Identifying Implementation-Based Testing Techniques for Classes

Peter Clarke and Brian Malloy [a]
Clemson University

Abstract

We present an algorithm that automates the process of identifying implementation-based testing techniques that are suitable for testing a given class. The algorithm accepts a summary of the class under test and a set representing testing techniques available to the developer engaged in performing the test. The summary of the class is based on our taxonomy *that maps the characteristics of a class in an object-oriented system into our taxonomy, each entry consisting of a* nomenclature *and* feature properties. *Each element in the set of testing techniques supplied to the algorithm uses our taxonomy to summarize the class characteristics favored by that technique. Using the nomenclature and feature properties of the class under test, together with a set of available testing techniques, the algorithm identifies a subset of techniques that are appropriate for the class under test.*

Keywords Testing, Object-Oriented programming, class-based testing, white-box testing, software engineering.

1. Introduction

As software developers shift their priorities to the construction of complex, large scale systems that are easy to extend and modify, traditional approaches and methodologies fall short. Object orientation makes it possible to model systems that are close to their real world analogues. The goal of object-oriented design is to accurately identify the principle roles in a process, assign responsibilities to these roles and encapsulate them in an object. The object-oriented approach facilitates the extension and modification of these objects. There is an ever-expanding pool of applications and tools that exploit the object-oriented paradigm in their construction.

In the face of the burgeoning popularity of the object-oriented approach, there is also a demand for robust, correct functioning of the developed application. The plethora of new approaches to testing the class structure is evidence of this increased demand[**?**, **?**, **?**, **?**, **?**]. A remarkable fact about testing strategies is that no one strategy has emerged as the accepted approach[**?**]. Thus, the developer is faced with an ever-expanding choice of testing approaches and strategies from which to choose. The difficulty of the choice is further compounded by the fact that some testing strategies are appropriate for some kinds of classes but inappropriate for other kinds of classes.

In this paper, we present an *identification algorithm* that automates the process of identifying the testing techniques that are suitable for testing a given class[1]. The algorithm accepts a summary of the class under test (CUT) and a set of testing techniques available to the developer engaged in performing the test. The summary of the class is based on our *taxonomy* that maps the characteristics of a class in an object-oriented system into our taxonomy, each entry consisting of a *nomenclature* and *feature properties*[**?**]. Each element in the set of testing techniques supplied to the algorithm uses our taxonomy to summarize the class characteristics favored by that technique. Using the nomenclature and feature properties of the class under test, together with a set of available testing techniques, the algorithm identifies a subset of techniques that are appropriate for the class under test.

In the next section, we overview terminology and concepts about object technology, review the important strategies and techniques for testing classes, and overview our taxonomy for cataloging classes. In Section **??**, we overview five implementation-based testing techniques, provide a brief analysis of each technique then identify the category of classes that can be tested by that technique. In Section **??** we present our algorithm that identifies the testing strategy that is appropriate for the CUT and, in Section **??**, we present an example that summarizes and demonstrates our approach. In Section **??** we draw conclusions and discuss our ongoing work.

[a] Computer Science Department
Clemson, SC 29634, U.S.A.
{peterc,malloy}@cs.clemson.edu

[1] It is entirely possible that none of the available strategies is appropriate for the class under test. Our ongoing work focuses on investigating this issue

2. Background

In the next section, we overview terminology and concepts about object technology. We then review the important concepts about testing classes and overview a taxonomy that describes the features and concepts of classes as they appear in the object-oriented approach to software development. We conclude this section with a summarizing example.

2.1. Class Structure

The major entity in Object Oriented (OO) software development is the class. Meyer defines a *class* as a static entity that represents an abstract data type with a partial or total implementation[**?**]. Since the class is the corner stone of OO software it must incorporate the characteristics of the OO paradigm. These characteristics include method of computation, information hiding, typing mechanism, inheritance, polymorphism, dynamic binding, and deferred features and classes. We use Meyer's terminology to describe the characteristics of the OO paradigm[**?**].

A class is considered to be a type since it supplies a static description of certain dynamic entities known as *objects*. The creation of an object during program execution is known as *object instantiation*. The static description supplied by the class includes a specification of the *features* that each object will contain. These features fall into two categories, (1) *attributes*, which are allocated memory when an object is instantiated, and (2) *routines*, which define a certain method of computation applicable to all instances of a class. Attributes are referred to as *data items* in C++, *instance variables* in other OO languages; routines are referred to as *member functions* in C++, *methods* in other OO languages. An object that invokes a feature of an instance of a class *C* is said to be a *client* of *C*.

