Incremental Testing of Object-Oriented Class Structures†

Mary Jean Harrold, John D. McGregor and Kevin J. Fitzpatrick
Department of Computer Science
Clemson University
Clemson, SC 29634-1906

Abstract

Although there is much interest in creating libraries of well-designed, thoroughly-tested classes that can be confidently reused for many applications, few class testing techniques have been developed. In this paper, we present a class testing technique that exploits the hierarchical nature of the inheritance relation to test related groups of classes by reusing the testing information for a parent class to guide the testing of a subclass. We initially test base classes having no parents by designing a test suite that tests each member function individually and also tests the interactions among member functions. To design a test suite for a subclass, our algorithm incrementally updates the history of its parent to reflect both the modified, inherited attributes and the subclass's newly defined attributes. Only those new attributes or affected, inherited attributes are tested and the parent class' test suites are reused, if possible, for the testing. Inherited attributes are retested in their new context in a subclass by testing their interactions with the subclass's newly defined attributes. We have incorporated a data flow tester into Free Software Foundation, Inc's C++ compiler© and are using it for our experimentation.

1. Introduction

One of the main benefits of object-oriented programming is that it facilitates the reuse of instantiable, information-hiding modules, or *classes*. A class is a template that defines the *attributes* that an object of that class will possess. A class's attributes consist of (1) *data members* or *instance variables* that implement the object's state and (2) *member functions* or *methods* that implement the operations on the object's state. Classes are used to define new classes, or subclasses, through a relation known as *inheritance*. Inheritance imposes a hierarchical organization on the classes and permits a subclass to inherit attributes from its parent classes and either extend, restrict or redefine them. Subclasses may inherit attributes from a parent class, cancel attributes in the parent, contain new attributes not possessed by the parent, and/or redefine some of the parent's attributes. A goal of object-oriented programming is to create libraries of well designed and thoroughly tested classes that can be confidently reused for many applications.‡

Although there is much interest in creating class libraries, few class testing techniques have been developed. One approach is to validate each class in the library individually. However, this approach requires complete retesting of each subclass although many of its attributes were previously tested since they are identical to those in the parent class. Additionally, completely retesting each class does not exploit opportunities to reuse and share the design, construction and execution of test suites. Another approach to class testing is to utilize the hierarchical nature of classes related by inheritance to reduce the overhead of retesting each subclass. However, Perry and Kaiser[16] have shown that many inherited attributes in subclasses of well designed and thoroughly tested classes must be retested in the context of the subclasses. Thus, any subclass testing technique must ensure that this interaction of new attributes and inherited attributes is thoroughly tested. Fielder[3] presented a technique to test subclasses whose parent classes have been thoroughly tested. Part of his test design phase is an analysis of the effects of inheritance on the subclass. He suggests that only minimal testing may be required for inherited member functions whose functionality has not changed. Cheatham and Mellinger[2] also discuss the problem of subclass testing and present a more extensive analysis of the retesting required for a subclass. However, both of these subclass testing techniques require that the

† This work was partially supported by the National Science Foundation under Grant CCR-9109531 to Clemson University.

‡ More detailed discussion of object-oriented programming is given by Korson and McGregor[12].

analysis be performed by hand, which prohibits automating the design phase of the testing. Additionally, neither technique attempts to reuse the parent class's test suite to test the subclass.

In this paper, we present an incremental class testing technique that exploits the hierarchical nature of the inheritance relation to test related groups of classes by reusing the testing information for a parent class and incrementally updating it to guide the testing of the subclass. We initially test *base classes* having no parents by designing a test suite that tests each member function individually and also tests the interactions among member functions. A *testing history* associates each test case with the attributes that it tests. In addition to inheriting attributes from its parent, a newly defined subclass "inherits" its parent's testing history. Just as a subclass is derived from its parent class, a subclass's testing history is derived from the testing history of its parent class. The inherited testing history is incrementally updated to reflect differences from the parent and the result is a testing history for the subclass. A subclass's testing history guides the execution of the test cases since it indicates which test cases must be run to test the subclass. With this technique, we automatically identify new attributes in the subclass that must be tested along with inherited attributes that must be retested. We retest inherited attributes in the context of the subclass by identifying and testing their interactions with newly defined attributes in the subclass. We also identify which of the test cases in the parent class's test suite can be reused to validate the subclass and which attributes of the subclass require new test cases.

The main benefit of this approach is that completely testing a subclass is avoided since the testing history of the parent class is reused to design a test suite for a subclass. Only new or replaced attributes in the subclass or those affected, inherited attributes are tested. Additionally, test cases from the test suite of the parent class are reused, if possible, to test the subclass. Thus, there is a savings in the time to design test cases, the time to construct a new test suite and the actual time to execute the test suite since the entire subclass is not tested. Since our technique is automated, there is limited user intervention in the testing process.

The next section gives background information on procedural language testing. Section 3 discusses inheritance in object-oriented programs as an incremental modification technique. Section 4 presents our incremental testing technique by first giving an overview, and then detailing, both base class testing and subclass testing. At the end of this section, we discuss our implementation. Section 5 discusses experimentation, and concluding remarks are given in Section 6.

2. Testing

The overall goal of testing is to provide confidence in the correctness of a program. With testing, the only way to guarantee a program's correctness is to execute it on all possible inputs, which is usually impractical. Thus, systematic testing techniques generate a representative set of test cases to provide coverage of the program according to some selected criteria. There are two general forms of test case coverage: *specification-based* and *program-based*[9]. In specification-based or 'black-box' testing, test cases are generated to show that a program satisfies its functional and performance specifications. Specification-based test cases are usually developed manually by considering a program's requirements. In program-based or 'white-box' testing, the program's implementation is used to select test cases to exercise certain aspects of the code such as all statements, branches, data dependencies or paths. For program-based testing, analysis techniques are often automated. Since specification-based and program-based testing complement each other, both types are usually used to test a program.

