International Journal of Software Engineering and Knowledge Engineering

A Tool To Automatically Map Implementation-Based Testing Techniques to Classes

Peter J. Clarke, Junhua Ding, Djuradj Babich

School of Computer Science, Florida International University
Miami, FL 33199, USA *
{clakep,jding01,dbabi001}@cs.fiu.edu†

Brian A. Malloy

Computer Science Department,Clemson University
Clemson, SC 29634, USA
malloy@cs.fiu.edu

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

The object-oriented (OO) paradigm provides several benefits during analysis and design of large-scale software systems, but scores lower in terms of testability. The low testability score for OO software is due mainly to the composition of OO systems exhibiting the characteristics of abstraction, encapsulation, genericity, inheritance, polymorphism, concurrency and exception handling. To address the difficulty of testing the features of a class, a plethora of implementation-based testing techniques (IBTTs) have been developed. However, no one IBTT has emerged as the preferred technique to test the implementation of a class. In this paper we present a technique that automatically identify those IBTTs that are most *suitable* for testing a class based on the characteristics of that class. Our approach uses a taxonomy of OO classes that is used to succinctly abstract the characteristics of a class under test (CUT). We have implemented a tool that automates the process of mapping IBTTs to a class. In addition to identifying the IBTTs that would be best suited for testing a class, our tool provides feedback to the tester facilitating the identification of the characteristics of the class that are *not suitably tested* by any of the IBTTs in the list. We provide results of a study supporting the notion that using more than on IBTT during testing improves test coverage of a CUT.

Keywords: Object-oriented software testing, implementation-based testing, class abstraction.

*School of Computer Science, 11200 S.W 8th Street Miami, FL 33199, USA. Tel.: 305 348 2440; Fax: 305 348 3549.
†Typeset author e-mail address in single line.

1. Introduction

Software quality assurance (SQA) continues to be a major challenge in the software industry. Studies have suggested that 50 - 60% of the overall development effort is devoted to quality assurance activities [?]. Although SQA activities should be performed throughout the software development process, the testing of programs continues to be the main approach used to ensure the quality of software. Software systems are becoming larger and more complex, while software testing remains a challenge. Stobie states that although the computation power doubles every 18 months and the size of the software code doubles every seven years testing of the software does not appear to be keeping pace with the size and complexity of the software, even with the increased computational power [?]. In this paper when we refer to software testing we mean: (1) the use of techniques and methods to generate test cases, and (2) deciding whether or not the test cases adequately cover some predetermined test criteria [?,?]. Most of the testing techniques we cite fall into the second category.

To handle the complexities of developing large-scale software systems many developers are using the Object-Oriented (OO) paradigm. The OO paradigm provides several benefits during analysis and design of large-scale software systems, but scores lower in terms of testability when compared to systems developed using the procedural approach. The low testability score for OO software is due mainly to the composition of OO systems exhibiting the features of abstraction, encapsulation, genericity, inheritance, and polymorphism [?,?,?,?]. To address the low testability of OO systems a plethora of OO testing techniques have been developed [?,?,?,?,? ?,?,?,?,?]. We would like to stress that although the number of testing techniques continues to grow, none of the current OO testing techniques attempt to unify existing techniques to produce a superior technique. In addition, many of these testing techniques are not widely used in industry by testing practitioners. Two reasons for this lack of wide acceptance are: (1) many OO testing techniques are not scalable to real software applications, and (2) there is a lack of adequate tool support for OO testing techniques [?].

To address the problem of unification and the lack of widespread use of OO testing techniques, we present a tool that automates the process of matching a list of *Implementation-Based Testing Techniques*, IBTTs, to a *Class Under Test*, CUT. Our tool is based on an OO taxonomy of the classes in a software application [?,?]. This taxonomy can be used to quickly provide detailed information about the characteristics of a class (properties of the data attributes and routines, and class dependencies), and is able to handle large object-oriented systems that more fine-grained analysis tools cannot handle. Our tool not only maps IBTTs to a CUT, but it also allows the testing professional to select those IBTTs available in the current testing environment. Therefore, our tool scales to handle large OO systems in production today and provides the testing professional with the flexibility to select IBTTs depending on the type of application to be tested.

The matching process that we present in this paper identifies those IBTTs that can *suitably test* characteristics of the CUT and, more importantly, provides feedback to the tester that facilitates identification of the characteristics of the CUT that are *not suitably tested* by any of the IBTTs in the list. The matching process used in the tool is based on the mapping algorithm presented in reference [?]. The tool also provides the testing professional with the ability to initialize each IBTT, in the list of IBTTs selected, to the class characteristics that can be *suitably tested*. We provide results that illustrate the importance of the mapping process by using our tool to map three IBTTs to the classes in several benchmark programs. The results strongly support the notion that combining several IBTTs when testing a class does achieve a higher degree of test coverage.