Information hiding provides a class with the ability to deny its clients access to certain features. This is part of a more general mechanism that allows selective exporting of features to clients. In the OO languages C++ and Java, this mechanism is implemented using the access specifiers *private*, *protected*, and *public*.

Inheritance is one of the major concepts of OO software construction and can be used to provide extensibility and reusability. Inheritance is usually represented by a hierarchical structure. The class inheriting the reusable features is known as the *descendent* and the class supplying the features as the *parent*. The descendent class has the ability to add new features if so desired. The term *single inheritance* is used when a descendent has only one parent and *multiple inheritance* if a descendant has more than one parent.

OO constructs that derive from the use of inheritance are polymorphism, dynamic binding, and deferred features and classes. Meyer[**?**] describes *polymorphism* as the ability of a feature in a class in the inheritance hierarchy to have many forms; that is, the form of the feature invoked is dependent on the type of the object instantiated. It should be noted that polymorphism, if used, is statically defined in the classes contained in the inheritance structure. The binding of these polymorphic features at runtime is known as *dynamic binding*. *Deferred features and classes* provide a mechanism that allows the delayed implementation of features in the inheritance hierarchy. These deferred features must be implemented in one of the descendents (direct or indirect) of the class containing the deferred feature. If a class contains a deferred feature no objects of that class can be instantiated; such a class is referred to as an *abstract class*.

In this paper we further refine the features that are contained in descendant classes. Harrold et al.[**?**] identified six types of features that a descendant class in the C++ language may have. These include a *new feature*, *recursive feature*, *redefined feature*, *virtual-new feature*, *virtual-recursive feature* and *virtual-redefined feature*. The term *new* describes a feature not found in the parent class, *recursive* features are inherited unchanged in the descendent class, and *redefined* features are in the descendant class having the same declaration as the parent but a different implementation. The term *virtual* is pre-pended to each of the above to introduce the notion of polymorphism.

2.2. Class-based Testing

There are several definitions for class-based testing presented in the current literature by researchers and practitioners in the field of testing[**?**, **?**]. Our definition is based on the IEEE/ANSI definition for software testing[**?**]. We define *class-based testing* as the process of operating a class under specified conditions, observing or recording the results, and making an evaluation of some aspect of the class. The aspects of the class to be evaluated determine how successful the tests are. We refer to these aspects as the *adequacy criteria*. There is a spectrum of such criteria for class-based testing, with criteria based on the implementation of the class at one end of the spectrum and criteria based on the specification at the other, with hybrid versions at various locations in the middle.

In this paper we refer to test adequacy criteria that focus on the implementation of the class as *implementation-based* and criteria that focus on the specification of the class as *specification-based*. To adequately test a class some form of the specification and implementation are used during testing. The specification is required to ensure the correctness of the test case executed, which is known as the *oracle problem*, and the implementation is used to execute the test case.

Implementation-based testing[**?**], also known as *white-box testing*, *program-based testing*, or *structural testing*[**?**], traditionally focuses on two main types of coverage criteria: *control-flow* and *data-flow*. These criteria are based on control-flow analysis and data-flow analysis respectively, performed by a compiler during code optimization[**?**]. Although control-flow and data-flow coverage criteria were developed for pre-OO software they still play an important role in class-based testing. Typically control-flow and data-flow analyze the properties of the class under test (CUT), which is represented by some form of a class control-flow graph (CCFG)[**?**], and tries to develop test cases that will maximize the coverage of those properties.

Specification-based testing, also known as *functional testing*[**?**], *black-box testing* or *responsibility-based testing*[**?**], provides the tester with the ability to generate test cases based solely on the specification of the class. As mentioned earlier, the specification of any class should also provide enough information to determine the correctness of a test case. The specification of a class can be represented in several ways, ranging from a natural language description to a detailed functional specification.

To adequately test a piece of software, testing researchers and practitioners have suggested that both specification-based and implementation-based testing are needed[**?**, **?**]. We refer to testing strategies that use criteria based on the implementation and specification as *hybrid-based testing*. Hybrid-based testing attempts to provide fuller coverage than might be achieved by performing either specification-based or implementation-based testing.

2.3. Taxonomy

In this section we overview a taxonomy of classes we developed to assist the tester in identifying the implementation-based testing techniques that can be used to test a CUT. A detailed explanation of the taxonomy can be found in[**?**]. The taxonomy classifies an OO class according to the OO characteristics it exhibits, as well as the types used by its features (attributes and routines). Each entry of the taxonomy has a *nomenclature* and a set of *features properties*.

2.3.1 Nomenclature. The nomenclature is composed of two components: an *Object Oriented modifier* (or simply *OO modifier*) and the *class associated types*. The OO modifier component is used to identify the OO characteristics the class exhibit.