While most systematic testing techniques are used to validate program *units*, such as procedures, additional testing is required when the units are combined or integrated. For *integration* testing, the interface between the units is the focus of the testing. Interface problems include errors in input/output format, incorrect sequencing of subroutine calls, and misunderstood entry or exit parameter values[1]. Although many of the integration testing techniques are specification-based, some interprocedural program-based testing techniques have recently been developed[8, 13].

A test set is *adequate* for a selected criterion if it covers the program according to that criterion [20] and a program is deemed to be *adequately* tested if it has been tested with an adequate test set. Weyuker[20] developed a set of axioms for test data adequacy that expose insufficiencies in program-based adequacy criteria. Several of these axioms are specifically related to unit and integration testing. The *antiextensionality* axiom reminds us that two programs that compute the same function may have entirely different implementations. While the same specification-based test cases may be used to test each of the programs, different program-based test cases may be required. Thus, changing a program's implementation may require additional test cases. The *antidecomposition* axiom tells us that adequately testing a program P does not imply that each component of P is adequately tested. Adequately testing each program component is especially important for those components that may be used in other environments where the input values may differ. Thus, each

unit that may be used in another environment must be individually tested. The *anticomposition* axiom tells us that adequately testing each component Q of a program does not imply that the program has been adequately tested. Thus, after each component is individually tested, the interactions among the components must also be tested.

3. Inheritance in Object-Oriented Systems

Inheritance is a mechanism for both class specification and code sharing that supports developing new classes based on the implementation of existing classes. A subclass's definition is a *modifier* that defines attributes that differ from, or alter, the attributes in the parent class. The modifier and parent class along with the inheritance rules for the language are used to define the subclass. The class designer controls the specification of the modifier while the inheritance controls the combination of the modifier and the parent class to get the subclass. Figure 1 illustrates inheritance as an incremental modification technique[18] that transforms a parent class P with modifier M into a resulting class R. The composition operator ⊕ symbolically unites M and P to get R, where R = P ⊕ M.

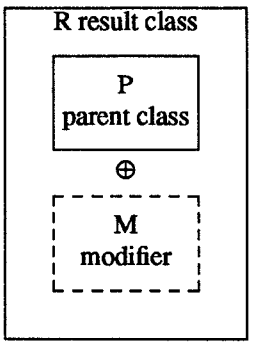

Figure 1. Incremental modification technique

The subclass designer specifies the modifier, which may contain various types of attributes that alter the parent class. These include the redefined, virtual and recursive attributes presented by Wegner[18] along with an additional type of attribute, the *new* attribute. We further classify Wegner's virtual attribute as *virtual-new*, *virtual-recursive* and *virtual-redefined*. In the following discussion, we reference Figure 1 and define these six types of attributes.

New attribute: (1) A is an attribute that is defined in M but not in P or (2) A is a member function attribute in M and P but A's argument list differs in M and P. In this case, A is bound to the locally defined attribute in the resulting class R but is not in P.

Recursive attribute: A is defined in P but not in M. In this case, A is bound to the locally defined attribute in P and is available in R.

Redefined attribute: A is defined in both P and M where A's argument list is the same in M and P. In this case, A is bound to the attribute definition in M which blocks the definition of the similarly named attribute in P.

Virtual-new attribute: (1) A is specified in M but its implementation may be incomplete in M to allow for later definitions or (2) A is specified in M and P and its implementation may be incomplete in P to allow for later definitions, but A's argument list differs in M and P. In this case, A is bound to the locally defined attribute in the resulting class R but is not in P.

Virtual-recursive attribute: A is specified in P but its implementation may be incomplete in P to allow for later definitions, and A is not defined in M. In this case, A is bound to the locally defined attribute in P and is available in R.

Virtual-redefined attribute: A is specified in P but its implementation may be incomplete in P to allow for later definition, and A is defined in M with the same argument list as A in P. In this case, A is bound to the attribute definition in M which blocks the definition of the similarly named attribute in P.

Although the modifier M transforms a parent class P into a resulting class R, M does not totally constrain R. We must also consider the inheritance relation since it determines the effects of composing the attributes of P and M and mapping them into R. The inheritance relation determines the visibility, availability and format of P's attributes in R. A language may support more than one inheritance mapping by allowing the specification of a parameter value to determine which mapping is used for a particular definition. For example, in C++, the *public* and *private* keywords as part of the specification of the inheritance relationship determine the visibility of the attributes in the subclass. Since inheritance is deterministic, it permits the construction of rules to identify the availability and visibility of each attribute. This feature supports automating the process of analyzing a class definition and determining which attributes require testing. To illustrate some of the different types of attributes, consider Figure 2, where *class* P is given on the left, the modifier that specifies R, a subclass of P, is given in the center, and

```
class P {                          class R : public P {
 private:                           private:
   int i;                             float i;
   int j;
 public:                            public:
   P(){}                              R(){}
   void A(int a,int b)
       {i=a; j=a+2*b;}

                                      void A(int a)
                                          {P::A(a,0);}
   virtual int B()                    virtual int B()
       {return i;}                        {return 3*P::B();}
   int C()                            int C()
       {return j;}                        {return 2*P::C();}
};                                 };
```

