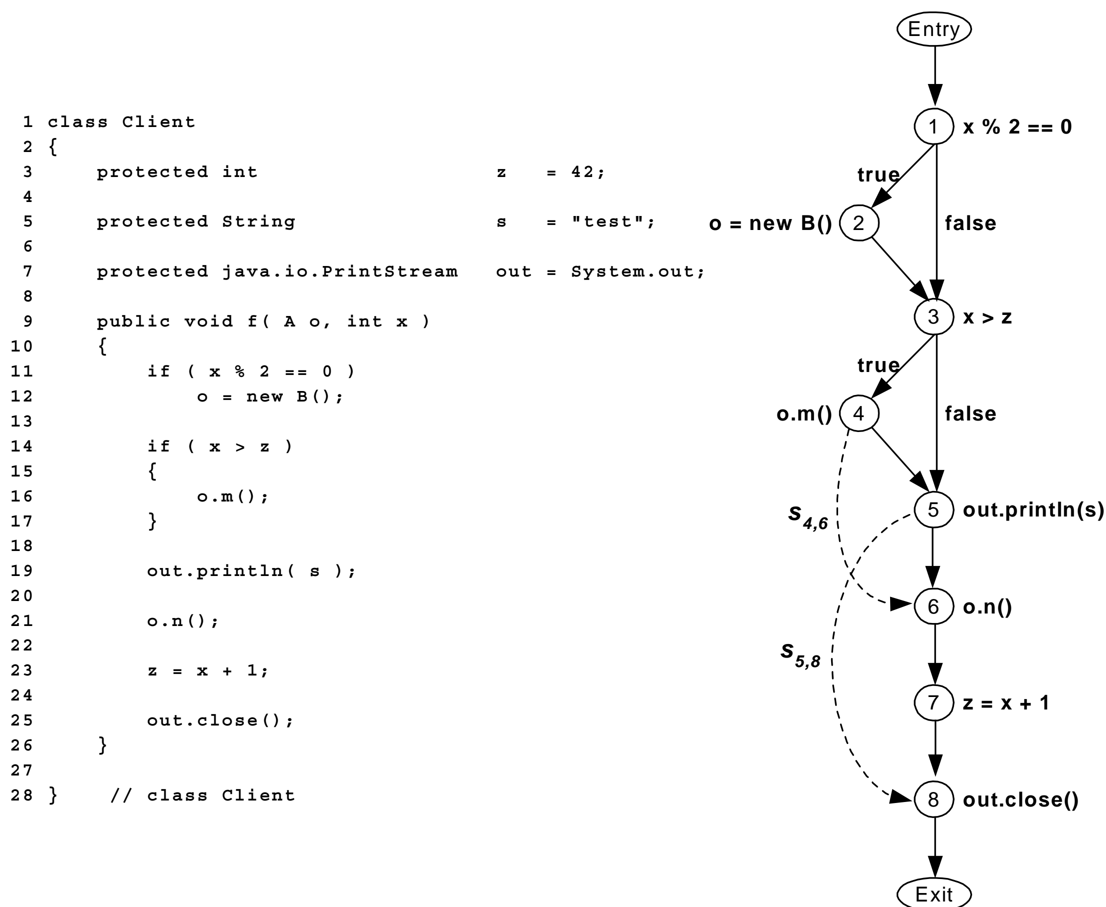



**Figure 6-3.** Method *Client.f* (without instrumentation) and corresponding CFG

Figure 6-4 on page 154 shows the instrumented version of *f* using the instrumentation instructions for Java. Each statement in the method is numbered down the left-hand side. The numbers in parentheses immediately to the right of the line numbers correspond to the original uninstrumented version of *f*. For example, adjacent to the line number 20 is “(11)”, indicating that line 20 corresponds to line 11 in Figure 6-3. The control flow nodes of the uninstrumented version of *f* are also shown, depicted as large circles containing the corresponding node of the control flow graph. The circles are shaded to identify each antecedent node and its corresponding consequent node. Directed arcs are drawn between call sites.

Immediately upon entry to the MUT and before any of the original instrumentation statements are executed, an instance of *DataCollector* must be created. This is shown at line 11

of Figure 6-4 where the instance is created with the name of the MUT passed as the argument to the constructor of *DataCollector*. Line 13 then records the event of entry to the method.

#### 6.5.2.1 Registration of Coupling Sequences

There are two locations in *f* where a binding to the context variable *o* occurs. The first is an implicit binding at the entry node to *f* where *o* is a formal argument, and the other is at line 23 in the instrumented version (12 in the original) of Figure 6-4. It is at these points that the instance for any coupling sequence involving *o* is determined. That is, for a given coupling sequence $s_{j,k}$ having *o* as the context variable, the binding of *o* occurring at one of the binding sites must reach $s_{j,k}$. Otherwise, there would be no coupling sequence. Since we cannot know at analysis time which input will cause which binding to reach a particular coupling sequence, we assume without loss of generality that every binding of the same context variable will reach every coupling sequence that uses that same context variable. Thus, we instrument all locations where a binding site occurs to register all possible coupling sequences that use the same context variable.[1] This occurs at lines 15 and 16 for the binding that occurs at entry to *f*, and at lines 24 and 25, immediately following the binding of *o* at line 23.

Observe that the hash code associated with the instance bound to the context variable *o* is passed as the third argument to the instrumentation method *registerCouplingSeq*().[2] This corresponds to the *object id* parameter given in Figure 6-1, and allows for the discrimination between potentially different instances bound to the same context variable.

---

1. This applies only to those coupling sequences that a particular binding site reaches.

2. In Java, every object (instance) has an associated hash code that is guaranteed to be unique at runtime. The method *hashCode*() is defined by class *Object* in the *java.lang* package.

#### 6.5.2.2 Collection of use and definition information

The first use of the formal argument *x* occurs at line 20 of Figure 6-4. This event is recorded by the instrumentation instruction at line 19 in the call to *firstUse*, which also automatically records the event as a use of *x*. The fact that *x* is used in a conditional expression is captured by the fourth actual argument to *firstUse*, which is the expression *true*. A first indirect use of the state variable *z* occurs at line 31. This is captured by the instrumentation instruction *firstIUse* at line 29. This event is also recorded as a use at line 30. An indirect use of state variable s occurs at line 44, which is recorded by the instrumentation instruction iUse at line 41. Note that the control flow node corresponding to the original statement in the uninstrumented version of *f* is used as the node id passed as the second actual argument to these methods.[1]

Definition information is captured in a similar fashion to use information. A last indirect definition of state variable *z* occurs at line 55. This event is recorded by the instrumentation instruction *lastIDef* at line 53.

#### 6.5.2.3 Identifying execution of coupling sequences

Each call to an antecedent method or consequent method is recorded by the instrumentation instructions *antecedentCall* and *consequentCall*, respectively. Examples of this are shown at lines 36 and 43 for a call to an antecedent method, and at lines 50 and 61 for a call to a consequent method.

---

1. This is true of all the instrumentation instructions that capture definition, use, call, and trace information.

```
9          public void f( A o, int x ) throws UndefinedCouplingSequenceException, java.io.IOException
10         {
11             DataCollector dc = new DataCollector( "Client.f" );
12
13             dc.trace( "entry" );
14
15             dc.registerCouplingSeq( "o", o.getClass().getName(), o.hashCode(), "4", "6", "Client::f" );
16             dc.registerCouplingSeq( "out", out.getClass().getName(), out.hashCode(), "5", "8", "Client::f" );
17
18             dc.trace( "1" );
19             dc.firstUse( "x", "1", 11, true, "Client::f" );
20 (11) 1      if ( x % 2 == 0 )
21             {
22                 c.lastDef( "o", "2", 12, "Client::f" );
23 (12) 2          o = new B();
24                 dc.registerCouplingSeq( "o", o.getClass().getName(), o.hashCode(), "4", "6", "Client::f" );
25                 dc.registerCouplingSeq( "out", out.getClass().getName(), out.hashCode(), "5", "8", "Client::f" );
26             }
27
28             dc.trace( "3" );
29             dc.firstIUse( "z", "3", 14, true, "Client::f" );
30             dc.use( "x", "3", 14, true, "Client::f" );
31 (14) 3      if ( x > z )
32             {
33                 dc.trace( "4" );
34                 dc.use( "o", "4", 16, false, "Client::f" );
35                 dc.call( "o", "m", "4", 16, o.getClass().getName(), o.hashCode(), "Client::f" );
36                 dc.antecedentCall( "o", o.getClass().getName(), o.hashCode(), "4", "6", 16, "Client::f" );
37 (16) 4          o.m();
38             }
39
40             dc.trace( "5" );
41             dc.iUse( "out", "5", 19, false, "Client::f" );
42             dc.call( "out", "println", "5", 19, out.getClass().getName(), out.hashCode(), "Client::f" );
43             dc.antecedentCall( "out", out.getClass().getName(), out.hashCode(), "5", "8", 19, "Client::f" );
44 (19) 5      out.println( s );
45
46             dc.trace( "6" );
47             dc.firstUse( "o", "6", 21, false, "Client::f" );
48             dc.call( "o", "n", "6", 21, o.getClass().getName(), o.hashCode(), "Client::f" );
49             dc.consequentCall( "o", o.getClass().getName(), o.hashCode(), "4", "6", 21, "Client::f" );
50 (21) 6      o.n();
51
52             dc.trace( "7" );
53             dc.lastIDef( "Z", "7", 23, "Client::f" );
54             dc.use( "x", "7", 23, false, "Client::f" );
55 (23) 7      z = x + 1;
56
57             dc.trace( "8" );
58             dc.iUse( "out", "8", 25, false, "Client::f" );
59             dc.call( "out", "println", "8", 25, out.getClass().getName(), out.hashCode(), "Client::f" );
60             dc.consequentCall( "out", out.getClass().getName(), out.hashCode(), "5", "8", 25, "Client::f" );
61 (25) 8      out.close();
62
63             dc.trace( "exit" );
64
65             dc.finish();
66         }
```

$S_{4,6}$ $S_{5,8}$

**Figure 6-4.** Method *Client.f* instrumented for coupling coverage

# 6.6 Summary

This chapter has presented the algorithms and data structures used to perform static and dynamic coupling-based analysis of object-oriented programs. Included with this are cov-

erage mappings that define a set of elements used to represent coupling analysis information. Each element contains a form and interpretation that describes the structural representation and meaning of the information the elements contain.

This chapter has also described the instrumentation requirements used to instrument object-oriented programs for the collection of coupling information. The requirements are based on the coverage mappings and yield an instruction set for each object-oriented programming language. Elements of this instruction set are used to inject instrumentation into methods under test, antecedent methods, and consequent methods. The instrumentation requirements along with the coverage mappings are used as the basis for generation and instrumentation activities, and are incorporated into the research proof of concept tool developed over the course of this research.

# 7. CBAT - Coupling-based Analysis Tool

CBAT is a research proof of concept tool developed for the purpose of demonstrating the practicality of the coupling-based testing approach for typed object-oriented languages. In its present form, CBAT is capable of analyzing programs written in Java. However, its representations are sufficiently rich to support other object-oriented and object-based languages such as C++, C#, Eiffel, Modula-2 and Ada. The following sections describe the representations supported by CBAT, its architecture, and how it is implemented.

## 7.1 Objectives of CBAT

CBAT satisfies a number of objectives. First, it supports the coupling-based testing of object-oriented programs. CBAT includes representations, algorithms, and utilities for producing programs that are instrumented for collecting coverage information for the coupling-based testing criteria presented in Chapter 5.

Second, CBAT is intended to be a general purpose research tool. While its initial objectives lie squarely at supporting the research described by this dissertation, it also is intended o support future research activities that both involve and do not involve coupling-based testing. For example, it is envisaged that CBAT will be used to extend the coupling-based testing techniques to include inter-method coupling sequences and also for the reverse engineering of software contracts (i.e. preconditions, postconditions, etc.) from existing object-oriented and procedural programs.

Finally, CBAT is intended to provide a general purpose analysis platform for conducting many different types of static and dynamic program analysis, such as slicing, code coverage, change impact analysis, dependency analysis, slicing, and the collection of OO metrics.

# 7.2 Representations provided by CBAT

CBAT includes a number of high-level abstractions that represent all of the syntactic and semantic entities specific to the Java programming language. However, the abstractions are generalized to support similar entities found in other object-oriented languages.

The primary representations of CBAT are the class and method graphs. The *class graph* contains abstractions that model the high-level entities of an object-oriented program, such as packages, classes, and interfaces. The *method graph* provides a representation for individual functions and procedures that implement control flow. Each of these graphs is described below.

## 7.2.1 Class Graph

The *class graph* models the high-level structural elements of an object-oriented program. It includes abstractions that model packages, classes, and interfaces, as well as abstractions that model program variables, functions. and procedures. Figure 7-1 depicts a UML class diagram that contains the classes that correspond to the key abstractions of the class graph and their structural relationships. The primary classes include *Package*, *ClassElement*, *Interface*, *Variable*, and *Method*. Instances of *Package* define individual name spaces that may contain instances of *Package*, *ClassElement*, or *Interface*.[1] Instances of *ClassElement* and *Interface* represent user-defined types, with *ClassElement* being an instantiable type that defines both state and behavior.[2] *Interface* corresponds to non-instantiable types that only define method signatures and variable constants. Instances of *Variable* are used to model variables that defined the state space of a class, and also variables that appear as arguments to methods or that are local to a block of code. Finally, instances of *Method* are used to model the procedures and functions (i.e. *methods*) that occur in the definition of a class.

---

1. Packages can also contain other packages, though this is only a notional concept in Java since there is no semantic or syntactic relationship between a package and sub-package in Java.

2. Built-in types, such as *int*, *float*, and *char* found in Java are represented by instances of class *BuiltinType* shown in Figure 7-1.

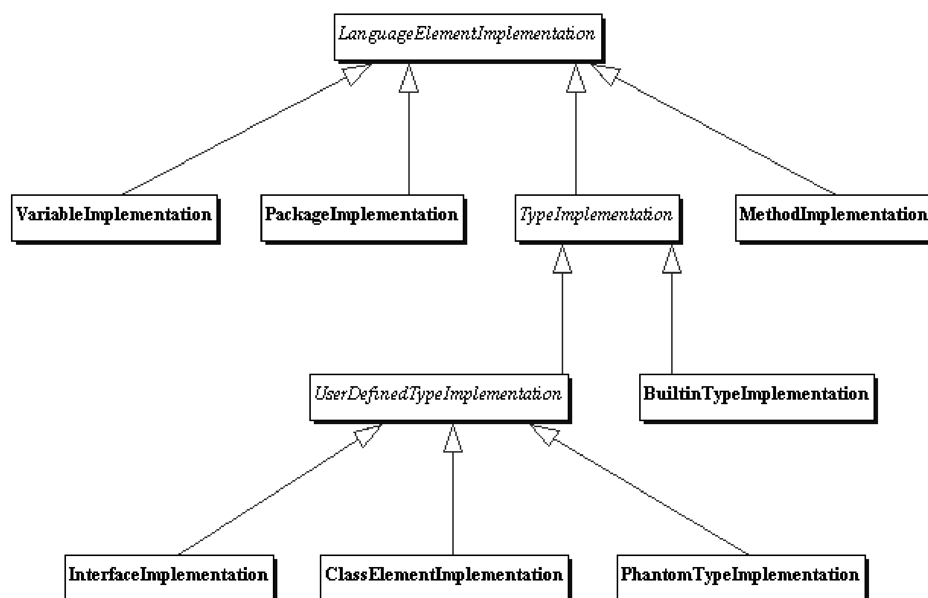



**Figure 7-1.** UML Class Diagram for package *ClassGraph*

#### 7.2.1.1 Method Graph

The *method graph* is the structure used by CBAT to represent the program text of individual methods. In reality, the method graph is a *multi-graph* that consists of several different sub-graphs. These sub-graphs represent various aspects of a method's structure, such as control and data flow. A method graph consists of three key components: an *abstract syntax tree*, a set of *intra-method representations*, and a set of *inter-method representations*. A method graph also contains a robust set of analysis data. Each of these elements are described in detail in the following subsections.