The important OO characteristics are:

- *Inheritance-free* - indicates that the class is not part of an inheritance hierarchy.
- *Parent* - identifies a class that is the root class of an inheritance hierarchy.
- *Derived* - identifies a class that is a descendant of a single parent.
- *Derived parent* - identifies a class that is a descendant as well as a parent.
- *Abstract* - identifies a class that contains deferred features.

The above is not a complete list of the OO characteristics that can be exhibited by a class, elided are characteristics such as genericity and multiple inheritance. Note that the OO modifier component of the nomenclature is a combination of OO characteristics.

The *class associated type* component of the nomenclature reflects the categories of types used in the class. Note here we use a relaxed definition of the term *type*. These types can include basic types, as well as user defined types; we give a detailed explanation of the types in [**?**].

The types used in the class taxonomy are:

- *Type 0* - no associated types used.
- *Type I* - scalar basic types.
- *Type II* - non-scalar basic types, these include access types and arrays for basic types.
- *Type III* - user-defined types i.e. classes.
- *Type IV* - access types for user-defined types, these include arrays for user defined types.
- *Type V* - class libraries, e.g., STL[**?**].

2.3.2 Feature Properties. The *feature properties* identify the type and scope information for each attribute, the types of the *routine locals* (parameters and local variable), and a classification of the inheritance features if the class is involved in an inheritance relationship. The access specifiers *private*, *protected* and *public* are used to specify the visibility of an attribute. We refer to the classification of the features for a class in an inheritance hierarchy, as *feature classification*. The feature classification is based on the refinement of features overviewed in the last paragraph of section 2.1. The features in a derived class are classified as *new, recursive, redefined, virtual-new, virtual-recursive or virtual- redefined*[**?**].

2.3.3 Example. Figure **??** illustrates the C++ code for a class Point and Figure **??** catalogs the class Point using our taxonomy. Class Point represents a point in the Cartesian plane. The cataloged class shown in Figure **??** is constructed using the taxonomy overviewed in sections 2.3.1 and 2.3.2. The nomenclature for class Point, shown in Figure **??**, is *Inheritance-free Types I, III* for the following reasons. Point is not part of an inheritance hierarchy and all of

```
1 class Point{
2   // Represents a point in the
3   // Cartesian plane
4 private:
5  int x, y;
6 public:
7   Point(): x(0), y(0){}
8   Point(int inX, int inY):
9         x(inX), y(inY){}
10  void print() const {
11    cout << "("<< x << ", "
12         << y << ")"<< endl;
13  }
14  double distance(Point p1) const {
15   return sqrt(pow(x - p1.x, 2)
16              + pow(y - p1.y, 2));
17  }
18 };
```

Figure 1. C++ code for class Point**.**

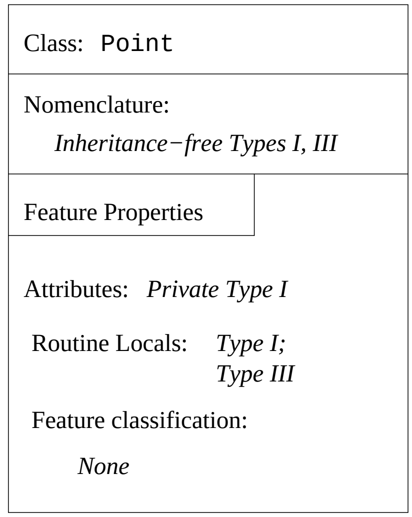

Figure 2. Application of taxonomy cataloging class Point**.**

its attributes, parameters and local variables are either basic scalar types (*Type I*) or user defined types (*Type III*). The feature properties identify the attributes defined on line 5 of Figure **??** as private to the class and of basic scalar types. The constructor for Point on line 8 has parameters that are basic scalar types (int) and the routine *distance* on line 14, has a parameter that is a user-defined type (Point). Therefore the routine locals are classified as *Type I* and *Type III*. Since Point is not a derived class there are no feature classifications.

3. Taxonomy and Implementation-based Testing

There are several implementation-based testing techniques that target OO software, however there is no one technique that seems to adequately test all the characteristics of a class. In this section we perform a brief analysis on some of the current implementation-based techniques resulting in the identification of techniques that are most suited to testing a particular category of class. We use the results of this analysis later in the paper to develop an algorithm to identify current implementation-based techniques using our taxonomy as the basis.

3.1. Overview of Testing Techniques

In this section we overview five testing techniques, four implementation-based[**?**, **?**, **?**, **?**] and one hybrid-based technique[**?**]. We include the hybrid-based technique since its structure allows us to easily extract the implementation-based component. The four implementation-based techniques are divided into two groups: techniques that generate *test tuples* [**?**, **?**] and those that generate *message sequences* [**?**, **?**]. These test tuples and message sequences are then used to create appropriate test cases.