R's attributes after the mapping

```
private:
   float i;                          //new

public:

   void A(int a, int b)              //recursive
       {i=a; j=a+2*b;}
   void A(int a)                     //new
       {P::A(a,0);}
   virtual int B()                   //virtual-redefined
       {return 3*P::B();}
   int C()                           //redefined
       {return 2*P::C();}
```

hidden

```
   int i;
   int j;
```

Figure 2. Class P on the left, subclass R's specification (modificr) in the center, subclass R's attributes on the right.

the attributes for the resulting class R are given on the right. P has two data members, *i* and *j*, both integers, and three member functions, A, B and C; B is a virtual member function. The modifier for class R contains one real data member, *i*, and three member functions, A, B and C. The modifier is combined with P under the inheritance rules to get R. Data member *float i* is a new attribute in R since is does not appear in P. Member function A that is defined in M, is a new attribute in R since its argument list does not agree with A's argument list in P. Member function A in P is recursive in R since it is inherited unchanged from P. Thus, R contains two member functions named A. Member function B is virtual in P and since it is redefined in M, it is virtual-redefined in R. Member function C is redefined in R since its implementation is changed by M and overrides member function C from P. Finally, data members *i* and *j* in P are inherited but hidden in R, which means they cannot be accessed by member function defined in the modifier.

The modifier approach permits a decomposition of the inheritance structure into overlapping sets of class inheritance relations. The left side of Figure 3 shows a simple three-level chain of inheritance relations while the center illustrates an incremental view of the relationship among the classes. Class B can be replaced by A ⊕ M1 since A's attributes and M1's attributes are combined to form B. Once B is defined, there is no distinction in B between A's attributes and M1's attributes. To define a subclass of B, the inheritance relation combines B and M2 in the same way. Thus, the three level inheritance relation can be decomposed into independent structures as illustrated on the right side of Figure 3. Decomposition of

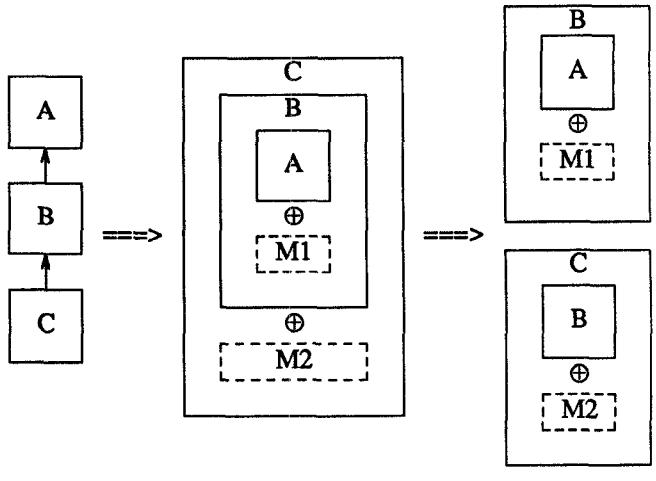

Figure 3. The inheritance hierarchy shown on the left indicates that A is a class with subclasses B and C, B is a class with subclass C. The figure in the center illustrates the incremental format for the class inheritance hierarchy where each new subclass is formed by combining the parent class with some modifier. The figure on the right shows how the class hierarchy can be decomposed into independent structures.

class hierarchies permits us to consider only the immediate parents and the modifier when testing a subclass.

The inheritance relation imposes an ordering on the classes in an inheritance structure. Class C can be deter-

mined without considering class A but the relation from A to B must be resolved prior to describing the relation from B to C. The order in which the classes must be defined is a partial ordering and any inheritance structure can be decomposed into a set of partially ordered pairs of classes. This permits us to consider only a class and its immediate parents to fully constrain the definition of that class.†

4. Hierarchical Incremental Class Testing

Our class testing technique initially tests a base class by testing each member function individually and then testing the interactions among the member functions. The test case design and execution information is saved in a testing history. Then, when a subclass is defined, the testing history of its parent, the definition of the modifier and the inheritance mapping of the implementation language are used to determine which attributes to (re)test in the subclass and which of the parent's test cases can be reused. The technique is hierarchical because it is guided by the partial ordering of the inheritance relation; it is incremental because it uses the results from testing at one level in the hierarchy to reduce the efforts needed by subsequent levels.

Our testing technique assumes a language model that is a generalization of the C++[17] model but is sufficiently flexible to support other languages such as Trellis[10] with similar features. Our language model is (1) strongly typed and permits polymorphic substitution to provide flexibility, (2) uses static binding whenever possible for efficiency, (3) supports three levels of attribute visibility with the same characteristics as C++'s *private*, *protected* and *public*, although the technique can handle any number of visibility levels, and (4) assumes a parameterized inheritance mapping with the two parametric values used in C++, *private* and *public*. The levels of visibility for attributes are ordered from most visible (*public*) to least visible (*private*) and the inheritance mapping maps an attribute to a level of visibility in the subclass that is at least as restrictive as its level in the parent class.

Our incremental testing technique addresses the test data adequacy concerns expressed by Perry and Kaiser[16]. The antidecomposition axiom cautions that adequate testing of a class does not imply that individual member functions have been adequately tested for use in all contexts. Our technique tests each member function independent of its place in the class. Conversely, the anticomposition axiom states that adequately testing a member function in isolation is not sufficient to assume that it has been adequately tested as part of a set of interacting member functions. Our technique uses integration testing techniques to test the interactions between member functions without retesting their internal implementations.

