Mapping Implementation-Based Testing Techniques to Object-Oriented Classes

Peter J. Clarke
School of Computer Science
Florida International University
Miami, FL 33199
U.S.A.
clarkep@cs.fiu.edu

Brian Malloy
Computer Science Department
Clemson University
Clemson, SC 29634
U.S.A.
malloy@cs.clemson.edu

Technical Report: 2004-08

Abstract

In this paper we describe our taxonomy of object-oriented classes that catalogs each class in an application based on the characteristics of that class. We present a mapping algorithm that matches the class under test to a list of implementation-based testing techniques, providing feedback to the tester describing the characteristics of the class that are suitably tested by each testing technique together with the characteristics that are not suitably tested by any of the testing techniques in the list. Our study shows that the number of entities to be analyzed by a tester when selecting implementation-based testing techniques can be reduced by 50% if the class under test is first cataloged using our taxonomy of OO classes.

1 Introduction

The cost of system nonperformance and failure for large scale systems is expensive, with sometimes catastrophic impact [22]. The trend in the development of large scale object-oriented systems has shifted toward testable, robust models, whose focus is on preventing errors. One process that supports the construction of robust software is testing. An advantage of software testing is the relative ease with which some of the testing activities can be performed, such as executing the program using a given set of inputs, or test cases, and then comparing the generated output to the expected output [10].

However, testing an object-oriented system presents some imposing problems. For example, the entire system must be available before the test can begin and this may occur late in the software life cycle. During the test of a complete system there are risks of complex interactions among the errors and of mutual destabilization of the corrected classes or components. Thus, many developers prefer a progressive approach where the first stage consists of testing individual classes and there is an abundance of class-based testing techniques described in the literature [3, 11, 12, 15, 16, 27, 28]. However, the data attributes and routines of a class complicate class-based testing in the presence of references, pointers,

inheritance, polymorphism, and deferred features. Each of the testing techniques described in the literature addresses one or more of these complications but no one technique has emerged as the accepted approach *de rigueur*, possibly because no single testing technique address all of the complications that classes may possess.

In this paper we describe our taxonomy of object-oriented classes that catalogs each class in an application based on the characteristics of that class, including the properties of the data attributes and routines as well as the relationships with other classes. We present a mapping algorithm that matches the class under test to a list of implementation-based testing techniques, providing feedback to the tester describing the characteristics of the class that are *suitably tested* by each testing technique together with the characteristics that are not *suitably tested* by any of the testing techniques in the list. We describe a study where applications are analyzed using our taxonomy tool that catalogs classes written in C++. Our study illustrates that the number of entities to be analyzed by a tester when selecting implementation-based testing techniques can be reduced by 50% if the class under test is first cataloged using our taxonomy of object-oriented classes.

In the next section we provide background and terminology about classes, class-based testing and our class abstraction technique. In Section 3 we describe our taxonomy and in Section 4 we summarize our results obtained using our taxonomy tool. In Section 5 we present our mapping algorithm together with an illustrative example showing the workings of the algorithm. We review the work related to ours in Section 6 and we draw conclusions in Section 7.

2 Background

In this section we introduce terminology and basic concepts of an object-oriented (OO) class, class-based testing, and class abstraction techniques (CATs).

2.1 Class Characteristics

Meyer defines a class as a static entity that represents an abstract data type with a partial or total implementation [21]. The static description supplied by a class includes a specification of the *features* that each object will contain. These features fall into two categories: (1) *attributes*, and (2) *routines*. Attributes are referred to as *data items* and *instance* variables in other OO languages while routines are referred to as *member functions* and *methods*. Throughout this paper we will use the terms attributes and routines.

We define the *characteristics* for a given class C as the properties of the features in C and the relationships C has with other classes in the implementation. The properties of the features in C describe how criteria such as types, accessibility, shared class data, polymorphism, dynamic binding, deferred features, exception handling, and concurrency are represented in the attributes and routines of C [21]. The relationships of C with other classes include associations, dependencies, and generalizations. We define these relationships based on the definitions given by Rumbaugh et al. [24].

Each attribute and routine local (variable or parameter) of a class can be declared as one of many possible *types*. These types include: built in types, user defined types, and types

provided by specialized libraries. Some OO languages also permit the use of *parameterized types*, also referred to as *generics*, that allow the actual type of the attribute or routine local to be known when an object of the class is instantiated. *Accessibility* to class attributes and routines provides a class with the ability to deny its clients access to certain features. In C++ [29] and Java [2] this mechanism is implemented using the access specifiers *private*, *protected* and *public*. There are other language specific mechanisms that allow for the selective exporting of features to client classes, such as *Friends* in C++. Some OO languages provide a mechanism for sharing data with all instances of a class. Class wide data is declared in C++ and Java using the keyword **static**.

Inheritance allows features of the class to be reused in another class and permits the class to be extended to include new features. The relationship between the classes involved in inheritance is known as *generalization* where the class inheriting the features is referred to as the *descendant* and the class supplying the features is referred to as the *parent*. If a descendant has more than one parent *multiple inheritance* is exhibited. *Polymorphism* is the ability of a feature (attribute or routine) of a class to have many forms; that is, the form of the feature invoked is dependent on the type of the object instantiated. As a consequence of polymorphism features can be bound to the object at runtime, referred to a *dynamic binding*. C++ uses the keyword **virtual** to signal the compiler that the member function should be dynamically bound. Harrold et al. identify six types of features that a descendant class in the C++ language may have [11]. These include *new feature*, *recursive feature*, *redefined routine*, *virtual-new routine*, *virtual-recursive routine* and *virtual-redefined routine*.

Deferred features and classes provide a mechanism whereby features of the parent class in the inheritance hierarchy need not be implemented in that class, but are required to be implemented in one of its descendants (direct or indirect descendant). If a parent class has deferred features no objects of this class can be instantiated, such a class is known as an *abstract class*.

Louden describes *exception handling* as the control of error conditions or other unusual events during the execution of a program [18]. When an exception occurs it is said to be *raised* or *signaled*. An *exception handler* is a routine or code sequence that is designed to be executed when an exception is raised. Many OO languages provide some form of *concurrency* that allows routines or code sequences in objects to execute simultaneously. Each concurrent routine or code segment represents a process (or thread) allocated by the operating system for independent execution. Languages that offer concurrency also provide some mechanism to synchronize processes, as well as a way for the processes to communicate [18].

2.2 Implementation-Based Testing of Classes

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. Our definition of *class-based testing* is based on the IEEE/ANSI definition for software testing [13]. The aspects of the class to be evaluated are usually based on some set of criteria. These criteria, used to generate test information, are based on either the specification, implementation, or a combination of both the specification and implementation

(hybrid). This paper focuses on testing techniques that generate test information based on the implementation, we refer to these techniques as *Implementation-Based Testing Techniques* or IBTTs. The following three subsections briefly describe several IBTTS. The final subsection provides a summary of the IBTTS.

2.2.1 Test Tuple Generation

Several IBTTs generate test information from tuples (referred to as *test tuples*) based on some type of coverage criteria. Harrold and Rothermel present a data flow testing technique for classes based on the procedural programming paradigm [12]. The technique described in [12] uses the class control flow graph (CCFG) to represent the classes in a program. Data-flow information computed from the CCFG is used to generate *intra-method, inter-method* and *intra-class* def-use pairs [12]. Sinha and Harrold describe a class of adequacy criteria that is used to test the behavior of exception-handling constructs in Java Programs [27]. The approach described in [27] is similar to that presented in [12], that is, data-flow analysis is performed on an inter-procedural control flow graph (ICFG) that incorporates exception-handling constructs resulting in the identification of test tuples. Souter and Pollock propose a testing technique known as OMEN (Object Manipulations in addition to using Escape Information) that uses data-flow analysis based on object manipulations to generate test tuples [28]. OMEN is based on a compiler optimization strategy for Java programs that creates a points-to-escape graph for a given region of the program.