### 7.2.2 Abstract Syntax Tree

The fundamental structure underlying the method graph is an abstract syntax tree that represents the structure of a program at the statement and expression level. This structure is derived directly from the source code of the program being analyzed, and consists of two parts: the *control tree* and the *expression tree*. The control tree is a high-level model of the

individual statements in a method, including those that alter control flow (selection and iteration statements) as well as individual statements that perform computation. Key abstractions of the control tree, shown in Figure 7-2, include *ControlNode*, *SequenceNode*, *BreakTransferNode*, and *ExceptionBlockNode*.

*ControlNode* defines a type family of nodes that represent method instructions that alter the flow of control in a program through the use of a conditional expression (loops and if statements). Instances of *SequenceNode* correspond to the individual instructions of a method, such as assignments and output statements, that whose control flows to the next instruction.

Instances of *BreakTransferNode* are used to represent those statements in a method that correspond to unconditional transfers of control. This includes *break*, *continue*, *goto*, and *return* statements. Instructions that yield the raising of an exception are also instances of *BreakTransferNode*. *ExceptionBlockNode* represents a single exception condition handler, such as a catch block in Java or C++.

Additional abstractions include nodes for representing case statements (*CaseNode*, *CaseDisjunctNode*, and *DefaultCaseDisjunctNode*), a node for represent statement labels (*LabelNode*), and a node that corresponds to a block of sequential statements (*BlockNode*). Certain elements of the control tree have an expression tree associated with them. These elements include members of the type family defined by *ControlNode*, and instances of *SequenceNode*, *ExceptionBlockNode* and *CaseNode*. An expression tree models the syntax of the computation that takes place at a given statement of a method. Examples include assignment statements, output statements, and conditional expressions. Together, the control tree and the expression tree model the complete syntax of a method's program text and are used by CBAT to generate higher-level graph representations for analysis purposes. The class diagram for the expression tree is shown in Figure 7-3.

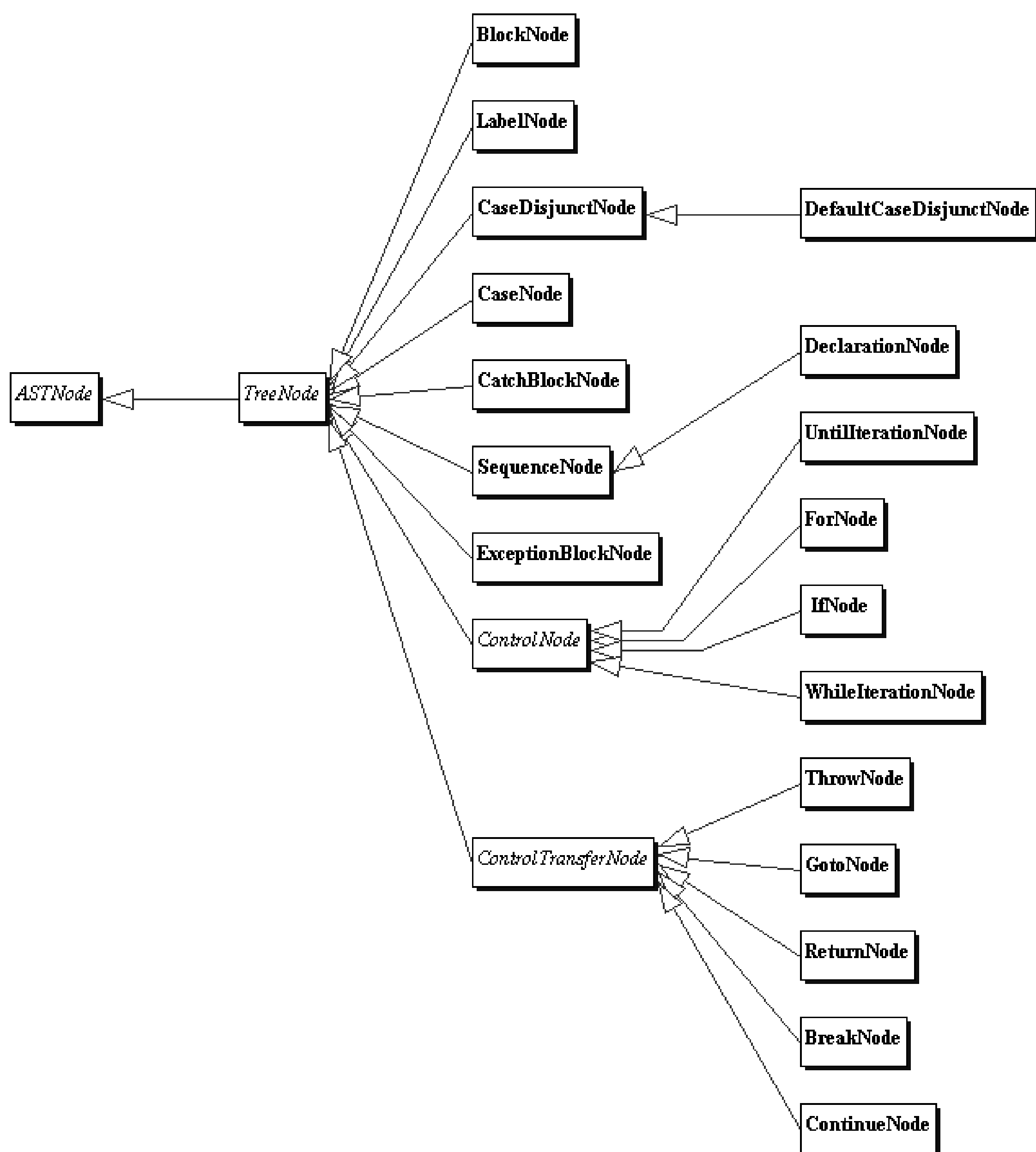



**Figure 7-2.** UML class diagram for *ControlTree*

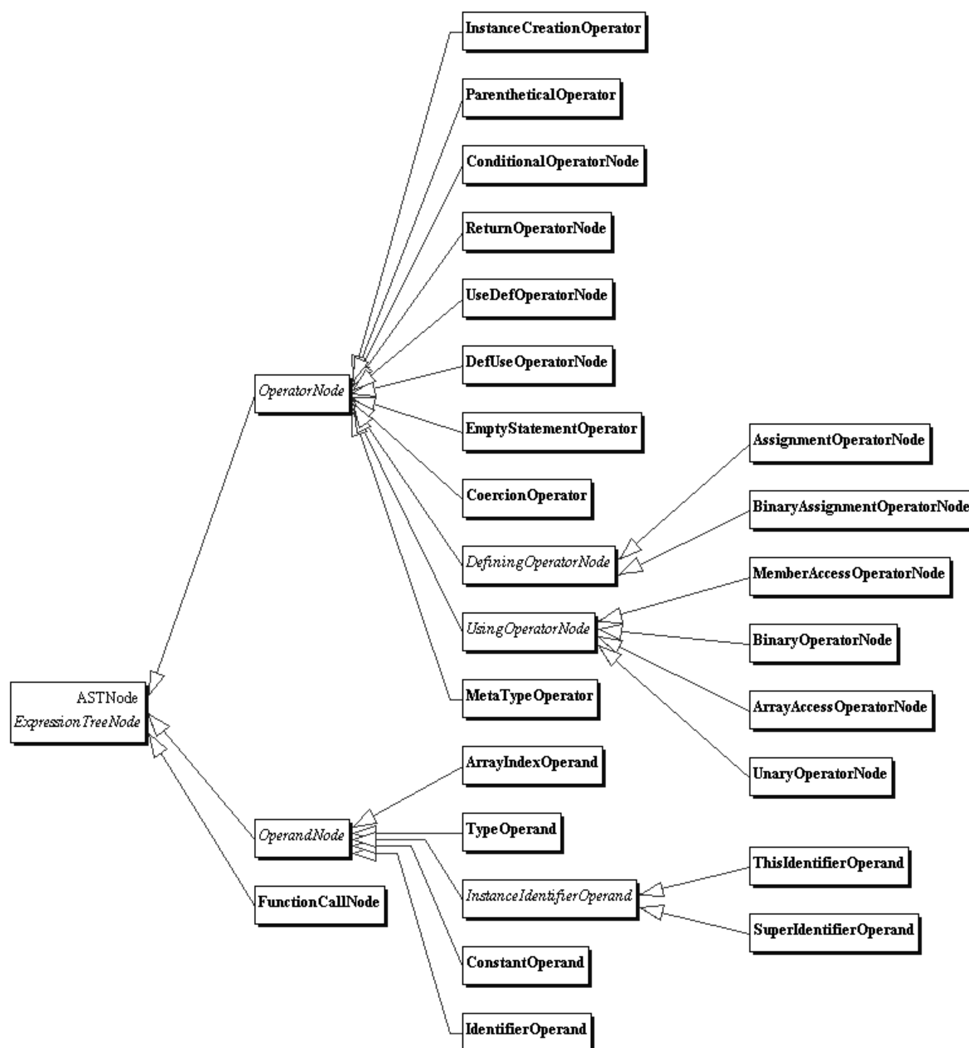



**Figure 7-3.** UML Class Diagram for AST *expression tree*

The key abstractions of the expression tree include *OperatorNode* (a family of types, each corresponding to a particular type of operator), *OperandNode* (a family of types that represent individual expression operands) and *FunctionCallNode*. *OperatorNode* is subdivided further into operators that define state (*DefiningOperatorNode*), those that use state (*UsingOperatorNode*), and other miscellaneous operators that support operations such as instance creation (*InstanceCreationOperator*), type coercion (*CoercionOperator*), conditional operators as found in Java and C++ (*ConditionalOperatorNode*), and so on. The operator family also includes the abstractions *DefUseOperator* and *UseDefOperator* for

representing prefix and postfix operators, respectively. Support for expression operands includes abstractions for representing types (*TypeOperand*), constants (*ConstantOperand*), and identifiers (*IdentifierOperand*).

#### 7.2.2.1 Intra-method Graph Representations

CBAT includes a number of higher-level representations that are used for various types of static analyses, including coupling-based testing. The most important of these is the control flow graph (CFG). The CFG is generated directly from the control tree and includes abstractions that model the various types of control flow statements at a high-degree of granularity. For example, each of the loop constructs *for*, *while*, and *until* are represented as distinct node types. This allows for ease in identifying distinct syntactic constructs for the purpose of analysis and code generation. These abstractions are language independent and the complete set is sufficiently rich to support the control flow constructs found in most object-oriented and procedural languages. New constructs can be easily added to account for new language features.

Figure 7-4 depicts the components of a method graph used to represent a control flow graph. The graph includes the following type families: *CallNode*, *CallReturnNode*, *ConditionNode*, *TransferNode*, and *PlaceholderNode*. The type family defined by CallNode includes lower level abstractions that are used to represent intra-method transfers of control that resemble procedure calls. These occur when an exception is thrown or when a code block protected by an exception handler successfully reaches the end of its body (the Java *finally* block is an example of the latter). At present, CBAT models the exception handling behavior found in Java, which is a generalization of that found in C++. When an exception is thrown in a block of code that is part of a *try* block, control first transfers to the catch handler associated with the try block. If no catch handler is present, control then transfers to either the nearest enclosing catch handler, or the method's exit node if none is present. If the exception is caught by the handler, then control is passed to the *finally* block, if present, and then to the first statement that follows the *try* block. Logically, this behavior can be

modeled as series of procedure calls that are internal to the method containing the try block, which is the approach adopted by Sinha and Harrold [67]. This is achieved in CBAT by modeling the throw of an exception first as a call to the catch handler, followed by a call to the finally block. Each call is represented as a pair of nodes, one that connects to the catch handler (a type of *CallNode* shown in Figure 7-4) and another (a type of *CallReturnNode*) that is connected to the last node of the catch handler and models the return of control. The *CallReturnNode* is then connected to another pair of nodes that model the call-return of a *finally* block.

Other high-level graph types maintained by CBAT include data dependency graphs, control dependency graphs, and coupling graphs. The representation for all of these graphs is straight forward and augments the control flow graph by adding a set of annotated edges between nodes that exhibit some relationship such as data or control dependency.

CBAT does not have an explicit abstraction for representing edges. Instead, edges are represented as a mapping between nodes. The set of mappings for a particular type of edge is implemented as a hash map. This affords the flexibility to treat all edges uniformly and to expand the type of edges supported by CBAT at any time. The sacrifice for this flexibility is added complexity for maintaining edge annotations, such as recording the fact that a particular edge has been traversed while conducting a depth first traversal (for example). CBAT's solution for handling this to maintain a hash map for each pair of nodes that is connected by some type of edge. This hash map preserves a mapping between a pair of nodes and another hash map that maintains the annotations. This later hash map is indexed by the kind of annotation that labels the node pair, and maps directly to the value of the label. The complexity of this solution is hidden behind a set of methods that allow edge annotations to be easily assigned and manipulated for a given pair of nodes.

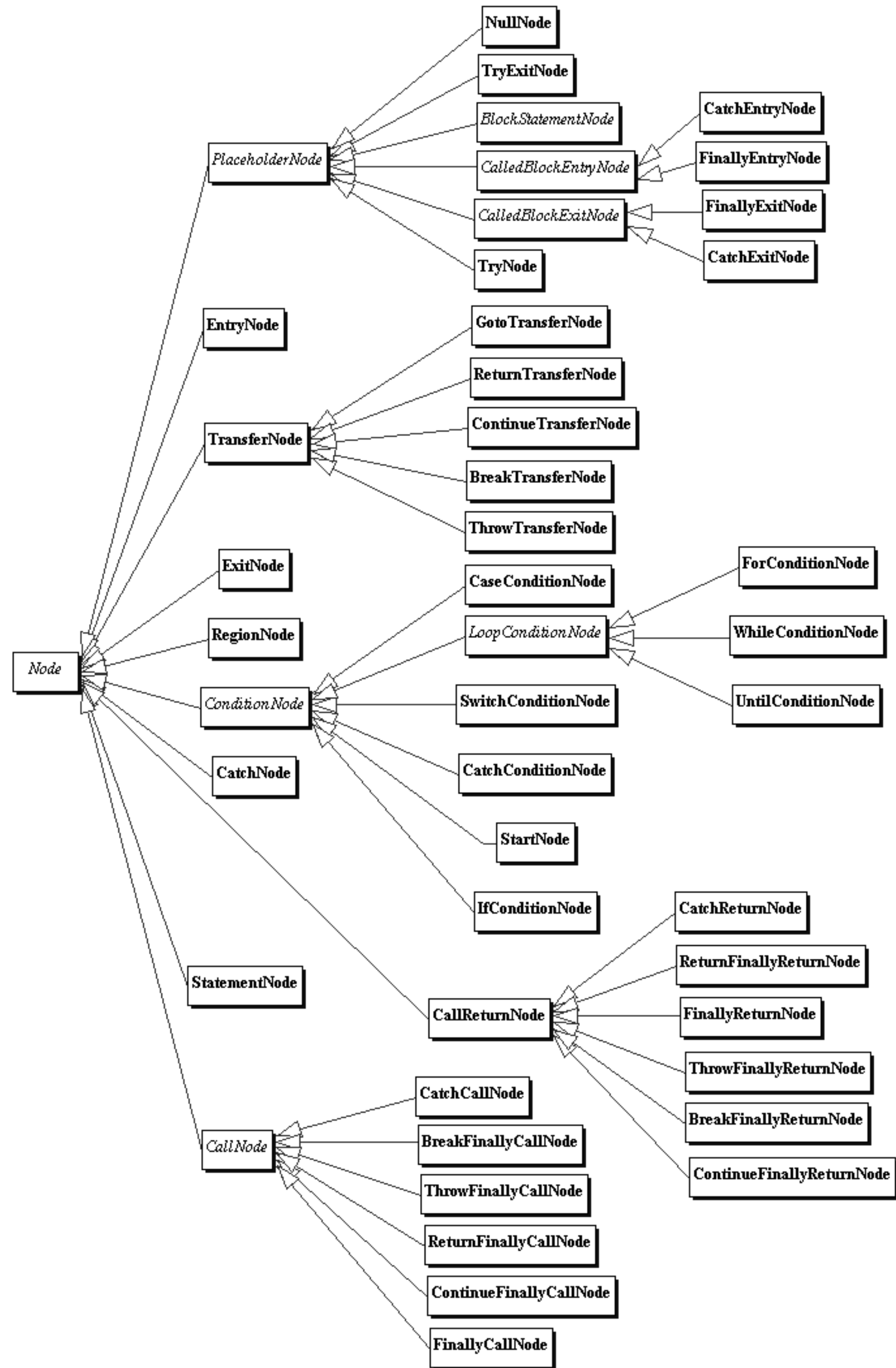



**Figure 7-4.** Class Diagram for package *MethodGraph*.

#### 7.2.2.2 Inter-method Graph Representations

In addition to the method level graph representations described above, CBAT also maintains a call graph that links each call site within a method to the called method. Each call site is actually represented as two distinct nodes in the calling method's control flow graph: a *call* node and *call-return* node. The call node is connected by a *call-return* edge that is incident upon the entry node of the called method, and the call-return node is connected by a call-return edge that emanates from the called method's exit node. Together, these call-return edges are used to represent the inter-method flow of control. CBAT also has the ability to support inter-method control and data dependencies, though the implementation of these are left as future work.