4.1. Base Class Testing

We first test base classes using traditional unit testing techniques to test individual member functions in the class. The antidecomposition axiom tells us that adequate testing of the class does not guarantee adequate testing of each member function. Adequately testing each member function is particularly important since member functions may be inherited by the subclasses and expected to operate in a new context. Thus, we individually test each member function in a class using a test suite that contains both specification-based and program-based test cases. The specification-based test cases can be constructed using existing approaches such as the one proposed by Frankl[4]. During this phase of testing, we follow the standard unit testing practice of handling calls to other member functions (procedures) by providing stubs representing called member functions and drivers representing calling member functions. The testing history for a class contains associations between each member function in the class and both a specification-based and a program-based test suite. Thus, the history contains triples, $\{m_i, (TS_i, test?), (TP_i, test?)\}$ where m_i is the member function, TS_i is the specification-based test suite, TP_i is the program-based test suite and 'test?' indicates whether the test suite is to be (re)run.

The anticomposition axiom implies that testing each member function individually does not mean that the class has been adequately tested. Thus, in addition to testing each member function, we must test the interactions among member functions in the same class, *intra-class* testing; we must also test the interactions among member functions that access member functions in other classes, *inter-class* testing. Intra-class testing is guided by a *class graph* where each node represents either a member function in the class or a primitive data member, and each edge represents a message. The class graph may be disconnected where each connected subgraph represents those attributes in the class that interact with each other. For intra-class integration testing, we combine the attributes as indicated by the class graph and develop test cases that test their interfaces. For intra-class testing, we develop both specification-based and program-based test suites. The history for the class contains the root nodes of the class graph subgraphs representing the interacting member functions along with the test suites for each of them. Thus, the second part of the history also consists of triples, $\{m_i, (TIS_i, test?), (TIP_i, test?)\}$ where m_i is the root nodes of the class graph subgraph, TIS_i is the specification-based integration test suite, and TIP_i is the program-based inte-

† Although a class may have several parents from which it can inherit attributes, for our discussion, we assume that each class has only one parent.

```
class Shape {
private:
        Point reference_point;
public:
        void put_reference_point(Point);
        point get_reference_point();
        void move_to(Point);
        void erase();
virtual  void draw() = 0;
virtual  float area();
        shape(Point);
        shape();
}
```

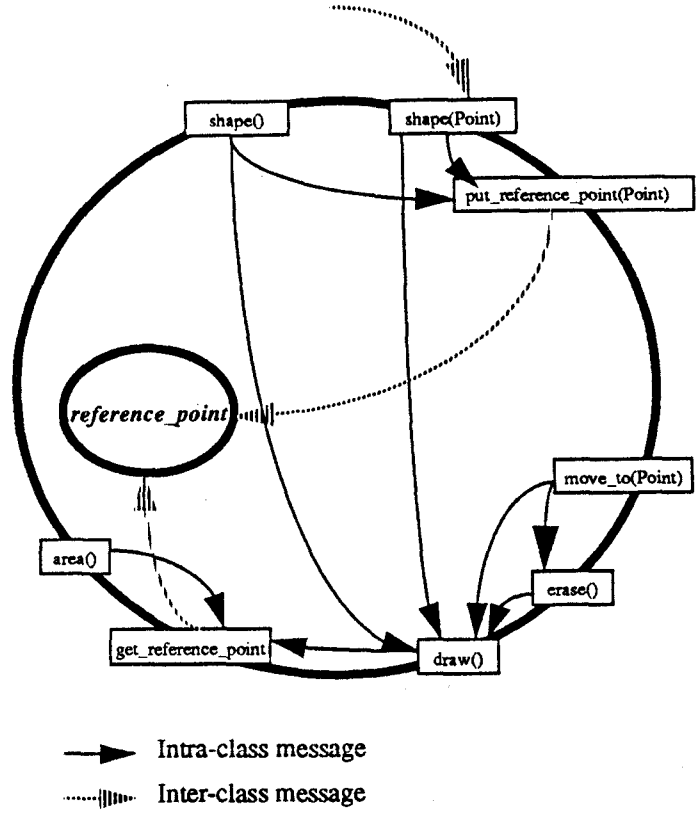

Figure 4. Definition for class *Shape* on the left and its class graph on the right.

gration test suite and 'test?' indicates if the test suite is to be totally (re)run (Y), partially (re)run (P), or not (re)run (N).

Inter-class testing is guided by the interactions of the classes that result when member functions in one class interact with member functions in another class. Inter-class interactions occur when (1) a member function in one class is passed an instance of another class as a parameter and then sends that instance a message or (2) when an instance of one class is part of the representation of another class and then sends that instance as a message. The application's design provides a relationship among the class instances that is similar to the class graph produced for intra-class testing. The techniques for handling these interactions are like those described above for intra-class interactions except that interacting attributes are in different classes. We omit the details for inter-class testing since it is analogous to intra-class testing.

To illustrate our technique for testing base classes, consider the simplified example of a hierarchy of graphical shape classes implemented in C++[17]. Class *Shape*, given in Figure 4, is an abstract class that facilitates the creation of classes of various shapes for graphics display. The class definition is abbreviated for the purpose of illustrating the testing algorithm and we omit the bodies of the member functions. Each 'shape' that can be drawn in the graphics system has a reference_point that is used to locate the position where the shape is drawn in the program's coordinate system.

