To Appear in Proc. Int. Conf. Software Reliability Engineering (ISSRE 2002) pp. 207--218, 2002.

Fault Detection Capabilities of Coupling-based OO Testing[1]

Roger T. Alexander
Colorado State University
Department of Computer Science
Fort Collins, Colorado 80523
rta@cs.colostate.edu

Jeff Offutt
George Mason University
ISE Department
Fairfax, Virginia 22030
ofut@gmu.edu

James M. Bieman
Colorado State University
Department of Computer Science
Fort Collins, Colorado 80523
bieman@cs.colostate.edu

Abstract

Object-oriented programs cause a shift in focus from software units to the way software classes and 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, which can be covered using testing criteria that take the effects of inheritance and polymorphism into account. This paper demonstrates, via a set of experiments, the relative effectiveness of several coupling-based OO testing criteria and branch coverage. OO criteria are all more effective at detecting faults due to the use of inheritance and polymorphism than branch coverage.

1. Introduction

The emphasis in object-oriented languages is on defining abstractions (e.g. abstract data types) that model aspects of a problem [11]. These abstractions are implemented as user-defined types that have both state and behavior. Although abstract data types can help achieve a higher quality design, their use may affect how software is tested. A major factor is that shifting from procedure-oriented to object-oriented software often changes where the complexity resides. Instead of procedures that have complicated control structures, object-oriented software often has simple procedures, with the complexity being in how the procedures and components are connected. Thus, testers are finding that less emphasis is needed on unit testing and more on integration testing.

The inherent complexity of the relationships found in object-oriented languages [6] also affects testing. The compositional relationships of inheritance and aggregation, combined with the power of polymorphism, can make it harder to detect faults in the way components are integrated. This is because component integration is different in object-oriented languages [5].

The primary distinction among the types of languages discussed in this paper is in the mechanisms used for abstraction. Procedure-oriented languages use procedures and functions as their primary abstraction mechanism, whereas object-oriented languages use data abstraction. In addition, object-oriented languages use the integration mechanisms of inheritance and polymorphism (dynamic binding), both of which can strongly affect component integration. Inheritance differs from aggregation in that a new type can have access to the internal representation of the ancestor types. When a call is made to a polymorphic method, which version is executed depends on the type of the object [11]. Thus inheritance and polymorphism provide two forms of integration that must be dealt with when testing object-oriented software, neither of which has a procedure-oriented counterpart.

This paper presents results from an ongoing research project that has the goal of improving the quality of object-oriented software. One previous paper [13] derives fault models for OO software based on the inheritance and polymorphism relationships in object-oriented software. Another prior paper [3] defines new coverage criteria, which allow routine aspects of testing at the integration level to be formalized. This paper evaluates the effectiveness of the criteria at detecting faults that result from the use of inheritance and polymorphism.

2. Background

The test adequacy criteria evaluated in this paper are based on our previous work in the area of coupling-based testing [2, 3, 10]. The following presents a brief overview of this work, beginning with concepts on object-orientation and the notion of the *coupling sequence*. The formal definitions of the criteria are then presented.

[1] This work is supported in pary by the US National Science Foundation under grants CCR-98-04111 and CCR-0098282 to George Mason University.

2.1. OO concepts

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

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

Polymorphism permits variable instances to be bound to references of different types according to the structure of the inheritance hierarchy. *Dynamic binding* permits different method implementations to execute. Which one executes depends on the actual type of an instance that is bound to a particular reference; this actual type is independent of its declared type [11].

Behavior in an object-oriented program is manifested through method calls, which can occur in two circumstances: (1) with respect to some instance (object), or (2) where there is no instance. *Instance methods* are called with respect to *instance variables*, and *class methods* have no instance. Instance methods can make the instance explicit, as in *o.m*(), or implicit, as in *p*(). For the call *o.m*(), *m*() executes in the context of the instance that is bound to the reference *o*. For convenience, we say that *m*() executes in the context of *o*, and we refer to *o* as *m*()'s instance context.