#### 7.2.2.3 Analysis Data

During the analysis of a method, CBAT collects an enormous quantity of data related to the method's control flow and data flow characteristics. Some of this information is collected specifically to support the coupling-based testing techniques described in this thesis. The following subsections describe the types of analysis data collected at the method level in terms of control flow, data flow, and coupling-related information.

#### 7.2.2.4 Control-flow information

The following control-flow information is collected for each method in a class:

**Anonymous Calls.** The set of *anonymous* calls made at a particular node. A call is anonymous if the instance context through which the call is made is not specified by a variable, but rather is supplied by the return value of a called method. For example, in the expression *o.m*().*n*(), the call to *n* is made through the anonymous context provided by the value returned from *o.m*().

**Binding Sets**. The set of types that a context variable can be bound to for a particular call site. For example, if *o* is declared to be of type *A*, and *A* is the base of an inheritance hierarchy containing *B* and *C*, then *o* can potentially be bound to instances of *A*, *B*, or *C*. The actual types comprising the binding set are determined by the binding mechanisms that reach a call site that uses *o* as the context variable. However, the actual binding set will be a subset of the type family defined by the declared type of the context variable.

**Call Sites**. The set of nodes in the control flow graph that contain calls to other methods.

**Called Class Methods.** The set of class methods called at a particular node.

**Called Instance Methods.** The set of instance methods called at a particular node.

**Control Flow Paths.** List of logical paths through the control flow graph. Each path is represented as a sequence of nodes that result from a depth-first traversal from the entry node to the exit node, traversing the body of any loops at most once.

**Explicit Calls.** Records the set of superclass methods called explicitly at a given node through the keyword "super", as in *super.m*().

**Indirect Calls.** Records the list of calls made at a given node through an instance context, as in *o.m*().

**Node Predecessors**. Records the list of nodes that are control predecessors of a particular node.

**Node Successors**. Records the list of nodes that are control successors of a particular node.

#### 7.2.2.5 Data flow information

The following data-flow information is collected for each method in a class:

**Anonymous Definitions**. Records the set of anonymous variable definitions that are made at a particular node. Anonymous definitions occur when the instance that specifies the variable is not specified by a variable. This occur when the instance is returned by a method call. For example, in the expression $o.m().v = 0$, the definition of $v$ is anonymous since the instance context is determined by the value returned by $m$.

**Anonymous Uses**. Records the set of anonymous variable uses that are made at a particular node. Anonymous uses occur when the instance that specifies the variable is not specified by a variable. This occurs when the instance is returned by a method call. For example, in the expression $b = o.m().v + 0$, the use of $v$ is anonymous since the instance context is determined by the value returned by $m$.

**Defined Class Variables**. Records the list of class variables defined at a particular node.

**Defined Instance Variables**. Records the list of instance variables defined at a particular node.

**Explicit Definitions**. Records the list of superclass methods called through the keyword *super* at a particular node.

**Explicit Uses**. Records superclass variables used through the keyword *super* for a particular method.

**Indirect Uses.** Records the list of state variables used at a given node that are referenced through an instance context, as in *o.v*.

**Live Aliases**. Records the list of variable aliases that reach a given node.

**Reaching Definitions**. Records the list of variable definitions that reach a particular node.

**Used Class Variables**. Records the list of class variables used by a particular node.

**Used Instance Variables**. Records the list of instance variables used by a particular node.

#### 7.2.2.6 Coupling-related information

The following data-flow information is collected for each method in a class:

**Coupling Sequences**. Records the set of coupling sequences for the coupling method. Each coupling sequence is represented by an instance of the type *CouplingSequence*. Instances of this type record the set of coupling variables in the sequence, the definition in the coupling method that provides the instance context, the antecedent and consequent call sites, and the set of transmission paths between the call sites.

**First-use Paths**. Records the set of paths that lead to each first-use of a state variable or formal method argument.

**First Uses**. Records the set of nodes that have first uses of variables (state or formal arguments). Each node is mapped to a set of pairs, with each pair containing the coupling path *p* and the set of first-use variables that *p* is definition-clear with respect to.

**Last Definition Paths**. Records the set of last-def paths that lead from the last definition of a state variable or formal method argument to the exit node of the method.

**Last Definitions**. Records the set of nodes that have last definitions of variables (state or formal arguments). Each node is mapped to a set of pairs, with each pair containing the coupling path *p* and the set of last-use variables that *p* is definition-clear with respect to.

## 7.3 Architecture of CBAT

Figure 7-5 presents the high-level architecture of CBAT. The principle components include the *CBAT Core*, the *Analysis Engine*, and the *Instrumentation Engine*. Each of these is discussed in the following sections.

### 7.3.1 CBAT Core

The CBAT Core consists of the *Parse Tree Generator*, and the *Class and Method Graph Generator*. Each of these components is described in the following subsections.

#### 7.3.1.1 Parse Tree Generator

The Parse Tree Generator transforms Java compilation units expressed in source form into a parse tree that is based on the language grammar (Java in this case). The transformation process maps nodes corresponding to non-terminals in the language grammar into nodes of the parse tree that represent the individual syntactic units of the language. Each non-leaf node of the parse tree corresponds to a single non-terminal in the language grammar, and

each leaf node corresponds to a terminal whose value is a lexeme generated by the parsing activity.

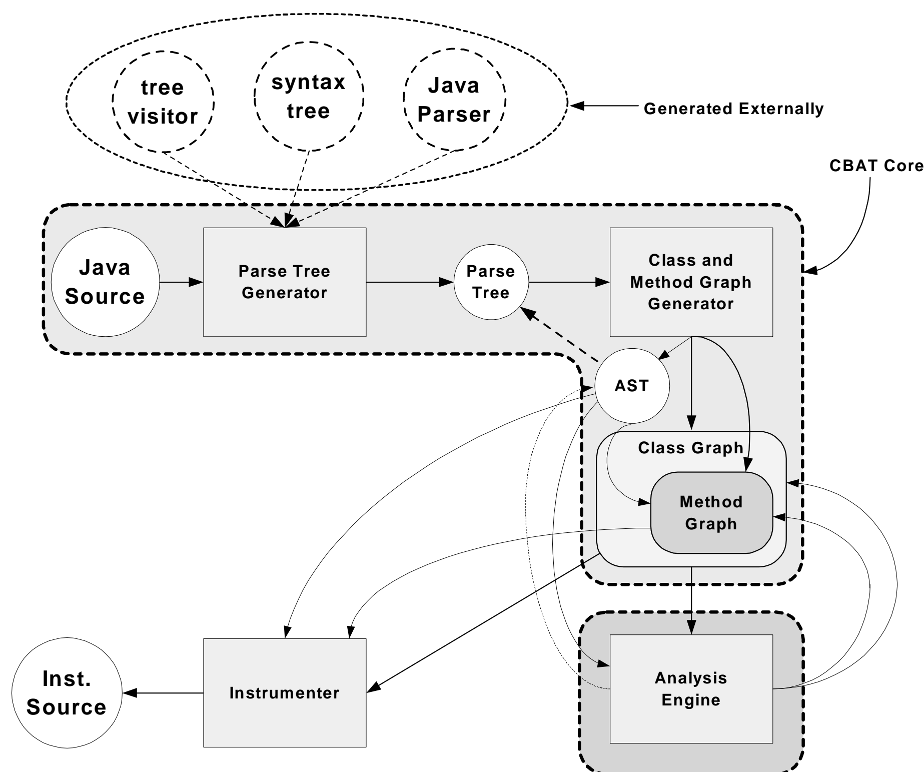



**Figure 7-5.** CBAT architecture

The parse tree is generated from a source file using a parser generated by the compiler generator JavaCC.[1] The grammar used by JavaCC was annotated by the Java Tree Builder (JTB) with semantic actions that yield a parse tree at runtime.[2] The resulting parse tree is expressed as a class hierarchy that represents the elements of the language. Each class in the hierarchy corresponds to a non-terminal or terminal in the language. The architecture of the parse tree is based on the *Visitor* design pattern [GOF 95].

1. JavaCC is available from http://www.metamata.com.
2. The Java Tree Builder is available from http://www.cs.purdue.edu/jtb.

#### 7.3.1.2 Class and Method Graph Generator

The Class and Method Generator is implemented as a pair of visitor-based utilities that process the parse tree and generate a corresponding *Class Graph* and *Method Graph*. The *Class Graph Generator* (CGG) walks the parse tree produced by the parser and creates instances of *Package*, *ClassElement*, *Interface*, *Variable*, and *Method* as the corresponding syntactic elements are encountered. When a method body is found, the *Method Graph Generator* (MGG) creates the corresponding control tree that represents the individual statements of the method. Individual expressions are transformed by the MGG into expression trees and associated with the corresponding control tree nodes.

### 7.3.2 Analysis Engine

The Analysis Engine is a collection of static analysis tools that utilize the information represented in the class and method graphs (and parse tree in some circumstances) to produce additional or refine existing representations and to generate various analysis information. For example, the *ControlFlow Analyzer* uses the information contained in the *Method Graph* for a particular method to generate the corresponding control flow and data flow graphs. The corresponding *Method Graph* is then updated to reflect this newly derived information. The other analyzers that currently exist in CBAT include *ControlDependency Analyzer*, *Coupling Sequence Analyzer*, and *Call Graph Analyzer*. Additional analyzers can easily be added as new types of information are required.

### 7.3.3 Instrumentation Engine

The final component of CBAT is the Instrumentation Engine. Similar to the Analysis Engine, Instrumentation Engine is a collection of utilities that are used to generate instrumented source code.

The strategy used for generated instrumented programs does not produce a new program directly from the Class and Method Graphs. Rather, a set of *instrumentation instructions* expressed in XML are generated that are subsequently used to produced the instrumented code. This is illustrated in Figure 7-6. As shown, a instruction generator is utilized that gen-

erates instructions for instrumenting a Java program to collect coverage information for the *All-Coupling-Sequences* test adequacy criterion. The instructions are then used to generate

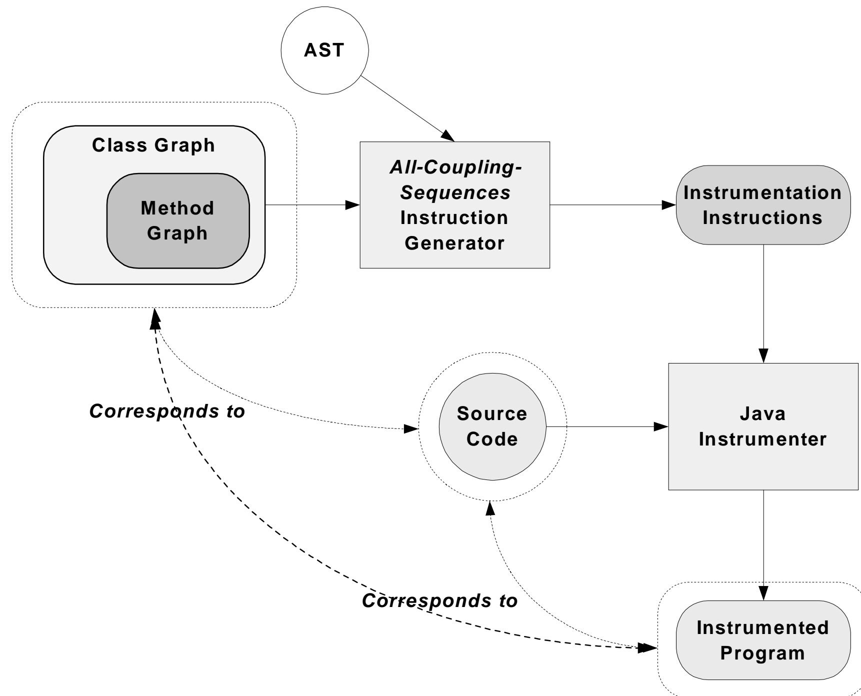



**Figure 7-6.** CBAT Instrumenter

an instrumented source file by the Java Instrumenter, along with the original source code corresponding to the class and method graphs of the unit under test.

# 7.4 Implementation

The CBAT core and Analysis Engine are implemented in Java and consist of approximately 90,000 lines of code. Approximately 20,000 lines were generated by the JavaCC and JTB tools. The Instrumentation Engine is written in Perl and consists of approximately 2,000 lines of code.

The following tools were used in the development of CBAT:

- Java Development Kit 1.1.2
- javacc (parser generator)
- Java Tree Builder (JTB) version 1.2.2

- Java Generic Library (JGL) version 3.1
- OROMatcher (regular expression package) version 1.1.0a
- CodeWright (program editor) version 6.5
- BugSeeker2 (source debugger) version 1.0.2
- TogetherJ (UML modeling tool) version 4.1
- Perl version 5.6

# 8. Validation

This chapter describes the experiments used to empirically validate the efficacy of the object-oriented coupling-based testing criteria described in Chapter 4. It begins with discussion of the experimental design, which includes a description of the experimental subjects, the test adequacy criteria used for comparative purposes, the test data, and the fault types that are used for evaluating the effectiveness of the criteria. The procedures for the actual conduct of the experiment are then described, followed by the results of the experiments. Finally, the chapter concludes with a discussion of the significance of the results.

## 8.1 Experimental design

The following sections describe the experimental design used to validate the research contained in this thesis.

### 8.1.1 Subject programs

Each subject program used in these experiments consists of a collection of classes that are integrated with a client method, the *method under test*. Each of these classes includes at least one method having one or more coupling sequences with respect to a particular class hierarchy, referred to as the *subject hierarchy*.