Class *Shape* defines several member functions that describe the behavior of a shape and includes two virtual member functions, area() and draw(), that contribute to the common interface for all classes in the inheritance structure. Since draw() is a pure virtual member function, it has an initial value of 0 and no implementation. Virtual member function area() is assumed to have an initial implementation that can be changed in subsequent subclass declarations. The put_reference_point() and get_reference_point() member functions provide controlled access to the values of the data members, the shape(Point) and shape() member functions are constructors of instances of the class, and the move_to() member function moves the shape to a new location. Move_to() can be totally defined in terms of member functions in class *Shape* even though some of these member functions are virtual and have no implementation. The erase() member function may be implemented in several ways but if an 'xor' drawing mode is used, erase only calls draw to overwrite, and thus erase, the existing figure. The class graph for *Shape* is also given in Figure 4. Rectangles represent member functions and ovals represent instances of classes. Solid lines indicate intra-class messages while dashed lines indicate inter-class messages. The table in

Testing History for *Shape*		
attribute	specification-based test suite	program-based test suite
individual member functions		
put_reference_point	(TS_1,Y)	(TP_1,Y)
get_reference_point	(TS_2,Y)	(TP_2,Y)
move_to	(TS_3,Y)	(TP_3,Y)
erase	(TS_4,Y)	(TP_4,Y)
draw	(TS_5,Y)	(--)
area	(TS_6,Y)	(TP_6,Y)
shape	(TS_7,Y)	(TP_7,Y)
shape	(TS_8,Y)	(TP_8,Y)
interacting member functions		
move_to	(TIS_9,Y)	(TIP_9,Y)
erase	(TIS_{10},Y)	(TP_{10},Y)

Figure 5. Testing history for Class *Shape* of Figure 4.

Figure 5 shows the testing history for class *Shape*. The analysis of *Shape* is very straight forward. Since *Shape* is a base class, we must test each of its available definitions. The specification-based test suite for draw() can be generated but cannot be run since there is no implementation for draw(). The program-based test suite for draw() cannot be generated since no implementation exists. Since there is an initial implementation for area(), both its specification-based and program-based test suites can be generated and run. The specification-based and program-based test suites for move_to(), erase() and the two shape() constructors are generated. The test suites for the constructors are independently tested since they do not rely on the implementations of either draw() or area(). The advantage of developing the specification-based test suites for draw() and area() in the base class is that these test suites can be 'inherited' in the histories of subclasses.

The class's specification describes how the individual member functions are intended to work together. The input values that test these interactions belong to the integration test suites TIS_i and TIP_i that are part of our intra-class testing member functionality. In addition to the test suites for the individual member functions in Shape, the interface test suites are shown in the history in Figure 5. Member functions move_to(), erase(), shape() and shape(Point) call other class member functions: move_to() calls both erase() and draw(), erase() also calls draw(), shape() and shape(Point) both call put_reference_point() and draw(). The class graph for Shape, shown in Figure 4, illustrates these interactions. The class graph serves as a guide for generating program-based test cases to test each of the interactions.

There are several inter-class messages: messages to construct instances of shape as well as the messages between member functions of class shape and reference_point, which is an instance of class Point. Integration test cases are used to validate these messages but for brevity, we omit them from this example.

4.2. Testing Subclasses

Our testing algorithm, *TestSubclass* given in Figure 6, uses an incremental technique that transforms the testing history for the parent class P to the testing history for the subclass R. *TestSubclass* inputs P's history, HISTORY(P), P's class graph, G(P), and modifier M and outputs an updated HISTORY for the subclass R. The actions taken by *TestSubclass* depend on the attribute type and the type of modification made to that attribute by the inheritance mapping. In section 3, we discussed six types of attributes: new, recursive, redefined, virtual-new, virtual-recursive and virtual-redefined. For each of these types of attributes, different actions may occur. Algorithm *TestSubclass* begins by initializing R's history to that of its parent class P, which has already been tested. The algorithm then inspects each attribute A in the modifier M, and takes appropriate action to update R's history and determine the required testing.

Any NEW or VIRTUAL-NEW member function attribute A must also be completely, individually tested since it was not defined in P. Since the anticomposition axiom tells us that it is necessary to retest each new member function in its new context, A must be integration tested with other member functions in R with which it interacts. We thoroughly test A individually so that when A is inherited by some subclass, only integration testing will need to be repeated. To individually test A, new specification-based and program-based test suites are developed, added to R's HISTORY and marked for testing

```
algorithm TestSubclass(HISTORY(P),G(P),M);

input:    HISTORY(P):P's testing history; G(P):P's class graph;
          M:modifier that specifies subclass R;
output:   HISTORY(R):testing history for R indicating what to rerun; G(R):class graph for subclass R;
begin
  HISTORY(R) := HISTORY(P);                    /* initialize R's history to that of P */
  G(R) := G(P);                                /* initialize R's class graph to that of P */
  foreach attribute A ∈ M do
    case A is NEW or NEW-VIRTUAL:              /* A is a new/virtual-new attribute */
        Generate TS , TP for A;
        Add {A, (TS, Y), (TP,Y)} to HISTORY(R);
        Integrate A into G(R);
        Generate any new TIS and TIP;
        Add {A, (TIS, Y), (TIP,Y)} to HISTORY(R);
    case A is RECURSIVE or RECURSIVE-VIRTUAL:  /* A is recursive/virtual-recursive */
        if A accesses data in R's scope then
          Identify interface tests to reuse;
          Add (TIS,P) and (TIP,P)} to HISTORY(R);
    case A is REDEFINED or REDEFINED-VIRTUAL:  /* A is redefined /virtual-redefined */
        Generate TP for A;
        Reuse TS from P if it exists or Generate TS for A;
        Add {A, (TS,Y), (TP,Y)} to HISTORY(R);
        Integrate A into G(R);
        Generate TIP for G(R) with respect to A;
        Reuse TIS from P;
        Add {A, (TIS,P), (TIP,P)} to HISTORY(R);
end TestSubclass.
```

Figure 6. Algorithm *TestSubclass* that determines a testing HISTORY for R by incrementally updating the HISTORY for its parent class. The HISTORY is used to test the subclass.