Alexander and Offutt, present OO coupling criteria that focuses on the effects of inheritance and polymorphism [1]. This criteria uses quasi-interprocedural data flow analysis, that is complete information about data flows between units are not needed. This approach requires data flow information from definitions to call sites, from call sites to uses, and from entry definitions to exit nodes [1]. Koppol et al. describe a testing technique that generates test sequences selected from labeled transition systems (LTS) [15]. An LTS is a type of state machine used to model programs. An LTS can be obtained from the source code of a concurrent program via the program's control flow graph [15]. The common approach to selecting test sequences is from a reachability graph. To overcome the state explosion problem with traditional reachability graphs, Koppol et al. defined a new type of graph for incremental analysis called an *annotated labeled transition system* (ALTS). During incremental analysis test paths can be selected from the intermediate graphs or from the final reduced graph.

2.2.2 Message Sequence Generation

Some IBTTs generate message sequences that are executed by instances of the CUT. These massage sequences are generated based on criteria associated with the implementation of the CUT. Buy et al. propose an automated testing strategy for classes that uses data-flow analysis, symbolic execution, and automatic deduction [3]. This IBTT generates message sequences seeking to reveal failures dependent on the current state of the object. Kung et al. use symbolic execution to generate an *object state test model* that is used to construct a *test tree* [16]. The method 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. A test tree is generated from the OSD and message sequences produced.

2.2.3 Test Case Reuse

Hybrid testing techniques (HTTs) use both the specification and the implementation of the software to develop test cases. Harrold et al. propose a HTT that uses an incremental approach to testing OO software dependent on the inheritance hierarchy component of the class structure [11]. The incremental approach reuses specification-based and implementation-based test sets created for the class at the root of the inheritance hierarchy, to possibly test derived features. The test sets use to test the base class at the root of the inheritance hierarchy are collectively referred to as the *test history*. The features in a derived class are classified as: *new* or *recursive* for both attributes and routines, or *redefined, virtual-new, virtual-recursive,* or *virtual-redefined* for routines [11]. Depending on the type of feature in the derived class: (1) new test cases are generated, (2) the test cases from the test history for that inherited feature are totally or partially reused, or (3) the feature is considered to be properly tested in the parent and none of the tests are reused.

2.2.4 Summary of IBTTs

Table 1 summarizes the IBTTs described in the previous three subsections identifying the characteristics of the CUT that can be *suitably tested* by the IBTT and those characteristics of the CUT *not suitably tested* by that IBTT. Column 1 of Table 1 identifies the main researcher that developed the IBTT, described in the corresponding row, and the name we associated with the respective IBTT. Column 2 identifies the class characteristics that can be *suitably tested* by the respective IBTT listed in Column 1, and Column 3 the class characteristics not *suitably tested* by the respective IBTT listed in Column 1. Note the class characteristics listed in Columns 2 and 3 are deduced from the references cited in Column 1.

Each row in the table, delimited by a hard line, represents the information pertaining to a given IBTT. For example, Row 2 represents the information for the IBTT developed by Buy et al.[3]. The name assigned to the IBTT developed by Buy et al. is *Automated*, Column 1. Column 2 of Row 2 states that the class characteristics (Private Attributes, Primitive Types, and Simple Control Flow) can be *suitably tested* by the *Automated* technique. Column 3 lists the characteristics (Recursive Protected Attributes, and Public Attributes) that cannot be *suitably tested* by the *Automated* IBTT. Note that the set of class characteristics listed in Column 3 of Table 1, i.e., those characteristics not suited to an IBTT, is not exhaustive. For additional information regarding the description of the class characteristics listed in Columns 2 and 3 of Table 1 see the cited references.

2.3 Class Abstraction Techniques

Several class abstraction techniques (CATs) exist that allow a tester to abstract away details of the source code providing an alternative view of the entities represented in the code. These abstract views include: (1) various graphical representations, such as class

Researchers *(IBTT Name)*	Class Characteristics	
	Suited To	*Not Suited To*
Alexander et al.[1] *(Polymorphic Relationships)*	Primitive and user-defined types, polymorphism, dynamic binding	Local variables of routines assigned return values
Buy et al. [3] *(Automated)*	Primitive types, simple control flow	Complex variables e.g., arrays, structs; references
Harrold et al. [11] *(Incremental)*	Inherited classes	Classes with no parents
Harrold et al. [12] *(Data-Flow)*	Primitive types, new attributes	Complex variables e.g., arrays, structs; references, polymorphism, dynamic binding
Koppol et al. [15] *(Concurrent Programs)*	Features exhibiting concurrency and synchronization	Features not exhibiting concurrency and synchronization
Kung et al. [16] *(Object State)*	Primitive types, simple control flow	Large number of attributes
Sinha et al. [27] *(Exception-Handling)*	Exception objects and variables, references to exception objects	Attributes/local variables outside exception mechanism
Souter et al. [28] *(OMEN)*	Objects in the presence of polymorphism, aliasing, and inheritance	Primitive types, references to primitive types

Table 1. Summary of IBTTs identifying the class characteristics that are *suited to* and *not suited to* the respective IBTT. The characteristics in Columns 2 and 3, and the scope in Column 4 are extracted from references cited in Column 1.

diagrams [24, 19] and object models [16], (2) various graphs, such as control flow graphs (CFGs) [12, 28], (3) object-oriented design metrics (OODMS) [9], and (4) the classification of OO characteristics [11]. Although the aforementioned CATs provide useful views for different aspects of the software development process, none of them are well suited to identifying the combination of class characteristics that allows the tester to identify those IBTTs *most suitable* for testing a specific class.

One CAT not fully exploited in reverse engineering tools is the classification of OO classes based on some type of comprehensive taxonomy. A *taxonomy* is the science of classifying elements according to an established system in a specific domain, with the resulting catalog used to provide a framework for analysis. Any taxonomy should take into account the importance of separating elements of a group (*taxon*) into subgroups (*taxa*) that are mutually exclusive, unambiguous, and taken together include all possibilities [32].

3 Taxonomy of OO Classes

In this section we describe our *Taxonomy of OO Classes* that catalogs each class in an OO software application based on the characteristics of that class. These characteristics include

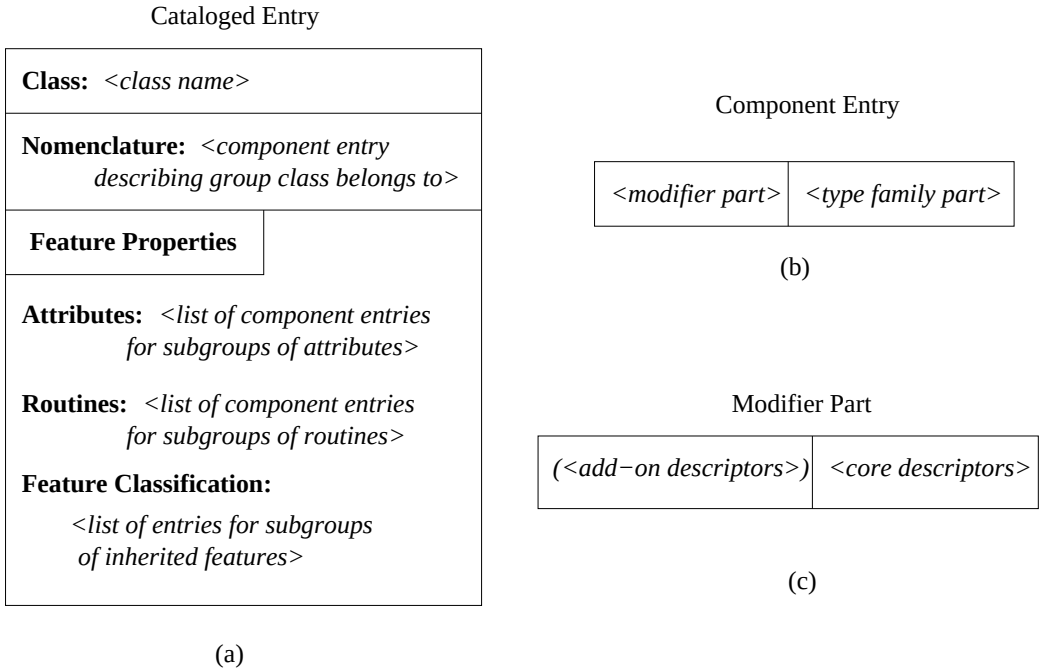

Figure 1. Diagram showing the structure of various entities used used to represent a cataloged class. **(a)** Cataloged Entry. **(b)** Component Entry, consisting of the modifier and type family parts. **(c)** Modifier Part, consisting of add-on descriptors, enclosed in parentheses, and core descriptors.

the properties of the features (attributes and routines), as well as relationships with other classes. The properties of the features cataloged include criteria such as: types, accessibility, shared class features, deferred features, binding of routines, polymorphic attributes and routine locals, exception handling, and concurrency (see Subsection 2.1).