Table 8-1 summarizes the subject programs used in these experiments. The column labeled $f$ identifies the method under test and $|s_f|$ is the number of coupling sequences contained within $f$. Each coupling sequence has a context variable, which defines induces a type family. The column labeled $F_{S_f}$ gives the number of classes in this type family (inheritance hierarchy) for the corresponding program.[1] The column labeled *Description* indicates the source from which each program was obtained. Five programs (P1, P2, P3, P5, and P6)

---

1. The term program includes $f$ (the method under test), the class that specifies $f$, and all classes in the type family specified by the context variable of each coupling sequence.

were examples created specifically to ensure that all of the subject faults were tested by at least one experiment. Of the remaining five subject programs, one was developed by a graduate student (P4), two were developed by a professional programmer having 15 years of experience (P7 and P8). The remaining two are open source products: ANTLR (a parser generator) and JMK (a build system, similar to make).[1]

**Table 8-1.** Subject program characteristics

| $f$ | $\lvert s_f \rvert$ | $F_{S_f}$ | **Description** |
|---|---|---|---|
| P1 | 4 | 4 | Polymorphic Example |
| P2 | 5 | 5 | Polymorphic Example |
| P3 | 1 | 5 | Polymorphic Example |
| P4 | 1 | 4 | Student Developer |
| P5 | 3 | 4 | Polymorphic Example |
| P6 | 3 | 5 | Polymorphic Example |
| P7 | 6 | 4 | Professional Developer |
| P8 | 20 | 5 | Professional Developer |
| P9 | 11 | 16 | Open Source (ANTLR) |
| P10 | 7 | 9 | Open Source (JMK) |

## 8.1.2 Test adequacy criteria

This experiment evaluated the following four test adequacy criteria:

1. All-Coupling-Sequences
2. All-Poly-Classes
3. All-Poly-Defs-Uses
4. Branch Coverage

---

1. ANTLR is available from http://www.antlr.org/ and JMK from http://sourceforge.net/projects/jmk.

The first three of these are the primary coupling-based testing criteria presented in Chapter 5 and are the subjects of the investigation for the experiments described in this chapter. The fourth, Branch Coverage, is used as the control to determine if the other criteria are effective at detecting faults. Branch testing is a unit-level white box testing technique, and seeks to "execute enough tests to assure that every branch alternative has been executed at least once" [9]. Attaining this goal yields a branch coverage measure of 100 percent. The justification for selecting Branch Coverage as the control is that Branch Coverage is a commonly used white box testing technique and is commonly used to test individual procedures and functions, and the coupling-based testing criteria are also white box testing approaches.

### 8.1.3 Test data

The test data used in the experiments were drawn randomly according to a uniform distribution. The data itself was produced from custom test data generators developed in Perl for each of the test adequacy criteria. In all cases, sufficient data was generated to achieve 100% coverage for a given criterion.

The strategy used to select test cases is similar to how test cases are selected for the Branch Coverage test adequacy criterion. For each coupling sequence, the path expression [9] necessary to execute the sequence was identified. These expressions were then used to create Perl programs that would generate the test data necessary to execute the set of sequences for the method under test. A similar procedure was followed for testing the state space interactions between antecedent and consequent methods. These path expressions ensured that the required coupling paths were covered. Table 8-2 summarizes the number of test cases for each combination of subject program and test adequacy criterion. For ACS, the number

**Table 8-2.** Number of test cases per subject program and criterion

| *f* | **ACS** | **APC** | **APDU** | **BC** |
|---|---|---|---|---|
| P1 | 2 | 4 | 6 | 1 |
| P2 | 2 | 5 | 320 | 2 |
| P3 | 2 | 5 | 80 | 2 |

**Table 8-2.** Number of test cases per subject program and criterion

| *f* | **ACS** | **APC** | **APDU** | **BC** |
|---|---|---|---|---|
| P4 | 1 | 3 | 3 | 1 |
| P5 | 2 | 5 | 75 | 1 |
| P6 | 2 | 5 | 105 | 1 |
| P7 | 1 | 2 | 64 | 1 |
| P8 | 4 | 2 | 42 | 4 |
| P9 | 6 | 15 | 95 | 6 |
| P10 | 4 | 9 | 27 | 4 |

of test cases is determined by the number of coupling sequences and control flow paths present in the method under test. For APC, the number of test cases is also determined by the size of the type family for the coupling variable. Finally, for APDU, the number of test cases is determined by adding the number of control flow paths in the antecedent and consequent methods to the test cases for APC and ACS.

### 8.1.4 Injected Faults

Each subject program *P* was seeded by injecting faults into the bodies of the antecedent and consequent methods for each member of each type family induced by the declared type of the coupling sequences in *P*. The types of faults injected into each unit under test are a subset of those described in Chapter 3.[1] Not all of those fault types will manifest failures as a result of integration of a type hierarchy with a method under test. For example, the fault types that involve anomalous construction behavior (e.g. ACB1 and ACB2) require testing approaches that focus on integrating a new class into an existing hierarchy, which is different from the testing approach required for integration of a hierarchy with a calling client method. Consequently, different testing approaches (possibly using a derivation of the coupling-based techniques) will be required, and these are left as future work. Table 8-3 summarizes both the number and type of faults that were injected.

---

1. *Unit under test* refers the method under test *f* and all artifacts associated with the integration of a particular type family through the coupling sequences defined by *f*. This includes all types, and the corresponding state variables and methods specified by those types.

**Table 8-3.** Number of faults injected into method under test

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

# 8.2 Conduct of Experiments

The testing and evaluation procedure used to validate the research in this thesis consists of four essential steps: (1) *test oracle derivation*, (2) *fault injection*, (3) *test execution*, and (4) *result evaluation*. The objective of the first step is to create a test oracle that can be used to evaluate the results of subsequent tests. That is, given a test result associated with a particular test case, the oracle determines if the test passes or fails. For the second step, fault injection, each subject program is injected with faults that yield a seeded version. This seeded version is used as the primary experimental subject. The third step executes each subject program using the test cases and records the outcome. The final step uses the test oracle to determine if the outcome of each execution for the corresponding test case detects a fault. The actual procedures in some of these steps vary according to the test adequacy criterion being evaluated.

The testing and evaluation procedure is discussed in detail in the following subsections. The steps of the procedure that are specific to particular criteria are labeled with the names of the applicable criteria in parentheses at the beginning of each step. Those steps not having this list are applicable to all of the subject criteria.

### 8.2.1 Test oracle derivation

For each $s_{j,k} \in S_f$, where $S_f$ is the set of coupling sequences for the method under test *f*:

1. Execute *f* using at least one test case $c \in C_{f, s_{j,k}}$ such that the context variable *o* of $s_{j,k}$ is bound to an instance of the declared type of *o*, where $C_{f, s_{j,k}}$ is the set of test cases for *f* associated with coupling sequence $s_{j,k}$.[1]

2. Record this result as $R^T_{s_{j,k}} = f_{s_{j,k}}(c)$, where $S_f$ is the set of coupling sequences in the method under test, and *T* is the declared type of the context variable of $s_{j,k}$. Add the result to the test oracle for *f*, $\Omega_f$.

3. (*All-Poly-Def-Uses*): For each $t \prec T$, each $v \in \Theta^t_{s_{j,k}}$, for each last-definition $d_v$ of $u_v$ in the antecedent method $\alpha_{s_{j,k}}$ and each corresponding first-use of *v* in the consequent method $\omega_{s_{j,k}}$, where $\Theta^t_{s_{j,k}}$ is the coupling set that results when the context variable of $s_{j,k}$ is bound to an instance of *t*:[2]

   - Execute *f* using at least one test case $c \in C_{f, s_{j,k}}$ such that the context variable is bound to an instance of its declared type, and there is a coupling path from $d_v$ to $u_v$ with respect to *v*.

4. (*All-Poly-Def-Uses*): Record the state of the instance bound to the context variable after the antecedent has executed, $\alpha^{d_v}_{s_{s,k}} = f(c)$, and immediately after each first-use in the consequent method, $\omega^{u_v}_{s_{j,k}} = f(c)$. Add the result tuple $[(t, v), (\alpha^{d_v}_{s_{s,k}}, \omega^{u_v}_{s_{j,k}})]$ to the test oracle $\Omega_f$, where the pair $(t,v)$ records the type of the instance bound to the context variable and also the coupling variable *v* that corresponds to the coupling path being tested; $(\alpha^{d_v}_{s_{s,k}}, \omega^{u_v}_{s_{j,k}})$ is the result pair for the test.

---

1. Traditionally, in the scientific testing literature, the letter "*t*" is used to represent a test case, In this thesis, we reserve *t* for representing the type of the instance bound to a coupling variable. Therefore, "*c*" is used to represent a test case.

2. For the *All-Poly-Classes* criterion, this same procedure is used except that the state variables $v \in \Theta^t_{s_{j,k}}$, are not captured and recorded as part of the test oracle $\Omega_f$.

### 8.2.2 Fault injection

1. For each $s_{j,k} \in S_f$ and $t \in (family(T) - \{T\})$, inject faults into each method of $t$ that overrides either the antecedent or consequent methods of $s_{j,k}$. This yields the fault-seeded type $t' \prec T$. This results in a *shadow inheritance hierarchy* rooted at $T$, where $T$ is the declared type of the context variable of $s_{j,k}$, as illustrated in Figure 8-1. The shadow hierarchy mirrors the original hierarchy in structure below the root, but is seeded with faults.

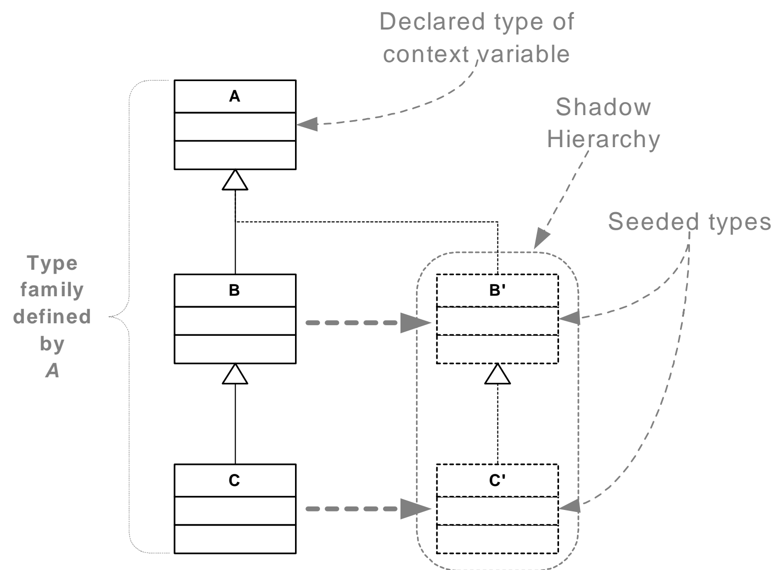



**Figure 8-1.** Class hierarchy with seeded shadow hierarchy for *All-Poly-Classes*

2. (*All-Poly-Def-Uses*): For each $s_{j,k} \in S_f$, $t \in (family(T) - \{T\})$, and $v \in \Theta^t_{s_{j,k}}$, where $\Theta^t_{s_{j,k}}$ is the coupling set that results when the context variable of $s_{j,k}$ is bound to an instance of $t$, inject corresponding faults into the antecedent and consequent methods, yielding the fault seeded type $t'' \prec T$ (i.e. $t''$ is a subtype of $T$) contains methods $\alpha'_{s_{j,k}}$ and $\omega'_{s_{j,k}}$, respectively. This results in a shadow inheritance hierarchy rooted at $T$, where $T$ is the declared type of the context variable of $s_{j,k}$, as illustrated in Figure 8-2.

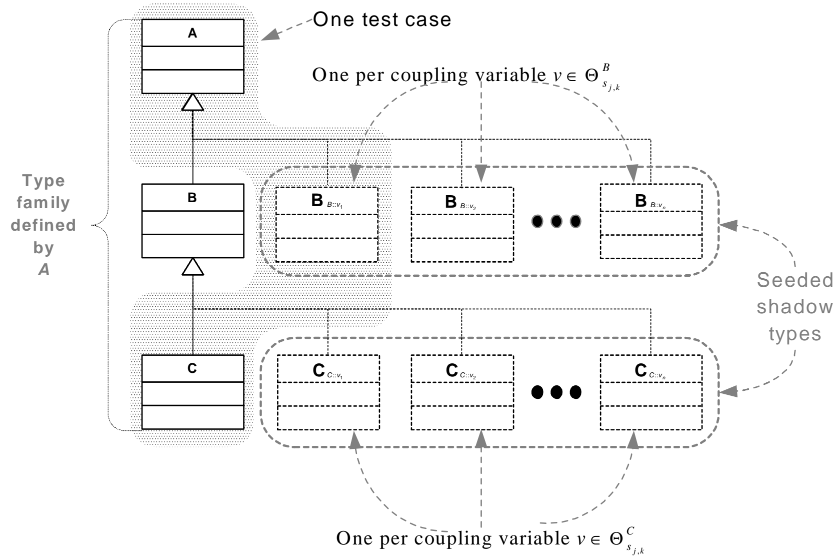



**Figure 8-2.** Class hierarchy with seeded shadow types for *All-Coupling-Defs-Uses*

## 8.2.3 Test execution

1. For each $s_{j,k} \in S_f$ and $t \in (family(T) - \{T\})$, execute $f$ using a test case c that binds the context variable to the corresponding fault-seeded type $t'$. Add the result $R^{t'}_{s_{j,k}} = f_{s_{j,k}}(c)$ to the test result set for $f$, $\Psi_f$.

2. For each test case $c \in C_{f, s_{j,k}}$, execute $f$ using $c$, and record the state of the instance bound to the context variable for the corresponding pairs of last-definitions and first-uses: $\alpha'^{d_v}_{s_{s,k}} = f(c)$ and $\omega'^{u_v}_{s_{j,k}} = f(c)$, respectively. Add $(\alpha^{d_v}_{s_{s,k}}, \omega^{u_v}_{s_{j,k}})$ to the test result set for $f$, $\Psi_f$.

## 8.2.4 Result evaluation

1. Compare each test result $R^{t'}_{s_{j,k}} \in \Psi_f$ with the corresponding pair in the test oracle: $R^{t'}_{s_{j,k}} = R^{T}_{s_{j,k}} \Rightarrow pass_t$. This ascertains whether or not an instance of the descendant type $t$ can be substituted freely for an instance of the declared type $T$ of the context variable.

2. (*All-Poly-Def-Uses*): Compare each test result $(\alpha'^{d_v}_{s_{s,k}}, \omega'^{u_v}_{s_{j,k}}) \in \Psi_f$ with the corresponding pair in the test oracle: $(\alpha'^{d_v}_{s_{s,k}} = \alpha^{d_v}_{s_{s,k}}) \wedge (\omega'^{u_v}_{s_{j,k}} = \omega^{u_v}_{s_{j,k}}) \Rightarrow pass_v$. This ascertains if the method under test preserves the fidelity of the interactions between the antecedent and consequent methods when the context variable *o* is bound to an instance of a particular type in the family determined by the declared type of *o*.

3. $pass_t \wedge pass_v \Rightarrow pass_{test}$

# 8.3 Results

Table 8-4 summarizes the results of each experiment. The table shows for each fault type the number of faults seeded, the number of faults detected, and the detection effectiveness (see Table 3-1 on page 56 for a summary description of the fault types) . The last column presents the average detection effectiveness per combination of criterion and fault type for each program. Effectiveness is defined as a ratio of the number of faults detected to the number of faults seeded. The shaded blocks correspond to combinations of program and fault type that were not tested. In these cases, the subject programs did not exhibit the structural characteristics necessary to support the syntactic pattern for the fault type. The last group of rows in the table summarizes by criterion the number of faults that were seeded, the number of faults detected, and the average detection effectiveness.

**Table 8-4.** Experimental Results

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

# 8.4 Analysis and Discussion

Figure 8-3 shows a plot of the detection effectiveness per criterion for each fault type averaged (i.e. the mean) over all programs. The individual data points were weighted to reflect the differences in the number of faults seeded for each combination of program and test adequacy criterion.Thus, the data points are comparable.

A cursory examination of the plot reveals that apparently the most effective of the coupling-based test adequacy criteria within the experimental is *All-Poly-Def-Uses* (APDU), having an average detection effectiveness across fault types of $\bar{X}_{APDU} = 0.66$. The other coupling-based criteria have average detection effectiveness of 0.45 (APC) and 0.25 (ACS), with Branch Coverage having the lowest detection effectiveness of 0.11. Plots for the average effectiveness of each program across all subject criteria are given in the figures 8-4 through 8-11.