---

by setting the 'test?' field to 'Y'. A new member function may send messages to existing member functions of the class or may reference existing data members. Thus, A is tested with any other member functions in R with which it interacts by first integrating it into the G(R). Then integration tests are generated, added to HISTORY(R) and marked for testing by setting the 'test?' field to 'Y'. A new data attribute is tested during integration testing when it is integrated into G(R) by testing A with member functions with which it interacts.

A RECURSIVE or VIRTUAL-RECURSIVE member function attribute A requires very limited retesting since it was previously individually tested in P and the specification and implementation remain unchanged. Thus, the specification-based and program-based test suites for A are not rerun. The antidecomposition axiom reminds us that it is necessary to test A in its new context in the subclass. Integration test cases are not reused if they only test the interaction of this recursive attribute with other recursive attributes since this interaction has also been previously tested. However, A may interact with new or redefined attributes or access the same instances in the class's representation as other member functions. A's interaction with new or redefined attributes is tested when those member functions are integrated into the subclass so there is no need to retest here. This limited testing adequately tests the attributes in the subclass because of the extensive testing that occurred when the attribute was defined.

Consider the case in which a recursive member function accesses the same data as a new attribute. The recursive member function was tested in P and the specification of the recursive member function remains unchanged. Thus, specification-based and program-based test cases need not be rerun. The only test cases that are rerun are those that test the interactions between A and any new member function(s). TestSubclass uses incremental techniques to identify those test cases, marks them for retesting by setting 'test?' to 'P' and updates HISTORY(R) to reflect the changes. 'Y' indicates that the entire test suite is reused while 'P' indicates that only the test cases identified as testing affected parts of the subclass are reused.

```
class Triangle: public Shape {
    private:
        Point vertex2;
        Point vertex3;
    public:
        Point get_vertex1();              //new
        Point get_vertex2();              //new
        Point get_vertex3();              //new
        void set_vertex1(Point);          //new
        void set_vertex2(Point);          //new
        void set_vertex3(Point);          //new
        void draw();      //virtual-redefined
        float area();     //virtual-redefined
        triangle();                       //new
        triangle(Point,Point,Point);      //new
}
```

Testing History for *Triangle*		
attribute	specification test suite	program-based test suite
individual member functions		
put_reference_point	(TS_1,N)	(TP_1,N)
get_reference_point	(TS_2,N)	(TP_2,N)
move_to	(TS_3,N)	(TP_3,N)
erase	(TS_4,N)	(TP_4,N)
draw	(TS_5,Y)	(TP'_5,Y)
area	(TS_6,Y)	(TP'_6,Y)
shape	(TS_7,N)	(TP_7,N)
shape	(TS_8,N)	(TP_8,N)
get_vertex1	(TS'_{11},Y)	(TP'_{11},Y)
get_vertex2	(TS'_{12},Y)	(TP'_{12},Y)
get_vertex3	(TS'_{13},Y)	(TP'_{13},Y)
put_vertex1	(TS'_{14},Y)	(TP'_{14},Y)
put_vertex2	(TS'_{15},Y)	(TP'_{15},Y)
put_vertex3	(TS'_{16},Y)	(TP'_{16},Y)
triangle	(TS'_{17},Y)	(TP'_{17},Y)
triangle	(TS'_{18},Y)	(TP'_{18},Y)
interacting member functions		
move_to	(TIS''_9,P)	(TIP''_9,P)
erase	(TIS''_{10},P)	(TIP''_{10},P)
area	(TIS'_{19},Y)	(TIP'_{19},Y)
get_vertex1	(TIS'_{20},Y)	(TIP'_{20},Y)
put_vertex1	(TIS'_{21},Y)	(TIP'_{21},Y)

Figure 7. Definition and History for Class *Triangle*. Test suites marked with 'Y' or 'P' are reused to test the subclass; those marked with 'N' are not rerun. 'Y' indicates that all test cases in the test suite are reused; 'P' means that only part of that test suite is reused. A comment with each of the public attributes indicates its type; all other inherited attributes are recursive.

Both data member and member function attributes are defined in the parent class and inherited by the subclass. The inheritance mapping from P into R may change the visibility of a data member attribute. For example, if the attribute has moved from a visible level to one that is not visible then it cannot interact with any new or redefined attribute. The data attributes that are hidden and the member functions on that data form a tested unit that need not be retested. If the data attribute is visible to any new member functions that are defined in R, then the interfaces between the new member functions and the existing member functions that access the data attributes must be tested. This testing is performed when a new member function that interacts with data attributes accessed by existing member functions is integrated into G(R).

A REDEFINED or VIRTUAL-REDEFINED attribute A in M requires extensive retesting but many existing specification-based test cases may be reused since only the implementation has changed. The antiextensionality axiom tells us that since the the implementation has changed, new program-based test cases may be required. If A is a data member (i.e. an instance of a class) we assume that the class to which the instance belongs has been tested. No other individual testing is performed on A although it may participate in the integration testing of the member functions defined in M. If A is a member function, the specification of A remains unchanged but the implementation of A will have changed. Thus, A is individually retested by generating new program-based test cases to test the implementation of A. The specification-based test cases stored in HISTORY(R) for the previous definition of A are still valid and are reused. HISTORY(R) is updated to reflect the new test cases and reused existing test cases, and these test cases are marked for reusing by setting 'test?' to 'Y'. Then, A is integrated into G(R). New program-based interface test cases are generated and marked for testing by setting 'test?' to 'Y' or 'P'. HISTORY(R) is again updated to reflect the changes.