Each class cataloged using our taxonomy of OO classes generates an entry, referred to as a *Cataloged Entry*, consisting of five main components: (1) *Class* - fully qualified name of the class, (2) *Nomenclature* - identifies the group (or taxon) the class belongs to, (3) *Attributes* - a list that identifies the subgroup each attribute belongs to, (4) *Routines* - a list that identifies the subgroup each routine belongs to, and (5) *Feature Classification* - summarizes the features of an inherited class. The Attributes, Routines and Feature Classification components are grouped together to form the *Feature Properties* section. Figure 1(a) illustrates the structure of a cataloged entry.

Each group or subgroup of a cataloged entry is represented by a *component entry*. The component entry is divided into two parts: (1) the *modifier* that describes the characteristics exhibited by the class, and (2) the *type family* that identifies the types associated with the class. Figure 1(b) illustrates how the parts of the component entry are combined. Each modifier is composed of a set of *descriptors* that summarizes the characteristics exhibited by a given class. To describe a class written in virtually any OO language the descriptors are divided into two groups: (1) *core* - identifies characteristics found in most OO languages, and (2) *add-ons* - descriptors specific to a language. Figure 1(c) shows the structure of the modifier part of a component entry. In this paper we present the add-on descriptors for the C++ language.

A formal definition for our taxonomy of OO classes is presented in reference [4]. In addition, we show that our taxonomy of OO classes exhibit the following properties: (1) partitions the set of all C++ classes into mutual exclusive sets, (2) can catalog any class written in C++, and (3) each component entry is represented in an unambiguous manner, i.e., a regular grammar is given in reference [4]. The descriptors and type families used in the Nomenclature and Feature Properties components are described in the following subsections.

3.1 Nomenclature

The *nomenclature* is the name that identifies a group of classes sharing similar characteristics, i.e. classes belonging to the same taxon. The following descriptors are used in the *modifier* part of the nomenclature.

Core descriptors:

- *Generic* - identifies a class that takes formal generic parameters representing arbitrary types.
- *Concurrent* - identifies a class whose instances are threads/processes.
- *Inheritance-free* - identifies a class that is not part of an inheritance hierarchy, i.e., a class that is not a parent and is not a descendant of another class.
- *Parent* - identifies a class that is the root of an inheritance hierarchy.
- *External Child* - identifies a class that is a descendant and not a parent.
- *Internal Child* - identifies a class that is a descendant and a parent.
- *Abstract* - identifies a class that contains deferred features.

Add-on descriptors for C++:

- *Nested* - identifies a class that contains the definition of another class.
- *Multi-Parents* - identifies a class with more than one parent.
- *Friend* - identifies a class that is a friend of another class.
- *Has-Friend* - identifies a class that has friends.

The modifier part of the nomenclature is a combination of the above descriptors. It should be noted that there are certain combinations of descriptors that are prohibited e.g. *Inheritance-free, Parent, External Child,* and *Internal Child* are mutually exclusive and cannot be used together in the same nomenclature entry. If the modifier contains an add-on descriptor, it is enclosed in parentheses and precedes the core descriptor(s).

The *type family* part of the nomenclature reflects the types used in the class. That is, it represents a summary of the types of the attributes and locals (parameters and local variables) of the routines. The type families (or simply *families*) used in the taxonomy are:

- *Family NA* - no associated types used.
- *Family P* - scalar primitive types e.g., *int*.

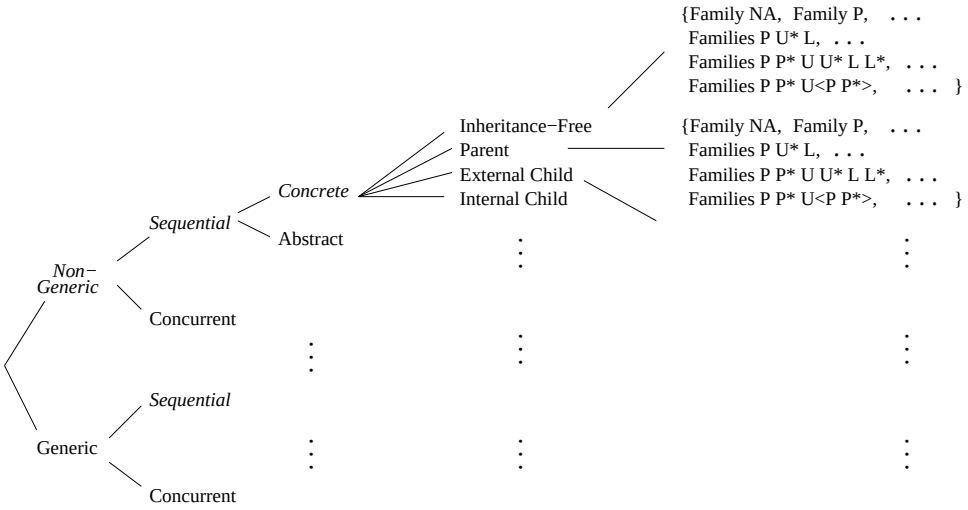

Figure 2. Tree structure representing the possible combinations of core descriptors for the Nomenclature component entry. The descriptors in italics are defaults and therefore not stated in the Nomenclature. The type families are enclosed in braces representing all possible combinations, of which one is chosen. Vertical ellipses implies repetition of the tree structure. The type families for Generic classes include families A and A*, representing the unknown parameters in the generic classes

- *Family P** - references to primitive types.
- *Family U* - user-defined types i.e., classes.
- *Family U** - references to user-defined types.
- *Family L* - standard class libraries.
- *Family L** - references to class libraries.
- *Family A* - any type (used as parameters for generics).
- *Family A** - references to any type.

The type family part of the Nomenclature component entry usually consists of a combination of the *type families* described. Following are the only possible combinations of type families used in the Nomenclature component entry:

- *Family NA* - means there are no attributes, or routine locals declared in the class.
- *Families t* - represents the type families of the attributes and routine locals, where t is one or more of the following *P, P*, U, U*, L, L*, A, A*, m<n>, or m<n>**. Note *m<n>* represents attributes or routine locals that are instantiated as generic types, where m is type family U or L, and n represents any combination of the type families *P, P*, U, U*, L, L*, A, or A*.* *m<n>** is a reference to a generic type.

Note that the above *families* are grouped with testing in mind. For example, knowing whether a class belongs to a class library or not can be helpful to the tester, hence the groups U and L. Class libraries are assumed to be *trusted frameworks* and used with a high degree of confidence [20].

Figure 2 illustrates how the descriptors and type families are combined in the Nomenclature component. In addition, Figure 2 shows how our taxonomy catalogs OO classes into mutually exclusive groups (or *taxa*). An example of one such group is *Non-Generic Sequential Concrete Inheritance-Free Family P*, shown along the top branch of the tree in Figure 2. Since we consider the descriptors *Non-Generic, Sequential,* and *Concrete*, as defaults descriptors, the Nomenclature becomes *Inheritance-Free Family P*. This group represents classes that are not part of an inheritance hierarchy and contain attributes and routine locals whose types are primitive. The default descriptors are italicized in Figure 2.

3.2 Feature Properties

The *Feature Properties* section of the taxonomy consists of three components: (1) *Attributes* - a list of subgroups categorizing the attributes, (2) *Routines* - a list of subgroups categorizing the routines, and (3) *Feature Classification* - a summary of the inherited features. The properties of the attributes of a class are described using the following descriptors.