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

2.2. Coupling sequences

Although one of the motivating goals of object-oriented design and programming was to reduce the amount of coupling between software components, the new language features also introduce new ways for components to be coupled. To handle these couplings in analysis techniques, the idea of a *coupling sequence* was previously introduced [2]. A coupling sequence is a sequence of method calls between a method under test *m* and an object *o* that *m* references that establishes a data coupling. The objective is not to determine if *o* is correct, but rather to determine if *m* is using *o* correctly. Coupling sequences represent those locations in the text of *m* where faults are likely to occur with respect to *o*.

Intra-method coupling sequences are defined by pairs of method calls made within the context of a particular method, referred to as the *coupling method*. When a method *m* is called through an object *o* (i.e. *o.m*()), we say that *m* executes in the *instance context* provided by *o* (referred to as the *context variable*). The two method calls of an intra-method coupling sequence are made through the same context variable, so they share a common instance context. Further, there is at least one path between the two method calls that is definition-clear with respect to the context variable and to at least one state variable that is defined by the first method and used by the second. Such a path is referred to as a *coupling path*. The intra-method coupling sequence is similar to a *def-use* pair [9], and serves to relate a definition of a state variable to a corresponding use across a procedural boundary in the context of a particular object and method. On the surface, it might appear that testing one such path is sufficient to test the class of object bound to *o*. However, it is not the class that is being tested, but rather the method that makes use of the object and the corresponding methods.

An example of an intra-method coupling sequence is illustrated in Figure 1. Method *f*, the coupling method, contains a single coupling sequence, $s_{j,k}$, that starts at node *j* with a call to *o.m* (the *antecedent method*) and extends through paths that end at node *k* where the sequence ends with a call to *o.n* (the *consequent method*). The nodes containing the *antecedent method* and *consequent method* are called the *antecedent node* and *consequent node* of $s_{j,k}$. Note that there must be at least one path between the call sites that is definition clear with respect to *o* and to the state variable *definitions* made in the antecedent method that have corresponding *uses* in the consequent method.

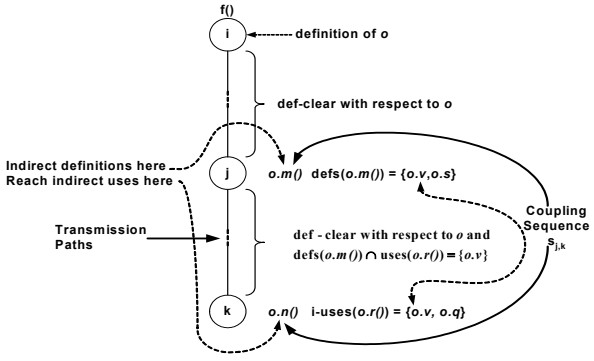

Figure 1. Example coupling sequence $s_{j,k}$

A coupling sequence $s_{j,k}$ is defined with respect to a set of state variables that are defined by the antecedent method and used by the consequent method. This set of variables is referred to as the coupling set $T_{s_{j,k}}$ of $s_{j,k}$, and each member of this set is a coupling variable. The coupling set for the sequence $s_{j,k}$ shown in Figure 1 is:

$$\Theta_{s_{j,k}} = \{v_{type(o)}\}$$

where the subscript *type*(*o*) is the declared type of the context variable *o*, and this $v_{type(o)}$ is a state variable contained in the definition of *o*'s declared type. $T_{s_{j,k}}$ contains the state variables referenced through *o* that are defined by the antecedent method and used by the consequent method in the coupling sequence $s_{j,k}$.

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

2.3. Coupling criteria

This paper evaluates three object-oriented coupling criteria for integration testing: *All-Coupling-Sequences*, *All-Poly-Classes,* and *All-Poly-Coupling-Defs-Uses*. In the following subsections, $T_{s_{j,k}}$ represents the set of test cases for coupling sequence $s_{j,k}$.

2.3.1. All-Coupling-Sequences (ACS). Ideally, during integration testing, at least every coupling sequence in every method of every class should be covered. Here, coverage means that each coupling sequence is executed by at least one test case. The *All-Coupling-Sequences* requires that every coupling sequence be covered by at least one test case.