Harrold and Rothermel present a data flow technique for classes based on the procedural approach[**?**]. This testing technique performs data-flow analysis on a class using a class control flow graph (CCFG) to produce test tuples of the form *(d, u)*, where *d* represents the line number of the variable definition and *u* the line number of its use. To handle the complexity of testing a class as compared to a simple procedure, Harrold et al. suggest four levels of testing *intra-method*, *inter-method*, *intra-class* and *inter-class* testing.

Souter and Pollock propose a testing strategy known as OMEN (Object Manipulations and Escape Information) that uses data-flow analysis based on object manipulations to generate test tuples[**?**]. These test tuples are of the form *object-name(store, load, object creation site)* and are based on the analyzed code. This technique builds an Annotated Points-to-Escape (APE) graph that identifies the references to a particular object in the analyzed section of code, the objects that can be read at a given point, and the possible reaching writes to an object prior to that point in the code.

Buy et al. propose an automated testing technique for classes that uses data-flow analysis[**?**], symbolic execution and automatic deduction to generate message sequences, seeking to reveal failures dependent on the current state of the object[**?**]. Symbolic execution identifies conditions related to path executions and variable definitions for each method in the class. The automatic deduction component generates a method sequence in reverse order by applying a set of backward-chained deductions.

Kung et al. use symbolic execution to generate an *object* *state test model* that is used to construct a *test tree*[**?**]. Test sequences are then generated from the test tree. The object state test model is represented as a *hierarchical, concurrent object state diagram* (OSD), which identifies the possible states an object can enter during execution[**?**]. Symbolic execution is used to identify the states and transitions for an OSD. After the construction of the OSD, the test tree is generated so that the nodes represent the states and the edges the transition between states in the OSD, with the initial state as the root.

The hybrid-based technique proposed by Harrold et al. is an incremental approach to testing OO software dependent on the class structure[**?**]. This technique focuses on the inheritance hierarchy of classes. Initially a test suite is created for the root class in the hierarchy using specification-based and/or implementation-based techniques, known as the *test history*. The features of each derived class from the root class are then analyzed and classified as one of the six types; we discussed these types in the last paragraph of section 2.1. Depending on the type of feature in the derived class, new test cases are generated, the test cases from the test history for that inherited feature are totally or partially rerun, or the feature is considered to be properly tested in the parent and none of the tests are rerun.

3.2. Mapping of Classes to Testing Techniques

In order to map the category of class most suited to each of the five testing techniques overviewed in section **??**, we first briefly analyze each technique, then identify the category of class using our taxonomy that can be tested by the technique.

The testing techniques presented in [**?**, **?**, **?**] use data-flow-based adequacy criteria[**?**] to either identify the test tuples[**?**, **?**] or the required message sequences that exercise the test tuples[**?**]. These techniques use the 'all-uses' criterion that require each feasible def-use pair in the class to be tested. Rapps and Weyuker propose a partial order ranking that suggests the 'all-uses' criterion is second only to the infeasible criteria of 'all-du-paths' and 'all-paths'[**?**]. The technique in [**?**] is based on the state of the object and detects errors due to interactions between routines through object state behavior. This technique[**?**, **?**] builds a test tree that covers a subset of the paths in the OSD (object state diagram). In some cases several trees are used to represent an OSD. The hybrid-based technique in [**?**] does not provide any adequacy based criteria since it only suggests which test cases from the test history should be re-run.

In the following list we present a brief analysis for each testing technique and identify the most suitable category of class for that technique. This analysis is based on the information found in the respective references describing each technique.

- Data-flow Testing[**?**]:
  Identifies test tuples for scalar basic types (*Type I*).
  Treats individual elements of aggregate objects, such as arrays, as a single object.
  Misses some test tuples resulting from specific aliases (*Type II, IV*).
  Does not consider classes that inherit attributes from their parents.
  Does not generate inter-class test tuples e.g., does not address public attributes.
  MOST SUITABLE CLASS: *Type I* (*inheritance-free* or any type of *parent*).

- OMEN[**?**]:
  Handles the problem of aliasing for references to user defined types (*Type IV*).
  Treats individual elements of aggregate objects, such as arrays, as a single object.
  Does not consider scalar basic types (*Type I*), non-scalar basic types (*Type II*), or user-defined types (*Type III*).
  Does not consider classes that inherit attributes from their parents.
  MOST SUITABLE CLASS: *Type IV (inheritance-free* or any type of *parent*).