Core descriptors:

- *Concurrent* - attribute is a thread/process.
- *Polymorphic* - attribute is polymorphic i.e., the attribute is a reference to a user defined type (*U**), and the user defined type (class) has descendants.
- *Private, Protected or Public* - the accessibility of the attribute.
- *Constant* - attribute is a named constant.
- *Static* - attribute is shared class data.
- Type family information of the attribute:
  - *Family NA* - represents no class attributes.
  - *Family t* - represents the type family of the attribute, where *t* is one of following *P, P*, U, U*, L, L*, A, A*, m<n>,* or *m<n>**.
  - *Family m<n>* - represents attributes that are instantiated generic types, where *m* is type family *U* or *L*, and *n* represents any combination of type families *P, P*, U, U*, L, L*, A, A**.

The properties of a routine in the class are described using the following descriptors:

Core descriptors:

- *Concurrent* - routine instantiates a thread/process.
- *Synchronized* - routine contains code that is synchronized.
- *Exception-R* - routine contains code that raises an exception.
- *Exception-H* - routine contains code that handles an exception.
- *Has-Polymorphic* - routine contains polymorphic variables.
- *Non-Virtual* - routine is statically bound.
- *Virtual* - routine is dynamically bound.

- *Deferred* - implementation of the routine is deferred.
- *Private, Protected or Public* - depending on the accessibility of the routine.
- *Static* - if the routine is a shared class routine.
- Type family information for parameters and local variables.
  - *Family NA* - represents no parameters or local variables.
  - *Families t* - represents the type families of the parameters or local variables, where *t* is one or more of the following *P, P*, U, U*, L, L*, A, A*, m<n>,* or *m<n>**. *m<n>* represents parameters or local variables that are instantiated generic types, where *m* is type family *U* or *L*, and *n* represents any combination of type families *P, P*, U, U*, L, L*, A, A**.

Add-on descriptors for C++:

- *Constant* - routine does not allow attributes of the class to be modified.

The properties of the attributes and routines are represented as a combination of the appropriate descriptors.

We classify the inherited features of a class as outlined in [11]:

- *New* - feature is declared in the child class.
- *Recursive* - feature is inherited from the parent unchanged.
- *Redefined* - feature is a routine and has the same signature as the one declared in the parent but with a different implementation.

The entries in the *Attributes* and *Routines* components of the taxonomy are prefixed with either *New*, *Recursive* or *Redefined* as appropriate. A summary of the inherited features for derived classes is presented in the *Feature Classification* component of the taxonomy. The inherited features are categorized as either an attribute or a routine. In addition to identifying a derived feature as a routine in the feature classification component, we also identify it as *non-virtual* if it is statically bound, or *virtual* if it is dynamically bound. A value of *None* is assigned to the feature classification component for *Inheritance-free* classes and *Unknown* for classes derived from classes in the standard library.

Each entry in the *Feature Properties* section of a cataloged entry is prefixed with a numerical value enclosed in square brackets representing the number of times that category of feature occurred in the class. The add-on descriptors used in the *Attributes* and *Routines* components are enclosed in parentheses.

3.3 Illustrative Example

Figure 3 illustrates an application of our taxonomy to a C++ class. Figure 3(a) shows the C++ code for classes Point, Cartesian and Polar. Class Point declares two protected attributes, x and y, both of type int and five public routines three constructors, a virtual destructor, and the constant virtual routine print. Classes Cartesian and Polar inherit from Point.

```
1  class Point{
2  protected:
3    int x, y;
4  public:
5    Point(): x(0), y(0){}
6    Point(int inX, int inY):
7              x(inX), y(inY){}
8    Point(const Point & p):
9      x(p.x),y(p.y){}
10   virtual ~Point(){}
11   virtual void print() const {
12     cout << " x = " << x << ","
13     << " y = " << y << endl;}
14 };
15
16 class Cartesian: public Point{
17   void print() const {cout <<
18     " abscissa = " << x << "," <<
19     " ordinate = " << y << endl;}
20 };
21
22 class Polar: public Point{
23   void print() const {cout <<
24     " distance = " << x << " ,"
25     << " angle = " << y << endl;}
26 };
```

(a)

Class: Point

Nomenclature: *Parent Families P, U**

Feature Properties

Attributes: *[2] Protected Family P*
{x, y}

Routines:

[1] Non−Virtual Public Family NA
{Point()}

[1] Non−Virtual Public Family P
{Point(int inX, int inY)}

*[1] Has−Polymorphic Non−Virtual Public Family U**
{Point(const Point & p)}

[1] Virtual Public Family NA
{~Point()}

[1] (Constant) Virtual Public Family NA
{print()}

Feature Classification: *None*

(b)

Figure 3. (a) C++ code for classes Point, Cartesian, and Polar. (b) Cataloged entry for class Point. Each numbered component entry in the Attributes and Routines components represents the number of features with the same characteristics. The entities in braces are the features that belong to that subgroup, shown here for illustration purposes only.

Figure 3(b) illustrates the cataloged entry for class Point. The nomenclature for class Point is *Parent Families P, U** for the following reasons: Point is the root of an inheritance hierarchy (*Parent*), and the data types used in declarations are the primitive type int (*Family P*) and a reference to the user-defined type Point (*Family U**). The entry in the Attributes component, is *[2] Protected Family P* that represents the attributes x and y, line 3 Figure 3(a).

The entries in the Routines component for the constructors are: *[1] Non-Virtual Public Family NA* representing the zero argument constructor Point(), line 5, *[1] Non-Virtual Public Family P* representing the two argument constructor Point(int inX, int inY), line 6, and *[1] Has-Polymorphic Non-Virtual Public Family U** for the one argument constructor Point(Point & p), line 8. The destructor ~Point() is cataloged as *Virtual Public Family NA* and routine print() as *(Constant) Virtual Public Family NA*. The descriptors *Protected* and *Public* reflect the accessibility for the attributes and routines in class Point. The binding of the routines are represented by the descriptors *Non-Virtual* - static and *Virtual* - dynamic. The add-on descriptor *Constant*, peculiar to C++, states that the routine print() prevents attributes of the class from being modified. The Feature Classification component in the cataloged entry for class Point contains the entry *None* because class Point is not a derived class, thereby not containing any derived features.

Application Name	Library Release	No. Lines	Cataloged			Avg. CEs/Class	
			Classes	Attrs.	Routs.	Attrs.	Routs.*
gauss elim	ADOL-C 1.6	737	11 (11)	22	131	1.3	6.3
IV graphdraw	IV Tools 1.0.1	4,301	151 (64)	421	1171	3.8	8.2
ep matrix	NTL 5.2	7,068	50 (40)	118	467	1.6	5.0
vkey	V GUI 1.2.5	8,588	46 (24)	293	435	3.2	6.6

Table 2. Summary of the test cases analyzed using TaxTOOL. The number in parentheses in Column 2 indicates the number of classes with routines. The abbreviation *CEs* represents Component Entries. *The number of classes with routines is used to compute the average number of Routines component entries per class.

4 Analysis of C++ Applications

In this section we present a summary of the results obtained when several C++ applications are analyzed using *TaxTOOL*. TaxTOOL - A Taxonomy Tool for an OO Language, is a reverse engineering tool that catalogs classes written using the C++ language [4, 6]. A more comprehensive set of results relating to the cataloged entries is presented in [4].

Table 2 summarizes the application suite analyzed using TaxTOOL. Row 1 summarizes the data for the first application gauss elim, Column 1, a program that performs Guassian Elimination using the library ADOL-C [8], Column 2. The second application, Row 2 Table 2 , is IV graphdraw a drawing application that uses the IV Tools library [14]. The third application, Row 3, is ep matrix, an extended precision matrix application that uses NTL [26]. The fourth application is vkey [30, p. 760], a GUI application that uses the V GUI library [31], a multi-platform C++ graphical interface framework to facilitate construction of GUI applications.