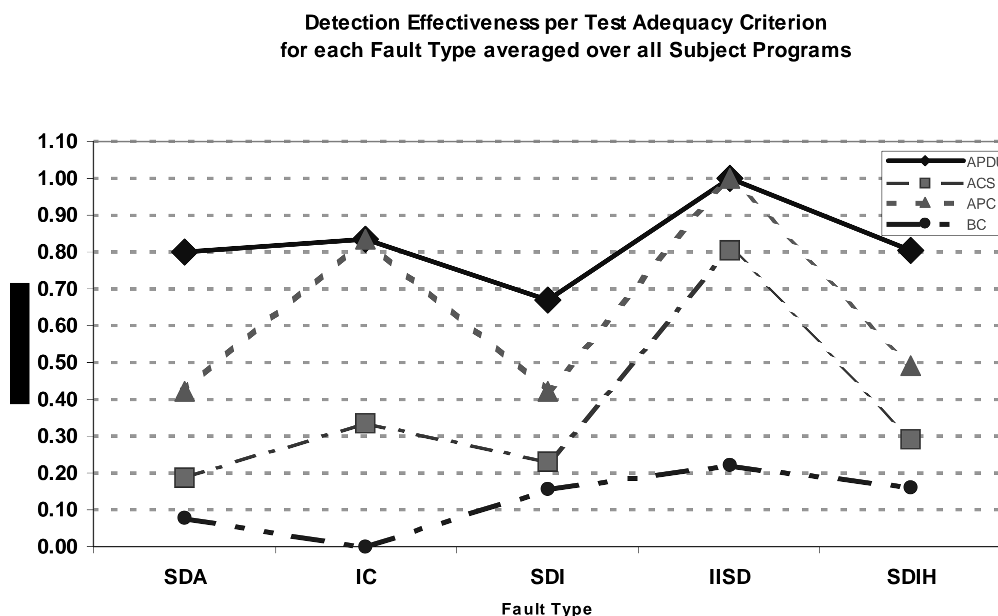



**Figure 8-3.** Average detection effectiveness by fault type

All three of the coupling-based testing criteria exhibit basically the same fault detection pattern. That is, each is more or less effective for the same fault types. For example, all three do reasonably well at detecting faults of type SDA, SDI, and SDIH, with the corresponding detection effectiveness across this sequence being monotonically increasing. In contrast, all three are much less effective at detecting faults of type IC and IISD. Note that in all cases,

across all fault types all four criteria appear to exhibit an ordering with respect to the average detection effectiveness across fault types (i.e. BC < ACS < APC < ADIH).