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

[1] A last-definition of a variable *v* is a node *n* along some path of a method, where *n* is the last node that defines *v* prior to the exit node of the method [10]. Similarly, a first use of *v* is the first node *n* along some path that begins with the entry node a method, such that there is no other node that uses *v* before the use of *n*.

2.3.2. All-Poly-Classes (APC). The *All-Poly-Classes* criterion strengthens *All-Coupling-Sequences* by considering inheritance and polymorphism. This is achieved by ensuring there is at least one test for every class that could provide an instance context for each coupling sequence. The idea is that the coupling sequence should be tested with every possible type substitution that can occur in a given coupling context. The *All-Poly-Classes* criterion requires that for every coupling sequence $s_{j,k}$ in a method *f*, and for every class *c* in the type family defined by the context of $s_{j,k}$, there is at least one test that covers every feasible combination of *c* and $s_{j,k}$ for *f*. The combination (c, $s_{j,k}$) is feasible if and only if *c* is the same as the declared type of the context variable for $s_{j,k}$, or *c* is a child of the declared type and it defines an overriding method for the antecedent or consequent method. Thus, only classes that override the antecedent and consequent methods are considered.

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

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

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

3. Experimental Design

This experiment evaluated the three coupling-based test adequacy criteria. 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" [4].

3.1. 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 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 whose declared type *T* defines a family of types that are descendants of *T*. The column labeled F_{S_f} gives the number of classes in this type family (inheritance hierarchy) for the corresponding program.[1] The column labeled *Description* indicates the source from which each program was obtained. Five programs (P1, P2, P3, P5, and P6) 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, 1 was developed by a graduate student (P4), and 2 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).[2]

Table 1: Subject program characteristics

f	$\|s_f\|$	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)

3.2. 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 actual structure of each test case varied from simple scalar type inputs and references to objects, to a Java program (for ANTLR) and a make file (for JMK).

The strategy used to select test cases is similar to a typical way test cases are selected for Branch Coverage. For each coupling sequence, the path expression [4] 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 3 summarizes the number of test cases for each combination of subject program and test adequacy criterion. For ACS, the number of test cases is determined by the number of coupling sequences and control flow paths present in the method under test. For APC, the number of test cases is also determined by the size of the type family for the coupling variable. Finally, for APDU, the number of test cases is determined by adding the number of control flow paths in the antecedent and consequent methods to the test cases for APC and ACS.

3.3. 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 described in detail in our previous work [13], and summarized in Table 2. The number of faults was determined by the syntactic characteristics of a particular subject program and the syntactic properties necessary for the manifestation of a failure for a given fault type. The details of the latter are described in Alexander's

Table 2: Faults due to inheritance and polymorphism

Acronym	**Fault**
SDA	State Definition Anomaly (possible postcondition violation)
SDIH	State Definition Inconsistency (due to variable hiding)
SDI	State Defined Incorrectly (possible post-condition violation)
IISD	Indirect Inconsistent State Definition
IC	Incomplete Construction

[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.

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

Table 3: Number of test cases

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

Table 4: Number of injected faults

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

dissertation [1]. Table 4 summarizes both the number and type of faults that were injected.

4. Conduct of Experiments

The testing and evaluation procedure consists of four steps: (1) *test oracle derivation*, (2) *fault injection*, (3) *test execution*, and (4) *result evaluation*. The first step creates a test oracle that can be used to evaluate the results of subsequent tests. For the second step 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.