Columns 3 through 8 of Table 2 list data obtained when the respective applications are reverse engineered using TaxTOOL. For example, gauss elim, shown in Row 1, has 737 non-blank lines, Column 3, and produces 11 cataloged entries summarizing the classes in gauss elim, Column 4 (all classes contain routines - number in parentheses). Application gauss elim has a total of 22 attributes and 131 routines as listed in Columns 5 and 6, respectively. In addition, the data in Columns 6 and 7 shows that application gauss elim produces on average 1.3 Attribute component entries and 6.3 Routine component entries per class, respectively. The averages for the Routines component entries are calculated using only classes that contain routines.

Our taxonomy of OO classes not only summarizes the characteristics of a class but also eliminates excessive analysis of a CUT when deciding on which IBTTs should be used to test a feature in the CUT. Before the tester can select the most appropriate IBTTs for a given CUT (class or cluster), i.e., the IBTT that provides the best coverage, an analysis of the CUT must be performed to identify its characteristics (see Table 1). Using TaxTOOL a succinct summary for each CUT is produced thereby eliminating the need for the tester to: (1) check the source code for the characteristics of the CUT, or (2) perform additional analysis on the artifacts generated by existing reverse engineering tools. Currently, there are a number of reverse engineering tools [7] that generate artifacts using the CATs stated in Subsection 2.3, however none of them produce a summary of the CUT that eliminates

excessive analysis of the source code or reversed engineered artifacts when mapping IBTTs to the CUT. For example, class diagrams reverse engineered from the source code [19] do not capture declarations of local variables in a routine or identify polymorphic attributes, therefore the tester must perform additional analysis on the source code or class diagram to extract this information.

In addition to the elimination of excessive analysis of a CUT when selecting IBTTs during class-based testing, Table 2 shows that the number of entities to be analyzed during the selection process can be reduced. For example, the values of the vkey application in Row 3 of Table 2 support this claim. The average number of routines per class in the vkey application is 18.1 (435/24, using the values in Columns 6 and 4 of Table 2 respectively), while the number of Routine component entries is 6.6 (Column 8 of Table 2). If the tester selects the appropriate IBTTs for a routine using the class characteristics in the Routine component entries there would be a 64% reduction in the number of entities analyzed i.e., 6.6 component entries versus 18.1 routines. In general there is a 50% reduction in the number of entities to be analyzed by the tester (attributes - 44% and routines - 55%).

5 Mapping IBTTs to OO Classes

In this section we describe the data structures used in the mapping process, describe the mapping algorithm, and provide an illustrative example showing how the mapping algorithm works [4].

5.1 Mapping Process

The mapping process accepts a summary of the CUT and a list summarizing the IBTTs available to the tester, then identifies those features of the CUT that can (cannot) be *suitably tested* by an IBTT. In this subsection we describe how the IBTTs available to the tester are represented using our taxonomy of OO classes, define the Boolean operator *matches* used to compare component entries, and describe the structure of the input parameters and values returned from the mapping algorithm.

5.1.1 Summary of an IBTT using Catalog Entries

The entry used by our taxonomy to catalog classes provides the basic structure when summarizing the class characteristics that can be *suitably tested* by an IBTT. Figure 4 shows a diagrammatic representation of how an IBTT is summarized using a list of catalog entries. Figure 4(a) shows the structure that contains the summary for each IBTT, referred to as an *IBTT entry*. Each IBTT entry consists of a unique identifier, *UniqueID*, and a list of catalog entries, *EntryList*. Figure 4(b) represents one of the catalog entries from the list of catalog entries in Figure 4(a). The catalog entry shown in Figure 4(b) contains the following fields: (1) *Priority* - the tester assigned priority, (2) *Nomenclature* - the group of classes that can be suitably tested by the IBTT, (3) *Attributes* - a list containing the groups of attributes suitably tested by the IBTT, and (4) *Routines* - a list containing the groups of routines suitably tested by the IBTT. The tester assigned priority is used to resolve the issue where IBTTs *suitably test* the same class characteristics.

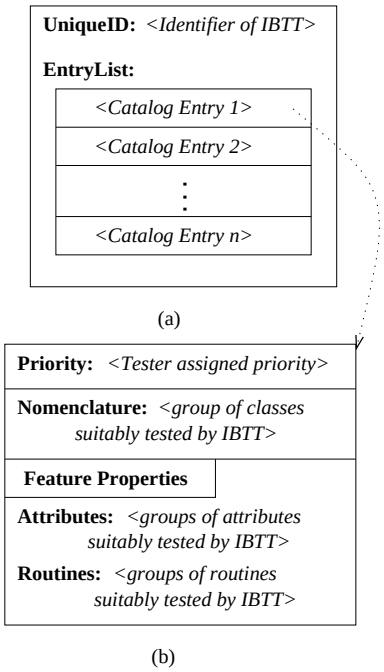

Figure 4. Summary for each IBTT input to the mapping process. **(a)** An IBTT entry. **(b)** A single catalog entry for the IBTT entry in part (a).

5.1.2 Matching Component Entries

The input to the mapping process is a list containing the summaries of IBTTs, described in Section 5.1.1, and a cataloged entry of the CUT, described in Section 3. Figure 3(b) illustrates a cataloged entry for the class Point shown in Figure 3(a). The mapping process requires component entries from the cataloged entries of the CUT and IBTT to be compared. We define the Boolean operator *matches*, denoted by $\simeq$, to compare entries from the same component in two different cataloged entries. The Boolean operator *matches* is defined for component entries A and B as follows:

Definition 1: *Boolean operator matches* ($\simeq$). The value of a component entry A *matches* a component entry B if and only if: (1) the string of descriptors in the modifier part of A is a string in the language generated by the modifier part of B, and (2) the intersection of type families in A and B is nonempty.

That is,

$$A \simeq B \iff A.modifier \in L(B.modifier) \text{ and } A.typeFamily \cap B.typeFamily \neq \varnothing$$

where $C.modifier$ is the string of descriptors of C, $L(C.modifier)$ is the language generated by the modifier component of C, and $C.typeFamily$ is the set containing the type families of C. □

The language for a modifier is expressed using EBNF notation [25]. The language describing the strings in the modifier is a subset of the strings generated by the grammar defined for the all the component entries [4].

An example of the *matches* operator is, *New Non-Virtual Public Families P U* $\simeq$ *New* (*Non-Virtual* | *Virtual*) (*Private* | *Protected* | *Protected*) [*Static*] *Family P*, that evaluates to *true*. In this example the entries used are from the Routines component of a cataloged entry. The component entry on the right-hand side of the $\simeq$ operator uses EBNF to describe the language representing the set of strings generated by the modifier component. Using Definition 1 the component entries in the example match, because *New Non-Virtual Public* $\in L($*New* (*Non-Virtual* | *Virtual*) (*Private* | *Protected* | *Public*) [*Static*]) and $\{P, U\} \cap \{P\} \neq \varnothing$.

5.1.3 Mapping IBTTS to CUT

The input to the mapping process is a list containing summaries of IBTTs and a catalog entry of the CUT. Figure 3(b) shows a cataloged entries for the CUT Point and Figure 4(a) shows the structure of a single entry in the list containing the summary of IBTTs. The output of the mapping process consists of a list of 4-tuples representing those CUT characteristics that can be *suitably tested* by some IBTT. In addition, the output of the mapping process provides feedback information informing the tester of any CUT characteristic that cannot be *suitably tested* by any IBTT. Each 4-tuple output in the list of characteristics that can be *suitably tested* by some IBTT consists of: (1) a characteristic of the CUT, (2) a list of feature pairs, (3) a unique identifier of the IBTT, and (4) a priority assigned to the IBTT by the tester. Each feature pair consists of the feature's name and whether it is an attribute or routine. The feedback information to the tester is in the form of a list, each element in the list is an ordered pair. Each ordered pair in the feedback information list consists of: (1) a characteristic of the CUT not *suitably tested* by any IBTT, and (2) a feature pair.