- Automated Testing of Classes[**?**]:
  Focuses mainly on inter-method def-use analysis.
  Only considers scalar basic types (*Type I*).
  Does not handle the problem of aliasing and parameter passing.
  Does not consider public attributes.
  Uses symbolic execution that can only be applied to routines with simple control flow.
  MOST SUITABLE CLASS: *Type I (inheritance-free* or any type of *parent*). There maybe be further restrictions due to complex control structures.

- Object State Testing[**?**]:
  Reasons similar to those mention for the 'Automated Testing of Classes' [**?**].
  MOST SUITABLE CLASS: *Type I (inheritance-free* or any type of *parent*). This technique might even be more restrictive than the technique referenced in[**?**], because of the computational complexity to generate the OSD for a class.

- Incremental Testing of Object Oriented Classes[**?**]:
  Depends on other testing techniques to generate test cases.
  MOST SUITABLE CLASS: *Derived*. Any descendant class in an inheritance hierarchy.

In [**?**] we use our taxonomy to catalog the classes in the major example as presented in the reference for each testing technique.

4. Identification of Implementation-based Testing Techniques

In this section we present our algorithm that automates the process of identifying the implementation-based testing technique(s) suitable for testing a given class. We refer to this algorithm as the *identification algorithm*. The algorithm accepts a summary of the contents of the class under test (CUT) and a set representing the available implementation-based testing techniques, then identifies the testing technique(s) that can be used to test the CUT. In the next section we describe the relationship between our taxonomy[**?**] and the process of selecting a suitable testing technique. The identification algorithm is presented in Section **??**.

4.1. Technique Identification Based on Taxonomy of Classes

Active research continues in the area of implementation-based testing of classes. Therefore, any algorithm that maps testing techniques to classes must be reusable with new techniques as they are developed. In addition, a tester must have the ability to choose an appropriate testing technique from the available set of techniques at the time of testing. Thus, we provide an algorithm that is reusable and flexible. The identification algorithm must also provide feedback to the tester in the event that there is no testing technique available to test specific characteristics of the CUT.

In order to provide reusability and flexibility, we permit the tester to create a set of available implementation-based testing techniques, used as input to the identification algorithm. Each entry in this set contains a summary of the class(es) that can be tested by that technique. In Section **??** we informally identified the most suitable category of class(es) for each of the five implementation-based testing techniques overviewed in Section **??**. We represent each class in the summary using the basic ideas of our taxonomy overviewed in Section 2.3. That is, each class in the summary is cataloged using nomenclature and feature properties[**?**]. Unlike the nomenclature used in reference [**?**] to identify classes, we restrict the nomenclature in each class cataloged in the summary for a given testing technique to one class associated type. For example, if the nomenclature of the most suitable classes for testing technique T was *inheritance-free Types I, II*, then the nomenclature for entry T in the list of test techniques would be *inheritance-free Type I* and *inheritance-free Type II*. In the event that a testing technique can be applied to all class associated types i.e., *Types I, II, III, IV* and *V*, we leave the class associated type component of the nomenclature empty. These restrictions allow the testing techniques to be more accurately described in the identification algorithm.

```
Identify(cutNomen, testList)
  currentTech ← ∅
  for each cutType ∈ cutNomen.type do
    tempTech ← ∅
    for each testEntry ∈ testList do
      for each suitableClass ∈ testEntry.testClass do
        if ((suitableClass.type = cutType or
              suitableClass.type is empty) and
          suitableClass.modifier = cutNomen.modifier)
          then
            tempTech ← tempTech ∪ testEntry
        endif
      endfor
    endfor
    if tempTech = ∅ then
      print " No Technique for "
              cutNomen.modifier + cutType
    else
      currentTech ← currentTech ∪ tempTech
    endif
  endfor
  return currentTech
endIdentify
```

Figure 3. Identification algorithm. This figure is a summary of our algorithm that takes as input a CUT, cataloged using our taxonomy, and a set of testing techniques supplied by the tester. The algorithm then searches the list of testing techniques and provides all the techniques suitable for testing the CUT.

As new testing techniques are developed the tester can update the summaries in the set of testing techniques appropriately. The tester also has the flexibility to restrict the set of testing techniques to those techniques that are truly automated or to the techniques that are practically available. In Section **??** we present an example that shows a list of testing techniques used as input to the identification algorithm.

The identification algorithm also accepts, as input, a summary of the CUT cataloged using our taxonomy; an example of such a summary was presented in Section 2.3. At present we are developing a tool to catalog C++ classes, using a C++ parser[**?**, **?**, **?**]. Using the cataloged CUT and the set of testing techniques, the algorithm identifies the technique(s) that can be use to test the CUT.