4.1. Test oracle derivation

To derive the test oracle for a particular combination of test adequacy criterion C and subject program f, we executed f against test cases drawn from the test data sets described in Section 3.2. For each combination (f,C) and each coupling sequence $s_{j,k}$ in f, the following procedure was used:

1. Execute f using at least one test case $c \in S_{C,f}$ drawn from the test set $S_{C,f}$, such that the context variable o of $s_{j,k}$ is bound to an instance of the declared type of o.
2. Record this result and add it to the test oracle for f, Ω_f.
3. For the *All-Poly-Classes* and *All-Poly-Def-Uses*: Execute f with at least one test case $c \in S_{C,f}$ for each combination of (t,v,d_v,u_v,p), where t is a descendant of the declared type of the context variable of $s_{j,k}$, v is a variable in $s_{j,k}$'s coupling set, d_v is a last definition of v by the antecedent method of $s_{j,k}$, u_v is a first use of v in the consequent method of $s_{j,k}$, and p is a definition clear path from d_v to u_v.
4. For *All-Poly-Classes* and *All-Poly-Def-Uses* criteria: Record the combination of t and v in Ω_f. Also, for *All-Poly-Def-Uses*, include the state of the instance bound to the context variable after execution of the antecedent method and immediately after each first-use in the consequent method.

4.2. Fault injection

The following steps were used to inject faults into each subject program f: for each coupling sequence $s_{j,k}$ in f, and each type t that is a subtype of the declared type T of $s_{j,k}$'s coupling variable:

1. 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' (also a subtype of T) and results in a *shadow inheritance hierarchy* rooted at T, as illustrated in Figure 2. The shadow

hierarchy mirrors the original hierarchy in structure below the root, but is seeded with faults.

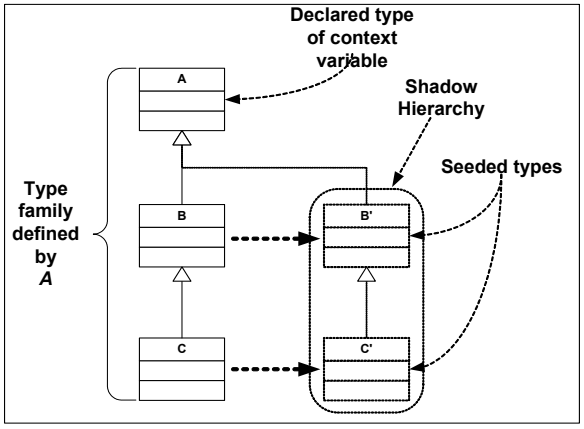

Figure 2. Class hierarchy with seeded shadow hierarchy for *All-Poly-Classes*

2. For *All-Poly-Def-Uses*: For each coupling variable in $s_{j,k}$, inject corresponding faults into the antecedent and consequent methods, yielding the fault seeded type t'' (also a subtype of T). This results in a shadow inheritance hierarchy rooted at T, as illustrated in Figure 3.

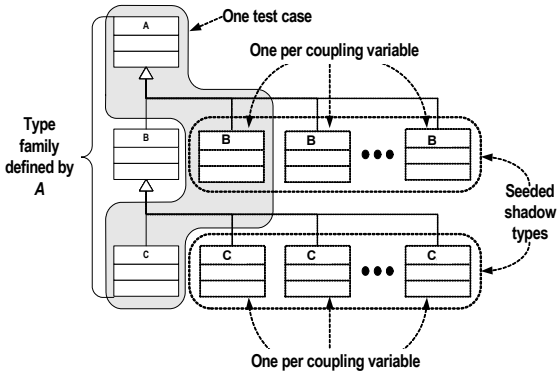

Figure 3. Class hierarchy with seeded shadow types for *All-Coupling-Defs-Uses*

4.3. Test execution

The following steps were used to execute each subject program f with the fault seeded types described in Section 4.1. For each coupling sequence $s_{j,k}$ in f, and each type t that is a subtype of the declared type T of $s_{j,k}$'s coupling variable:

1. Execute f using a test case c that binds $s_{j,k}$'s context variable to the corresponding fault-seeded type t'. Record the result in the test result set for f, Ψ_f.
2. For each test case $c \in S_{C,f}$ execute f using c, and record the state of the instance bound to $s_{j,k}$'s context variable for the corresponding pairs of last-definitions and first-uses of each coupling variable. Add this result to Ψ_f.