The rest of the paper is organized as follows: in Section 2 we overview implementation-based testing defining the term class characteristics and describing the IBTTs used in our study. In Section 3, we describe the taxonomy of OO classes used in the tool that maps IBTTs to a CUT. In Section 4, we describe the structure of the tool to map IBTTs to a CUT. In Section 5, we describe the mapping process and present an illustrative example showing the application of the tool to a class from a class library. In Section 6, we present the results of an empirical study and discuss the results. In Section 7, we discuss the related work and give concluding remarks in Section 8.

2. Implementation-Based Testing

Implementation-based testing (also referred to as program-based testing) of an OO class is the process of operating a class under specified conditions, observing or recording the results, and making an evaluation of the class based on aspects of its implementation. This definition is based on the IEEE/ANSI definition for software testing [?]. The aspects of the class to be evaluated are specified as a set of test requirements and are measured against a set of corresponding adequacy criteria [?]. The test requirements for *implementation-based testing techniques*, or IBTTs, are usually defined in terms of the class (program) structure [?]. In this section we define the term *Class Characteristics* and overview how IBTTs are used during class-based testing, focusing on three of the IBTTs used in the study presented in this paper.

2.1. *Class Characteristics*

Efforts to adequately test the basic unit of OO software, the class (or class cluster), have resulted in an abundance of IBTTs [?,?,?,?,?] [?,?,?,?,?]. Each IBTT focuses on testing one or more of the properties associated with the features in the class and/or the dependencies between the class and other classes in the software system (*Class Characteristics*). The features of the class include the *attributes* (data items or instance variables) and the *routines* (member functions or methods) [?].

We define the *class characteristics* for a given class C as the properties of the features in C and the dependencies C has with other types (built-in and user-

defined) in the implementation. The properties of the features in C describe how criteria such as types, accessibility, shared class features, polymorphism, dynamic binding, deferred features, exception handling, and concurrency are represented in the attributes and routines of C. The dependencies of C with other types are realized through declarations and definitions of *C's* features and *C's* role in an inheritance hierarchy [?].

2.2. *Testing Techniques*

The IBTTs used to generate test information for a class are usually classified into three categories, these are: (1) test tuple generation, (2) message sequence generation, and (3) test case reuse. Test tuple generation techniques usually obtain test information during program analysis and this information is represented in the form of tuples. These tuples represent either ordered pairs or ordered triples that represent line numbers associated with entities in the program that should be exercise during program testing [?,?,?,? ?,?,?]. Message sequence generation techniques usually involve some combination of program analysis, automatic deduction, and symbolic execution resulting in a sequences of message calls to an object being created [?,?]. Test case reuse techniques use test cases generated to test some part of a program to test other parts of the program that have not yet been tested [?].

In this paper we describe how three IBTTs can be automatically mapped to a class under test (CUT) based on the characteristics of the CUT. The three IBTTS include: (1) *Data Flow* IBTT [?], (2) the *OMEN* (Object Manipulations in addition to using Escape Information) IBTT [?], and (3) *Incremental* IBTT [?]. The *Data Flow* and *OMEN* IBTTs generate test information in the form of test tuples and the *Incremental* test information is the identification of test cases that may be reused. In the following paragraphs we overview each of these IBTTs and identify the class characteristics that can (cannot) be *suitably tested* by each IBTT. Note that these class characteristics are used to initialized the appropriate data structures used in the mapping process *before* the IBTTs are automatically mapped to the classes under test. In this paper the class characteristics that can (cannot) be *suitable tested* were extracted from the cited reference for each IBTT, however, in practice the tester would identify the class characteristics that can (cannot) be *suitably tested* by each IBTT before the automated process begins. Weiss supports this approach by stating that the notion of deciding which IBTT is "better" should be left to the tester [?].

The Data Flow IBTT uses a class control flow graph (CCFG) to generate *intra-method, inter-method* and *intra-class* def-use pairs, then evaluates the coverage for a given test set based on the all-uses test adequacy criterion [?]. The Data Flow IBTT identifies the def-use pairs for variables of primitive types e.g., ints. These variables can occur as attributes of a class, new attributes of a derived class, and locals (variables and parameters) of routines. The drawbacks of using the Data Flow IBTT include: (1) missing some intra-method, inter-method or intra-class

def-use pairs resulting from specific aliases (complex variables e.g., arrays, structs and pointers), and (2) the handling of specific object oriented features such as polymorphism and dynamic binding [?].

The OMEN IBTT uses data-flow analysis on object manipulations to generate test tuples of the form *object-name(store, load, object creation site)*, then evaluates the coverage for a given test set based on whether or not these tuples were exercised [?]. OMEN generates test tuples for inter-class testing and presents a technique to deal with the problem of aliasing. The drawbacks of this approach are: (1) it does not consider variables of primitive data types (including pointers to primitive types), and (2) individual elements of aggregated objects, such as arrays, created at one site are considered as one object [?].