The first step in the process is to compare the Nomenclature component entries of the CUT and the Nomenclature component entry of each catalog entry of each IBTT. The field EntryList in the IBTT summary entry, shown in Figure 4(a), contains the list of catalog entries. If the Nomenclature component entry of the CUT *matches* the Nomenclature component entry of a catalog entry for an IBTT, say $IBTT_i$, then the mapping process goes to the second step. In the second step of the mapping process, the entries of the Attributes and Routines components for the CUT are compared to the corresponding entries in the catalog entry of $IBTT_i$. If there is a match then that entry can be *suitably tested* by $IBTT_i$.

During the mapping process two lists are maintained that keep track of: (1) the 4-tuples representing those characteristics of the CUT that can be *suitably tested* by some IBTT, and (2) the characteristics of the CUT that cannot be *suitably tested* by any IBTT. If there is a match between any of the entries in the Attributes or Routines components of the CUT and the corresponding IBTT catalog entry, a 4-tuple is created and added to the list of 4-tuples. If there is a match between a CUT entry and the corresponding entry in the IBTT and the difference between the type families of the entries is the empty set, then that entry is removed from the list of characteristics of the CUT that cannot be *suitably tested*. Otherwise, the entry for the CUT is updated by removing the type families that can be *suitably tested* and the revised entry remains in the list of characteristics of the CUT that cannot be *suitably tested*.

5.2 Mapping Algorithm

The algorithm IBTT_CUTMap shown in Figure 5 maps IBTTs to a OO class under test (CUT). An extended version of the algorithm is given in Appendix A [4]. The mapping relation from IBTTs to a CUT is *suitably test*. The parameters passed to algorithm IBTT_CUTMap, Figure 5, consist of: (1) cutEntry - a cataloged entry for the CUT that summarizes the characteristics of the CUT, and (2) ibttList - a list containing a summary of the IBTTs available to the tester. Note that the tester is responsible for initializing ibttList to those IBTTs available to the tester. The data returned from algorithm IBTT_CUTMap is stored in the variable ibttCutMap. The data in ibttCutMap consists of: (1) tupleList - a list of tuples representing the mapping from IBTTs to the CUT, and (2) charsNotTestedList - a list of the class characteristics that cannot be *suitably tested* by any IBTTs in ibttList. The output produced from the mapping process is described in Section 5.1.2.

Line 3 of algorithm IBTT_CUTMap shown in Figure 5 initializes the fields in ibttCutMap. Line 4 creates ordered pairs for all the entries in the Attributes and Routines components of cutEntry and adds them to the variable ibttCutMap.charsNotTestedList (see Subsection 5.1.3). The loop lines 5 through 23 sequences through each IBTT in ibttList. The entries in the various components of the IBTT under consideration are compared to the entries in the corresponding components in the CUT seeking a match. If there is a match between the corresponding Nomenclature component entries, line 8, then the Attributes and Routines component entries are compared. A match between corresponding entries in the Attributes component, line 11, creates a 4-tuple, adds it to the tuple list ibbtCutMap.tupleList, and the component entry of the CUT, cutEntry.Attr, removed from ibttCutMap.charsNotTestedList. If more than one feature can be *suitably tested* by an IBTT then ibbtCutMap.tupleList contains the IBTT with the lowest priority.

Similar actions, to those for the Attributes component, are performed for the entries in the Routines component, line 17. The only difference is that a match results in the an update of the cutEntry.Rout in ibttCutMap.charsNotTestedList. This update involves removing the component entry from ibttCutMap.charsNotTestedList if the difference between the type families of the corresponding component entries is the empty set, otherwise only the type families that intersect are removed from the component entry in ibttCutMap.charsNotTestedList.

The running time of algorithm IBTT_CUTMap is $O(j * n * max(a + r))$. The value j represents the number of IBTTs in the input list ibttList and n is the cost required to check if Nomenclature component entries match, line 8 of Figure 5. The value a is the cost to compare each entry in the Attributes component of the CUT and the IBTT entry line 11. The value r is the cost to compare each entry in the Routines component of the CUT and the IBTT entry, line 17 of Figure 5. Note that although the running time appear to be a cubic function, the values of a and r are expected to be small, as suggested by the results reported in Table 2, Columns 7 and 8.

5.3 Mapping Example

In this subsection we describe an application of algorithm IBTT_CUTMap shown in Figure 5. The input to algorithm IBTT_CUTMap consist of: (1) a cataloged entry for class Point,

```
1: IBTT_CUTMap(cutEntry, ibttList)
2: /*Input: cutEntry - A cataloged entry for the CUT
          ibttList - A list containing summaries of IBTTs
     Output: ibbtCutMap - variable containing:
          (1) tupleList, list of tuples mapping IBTTs
              to characteristics of CUT, and
          (2) charsNotTestedList,list of characteristics
              of CUT with no suitable IBTT. */
3: Initialize fields of ibbtCutMap
4: Create tuples for entries in Attributes and Routine
     components and add them to
     ibbtCutMap.charsNotTestedList
5: for all ibtt in ibttList do
6:  for all ibttEntry in ibtt do
7:   /* Nomen - Nomenclature entry */
8:   if cutEntry.Nomen ≃ ibttEntry.Nomen then
9:    for all Attributes in cutEntry do
10:    /* Attr - Attribute component entry */
11:    if cutEntry.Attr ≃ ibttEntry.Attr then
12:     Create 4-tuple, update ibbtCutMap.tupleList using
            ibttEntry.Priority, and remove cutEntry.Attr
            from ibbtCutMap.charsNotTestedList
13:    end if
14:   end for
15:   for all Routines in cutEntry do
16:    /* Rout - Routine component entry */
17:    if cutEntry.Rout ≃ ibttEntry.Rout then
18:     Create 4-tuple, update ibbtCutMap.tupleList using
            ibttEntry.Priority, and update
            ibbtCutMap.charsNotTestedList
19:    end if
20:   end for
21:  end if
22: end for
23: end for
24: return ibttCutMap
25: end IBTT_CUTMap
```

Figure 5. Overview of Algorithm to map IBTTs to a CUT.

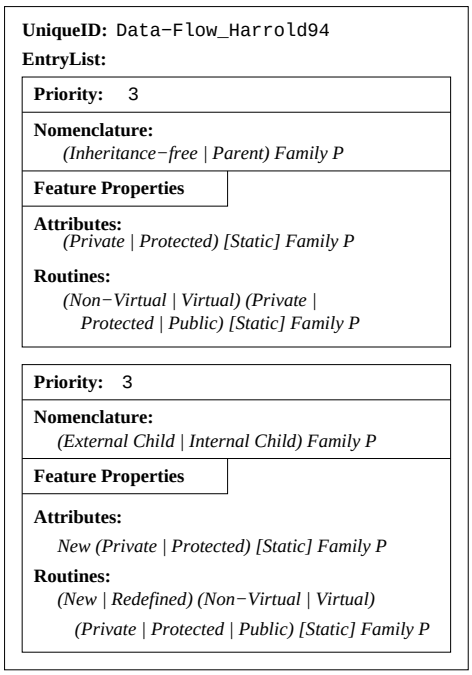

Figure 6. Summary of the Data-flow IBTT, the single IBTT stored in ibttList, used as input to algorithm IBTT_CUTMap.

Figure 3(b), and (2) a summary of the *Data-Flow* IBTT by Harrold et al. [12], Figure 6. In this example the *Data-Flow* IBTT is assigned the unique identifier Data-Flow_Harrold94. We stress the fact that the summary of IBTTs in ibttList, the second input parameter of algorithm IBTT_CUTMap, is supplied by the tester. For the purpose of this example we assigned possible values to the component entries of the *Data-Flow* IBTT in Figure 6.

The component entries of each cataloged entry in Figure 6 is written using EBNF notation [25]. For example, the Nomenclature component entry in the first cataloged entry of the IBTT summary for Data-Flow_Harrold94 is *(Inheritance-free | Parent) Family P*. This Nomenclature entry says that this IBTT can be used to *suitably test*: (1) any class that is not part of an inheritance hierarchy and contains primitive data types, or (2) any class that is the root of an inheritance hierarchy and contains primitive data types. The Attributes entry for the first cataloged entry in Data-Flow_Harrold94, Figure 6, is *(Private | Protected)* [*Static*] *Family P*. The entry states that Data-Flow_Harrold94 can *suitably test* attributes that are either private or protected, can be static, and are primitive types.