4.4. Result evaluation

For each coupling sequence $s_{j,k}$ in f, and each type t that is a subtype of the declared type T of $s_{j,k}$'s coupling variable:

1. Compare each test result in Ψ_f with the corresponding result in the test oracle Ω_f. If the two results are equal, then the test was passed. 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. For *All-Poly-Def-Uses*: Compare each test result in Ψ_f for each coupling variable v in $s_{j,k}$ with the corresponding pair in the test oracle Ω_f. If the results are equal, the test was passed. 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 that is a subtype of the declared type of o.

5. Results

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

Figure 4 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

Table 5: Experimental Results

		Faults Seeded					Faults Detected					Detection Effectiveness					$\overline{X}$
Program	Criterion	SDA	IC	SDI	IISD	SDIH	SDA	IC	SDI	IISD	SDIH	SDA	IC	SDI	IISD	SDIH	
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

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), which, as shown in Table 5, has average detection effectiveness across fault types of $\overline{X}_{APDU} = 0.82$. The other coupling-based criteria have average detection effectiveness of 0.63 (APC) and 0.37 (ACS), with Branch Coverage having the lowest detection effectiveness of 0.12. Plots for the average effectiveness of each program across all subject criteria are given in Alexander's dissertation [1].

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

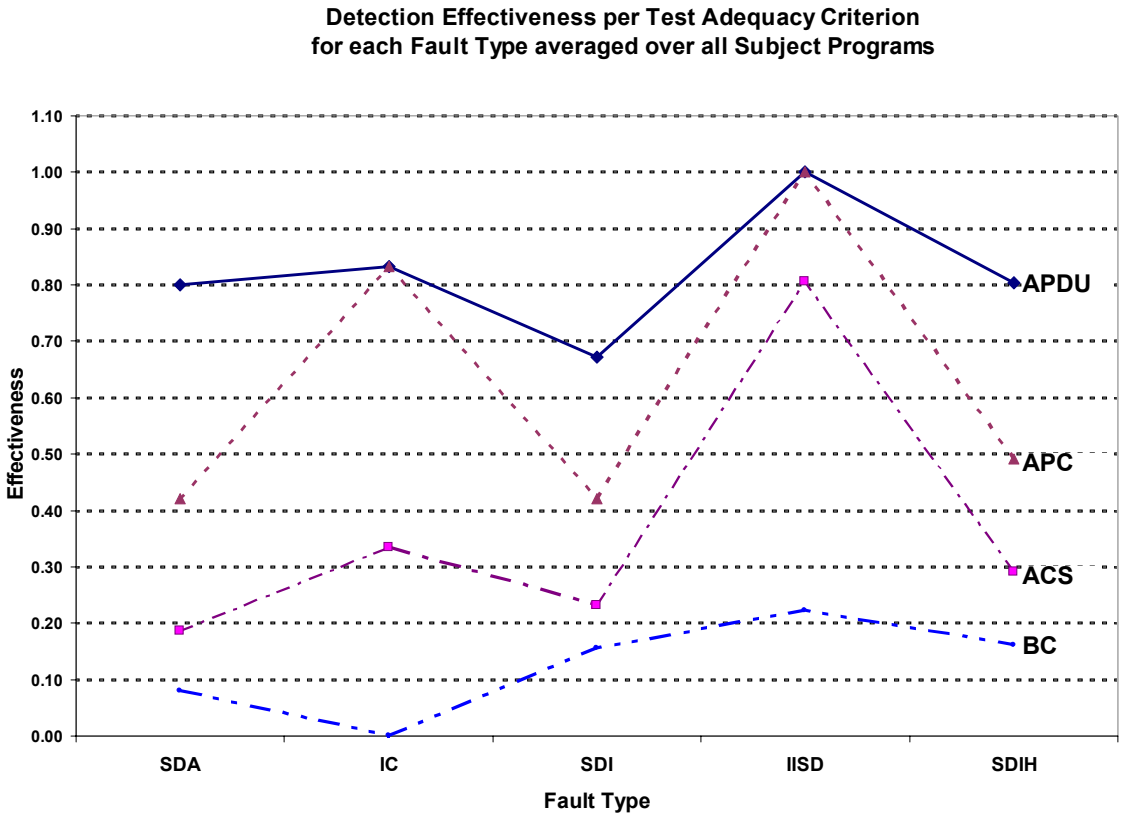

Figure 4. Average detection effectiveness by fault type

average detection effectiveness across fault types (i.e. BC < ACS < APC < APDU).