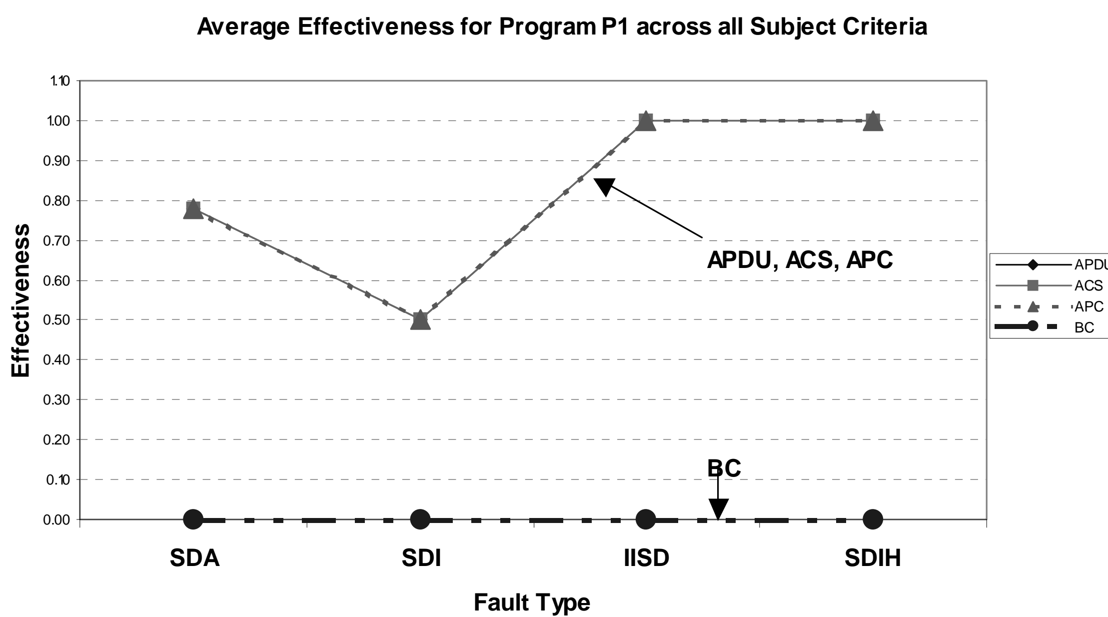



**Figure 8-4.** Average effectiveness for program P1 across all subject criteria

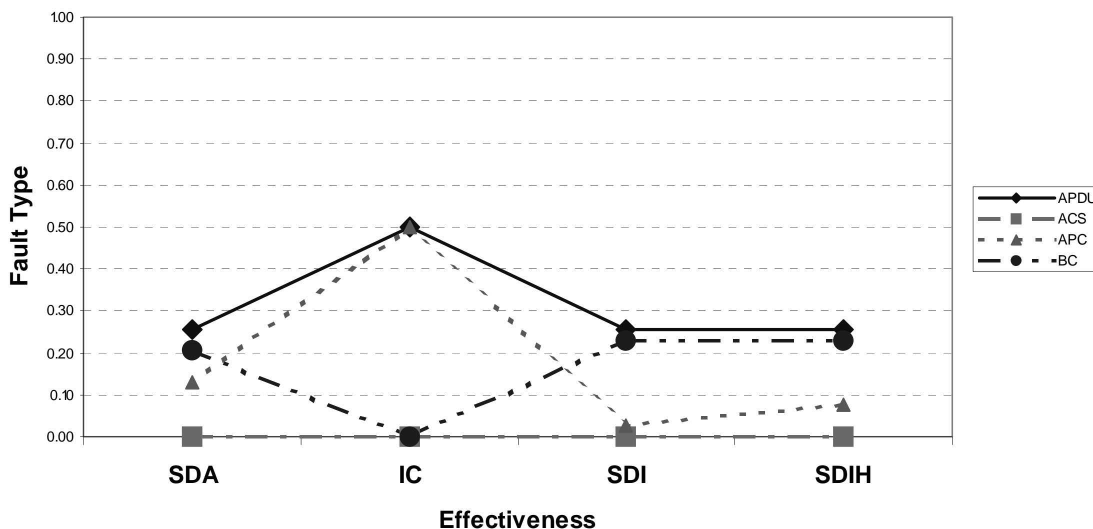



**Figure 8-5.** Average effectiveness for program P2 across all subject criteria

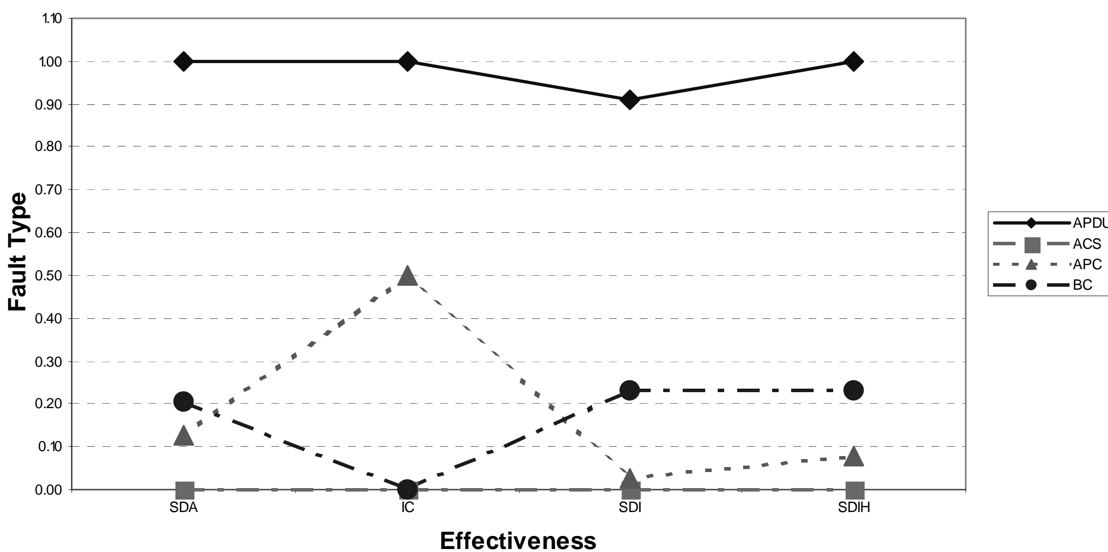



**Figure 8-6.** Average effectiveness for program P3 across all subject criteria

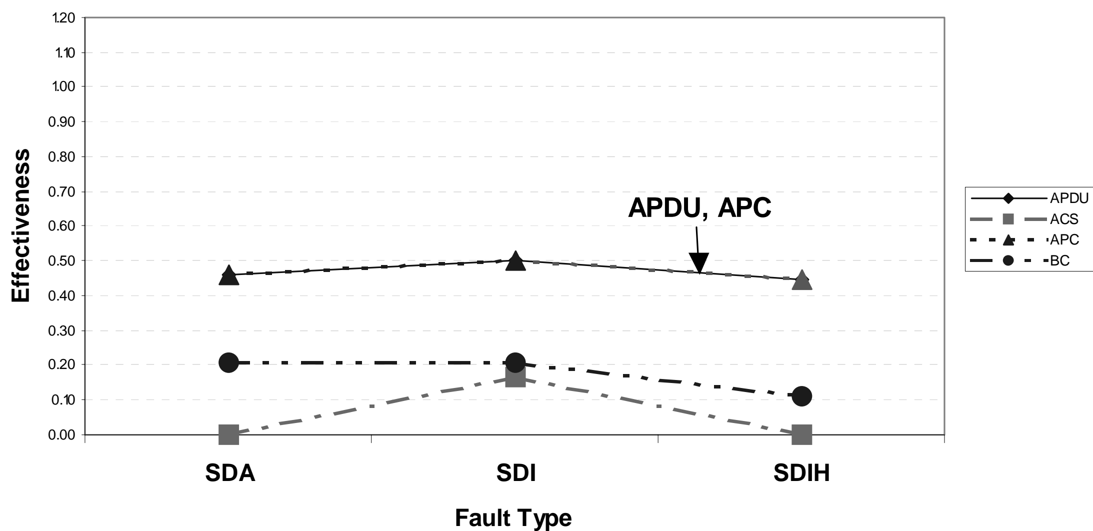



**Figure 8-7.** Average effectiveness for program P4 across all subject criteria

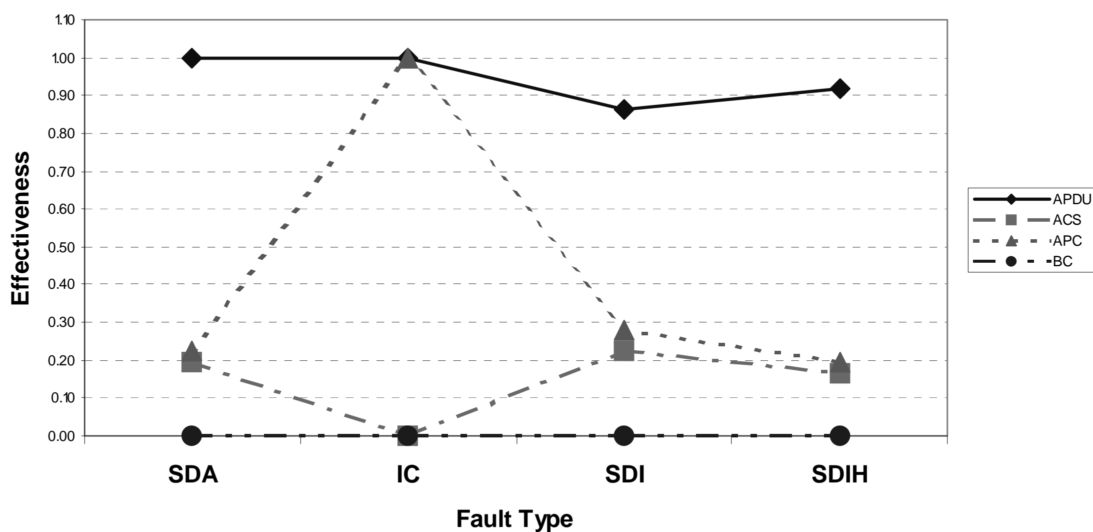



**Figure 8-8.** Average Effectiveness for Program P5 across all Subject Criteria

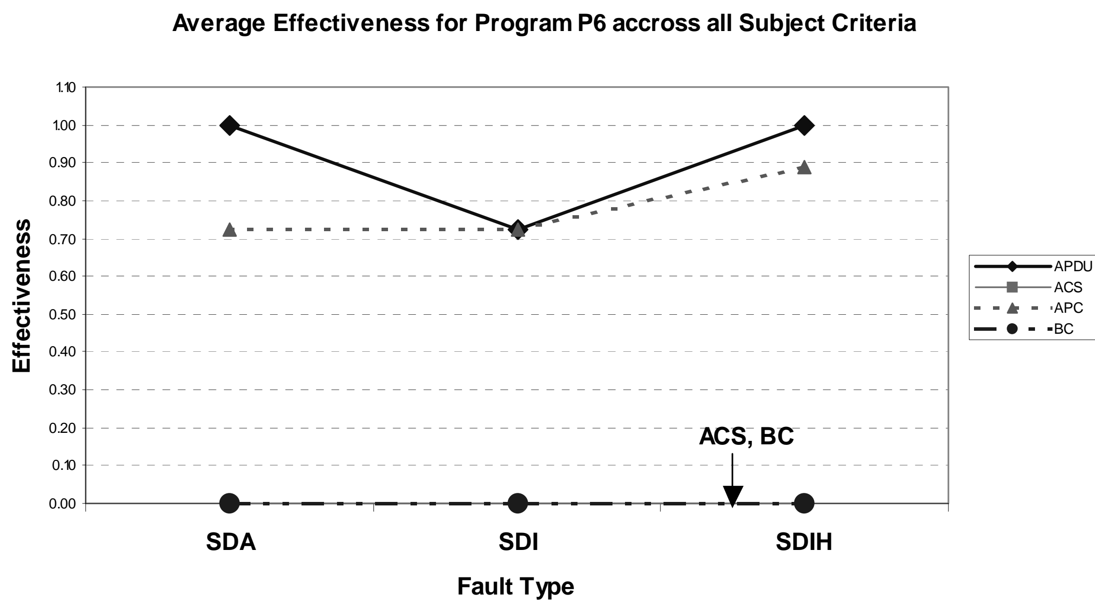



**Figure 8-9.** Average effectiveness for program P6 across all subject criteria

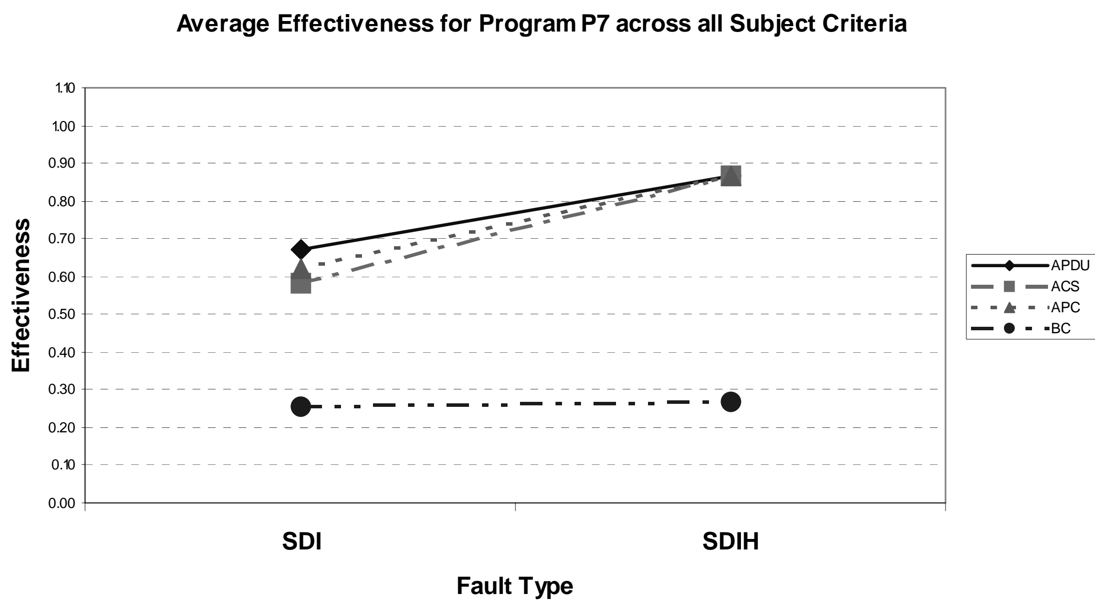



**Figure 8-10.** Average effectiveness for program P7 across all subject criteria

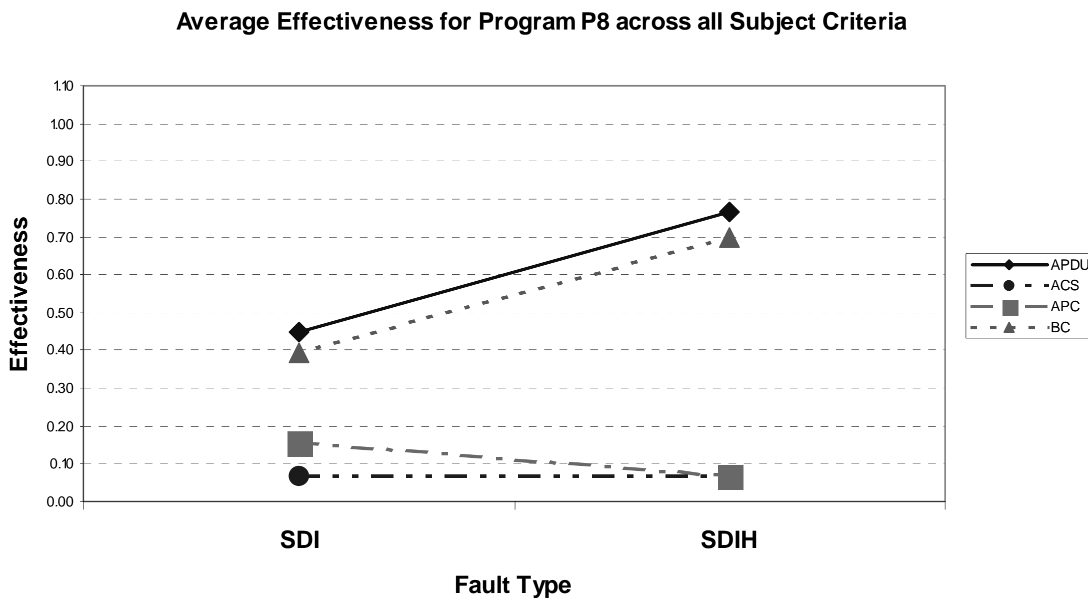



**Figure 8-11.** Average effectiveness for program P8 across all subject criteria

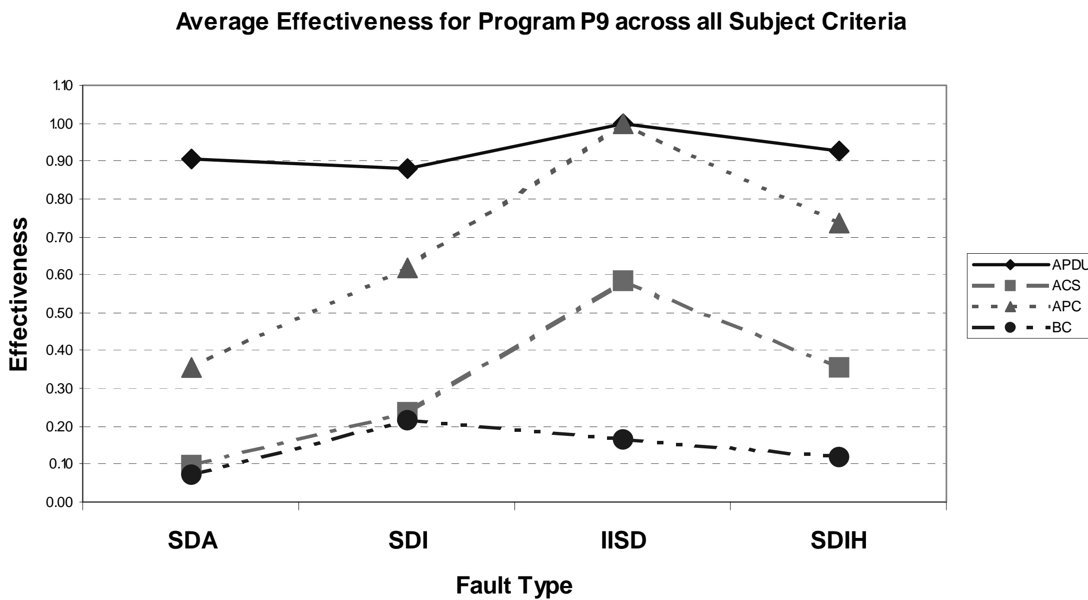



**Figure 8-12.** Average effectiveness for program P9 across all subject criteria

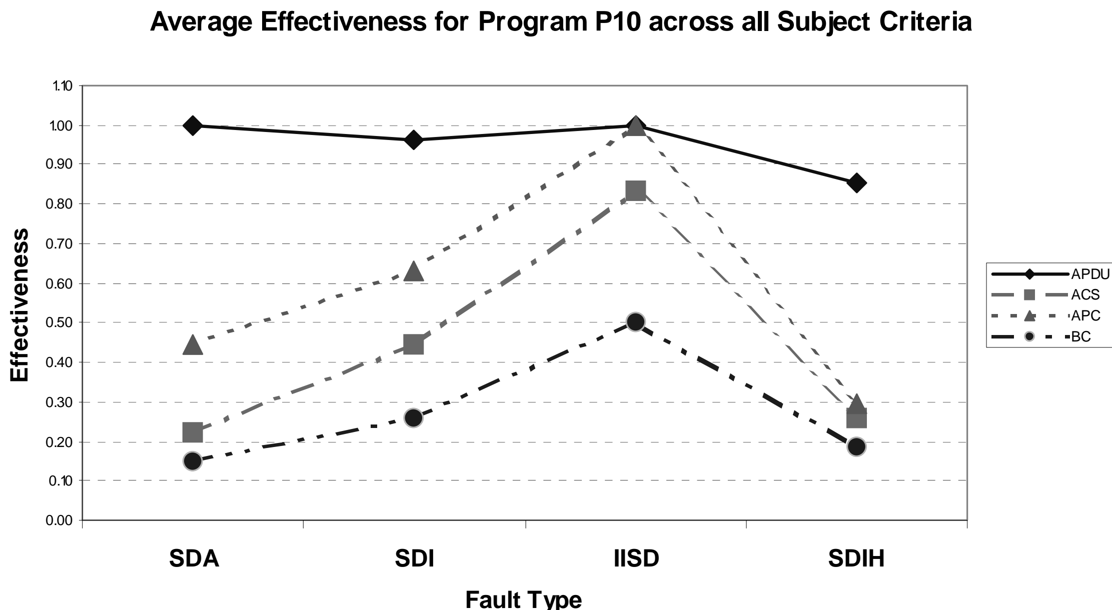



**Figure 8-13.** Average effectiveness for program P10 across all subject criteria

## 8.4.1 Analysis of the coupling-based criteria

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

For the SDI fault type, the average detection effectiveness of APDU is 0.66 which is approximately an 18 percent reduction, making it not quite as effective as for SDIH faults. Similarly, the remaining coupling criteria also reflect a reduced average detection effectiveness. APC is reduced by approximately 14 percent, yielding 0.42, and for ACS, the reduction is sightly more at 21 percent, yielding a detection effectiveness of 0.23. The detection effectiveness for Branch Coverage remains the same at 0.16.

For the SDA fault type, APDU remains the most effective, having the same average detection effectiveness as SDIH fault types (0.80). Both APC and ACS suffer reductions, having

a detection effectiveness of approximately 0.42 and 0.19. This represents a decrease of approximately 14 percent for APC as compared to its average detection effectiveness for SDIH faults. APC did no worse for SDA faults than it did for SDI Faults. For ACS, the reduction is approximately 34 percent from its detection effectiveness for SDIH faults, and a decrease of approximately 17 percent as compared to its effectiveness for SDI faults. Branch coverage drops to a detection effectiveness of 0.08, a decrease of 50 percent.

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

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

### 8.4.2 Explanation of effects

The variation in the detection effectiveness among the coupling criteria is of no surprise. The weakest of the coupling criteria, ACS, does not consider the effects on state space interactions caused by inheritance and polymorphism, and this could account for its relatively poor performance as compared to the remaining two. As described in Section 3.1 on page 53, the first condition of the fault/failure model is that a location that contains a fault must be reached before the fault can manifest a failure. The shortcoming of ACS is that its

requirements are weak in that not all locations that can contain faults due to inheritance and polymorphism must be executed. By their very nature, these faults will be located within the hierarchy associated with the objects being integrated, not in the method under test. Thus, faults at these locations will not necessarily be executed as a result of testing according to the ACS criterion.

As expected, the APC criterion performs better than ACS. This is due to the stronger testing requirements imposed by APC. As described in Section 5.1.3 on page 136, APC requires that all possible type substitutions be tested for each coupling sequence appearing in the method under test. Thus, the possibility of executing a fault located in the hierarchy being integrated is increased simply because control flow enters each type at least once. However, this is not sufficient to ensure all feasible locations containing faults will be executed.[1]

The most effective of the three coupling-based test adequacy criteria is APDU. This too is of no surprise since its requirements are stronger than ACS and APC. In particular, it requires that all state interactions be tested with respect to the coupling variable for each coupling sequence, and for all types of instances that can be bound to the coupling variable (see Section 5.1.7 on page 139). In terms of the fault/failure model, the requirements imposed by APDU have the greatest chance of causing a fault to be executed, and this accounts for the better performance observed over all the experimental trials.

### 8.4.3 Effectiveness of the Coupling-based Criteria

Log-linear analysis permits one to analyze categorical data in much the same manner as in analysis of variance. The sampling distribution underlying Table 8-4 is a product of independent multinomials. According to Bishop, Fienberg and Holland, the kernel of the appropriate likelihood function is the same as that for a simple multinomial or a simple Poisson [16]. Therefore the estimation procedures for the simpler sampling distributions may be

---

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

used, at least for large samples. The resulting estimates are close to the correct maximum likelihood estimates and the usual goodness of fit statistics are asymptotically chi-square.

**8.4.3.1 Details of the Hypothesis Tests**

We first fitted the experimental results to a model corresponding to a 4-way contingency table with $i$, $k$ marginals fixed. The model consists of the dimensions *Fault* × *Response*, *Fault* × *Program*, *Program* × *Criterion* × *Response*, and all lower level nested factors. The factor *Response* consists of two levels, each corresponding to success or failure of a particular test case. Denote these four factors by $u_1$ (Program), $u_2$ (Fault Type), $u_3$ (Criterion), and $u_4$ (Response). Denote cell counts by $m_{i,j,k,l}$, where $i$, $j$, $k$, and $l$ correspond to the four factors. The best fitting model was found to be:

$$Log(m_{i,j,k,l}) = u_0 + u_1 + u_2 + u_{1,3} + u_{1,4} + u_{2,4} + u_{1,2} + u_{3,4} + u_{1,3,4} + \ldots$$

The terms with one subscript represent main effects; the terms with two subscripts represent two-factor interactions; and the terms with three subscripts represent three-factor interactions. In Figure 8-14, we can see that the fitted cell counts closely match the observed cell counts.

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

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

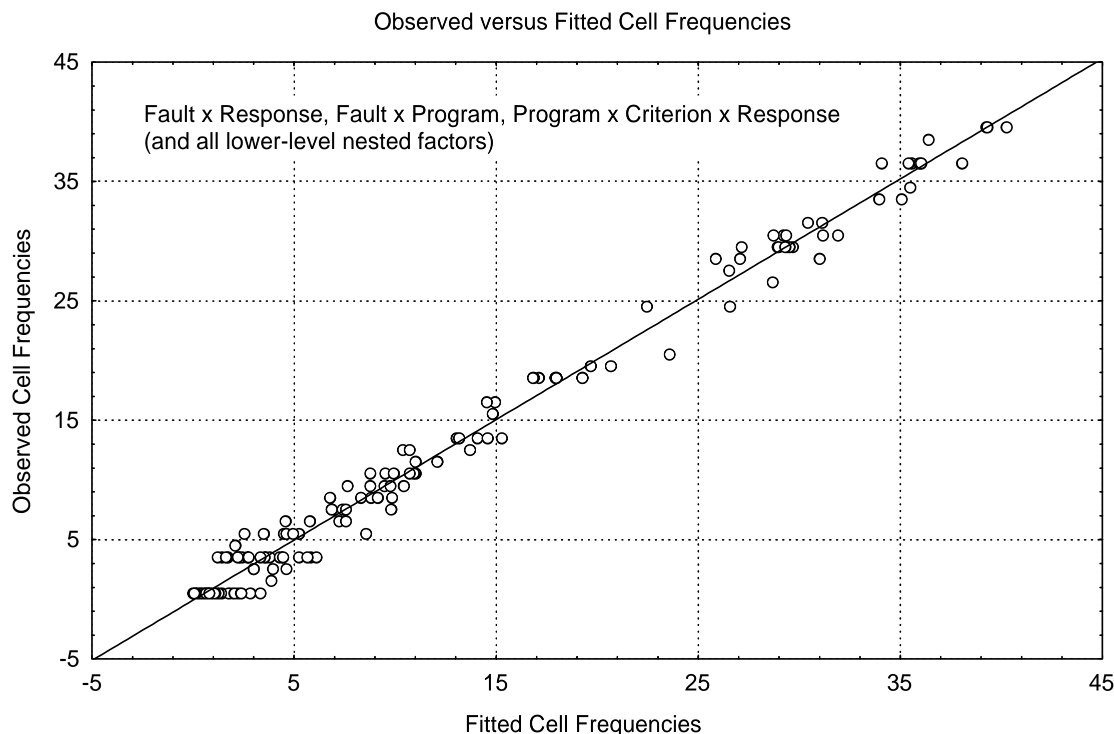



**Figure 8-14.** Observed versus fitted cell frequencies

For the remaining hypothesis tests, we selected out only the data for a particular pair of criteria (indicated by the column labeled *Hypothesis* in Table 8-5) and then tested for an interaction between these two by fitting the model described with and without the corresponding fault-type/criterion term. Table 8-5 on page 195 summarizes the results of these tests. The column labeled *Hypothesis* states the null ($H_0$) and alternative hypothesis ($H_1$) for each test. The columns labeled $\chi^2$ and $\Delta\chi^2$ give the change in value of the chi-square goodness-of-fit statistic, and the columns labeled *df* and $\Delta df$ give the corresponding change in degrees of freedom. Finally, the last column gives the result of each test, indicating whether the null hypothesis is rejected or not.