To illustrate the way in which algorithm *TestSubclass* works, we consider subclasses of Class *Shape* that was given in Figure 4. The benefits of hierarchical incremental testing can be seen in the testing history for class *Triangle*, given in Figure 7. None of the test suites for the put_() and get_() member functions for reference_point are rerun because they are recursive attributes since no

```
class EquiTriangle: public Triangle{
    public:
        float area();                          //redefined
        equi_triangle(Point,Point,Point);      //new
        equi_triangle();                       //new
}
```

Testing History for *EquiTriangle*		
attribute	specification test suite	program-based test suite
individual member functions		
put_reference_point	(TS_1,N)	(TP_1,N)
get_reference_point	(TS_2,N)	(TP_2,N)
move_to	(TS_3,N)	(TP_3,N)
erase	(TS_4,N)	(TP_4,N)
draw	(TS_5,N)	(TP_5,N)
area	(TS_6,Y)	(TP'_6,Y)
shape	(TS_7,N)	(TP_7,N)
shape	(TS_8,N)	(TP_8,N)
get_vertex1	(TS_{11},N)	(TP_{11},N)
get_vertex2	(TS_{12},N)	(TP_{12},N)
get_vertex3	(TS_{13},N)	(TP_{13},N)
put_vertex1	(TS_{14},N)	(TP_{14},N)
put_vertex2	(TS_{15},N)	(TP_{15},N)
put_vertex3	(TS_{16},N)	(TP_{16},N)
triangle	(TS_{17},N)	(TP_{17},N)
triangle	(TS_{18},N)	(TP_{18},N)
equi_triangle	(TS_{22},Y)	(TP_{22},Y)
equi_triangle	(TS_{23},Y)	(TP_{23},Y)
interacting member functions		
move_to	(TIS''_9,P)	(TIP''_9,P)
erase	(TIS''_{10},P)	(TIP''_{10},P)
area	(TIS''_{19},P)	(TIP''_{19},P)

Figure 8. Definition and History for Class *EquiTriangle*. Test suites marked with 'Y' or 'P' are reused to test the subclass; those marked with 'N' are not rerun. 'Y' indicates that all test cases in the test suite are reused; 'P' means that only part of that test suite is reused. A comment with each of the public attributes indicates its type; all other inherited attributes are recursive.

changes are made in their definitions. Virtual-redefined member functions draw() and area() are retested since they have new implementations. However, only new program-based test cases are developed since existing specification-based test cases can be reused. Test suites are developed and run for the new member functions defined in *Triangle*. Three member functions have been added to the list of interacting member functions: area() calls the get_vertex() member functions and both get_vertex1() and put_vertex1() call the get_() and put_() member functions for reference_point respectively. The interfaces between these pairs of member functions must be tested, but get_reference_point() and put_reference_point() need no further individual testing. No member function-interaction test suites for move_to() and get_reference_point() are executed since no member function defined in *Triangle* can directly access reference_point. In Figure 7, test suites marked with ' are newly developed, test suites marked with " may have newly developed test cases, while all others are reused from the parent.

The last class in the hierarchy is *EquiTriangle* which adds no new member functions other than the constructors for the class. However, *EquiTriangle* redefines the implementation of area() to provide more efficiency. Only the program-based test cases for area() are regenerated, although all test cases for area() are rerun. Integration test cases to test the interactions of the new and redefined member functions with the inherited attributes are also run. The definition of class *EquiTriangle* and its testing history are given in Figure 8.

4.3. Implementation

The implementation of our testing system consists of two main parts. The first part uses our algorithm *TestSubClass* to automatically identify the required retesting in a subclass. The second part assists in performing the subclass testing. Although, our hierarchical incremental algorithm is independent of the testing methodology, we are using a type of program-based testing known as data

Table 1: Interactor Class Hierarchy							
class	*lines of code*	*number of attributes of each type*					
		new	*recursive*	*redefined*	*virtual new*	*virtual recursive*	*virtual redefined*
Interactor	908	79	0	0	14	0	0
Scene	195	21	59	0	8	14	1
MonoScene	98	1	73	0	4	16	4
Dialog	84	3	74	0	1	24	0

flow testing[6, 11, 15] to demonstrate the feasibility of our technique. The underlying premise of all of the data flow testing criteria is that confidence in the correctness of a variable assignment at a point in a program depends on whether some test data has caused execution of a path from the assignment (i.e., *definition*) to points where the variable's value is used (i.e., *use*). Definition-use pairs are determined by considering the reachable uses of each definition. Test data adequacy criteria are used to select particular definition-use pairs or subpaths that are identified as the *testing requirements* for a program. Test cases are generated that satisfy the testing requirements when used in a program's execution. One criterion, 'all-uses'[6], requires that each definition of a variable be tested on some path to each of its uses. The 'all-uses' criterion has been shown to be effective in uncovering errors [5] and feasible since relatively few test cases typically are required for its satisfaction[19].

Data flow testing is also used to validate the interfaces between procedures[7, 8]. When validating the interface, the focus of the testing is the definitions and uses of variables that extend across procedure boundaries and includes global variables and reference parameters. We use data flow testing to validate the class member functions individually and to test the interface among the member functions. Stubs and drivers are used to represent any incomplete implementations. We are incorporating our testing technique into the Free Software Foundation, Inc's C++ compiler© (g++) and are using it for our experimentation. We have modified the data flow analysis performed by the g++ compiler to gather the definition-use pairs for the testing and we can currently test individual member functions. Each member function is compiled with the modified g++ compiler and stubs are used to return appropriate values at runtime.