5.1. 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 18 percent less, making it not quite as effective as for SDIH faults. Similarly, the remaining coupling criteria also reflect less average detection effectiveness. APC is approximately 14 percent less, yielding 0.42, and the decrease for ACS is sightly less 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 are less for SDA faults, 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. ACS detected about 34 percent fewer SDIH faults, and approximately 17 percent fewer than 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 23 percent less average detection effectiveness across all fault types of 0.63, and for ACS, the effectiveness 0.37 is approximately 55 percent less as compared to APDU and 41 percent less 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, Branch Coverage, APC, and ACS are all less effective, approximately by 85 percent, 81 percent, and 66 percent, respectively.

5.2. 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. According to the first condition of the fault/failure model, a location that contains a fault must be reached before the fault can manifest a failure [8, 12]. The shortcoming of ACS is 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. 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.

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. 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. This validates our theory that state interactions need to be explicitly tested for.

5.3. Hypothesis tests

Log-linear analysis permits one to analyze categorical data in much the same manner as in analysis of variance. The sampling distribution underlying Table 6 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 [7]. Therefore the estimation procedures for the simpler sampling distributions may be used, at least for large samples. The resulting estimates are close to the correct maximum likelihood estimates and the usual goodness of fit statistics are asymptotically chi-square.

We first fitted the experimental results to a model

Table 6: Results of hypothesis tests

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

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 5, we can see that the fitted cell counts closely match the observed cell counts.

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

For the initial hypothesis test, we tested for an interaction between criterion and fault type by fitting the model described above with and without the fault-type/ criterion term. If there is no interaction, we can simply pick the best criterion and only use it for our testing. If there is an interaction, then we will have to use two or more of the criteria to adequately test for all of the fault types. For this test, the difference in the total χ^2 that the term of criterion × 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.

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 6) and then tested for an interaction between these two by fitting the model described with and without the corresponding fault-type/criterion term. Table 6 summarizes the results of these tests. The column labeled *Hypothesis* states the null (H_0) and alternative hypothesis (H_1) for each test. The columns labeled χ^2 and $\Delta\chi^2$ give the change in value of the chi-square goodness-of-fit statistic, and the columns labeled *df* and Δdf give the corresponding change in degrees of freedom. Finally, the last column gives the result of each test, indicating whether the null hypothesis is rejected or not.

As the table shows, for hypotheses one through six, there was a net change in the degrees of freedom and the χ^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. 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 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.

6. Discussion

The three hypotheses in Table 6 that tested the effectiveness of each coupling-based criteria against Branch Coverage indicate that the coupling criteria are better at detecting the object-oriented faults used in the experiment. A remaining question is which of the three coupling criteria is the most effective. Hypotheses one, two, and three have established that each of the coupling criteria are better than Branch Coverage. Observation of the plot in Figure 4 suggests that APDU is, on average, more effective that APC and ACS. Similarly, APC is, also, on average, more effective than ACS. This observation is 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 4 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, other factors must be considered when choosing a test adequacy criteria *C*. Cost can be defined in many ways, including the number of test cases required to satisfy *C* and the time required to analyze a program to determine if a desired level test coverage has been attained. An observation made in 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, however, our research offers a clear cost/benefit trade-off.