As the table shows, for hypotheses one through six, there was a net change in the degrees of freedom and $\chi^2$ goodness of fit value. In all cases, there is statistical significance at a *p*-

value less than 0.001. Therefore, we reject the null hypothesis ($H_0$) in favor of the alternative ($H_1$) for all six of these hypotheses. The first three hypotheses allow us to conclude that each of the three coupling-based criteria are more effective than Branch Coverage at detecting the types of faults seeded into the subject programs. The remaining three hypotheses allow us to compare the effectiveness among the coupling-based criterion. Since the null hypothesis ($H_0$) was rejected for each, we can conclude that there is statistical evidence to suggest that APDU is more effective than APC and ACS at detecting the subject fault types, and also that APC is more effective than ACS.

**Table 8-5.** Results of hypothesis tests

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

### 8.4.4 Discussion

The three hypotheses in Table 8-5 that tested the effectiveness of each coupling-based criteria against Branch Coverage indicate that the coupling criteria are better at detecting the object-oriented faults used in the experiment. A remaining question is which of the three coupling criteria is the most effective. Hypotheses one, two, and three have established that each of the coupling criteria are better than Branch Coverage. Observation of the plot in Figure 8-3 suggests that APDU is, on average, more effective that APC and ACS. Similarly, APC is, also on average, more effective that ACS. This observation is, in fact, supported by the last three hypothesis tests.

Given the above conclusion, a key question that remains is *which criterion or combination of criteria should be used*? The plot in Figure 8-3 also suggests that there is no coupling-based criterion that is particularly better for detecting one fault type versus another (i.e. the criterion do not specialize in the faults that they detect). If any criterion is good for a particular fault type, they all are. Therefore we could pick best of the coupling criteria and use that for all fault types.

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

## 8.5 Conclusion

The experiments described in this chapter show that coupling-based testing techniques can be (and have been) extended to detect the faults that result from the polymorphic relationships among components in an object-oriented program. Further, the results show that these techniques are an effective testing strategy for object-oriented programs that use inheritance and polymorphism. This is an important result for developers, testers, and consumers of software developed using object-oriented languages. Developers now have an approach, techniques, and guidelines for addressing certain aspects of integrating object-oriented components. Professional testers also have a repeatable and verifiable means of testing the work products produced by developers and a means of targeting specific types of faults peculiar to object-oriented software. Consumers of software-based and software-embedded products will also benefit by receiving products that are of higher quality.

# 9. Contributions and Future Work

In this dissertation, a new approach to integration testing of object-oriented programs has been presented. This approach takes into account those state interactions that result from the use of inheritance and polymorphism and their effect on a method under test. The approach is based heavily on the static and dynamic analysis of object-oriented programs, and solves a key problem in the area of testing object-oriented programs: how to effectively test programs that make use of inheritance and polymorphism.

The research presented in this thesis has several aspects that have not been explored in this dissertation. Some of these are related directly to coupling-based testing approach while others are specific to related areas of object-oriented software development. The following section discusses the contributions of this research in detail, and the final section of this chapter discusses future research related to the other aspects.

## 9.1 Contributions

A key contribution is a technique for analyzing and testing polymorphic relationships. The foundation of this technique is the **coupling sequence**, which is a new abstraction for representing state space interactions between pairs of method invocations. The coupling sequence provides the analytical focal point for methods under test, and is the foundation for the algorithms for identifying and representing polymorphic relationships for both static and dynamic analysis. With this abstraction and the algorithms, both testers and developers of object-oriented programs now have a means to analyze and better understand the interactions within their software. Though the coupling sequence has been cast for testing problems involving inheritance and polymorphism, is generally applicable to any program that makes uses of encapsulated data types (e.g. Modula-2, Ada83, etc.).

This thesis also contributes a set of **test-adequacy criteria** that are based on coupling sequences and that take inheritance and polymorphism into account. These criteria provide the tester and developer with a way of judging when a testing goal has been achieved. The criteria naturally vary in their effectiveness, but this variation also correlates with the required level of testing effort and is reflected by the subsumptive relationship among the criteria. In ideal circumstances, the level of required to achieved perfect or near-perfect software would be expended. In this case, only a single criterion would be necessary. However, the nature of the world dictates that limited amounts of effort can be expended. The variation of the criteria allow the tester and developer to develop test requirements that reflect this world view. Critical areas are less tolerant to failures, and thus more effective testing is required than in less critical areas. The coupling-based testing criteria for object-oriented programs presented in this thesis, combined with the original criteria of Jin and Offutt [38], allow this objective to be achieved.

Another contribution is a technique for **identifying data flow anomalies** within class hierarchies. As this thesis has shown, inheritance relationships within object-oriented programs yield greater complexity due to the implicit coupling throughout class hierarchies. This thesis has identified and defined specific patterns with respect to state space interactions that indicate anomalous and potentially faulty behaviors.

This thesis has also produced a **model of faults** associated with the use of inheritance and polymorphism. This model is based on the distinction between overriding methods that *extend* the behavior of a new class with respect to its parents, and overriding methods that make *refinements* to inherited behavior. The model takes into consideration both the semantics of inheritance and polymorphism, and also the syntactic patterns of inheritance that lead to anomalies and faults. This model benefits testers and developers by providing a specific set of fault types based on syntactic patterns that can be used for code inspections, and to guide the production of test cases. The model also benefits researchers by providing a foundation for understanding and reasoning how failures are manifested in object-oriented programs.

This thesis has also resulted in a **proof of concept tool** that demonstrates the practicality and effectiveness of the coupling-based analysis techniques.

Finally, this thesis has contributed a graphical model for analyzing and understanding the effects of polymorphism within a class hierarchy. The enhanced **yo-yo graph** clearly shows the control flow across class boundaries between descendants and ancestors that results from a method invocation. It is often difficult to see and understand the path taken by the flow of control because the apparent path is not always the path that is actually executed. Unfortunately, this is the rule rather than the exception, and is due to the non-determinism induced by polymorphism. The yo-yo graph provides a way to gain insight into such complexities.

## 9.2 Future Work

There are a number of additional problems that related to the research reported in this thesis that warrant further investigation. The following sub-sections discuss 11 of these in detail.

### 9.2.1 Testing inter-method coupling sequences

This thesis has focused on testing polymorphic relationships that manifest themselves through state space interactions resulting from pairs of method invocations within the same method. However, as described in Section 4.2.5 and illustrated by Figure 4-5 on page 100, there are other interactions that can occur between methods that are not invoked from the same methods. These interactions form *inter-method coupling sequences* and represent interactions that occur indirectly as the result of two or more separate method invocations. To accommodate this, the definition of the types of coupling sequences (Section 4.2) will have to be expanded along with the definitions for the coupling method, antecedent node and method, and consequent node and method. The algorithms for identifying coupling sequences and coupling sets must also be redefined (Algorithm 6-1 on page 143 and Algorithm 6-2 on page 144, respectively). The expected benefit of this research will be the detection of more faults, but at the cost of a more expensive analysis.

### 9.2.2 Specification and coupling-based testing of object-oriented programs

One of the assumptions underlying this research is that no formal or systematic specification is available for the classes and methods under test. The validity of this assumption rests upon common practice in industrial (and academic) settings. The result of this assumption is that a number of anomalies are identified that can only be characterized as potential faults (e.g. data flow anomalies due to inconsistent state space definitions and uses across methods). It is likely that if some type of formal or systematic specification were available (e.g. state transition diagram and method sequence diagram), static analysis using coupling-based techniques could precisely identify certain anomalies as faults or eliminate them from further consideration.

### 9.2.3 Integration testing within class hierarchies

Another key area not emphasized by this thesis is the integration testing of classes within a class hierarchy. Each class $C$ has the potential to define state and behavior. This serves the constrain the behavior of its set of descendants $D$, particularly if elements of $D$ can possibly be used in contexts where an instance of $C$ is expected. In this case, the behavior of each descendant $d \in D$ must be consistent with the externally observable behavior defined by $C$. Each $d$ may provide additional behaviors (i.e. extensions), but it must behave like $C$. It's not clear what testing strategies should be employed to determine if this in fact the case, but two criteria come to mind that can possibly bound the set of testing requirements: *All-Subtypes* and *All-Immediate-Subtypes*. The *All-Subtypes* criterion would require that for a given class $C$ that can be used in its own context (i.e. used as the declared type of a variable or the type of a type coercion), every descendant of $C$ be tested as if it were an instance of $C$. That is, every descendant would be subjected to the same set of tests required for $C$, with the expected outcome being state and behavior identical to $C$ (thus $C$ can be used as a test oracle for its descendants). From the perspective of practically, *All-Subtypes* has the unfortunate effect of requiring that for every $C$ in a hierarchy, each of its descendants is tested with $C$'s tests.

*All-Immediate-Subtypes* relaxes *All-Subtypes* by requiring that only the direct descendants of *C* be tested with *C*'s tests. This has the effect of reducing the testing effort while attempting to ensure that the each descendant is substitutable for all its ancestors, both direct and indirect (i.e. grandparents, great-grandparents, and so on). There is some basis for this since inheritance is a transitive relation. If *B* is a descendant of *A*, and *C* is a descendant of *B*, then by transitivity, *C* is also a descendant of *A*. Thus, if *B* is first tested using *A*'s test cases (and passes), and *C* is then tested using *B*'s (which includes *A*'s) and also passes, then *C* should be behaviorally compatible with *A* and an instance of *C* can safely be used where an instance of *A* is expected (conjecture). If this conjecture holds, then *All-Immediate-Subtypes* has the advantage of achieving the same testing objective as *All-Subtypes*, but with reduced effort because the number of test requirements is reduced. Unfortunately, this is not likely to hold in all cases given Weyuker's antidecomposition axiom [69] as described by Perry and Kaiser [64].

### 9.2.4 Coupling-based testing of concurrent object-oriented programs

Many object-oriented languages, such as Java, Eiffel, and Ada 95, incorporate some type of threading mechanism. This results in greater complexity of software, and likely exacerbates the number (and perhaps types) of faults that can occur in an object-oriented program. An interesting area of investigation that remains open is whether or not coupling-based testing techniques would be effective in the presence of multiple threads.

### 9.2.5 Testing of reflective object-oriented programs

Some object-oriented languages, such as Java and C#, provide runtime features that permit the utilization and manipulation of the underlying metadata within their runtime environments (i.e. class, variable, and method names, field type information, and method return types). For example, in Java it is possible to obtain a reference to an object whose type is unknown at some location within the program. The program can then query the underlying metadata to determine the object's type, including information such as the methods it provides, the interfaces implemented, and the variables contained in its state space. The latter, with few restrictions, can be manipulated even if its variables are not declared to be publicly

available. This type of dynamic semantic manipulation offers a number of testing issues and challenges. For example, at compile time, we do not know what the actual type of an object will be that is loaded through reflection, and thus we do not know what the test requirements should be. At least the problem is bounded with inheritance and polymorphism to the finite set of types that appear in a class hierarchy. With reflection, there are no restrictions on the type.

### 9.2.6 Generation of test cases for coupling-based testing

A key area of research related to this thesis is the generation of test cases that satisfy a particular coupling-based criterion. During the research reported in this thesis, test case generation was accomplished primarily through manual analysis of the subject programs used in the validation. While this is acceptable for a scientific investigation, it is of limited applicability in practical settings. Thus, there is a need to enhance the test case generation process through automation, to the extent possible.

### 9.2.7 Metrics for coupling-based testing

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

### 9.2.8 Mutation testing of object-oriented programs

One of the key questions that arises in testing is: *how effective are the test cases at detecting faults?* This question applies equally to the testing approach described in this thesis and to

all other approaches to testing software. A testing technique that has been used to answer this question for procedural programs is *mutation testing* [23]. Mutation testing is a fault-based approach, and represents the actual faults that occur in programs using a fault model. Testing techniques usually make assumptions about the types of faults that occur in programs, and are used to (hopefully) select test cases that detect faults of those types. In mutation testing, static changes called *mutations* are made to programs. Mutations are accomplished by the application of set of *mutation operators*. Examples of these operators include substituting one operator for another (e.g. subtraction for addition) and using a different constant or variable in an expression.

The changed programs, referred to as *mutants*, correspond to the original unchanged program but with the addition of a single fault. This process is repeated to produce a population of mutants that corresponds to a particular set of fault types. The mutants are then used to assess the adequacy of a set of test cases at detecting the faults. If the behavior of a mutant is different from the behavior of the original program when executed with test $t$, then $t$ is said to *kill* the mutant and thus is capable of detecting the type of fault associated with the mutant. In this manner, mutation testing is used to assess the effectiveness of a particular test set.

It seems plausible that mutation testing techniques can also be applied to object-oriented programs. However, there are some challenges that must be overcome. An underlying assumption in the process of mutating a program to model a particular fault is that the semantics of the mutation operators and the underlying types of the operands are well defined and understood. For example, to change the expression $x = y + z$ to $x = y - z$, where $x$, $y$, and $z$ are integers, it is necessary to understand that both addition and subtraction are binary operators, and also that addition and subtraction are valid for instances of integer. If the type of $x$, $y$, and $z$ is *string*, and where addition means concatenation, then subtraction is meaningless with respect to the type *string*, and thus the resulting mutated expression $x = y - z$ is semantically invalid. It follows from this argument that to extend mutation testing

to object-oriented programs also requires knowledge of the semantics of the mutation operators and of the types. Herein lies the challenge.

In object-oriented programs, types and operators are not static, but are an intentional side-effect of the design process, and hence are not fixed across problem and solution domains. Thus the key insight to applying mutation testing techniques requires that the variation of types and operators be taken into account. The consequence of this insight is that the mutation process cannot be treated solely as an instance of syntactic manipulation, as it can where the set of types is fixed and well-defined. A key challenge is coming up with a mutation process that can be generalized across object-oriented languages and problem/solution domains. A likely solution is the use of a mutation engine that can be customized to account for the variation of semantics among types and operators. For example, a special inheritance hierarchy *H* could be used to mirror a subset of the problem/solution domain hierarchy *D*. For each type *T* in *D* that is subject to mutation, *H* would have a corresponding type that "knows" how to mutate instances of *T* in a manner consistent with *T*'s semantics. Out of necessity, this would require the implementer of *D* to also implement *H* (or someone with knowledge of *D*). Clearly this has the potential to be an onerous task. However, it may be the case that certain subsets of *D* can be treated uniformly, thus allowing the use of inheritance and polymorphism to reduce the effort required to implement *H*.

### 9.2.9 CBAT Enhancements

There are a number of enhancements and modifications that need to be made to CBAT. In no particular order, these include:

- **Graphical user interface**. At present, CBAT is driven by a command line interface along with a set of properties maintained in a separate file. Using CBAT for a particular analysis problem requires that a number of steps be carried out manually. This is both time consuming and tedious. A graphical user interface (or an HTML forms-based interlace) would vastly improve the usability of CBAT.

- **Adding a database backend**. CBAT collects an enormous amount of information during an analysis, all of which is held in memory and discarded when the analysis is complete. Subsequent analyses requires that this information be re-computed, and much of it is common to prior analyses performed. Further, the fact that all of this information is held in memory necessarily places an upper bound on the size of the problem that CBAT can handle. Storing and staging this information in a database would increase the net performance of CBAT and increase the size of the problem that can be handled.
- **Support for additional languages**. CBAT was developed for analyzing programs written in the Java programming language. However, its design was intentionally generalized so that programs expressed in other object-oriented languages (e.g. C++, C#, Eiffel, Ada 95) could also be analyzed. To support a new language requires that a new language-specific front-end (i.e. parser) be provided. Also, a new language-specific code instrumentation engine on the back-end would also need to be provided.