The output generated after applying algorithm IBTT_CUTMap to cutEntry, containing a cataloged entry of class Point, and ibttList, containing catalog entries for Data-Flow_Harrold-94, is shown in Figure 7. Figure 7(a) represents the variable ibbtCutMap a local variable that references the two list returned from IBTT_CUTMap. These list include: (1) ibbt-CutMap.tupleList - the features of class Point *suitably test* by Data-Flow_Harrold94, Figure 7(b), and (2) ibttCutMap.charsNotTestedList - the features that cannot be *suitably tested* by Data-Flow_Harrold94, Figure 7(c). The first entry in Figure 7(b) is a 4-tuple consisting of: (1) *Characteristic* - a component entry representing the attributes x and y in class Point, (2) *Feature Pairs* - consisting of (x, ATTRIBUTE) and (y, ATTRIBUTE) for the two attributes in

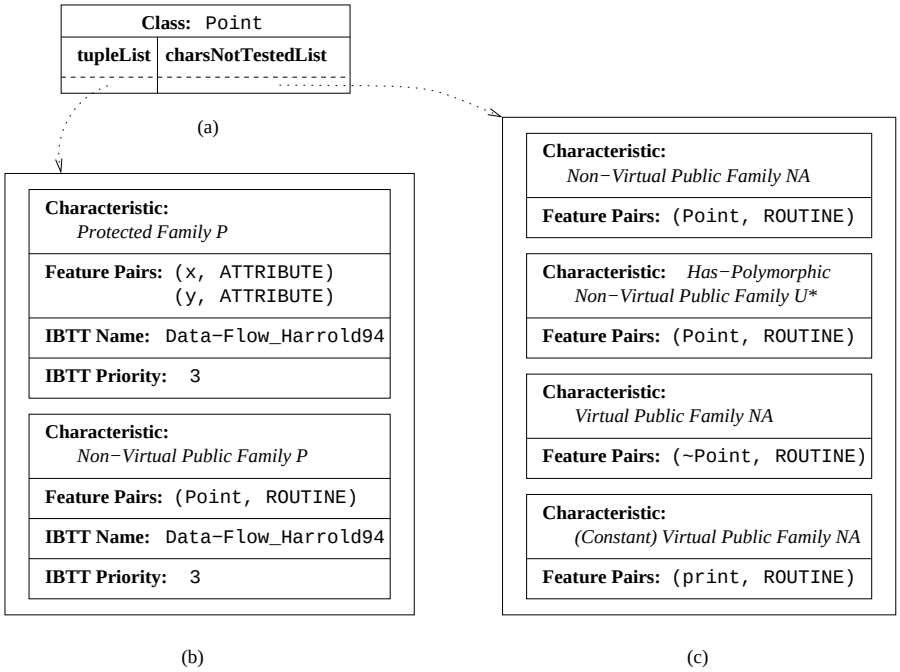

Figure 7. Output of algorithm IBTT_CUTMap after mapping the IBTT Data-Flow_Harrold94 to class Point. **(a)** Structure of variable ibttCutMap. **(b)** Contents of list ibttCutMap.tupleList. **(c)** Contents of list ibttCutMap.charsNotTestedList.

class Point, (3) IBTT Name - the IBTT Data-Flow_Harrold94, that can *suitably test* test the listed feature pairs, and (4) IBTT Priority - tester assigned priority, 3. Figure 7(c) contains a list 2-tuples, each 2-tuple consisting of the feature characteristics that cannot be *suitably tested* by any IBTT, and a list of the features with the stated characteristic.

Applying algorithm IBTT_CUTMap, Figure 5, to the input described in paragraph one of this subsection results in the following events. The variable ibbtCutMap is initialized on line 3 of algorithm IBTT_CUTMap. All the Attributes and Routines component entries from the CUT for class Point are copied to the variable ibbtCutMap.charsNotTestedList, line 4. There is only one ibtt in ibttList therefore the loop lines 5 through 27 is executed once. A match occurs on line 8 between the Nomenclature component entry of cutEntry and the first cataloged entry for the IBTT Data-Flow_Harrold94, Figure 3(b) and Figure 6 respectively. A match occurs between the Attribute component entries, line 12 of algorithm IBTT_CUTMap, resulting in a 4-tuple being created and added to ibbtCutMap.tupleList, the first entry in Figure 7(b). In addition, the component entry *Protected Family P* is removed from ibbtCutMap.charsNotTestedList. The second entry in ibbtCutMap.tupleList, Figure 7(b), is added as a result of a match between the Routine component entry for the two-argument constructor Point and the Routine component entry in the first cataloged entry for the IBTT Data-Flow_Harrold94. No other matches occur for the second cataloged entry of the IBTT Data-Flow_Harrold94.

6 Related Work

There are several class abstraction techniques (CATS) that provide alternative views of the software application to a tester. Harrison et al. [9] overview three OODM sets, including a cross-section of the set developed by Lorenz et al. [17]. The OODM set by Lorenz et al. contains metrics that reflect certain characteristics of a class closely related to our taxonomy of OO classes. Three of the metrics in the set by Lorenz et al. are: *Number of Public Methods (PM)*, *Number of Methods Inherited by a subclass (NMO)*, and *Number of Methods Overridden by a subclass (NMO)*. Although the OODM by Lorenz et al. identifies several class characteristics it does not show how these characteristics are combined in the class. Our taxonomy of OO classes can identify the number of routines (methods) in a derived class that are public and inherited (recursive). The combination of class characteristics are essential when mapping IBTTs to a CUT.

Harrold et al. [11] classifies the features of a derived class and use this classification to identify those test cases of the parent class that can be reused when testing the derived class. A brief summary of the testing technique by Harrold et al. is presented in Subsection 2.2.3. Our taxonomy of OO classes extends the classification presented by Harrold et al. [11] to include characteristics for all classes written in virtually any OO language.

We have extended the work presented in reference [5] to include a revision of the taxonomy of OO classes and improve the approach used to identify the IBTTs that can *suitably test* a CUT. The revision of the taxonomy includes: (1) extending the number of descriptors in the Nomenclature, Attributes, and Routines components to more accurately summarize the characteristics of the class, (2) using add-on descriptors to catalog classes written in virtually any OO language, (3) renaming the type families (class associated types) to be more meaningful, and (4) extending the type families to include parameterized types. The mapping process in reference [5] is based solely on the Nomenclature component entry. We have extended the mapping process to include the entries in the Attributes and Routines components. In addition, we have defined the Boolean operator *matches* that uses both parts of a component entry to identify if an IBTT can *suitably test* a feature in the CUT.

7 Concluding Remarks

In this paper we have presented a technique to map implementation-based testing techniques (IBTTs) to a class under test (CUT). This technique uses our class abstraction technique (CAT), *A Taxonomy of OO Classes*, that summarizes the characteristics of a class. The summary of characteristics for an OO class is represented using a catalog entry. The main purpose of our taxonomy of OO classes is to provide the tester with the ability to easily identify those class characteristics that can influence the selection of an IBTT.

There are five major components of a cataloged entry: Class Name, Nomenclature, Attributes, Routines, and Feature Classification. The Nomenclature component captures the relationships with the CUT has with other classes in the application, including associations, dependencies, and generalizations. The entries in the the Attributes and Routines components summarize how criteria such as types, accessibility, shared class data, polymorphism,

dynamic binding, deferred features, exception handling, and concurrency are combined in the attributes and routines of a CUT.

Using our taxonomy of OO classes we described an algorithm (IBTT_CUTMap) that maps IBTTs to a CUT. The input to the IBTT_CUTMap consists of catalog entries for the CUT and the IBTTs available to the tester. The output of algorithm IBTT_CUTMap identifies the features of the CUT that can be *suitably test* at least one IBTT. The algorithm also provides feedback to the tester by generating a list of class features for which there is no IBTT that can *suitably test* those feature in the CUT. It is assumed the tester will be responsible for identifying the class characteristics that can be *suitably tested* by an IBTT.