4.2. Identification Algorithm

In this section we describe the identification algorithm shown in Figure **??**. This algorithm identifies the implementation-based techniques that can be used to test a given class. The algorithm also provides feedback to the tester in the event there is no testing technique to adequately test specific characteristics of a class. For the purpose of simplicity we restricted the selection of the testing techniques to the information contained in the nomenclature of the CUT and the nomenclature of the class information stored in the set of testing techniques.

The input to the algorithm in Figure **??**, described in Section **??**, consists of the CUT cataloged using our taxonomy (cutNomen), and a set of testing techniques supplied by the user (testList). The main data structure is a record consisting of the fields modifier and type. We refer to this record structure as Nomenclature. The field type is a list of the class associated types, and modifier a string for the OO modifier. The parameter cutNomen is of type Nomenclature. testList is a list of records each of type TestTech. TestTech is a record structure consisting of the fields name, a string containing the name of the testing technique, and testClass containing a list of entries of type Nomenclature. Note that each entry of the list testClass contains only one value in the field type to facilitate the restriction of one class associated type per class summary for each testing technique. The reason for this restriction is given in Section **??**.

The algorithm Identify in Figure **??** starts by initializing the set variable currentTech in line 2 to the empty set. The variable currentTech will eventually contain the testing techniques to be returned by the routine Identify. The loop from lines 3 to 21 uses the variable cutType to iterate through each class associated type in the nomenclature of the CUT. This loop is required since the nomenclature of a class can have at most five class associated types. The temporary set variable tempTech, initialized in line 4, is used to store all the testing techniques that can be used to test the characteristics of the CUT represented by a combination of the variables cutNomen.modifier and cutType. Variable cutType, initialized in line 3, contains one of the class associated types of the CUT and cutNomen.modifier the OO modifier of the CUT.

The loop from lines 5 to 14 uses the variable testEntry to iterate through the list of testing techniques testList. Since each testEntry contains a list of class summaries, the loop from lines 6 to 13 is required to iterate through this list. The variable suitableClass initialized in line 6, is used as a temporary reference to each of the class summaries in the list stored in testEntry.testClass. The record field suitableClass.type in line 7 contains the class associated type and suitableClass.modifier in line 9 the OO modifier. Lines 7 to 9 of the if statement checks for a match between the CUT and the class summary of the testing technique referenced by suitableClass. If there is a match, then a copy of that entry for the testing technique referenced by testEntry is added to temporary set tempTech in line 11. Before the next class associated type of the CUT is considered, on line 3, a checked is made to see if the set tempTech is empty. If tempTech is empty this implies there is no testing technique to test the characteristics of the CUT described by cutNomen.modifier and cutType and an appropriate message is printed, lines 16 and 17. Otherwise, the union of the sets currentTech and testEntry is placed in currentTech as shown on line 19. When there are no more class associated types of the CUT to consider the set currentTech containing the applicable testing techniques are returned on line 22.

5. Application of Identification Algorithm

In this section we present an application of the identification algorithm described in Section **??**. The application uses the C++ example identified in Section 2.3.3 as the CUT and two of the implementation-based techniques summarized in Section **??** as the test list. The next section describes the structure of the input data to the algorithm Identify and Section **??** traces the application of the algorithm on the input.

5.1. Representation of the CUT and Test List

The current version of the algorithm Identify takes as input the nomenclature component of the CUT and a list representing the summaries of the testing techniques available to the tester. For simplicity, we use two of the implementation-based techniques described in Section **??**, these are *Data-flow Testing*[**?**], and *Incremental Testing of Object Oriented Classes*[**?**].

Figure **??** (a) shows the record structure of the CUT, class Point, which consists of the fields name, modifier, and type. The fields modifier and type represent the nomenclature component of the taxonomy as shown in Figure **??**. The field type is a list consisting of two entries. In Figure **??** we represent a list as a box structure consisting of several components. Figure **??** parts (b) and (c) show the record structures of the summaries for the two implementation-based testing techniques provided by the tester. Each of these record structures consists of the fields technique, the name of the technique, and testClass, the list of the classes that the technique can adequately test as determined by the tester. Note that in Figure **??** part (c) each field type of the list has a value of *empty*, indicating that the testing technique can be applied to all class associated types (see Section **??** for an explanation).