### 9.2.10 XML-based program representations for testing and analysis

One of the problems that must be faced by a developer of any program analysis tool is what representation to use for analysis and storage of information. This includes information that results from the analysis itself (e.g. the set of coupling sequences for a method), and also the information that describes the program itself (e.g. abstract syntax tree and inheritance hierarchy). Historically, the representations used to store this information are custom and specific to the tool at hand (as in the case of CBAT). While this is good in that it helps to ensure that the representation is optimized for the specific requirements of the analysis, it

usually limits the ability for additional tools and analyses to the specific representation. A better solution would be a generalized canonical form for describing program information. This would eliminate the necessity of having to start from scratch when developing an analysis tool, and also allow for easy exchange of analysis information among tools. Further, standardized tools could be developed for performing various analyzing tasks, such as the construction of program dependence graphs and slicing. Such a form called JavaML has been developed for Java programs using XML as the basis [6]. The question is how effective this representation is for analyzing programs and what are its limitations. A useful exercise to answer this question would be to extend CBAT to produce an instance of JavaML for program under analysis, and to use this for storage and analysis.

### 9.2.11 Reverse engineering of software contracts

If anything, object-oriented (and object-based) programing is about specifying and implementing types, and then acting on instances of those types to affect computation and achieve behavior. A useful technique for specifying types (classes) is *Design by Contract* wherein preconditions, postconditions, and invariants are specified for the type and each of its operations (method) [51]. These serve not only as specifications to constrain the implementation, but may also be used as passive testing mechanisms [52]. With suitable language support, preconditions and invariants can automatically be checked on entry to a method, and postconditions and invariants checked just prior to exiting [40, 52]. A precondition violation indicates a fault in the client of a type, and postcondition or invariant violation indicates a fault in the implementation of the type. Unfortunately, even with language support, such specifications are often not produced or become lost over time.

There is a need for a mechanism or tool that will reverse engineer and recover contracts from existing type implementations. Clearly the resulting contracts would be based solely on the implementation (at least initially), and an underlying implicit assumption would be that the implementation is correct. This, of course, is a dubious assumption, However, it is not without merit since the implementation does exists and the corresponding original specification does not. Furthermore, any maintenance changes to the code have to be made

locally in the context of that code without the benefit of a specification. Thus, without other reasons to the contrary, assuming the correctness of the code *as a starting point* is reasonable.

Once the contracts have been recovered and the type's implementation instrumented appropriately, testing can begin to determine if clients that use the type do so correctly. At this stage, violations of preconditions *may* indicate a fault in the client, but could equally indicate a fault in the implementation. This at least gives the basis for more focused probing to determine where the fault lies. Postconditions are more difficult since they are based directly on the code analyzed, and thus never should be violated. Determining their correctness, and hence the correctness of the implementation, would have to be determined by how instances of the type are used after the operation is complete. Anomalous behavior in a client after the method has returned might indicate a problem with the implementation of the method.

On the surface, contract recovery seems to be of limited value given the uncertainty that initially arises with respect to the postcondition and the correctness of the implementation. However, over time this concern should diminish, particularly as further maintenance activities are applied to the implementation. While a postcondition violation shortly after contract recovery raises suspicions about the contract (and the implementation) the likelihood that violations occurring during the course of maintenance result from a malformed postcondition decrease, while the likelihood of a problem in the maintenance increases. Overtime, one would expect that the recovered contracts would become the actual contracts, even if they are different from the original contracts.

# References

1. *Testability of Object-Oriented Systems*. 1994: Reliable Software Technologies.

2. *Unified Modeling Language, version 1.1*. 1997, Object Management Group.

3. Alexander, R. T., J. M. Bieman, and J. Viega, *Coping with Java Programming Stress.* Computer, 2000. **33**(4), April: p. 30-38.

4. Alexander, R. T. and A. J. Offutt. *Analysis Techniques for Testing Polymorphic Relationships*. In *Proceedings of Thirtieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS30 '99)*. 1999, August. Santa Barbara CA: IEEE Computer Society.

5. Alexander, R. T. and A. J. Offutt. *Criteria for Testing Polymorphic Relationships*. In *Proceedings of International Symposium on Software Reliability and Engineering (ISSRE00)*. 2000, October. San Jose CA: IEEE Computer Society.

6. Badros, G. J., *JavaML: A Markup Language for Java Source Code*. 2000, Dept. of Computer Science and Engineering. University of Washington. Seattle, WA USA.

7. Balcer, M. J., W. M. Hasling, and T. J. Ostrand. *Automatic generation of test scripts from formal test specifications*. In *Proceedings of ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification (TAV3)*. 1989. Key West, FL, USA.

8. Barbey, S. and A. Strohmeier. *The Problematics of Testing Object-Oriented Software*. In *Proceedings of SQM'94 Second Conference on Software Quality Management*. 1994. Edinburgh, Scotland, UK.

9. Beizer, B., *Software Testing Techniques*. 1990, New York, New York: Van Nostrand Reinhold.

10. Berard, E. *Issues in the Testing of Object-oriented Software*. in *Proceedings of Electro'94 International*. 1994: IEEE Computer Society Press.

11. Berard, E. V., *Essays on Object-Oriented Software Engineering*. Vol. 1. 1993: Prentice Hall.

12. Binder, R. V., *Testing Objects: Myth and Reality*. Object Magazine, 1995. **5**(2): p. 73-75.

13. Binder, R. V., *Trends in Testing Object-oriented Software.* Computer, 1995. **28**(10): p. 68-69.

14. Binder, R. V., *The FREE Approach for System Testing: Use-cases, Threads, and Relations.* Object Magazine, February, 1996. **6**(2).

15. Binder, R. V., *Testing Object-Oriented Software: A Survey.* Journal of Software Testing, Verification & Reliability, 1996. **6**(3/4), September / December: p. 125-252.

16. Bishop, Y. M. M., S. E. Fienberg, and P. W. Holland, *Discrete Multivariate Analysis: Theory and Practice*. 1975, Cambridge, Massachusetts: MIT Press.

17. Capper, N. P., R. J. Colgate, J. C. Hunter, and M. F. James, *The Impact of Object-oriented Technology on Software Quality: Three Case Histories.* IBM Systems Journal, 1994. **33**(1): p. 131-157.

18. Cheatham, T. J. and L. Mellinger. *Testing Object-oriented Software Systems*. In *Proceedings of ACM 18th Annual Computer Science Conference*. 1990, February: ACM Press.

19. Chen, H. Y., T. H. Tse, F. T. Chan, and T. Y. Chen, *In Black and White: An Integrated Approacht to Class-Level Testing of Object-Oriented Programming.* ACM Transactions on Software Engineering and Methodology, 1998. **7**(3): p. 250-295.

20. Chen, M.-H. and M.-H. H. Kao. *Testing Object-Oriented Programs - An Integrated Approach*. In *Proceedings of Tenth International Symposium on Software Reliability Engineering*. 1999, 1 - 4 November. Boca Raton, Florida.

21. Chow, T. S., *Testing software design modeled by finite-state machines.* IEEE Transactions on Software Engineering, 1978. **SE-4**(3): p. 178-87.

22. DeMillo, R. A., D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. *An Extended Overview of the Mothra Software Testing Environment*. in *Proceedings of Second Workshop on Software Testing, Analysis, and Verification*. 1988, July. Banff Alberta: IEEE Computer Society Press.

23. DeMillo, R. A. and A. J. Offutt, *Constraint-Based Automatic Test Data Generation.* IEEE Transactions on Software Engineering, 1991. **17**(9), September: p. 900-910.

24. Doong, R.-K. and P. G. Frankl, *The ASTOOT Approach to Testing Object-Oriented Programs.* ACM Transactions on Software Engineering and Methodology, 1994. **3**(4): p. 101-130.

25. Dorman, M. *Unit Testing of C++ Objects*. in *Proceedings of EuroSTAR 93*. 1993, October. Jacksonville, Flordia: SQE, Inc.

26. Fiedler, S. P., *Object-Oriented Unit Testing.* Hewlett-Packard Journal, 1989. **40**(2): p. 69-75.

27. Firesmith, D. G., *Testing Object-Oriented Software*. 1992, Advanced Technology Specialists.

28. Firesmith, D. G. *Testing Object-Oriented Software*. In *Proceedings of Eleventh International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA, '93)*. 1993: Prentice-Hall, Englewood Cliffs, New Jersey.

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

30. Frankl, P. G. and E. J. Weyuker, *An applicable family of data flow testing criteria.* IEEE Transactions on Software Engineering, 1988. **14**(10): p. 1483-98.

31. Freedman, R. S., *Testability of software components.* IEEE Transactions on Software Engineering, 1991. **17**(6): p. 553-64.

32. Harrold, M. J., J. McGregor, and K. Fitzpatrick. *Incremental Testing of Object-Oriented Class Structures*. In *Proceedings of 14th International Conference on Software Engineering*. 1992: IEEE Computer Society.

33. Harrold, M. J. and G. Rothermel. *Performing Data Flow Testing on Classes*. In *Proceedings of Second ACM SIGSOFT Symposium on Foundations of Software Engineering*. 1994: ACM Press, New York, New York.

34. Harrold, M. J. and M. L. Soffa, *Selecting and using data for integration testing.* IEEE Software, 1991. **8**(2): p. 58-65.

35. Hayes, J. H. *Testing of Object-Oriented Programming Systems (OOPS): A Fault-Based Approach*. In *Proceedings of Object-Oriented Methodologies and Systems*. 1994: Springer-Verlag.

36. Hong, H. S., Y. R. Kwon, and S. D. Cha. *Testing of Object-oriented Programs Based on Finite State Machines*. In *Proceedings of 1995 Asia Pacific Software Engineering Conference*. 1995: IEEE Computer Society Press, Los Alamitos, California.

37. Hong, H. S., Y. R. Kwon, and S. D. Cha, *A State-Based Testing Method for Classes.* Journal of Korea Information Science Society(B, 1996. **23**(11): p. 1145-1154.

38. Jin, Z. and A. J. Offutt, *Coupling-based Criteria for Integration Testing.* The Journal of Software Testing, Verification, and Reliability, 1998. **8**(3), September: p. 133-154.

39. Jorgenson, P. C. and C. Erickson, *Object-Oriented Integration Testing.* Communications of the ACM, 1994. **37**(9): p. 30-38.

40. Kramer, R. *iContract - The Java(tm) Design by Contract(tm) Tool*. In *Proceedings of Technology of Object-Oriented Languages and Systems*. 1998, August 3-7. Santa Barbara, California: IEEE Computer Society.

41. Kung, D., J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. *A Test Strategy for Object-Oriented Systems*. In *Proceedings of Nineteenth Annual International Computer Software and Applications Conference*. 1995: IEEE Computer Society Press, Los Alamitos, Calif.

42. Kung, D., N. Suchak, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. *On Object State Testing*. In *Proceedings of Eighteenth Annual International Computer Software \& Applications Conference*. 1993: IEEE Computer Society Press, Los Alamitos, Calif.

43. Leavens, G. T., *Modular Specification and Verification of Object-oriented Programs.* IEEE Software, 1991. **8**(4): p. 72-80.

44. Leavens, G. T. and W. E. Weihl, *Specification and Verification of Object-oriented Programs Using Supertype Abstraction.* Acta Informatica, 1995. **32**(8): p. 705-778.

45. Liskov, B. and J. M. Wing. *Specifications and their use in defining sub-types*. in *Proceedings of OOPSLA'93*. 1993. New York: ACM Press.

46. McGregor, J. D. *Constructing Functional Test Cases Using Incrementally Derived State Machines*. In *Proceedings of 11th International Conference on Testing Computer Software*. 1994: USPDI, Washington, DC.

47. McGregor, J. D. *Functional Testing of Classes*. in *Proceedings of 7th International Software Quality Week*. 1994: Software Research Institute, San Francisco.

48. McGregor, J. D. and D. M. Dyer, *A Note on Inheritance and State Machines.* Software Engineering Notes, 1993. **18**(4): p. 61-69.

49. McGregor, J. D. and D. M. Dyer. *Selecting Functional Test Cases for a Class*. In *Proceedings of 11th Annual Pacific Northwest Software Quality Conference*. 1993: PNSQC, Portland, Oregon.

50. Meyer, B., *Introduction to the Theory of Programming Languages*. 1990: Prentice Hall.

51. Meyer, B., *Design By Contract*, in *Advances in Object-Oriented Software Engineering*, D. Mandrioli and B. Meyer, Editor. 1991, Prentice Hall: Englewood Cliffs, N.J. p. 1-50.

52. Meyer, B., *Object-Oriented Software Construction*. 1997, Englewood Cliffs, New Jersey: Prentice Hall.

53. Meyer, S., *Effective C++*. 1992, Reading, Massachusetts: Addison-Wesley.

54. Morell, L. J. *Theoretical Insights into Fault-Based Testing*. In *Proceedings of ACM SIGSOFT '89 2nd Symposium on Software Testing Analysis and Verification (TAV2)*. 1988. Banff Alberta.

55. Morell, L. J., *A Theory of Fault-Based Testing.* IEEE Transactions on Software Engineering and Methodology, 1990. **16**(8), August: p. 844-857.

56. Offutt, A. J. *Software testing: State-of-the-art Vs. State-of-the-practice. Position Paper: Software Testing: From Theory to Practice*. In *Proceedings of 1997 Annual Conference on Computer Assurance (COMPASS 97)*. 1997, June. Gaithersburg MD: IEEE Computer Society Press.

57. Offutt, A. J. and A. Irvine. *Testing Object-Oriented Software Using the Category-Partition Method*. in *Proceedings of TOOLS USA'95*. 1995. Santa Barbara, California: Prentice Hall.

58. Orso, A., *Integration Testing of Object-Oriented Software*, in *Dipartimento di Elettronica e Informazione*. 1999, Politecnico Di Milano: Milan, Italy.

59. Ostrand, T. J. and M. J. Balcer, *The category-partition method for specifying and generating functional tests.* Communications of the ACM, 1988. **31**(6): p. 676-86.

60. Overbeck, J., *Integration Testing for Object-Oriented Software*. 1994, Vienna University of Technology.

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

62. Parnas, D. L., J. E. Shore, and D. Weiss. *Abstract types defined as classes of variables*. In *Proceedings of Conference on Data: Abstraction, Definition and Structure*. 1976. Salt Lake City, UT, USA.

63. Payne, J. E., R. T. Alexander, and C. D. Hutchinson, *Design-for-Testability for Object-Oriented Software.* Object Magazine, 1997. **7**(5), July: p. 34-43.

64. Perry, D. E. and G. E. Kaiser, *Adequate Testing and Object-Oriented Programming.* Journal of Object-Oriented Programming, 1990. **2**(5): p. 13-19.

65. Rapps, S. and E. J. Weyuker, *Selecting software test data using data flow information.* IEEE Transactions on Software Engineering, 1985. **SE-11**(4): p. 367-75.

66. Sanden, B., *Software Systems Construction with Applications in Ada*. 1993, Englewood Cliffs, New Jersey: Prentice Hall.

67. Sinha, S. and M. J. Harrold, *Analysis and Testing of Programs with Exception-Handling Constructs.* IEEE Transactions on Software Engineering, 2000. **26**(9), September: p. 849-871.

68. Smith, M. D. and D. J. Robson. *Object-oriented Programming: The Problems of Validation*. In *Proceedings of 6th International Conference on Software Maintenance*. 1990: IEEE Computer Society Press, Los Alamitos, Calif.

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

# CURRICULUM VITAE

Roger T. Alexander was born on December 9, 1958, in Louisville, Kentucky, and is an American Citizen. He graduated from Louisville Male High School, Louisville, Kentucky, in 1977. He received his Bachelor of Science from the University of the State of New York in 1991. He received his Master of Science from George Mason University in 1994.

Mr. Alexander has worked in the software industry for 23 years, primarily as a developer of software development tools. He is currently employed as a Senior Research Scientist in the Center for Secure Information Systems at George Mason University. Mr. Alexander is a Senior Member of the Institute of Electrical and Electronic Engineers, a Member of the IEEE Computer Society, and a Professional Member of the Association for Computing Machinery.