There are several limitations of our mapping process. One limitation is the fact that our taxonomy does not capture any information regarding the type of control structures used in the various routines in a class. Several pre-OO IBTTs used coverage criteria based on the analysis of control structures [23]. A second limitation is that algorithm IBTT_CUTMap does not fully exploit the priorities assigned to the IBTT, in particular the priorities assigned to individual catalog entries for an IBTT. We are investigating ways to fully utilize these priorities to provide better accuracy in the mapping process.

We have developed a tool, *TaxTOOL - A Taxonomy Tool for an Object Oriented Language*, that catalogs classes written in the C++ language [4]. TaxTOOL also identifies changes between different versions of the same software application with respect to class characteristics described in this paper [6]. Currently, we are extending TaxTOOL to map IBTTs to a CUT based on the algorithm presented in this paper.

Appendix A

```
 1: IBTT_CUTMap(cutEntry, ibttList)
 2: ibttCutMap.className ← cutEntry.className
 3: ibttCutMap.tupleList ← ∅
 4: /* charsNotTestedList - is the list of all characteristics of the CUT that cannot be
        suitably tested by any IBTT */
 5: ibttCutMap.charsNotTestedList ← ∅
 6: /* temporary lists to store feature information */
 7: cutAttrList ← ∅
 8: cutRoutList ← ∅
 9: /* addFormatted - formats each feature and adds it to the appropriate list */
10: cutAttrList.addFormatted(cutEntry.Attributes)
11: cutRoutList.addFormatted(cutEntry.Routines)
12: for all ibtt ∈ ibttList do
13:   for all ibttEntry ∈ ibtt.EntryList do
14:     if cutEntry.Nomen ≃ ibtt.Nomen  then
15:       for all cutAttrRef ∈ cutAttrList do
16:         if cutAttrRef ≃ ibttEntryAttr then
17:           /* getMatchedPart - returns the matching part of the cutAttrRef
                 i.e., the modifier and intersection of the type families */
18:           matchedPart ← getMatchedPart(cutAttrRef, ibttEntryAttr)
19:           /* update - add the attribute name and feature type, attribute
20:           matchedPart.update(cutAttrRef)
```

```
21:        /* getUNMatchedPart - returns the cutAttrRef.modifier and type families
               of cutAttrRef not in the intersection of type families */
22:        unMatchedPart ← getUnMatchedPart(cutAttrRef, ibttEntryAttr)
23:        unMatchedPart.update(cutAttrRef)
24:        /* add - creates a 4-tuple and adds it to the tuple list. If a tuple exist
               to test the matchedPart then the priorities are compared resulting
               in the IBTT with the highest priority in the list, the attribute is
               added to list of feature pairs */
25:        ibttCutMap.tupleList.add(matchedPart, ibtt.Name, ibttEntry.Priority)
26:        /* replace - replaces the attribute entry in cutAttrList with the unmatched
               part. If unMatchedPart is empty then the attribute entry is removed
               form the list, cutAttrList */
27:        cutAttrList.replace(unMatchedPart, cutAttrRef)
28:       end if{if cutAttrRef . . .}
29:      end for{for all cutAttrRef . . .}
30:      /* Routines are processed similar to Attributes. */
31:      for all cutRoutRef ∈ cutRoutList do
32:       if cutRoutRef ≃ ibttEntryRout then
33:        matchedPart ← getMatchedPart(cutRoutRef, ibttEntryRout)
34:        matchedPart.update(cutRoutRef)
35:        unMatchedPart ← getUnMatchedPart(cutRoutRef, ibttEntryRout)
36:        unMatchedPart.update(cutRoutRef)
37:        ibttCutMap.tupleList.add(matchedPart, ibtt.Name, ibttEntry.Priority)
38:        cutRoutList.replace(unMatchedPart, cutRoutRef)
39:       end if
40:      end for
41:     end if{if cutEntry.Nomen . . .}
42:   end for{for all ibttEntry . . .}
43: end for{for all ibtt . . .}
44: /* add - adds the contents of the list to charsNotTestedList */
45: ibttCutMap.charsNotTestedList.add(cutAttrList)
46: ibttCutMap.charsNotTestedList.add(cutRoutList)
47: return ibttCutMap
48: end IBTT_CUTMap
```

References

[1] R. T. Alexander and A. J. Offutt. Criteria for testing polymorphic relationships. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, pages 15–24. ACM, Oct. 2000.

[2] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison Wesley, Reading, Massachusetts, third edition, 2000.

[3] U. Buy, A. Orso, and M. Pezze. Automated testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 39–48. ACM, August 2000.

[4] P. J. Clarke. *A Taxonomy of Classes to Support Integration Testing and the Mapping of Implementation-based Testing Techniques to Classes*. PhD thesis, Clemson University, August 2003.

[5] P. J. Clarke and B. A. Malloy. Identifying implementation-based testing techniques for classes. *International Journal of Computers and Information Systems*, 3(3):195–204, September 2002.

[6] P. J. Clarke, B. A. Malloy, and P. Gibson. Using a taxonomy tool to identify changes in OO software. In *7th European Conference on Software Maintenance and Reengineering*, pages 213–222. IEEE, March 2003.

[7] G. C. Gannod and B. H. C. Cheng. A framework for classifying and comparing software reverse engineering and design recovery tools. In *In Proceedings of the 6th Working Conference on Reverse Engineering*, pages 77–78. IEEE, October 1999.

[8] A. Griewank and O. Vogel. http://www.math.tu-dresden.de/wir/project/adolc/, April 2003.

[9] R. Harrison, S. Counsell, and R. Nithi. An overview of object-oriented design metrics. In *8th International Workshop on Software Technology and Engineering Practice*, pages 230–237. IEEE, July 1997.

[10] M. J. Harrold. Testing: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 61–72, New York, Dec. 2000. ACM.

[11] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oriented class structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 68–80. ACM, May 1992.

[12] M. J. Harrold and G. Rothermel. Peforming data flow testing on classes. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 154–163. ACM, December 1994.

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

[14] S. Johnston, J. Gautier, B. Hogencamp, R. Kissh, and E. Kahler. http://www.ivtools.org/ivtools/index.html, April 2003.

[15] P. V. Koppol, R. H. Carver, and K. C. Tai. Incremental integration testing of concurrent programs. *IEEE Transactions on Software Engineering*, 28(6):607–623, June 2002.

[16] 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 *Proceedings of the 7th International Symposium on Reliability Engineering*, pages 239–244. IEEE, August 1996.

[17] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Printice Hall Object-Oriented Series, 1994.

[18] K. C. Louden. *Programming Languages Principles and Practice*. PWS Publishing Company, 1993.

[19] S. Matzko, P. Clarke, T. H. Gibbs, B. A. Malloy, J. F. Power, and R. Monahan. Reveal: A tool to reverse engineer class diagrams. In *Proceedings of the 40th International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, pages 13–21. ACM, Feb 2002.

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

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

[22] NIST. The economic impacts of inadequate infrastructure for software testing. Technical Report, May 2002.

[23] M. Roper. *Software Testing*. McGraw-Hill, 1994.

[24] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, Inc, 1999.

[25] R. Sethi. *Programming Languages Concepts and Constructs*. Addison-Wesley, 1986.

[26] V. Shoup. Number theory library. *http://www.shoup.net/ntl/*, March 2002.

[27] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, September 2000.

[28] A. L. Souter and L. L. Pollock. OMEN: A strategy for testing object-oriented software. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 49–59. ACM, August 2000.

[29] B. Stroustrup. *The C++ Programming Language (Special 3rd Edition)*. Addison-Wesley, 2000.

[30] T. Swan. *GNU C++ for linux*. Que Corporation, first edition, 2000.

[31] B. Wampler. The V C++ GUI framework. *http://www.objectcentral.com*, October 2001.

[32] Whatis. Whatis.com target *search*^{TM}. *http://whatis.techtarget.com/*, May 2002.