5. Experimentation

We are using a variety of existing C++ class hierarchies for our experiments to determine the savings in testing gained using our technique. We are considering the class hierarchies in InterViews 2.6 [14], which is a library of graphics interface classes. One representative class hierarchy in InterViews is base class *Interactor*, and its subclasses, *Scene*, *MonoScene* and *Dialog*, where *Scene* is a subclass of *Interactor*, *MonoScene* is a subclass of *Scene*, and *Dialog* is a subclass of *MonoScene*. Table 1 gives statistics about these classes.

We used our algorithm to determine which of the methods in each class required retesting. The results of that analysis are shown in Tables 2 and 3. The only comparison possible was with a technique that retests all methods. The results show that for this particular path, in one hierarchy, a significant amount of effort would be saved with our technique. This is a reasonable result for a well-designed hierarchy with a large amount of functionality defined at the top level and modifications and additions made in the lower levels.

This analysis did not consider another potential benefit of the technique; the benefits derived from reuse of the parent tests suites. Many of the methods that must be retested will reuse the specification-based test cases that were developed for their parent class. Reusing test suites from the parent class results in substantial additional savings of time for the testing process.

6. Conclusion

We have presented an incremental technique to validate classes that exploits the hierarchical structure of groups of classes related by inheritance. Our language model is a generalization of the C++ [17] language. Base classes are initially tested using both specification-based and program-based test cases, and a history of the testing information is saved. A subclass is then tested by incrementally updating the history of the parent class to reflect the differences from the parent. Only new attributes or those inherited, affected attributes and their interactions

Table 2: Number of member functions to be tested (specification/program-based)			
class	*retest all*	*our technique*	*our method/retest all*
Interactor	93	93	100%
Scene	96	30	31%
MonoScene	99	9	9%
Dialog	103	4	4%

Table 3: Number of member functions to be tested (interaction/interface)			
class	*retest all*	*our technique*	*our method/retest all*
Interactor	93	93	100%
Scene	96	36	38%
MonoScene	99	9	9%
Dialog	103	6	6%

are tested. The benefit of this technique is that it provides a savings both in the time to analyze the class to determine what must be tested and in the time to execute test cases. We are initially incorporating data flow testing into our hierarchical testing system for both individual member functions and interacting member functions to provide base class testing and subclass testing. Later, we will include a specification-based testing technique in the testing system.

References

1. B. Beizer, in *Software Testing Techniques*, Van Nostrand Reinhold Company, Inc., New York, 1990.
2. T. J. Cheatham and L. Mellinger, "Testing object-oriented software systems," *Proceedings of the 1990 Computer Science Conference*, pp. 161-165, 1990.
3. S. P. Fielder, "Object-oriented unit testing," *Hewlett-Packard Journal*, pp. 69-74, April 1989.
4. P. Frankl, "A framework for testing object-oriented programs," *Technical Report, Department of Electrical Engineering and Computer Science*, Polytechnic University, New York, 1989.
5. P. Frankl and S. Weiss, "Is data flow testing more effective than branch testing? An emperical study," *Proceedings of Quality Week 1991*, May 1991.
6. P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering*, vol. SE-14, no. 10, pp. 1483-1498, October 1988.
7. M. J. Harrold and M. L. Soffa, "Interprocedural data flow testing," *Proceedings of the Third Testing, Analysis, and Verification Symposium (TAV3 - SIGSOFT89)*, pp. 158-167, Key West, FL, December 1989.
8. M. J. Harrold and M. L. Soffa, "Selecting Data for Integration Testing," *IEEE Software, special issue on testing and debugging*, March 1991.
9. W. E. Howden, in *Software Engineering and Technology: Functional Program Testing and Analysis*, McGraw-Hill, New York, 1987.
10. M. Killian, "Trellis: Turning designs into programs," *CACM*, vol. 33, no. 9, pp. 65-67, September 1990.
11. B. Korel and J. Laski, "A tool for data flow oriented program testing," *ACM Softfair Proceedings*, pp. 35-37, December 1985.
12. T. Korson and J. D. McGregor, "Understanding object-oriented: A unifying paradigm," *Communications of the ACM*, vol. 33, no. 9, pp. 40-60, September 1990.
13. U. Linnenkugel and M. Mullerburg, "Test data selection criteria for integration testing," *Proceedings of the 1990 Conference on Systems Integration*, pp. 45-58, April 1990.
14. M. A. Linton and P. R. Calder, "The design and implementation of InterViews," *Proceedings of USNIX C++ Workship*, pp. 256-267, 1987.
15. S. C. Ntafos, "An evaluation of required element testing strategies," *Proceedings of 7th International Conference on Software Engineering*, pp. 250-256, March 1984.
16. D. E. Perry and G. E. Kaiser, "Adequate testing and object-oriented programming," *Journal of Object-*

Oriented Programming, vol. 2, pp. 13-19, January/February 1990.

17. B. Stroustrup, in *The C++ Programming Language*, Addison-Wesley Publishing Company, Massachusetts, 1986.
18. P. Wegner and S. B. Zdonik, "Inheritance as an incremental modification mechanism or what like is and isn't like," *Proceedings of ECOOP'88*, pp. 55-77, Springer-Verlag, 1988.
19. E. J. Weyuker, "The cost of data flow testing: An empirical study," *IEEE Transactions on Software Engineering*, vol. SE-16, no. 2, pp. 121-128, February 1990 .
20. E. J. Weyuker, "Axiomatizing software test data adequacy," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 12, pp. 1128-1138, December 1986.