5.2. Applying the Algorithm

Identify accepts the record structure cutNomen Figure **??** (a) and testList, a list consisting of record structures

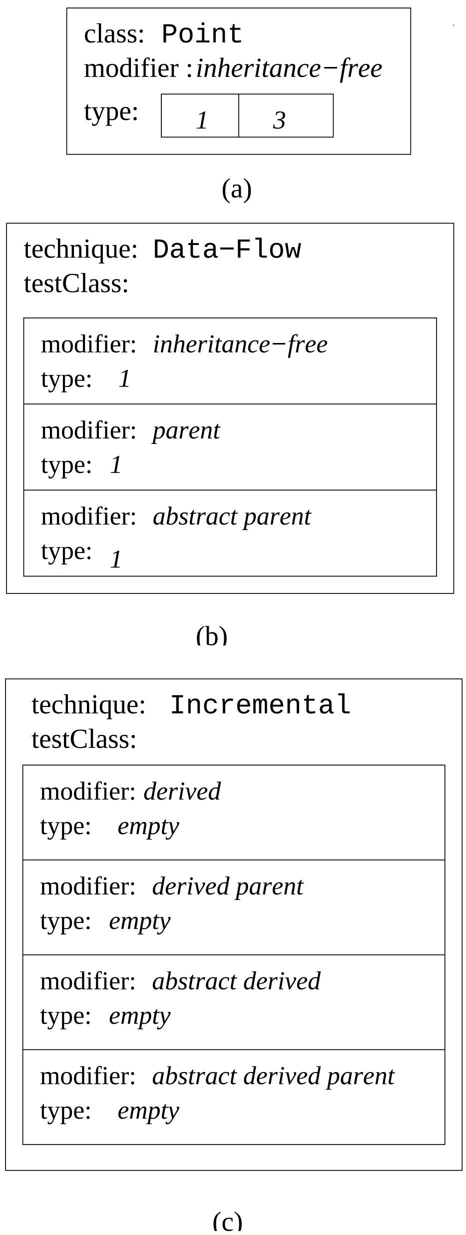

Figure 4. Representation of data structures. (a) CUT class Point. (b) Summary for Data-flow testing technique. (c) Summary for Incremental testing technique. A name preceding a colon is a field of the record structure e.g., modifier. If a field is followed by a box structure this represents a list e.g., type in (a) and testClass in (b) and (c).

shown in Figure **??** parts (b) and (c). In line 3 of Identify, Figure **??**, the loop variable cutType sequences through the contents of type field of Figure **??** (a) starting with a value of *1*. Variable testEntry in line 5 of Identify sequences through the list testList, starting with the technique *Data-flow*, Figure **??** part (b). The loop variable suitableClass, then sequences through the list in the field testClass of *Data-flow* starting with the record whose modifier value is *inheritance-free* and type value is *1*. The boolean expression in lines 7 to 9 of Identify evaluates to true since the content of the variables suitableClass.type and cutType are equivalent, as well as the variables suitableClass.modifier and cutNomen.modifier. Line 11 is then executed and a copy of the record containing the *Data-flow* technique summary is placed in tempTech. The variable suitableClass continues to iterate through the list testClass in the *Data-flow* record but the boolean expression on lines 7 through 9 evaluates to false in each case.

The algorithm continues by assigning the loop variable testEntry, on line 5 of Identify, the next record in the list of testing summaries, in this case the summary for the *Incremental* testing technique as shown in Figure **??** part (c). As the list in the testClass field of the *Incremental* record is traversed, the boolean expression in lines 7 through 9 never evaluates to true. Line 15 is then executed and the boolean expression evaluates to false since the set variable tempTech is not empty. This outcome results in the contents of tempTech i.e., the record for the technique *Data-flow*, being copied into the variable currentTech. Execution of the algorithm continues on line 3 and the loop variable cutType is assigned the next value in the type field of the CUT, which happens to be *3*. The variable tempTech is assigned to the empty set and the loops on lines 5 through 14 are executed. Execution of these loops traverse each record in the testClass list of each testing technique. Since the boolean expression in lines 7 through 9 never evaluates to true, the variable tempTech exits the loops on line 14 containing the empty set. The boolean expression in line 15 of the algorithm evaluates to true and a message is printed informing the tester that there are no techniques available to test the features *inheritance-free type 3* of the CUT. The algorithm then returns the contents of the set variable *currentTech*, the record containing the summary for the testing technique *Data-flow*, and terminates.

6. Concluding Remarks

In this paper we presented an approach that identifies implementation-based techniques that can be used to test classes. This approach includes the classification of the class under test (CUT), an approach to map categories of classes to implementation-based testing techniques, and an algorithm that identifies the testing techniques most suited to testing a given class.

The CUT is first cataloged using our taxonomy as outlined in reference [**?**]. This process analyses the CUT assigning a nomenclature and summarizing the properties of its features (feature properties). The nomenclature consists of an OO modifier that identifies the object oriented characteristics of the CUT, and a class associated type that specifies the types of its attributes, parameters and local variables. The feature properties provide additional information about the attributes, parameters and local variables. This information includes access specifiers for the attributes and a classification of features if the CUT is a derived class[**?**].