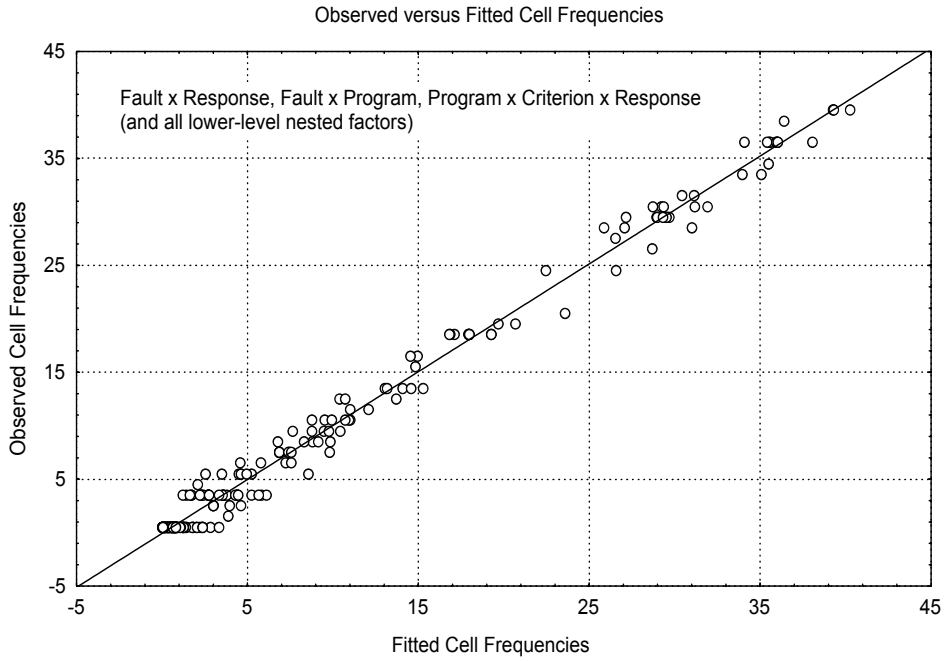

Figure 5. Observed versus fitted cell frequencies

7. Conclusions

The experiments 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.

Acknowledgments

We would like to thank Dr. Gene R. Lowrimore of the Center for Demographic Studies at Duke University for his assistance with the statistical analysis of the experimental results.

References

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

[2] Roger T. Alexander and A. Jefferson Offutt. *Analysis Techniques for Testing Polymorphic Relationships*. In *Proceedings of the Thirtieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS30 '99)*. 1999. Santa Barbara CA: IEEE Computer Society.

[3] Roger T. Alexander and A. Jefferson Offutt. *Criteria for Testing Polymorphic Relationships*. In *Proceedings of the International Symposium on Software Reliability and Engineering (ISSRE00)*. 2000. San Jose CA: IEEE Computer Society.

[4] Boris Beizer, *Software Testing Techniques*. 1990, New York, New York: Van Nostrand Reinhold.

[5] Edward V. Berard, *Essays on Object-Oriented Software Engineering*. Vol. 1. 1993: Prentice Hall.

[6] Robert V. Binder, *Testing Object-Oriented Software: A Survey.* Journal of Software Testing, Verification and Reliability, 1996. **6**(3/4): p. 125-252.

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

[8] R. A. DeMillo and A. J. Offutt, *Constraint-based Automatic Test Data Generation.* IEEE Transactions on Software Engineering, 1991. **17**(9): p. 900-910.

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

[10] Zhenyi Jin and A. Jefferson Offutt, *Coupling-based Criteria for Integration Testing.* The Journal of Software Testing, Verification, and Reliability, 1998. **8**(3): p. 133-154.

[11] Bertrand Meyer, *Object-Oriented Software Construction*. 1997, Englewood Cliffs, New Jersey: Prentice Hall.

[12] L. J. Morell. *Theoretical Insights into Fault-Based Testing*. In *Proceedings of the ACM SIGSOFT '89 2nd Symposium on Software Testing Analysis and Verification (TAV2)*. 1988. Banff Alberta.

[13] Jeff Offutt, Roger Alexander, Ye Wu, Quansheng Xiao, and Chuck Hutchinson. *A Fault Model for Subtype Inheritance and Polymorphism*. In *Proceedings of the Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE '01)*. 2001. Hong Kong, PRC.