We map categories of classes to each of the implementation-based techniques by reviewing the literature and identifying the properties of a class that are not tested by that technique. Using these properties, we used the naming system provided by our taxonomy to map the categories of classes to the respective implementation-based testing technique.

The main focus of this paper is the algorithm Identify that allows the selection of one or more implementation-based testing techniques most suited to test the features of the CUT. The tester provides the algorithm Identify with a list containing a summary of the available testing techniques and the CUT cataloged using our taxonomy. The algorithm returns a set of testing techniques that can be used to adequately test the CUT based on the input supplied by the tester. If no technique exists to test a given characteristic of the CUT the algorithm provides feedback to the tester.

At present, algorithm Identify bases its decision on which testing techniques are applicable solely on the nomenclature of the cataloged CUT, ignoring the additional information provided by feature properties. To improve the accuracy of identifying the most suitable testing technique the summaries of classes for each testing technique can be expanded to include feature properties. For example, knowing whether an attribute that is *Type 1* is private to the CUT is important in determining the applicability of a technique.

We are developing an implementation of our algorithm, as well as a tool to catalog classes based on our taxonomy[**?**] using a C++ parser [**?**, **?**, **?**]. We are also interested in using the cataloging tool to identify the percentages of different types of classes found in real world applications. This result will provide feedback about the practicality of the current implementation-based techniques for testing real world applications.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *"Compilers: Principles, Techniques, and Tools"*. Addison-Wesley, 1986.

[2] R. V. Binder. *"Testing Object-Oriented Systems: Models, Patterns, and Tools"*. Addison-Wesley, 2000.

[3] U. Buy, A. Orso, and M. Pezze. "Automated Testing of Classes". In *Proc. of ISSTA.*, Portland, OR, August 2000.

[4] P. J. Clarke and B. A. Malloy. "A Taxonomy of Classes for Implementation-Based Testing". Technical report, Clemson University, April 2001.

[5] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. "Incremental Testing of Object-Oriented Class Structures". In *Proc. of ICSE.*, Melbourne, Australia, pages 68–80, May 1992.

[6] M. J. Harrold and G. Rothermel. "Peforming Data Flow Testing on Classes". In *Proc. of ACM SIGSOFT Symp. FSE*, New Orleans, LA, pages 154–163, Dec. 1994.

[7] IEEE/ANSI Standards Committee. Std 610.12-1990. 1990.

[8] ISO/IEC JTC 1. *"International Standard: Programming Languages - C++"*. Number 14882:1998(E) in ASC X3. ANSI, first edition, September 1998.

[9] D. Kung, Y. Lu, N. Venugopalan, P. Hsia, Y. Toyoshima, C. Chen, and J. Gao. "Object State Testing and Fault Analysis for Reliable Software Systems". In *Proc. of 7th Int'l Symp. Software Reliability Eng.*, White Plains, NY, pages 76–86, Oct. 1996.

[10] D. Kung, N. Suchak, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. "On Object State Testing". In *Proc. of COMPSAC 94*, Los Alamitos, CA, pages 222–227, 1994.

[11] J. D. McGregor and D. A. Sykes. *"A Practical Guide To Testing Object-Oriented Software"*. Addison-Wesley, 2001.

[12] B. Meyer. *"Object-Oriented Software Construction"*. Prentice Hall PTR, 1997.

[13] J. F. Power and B. A. Malloy. "An Approach for Modeling the Name Lookup Problem in the C++ Programming Language". In *Proc. of ACM Symposium on Applied Computing, SAC'2000*, pages 792–796, Como, Italy, March 2000.

[14] J. F. Power and B. A. Malloy. "Metric-Based Analysis of Context-Free Grammars". In *Proc. of the 8th International Workshop on Program Comprehension*, pages 171–178, Limerick, Ireland, June 2000.

[15] J. F. Power and B. A. Malloy. "Symbol table construction and name lookup in ISO C++". In *Technology of Object-Oriented Languages and Systems, TOOLS 2000*, pages 57–68, Sydney, Australia, November 2001.

[16] S. Rapps and E. J. Weyuker. "Selecting Software Test Data Using Data Flow Information". *IEEE Trans. Softw. Eng.*, 11(4):367–375, April 1985.

[17] A. L. Souter and L. L. Pollock. "OMEN: A Strategy for Testing Object-Oriented Software". In *Proc. of ISSTA*, Portland, OR, pages 49–59, August 2000.

[18] H. Zhu, P. A. V. Hall, and J. H. R. May. "Software Unit Testing Coverage and Adequacy". *ACM Computing Surveys*, 29(4):366–427, December 1997.