The Incremental IBTT reuses test sets created for the class at the root of the inheritance hierarchy (or any class with descendants) based on derived features [?]. The attributes and routines of each class derived from the root (parent) class are analyzed and classified as one of six types. These types are: *new* and *recursive* for both attributes and routines, *redefined, virtual-new, virtual-recursive,* and *virtual-redefined* for routines [?]. New features are those features declared in the derived class. Recursive features are those features inherited unchanged from the parent class. Redefined (or overridden) routines define a new implementation for the signature of the routine defined in the parent class. The virtual and non-virtual modifiers represents dynamic and static binding of the routines respectively. Depending on the type of feature in the derived class: (1) new test cases are generated (new or virtual-new features), (2) the test cases from the test history for that inherited feature are totally or partially reused (redefined and virtual-redefined routines), or (3) the feature is considered to be properly tested in the parent and none of the tests are reused (recursive or virtual-recursive features). The Incremental IBTT cannot be applied to classes that are at the root of an inheritance hierarchy or those classes that are not part of an inheritance hierarchy.

3. A Taxonomy of OO Classes

Clarke et al. [?,?] propose a taxonomy of OO classes that is used to succinctly abstract the characteristics of an OO class. The taxonomy of OO classes is essential to the process of automatically mapping IBTTs to classes. Clarke et al. [?] defines a taxonomy of OO classes as follows:

Definition 1: *Taxonomy of OO Classes.* A *taxonomy of OO classes T*, classifies an OO class *C* into a group based on the dependencies *C* has with other types (built-in and user-defined) in the software application. The dependencies of *C* with other types are realized through declarations and definitions of *C's* features and *C's* role in an inheritance hierarchy. □

The properties of the taxonomy include: (1) it can be used to catalog a class written in virtually any OO language, (2) it partitions the set of all OO classes into the finite set of mutually exclusive groups (taxa), and (3) the groups of classes are

Table 1. Descriptors (core and add-on) and type families used in a cataloged entry. The descriptors in parentheses are the add-on descriptors used to describe the characteristics of a class peculiar to C++.

Descriptors			Type Families	
Nomenclature	*Attributes*	*Routines*		
(Nested)	New	(Constant)	NA	no type
(Multi-Parents)	Recursive	New	P	primitive type
(Friend)	Concurrent	Recursive	P*	reference to P
(Has-Friend)	Polymorphic	Redefined	U	user-defined type
Generic	Private	Concurrent	U*	reference to U
Concurrent	Protected	Synchronized	L	library
Abstract	Public	Exception-R	L*	reference to L
Inheritance-free	Constant	Exception-H	A	any type (generics)
Parent	Static	Has-Polymorphic	A*	reference to A
External Child	-	Non-Virtual	$m < n >$	parameterized type
Internal Child	-	Virtual	$m < n >^*$	reference to
-	-	Deferred		parameterized type
-	-	Private	where $m \in$ {U, L}	
-	-	Protected	n is any combination of	
-	-	Public	{P, P*, U, U*, L, L*, A, A*}	
-	-	Static	-	

described in an unambiguous manner using a regular grammar. Using the taxonomy to catalog an OO class generates a *cataloged entry* containing the name of the class, the group the class belongs to, and the subgroups representing the attributes and routines.

3.1. *Cataloged Entry*

A cataloged entry [?] is defined as a 5-tuple consisting of: (1) *Class Name* (2) *Nomenclature Component* - the group (or *taxon*) containing the a class, (3) *Attributes Component* - a list of entries representing the subgroups attributes, (4) *Routines Component* - a list of entries representing the routines, and (5) *Feature Classification Component* - a list summarizing the inherited features of the class. Each *component entry* consists of two parts: (1) a *modifier* - describing the properties of the class and its features (attributes and routines), and (2) the *type families* - types associated with the class. A modifier consists of a list of descriptors *(core and add-on)* representing the class characteristics. The core descriptors represent class characteristics found in most OO languages and the add-ons descriptors represent characteristics peculiar to a given language. Each add-on descriptor used in a component entry is enclosed in a pair of parentheses to distinguish it form the core descriptors.

Table 1 lists the descriptors and type families used in the component entries. Columns 1, 2, and 3 in Table 1 show the descriptors used in the modifier part of

```
1  class vBaseItem{
2  public:
3    vBaseItem(const vBaseItem& b);
4    virtual ~vBaseItem();
5    const Widget vHandle() {
6      return _vHandle; }
7    const char* name() {
8      return (const char*) _name;}
9  protected:
10   char* _name;
11   Widget _vHandle;
12   int _copied;
13   vBaseItem( char* name);};
14
15 vBaseItem::vBaseItem(const
16                      char* name){
17   _vHandle = NULL;
18   _name = new char[strlen(name)+1];
19   strcpy(_name,name);
20   _copied = 0;}
21
22 vBaseItem::vBaseItem(const
23                      vBaseItem& b){
24   _name = b._name;
25   _vHandle = b._vHandle;
26   _copied = 1;}
27
28 vBaseItem::~vBaseItem(){
29   if (_copied){
30     // Debugging system call
31     return;}
32   if (_vHandle){
33     XtDestroyWidget(_vHandle);
34     _vHandle = NULL;}
35   delete [] _name;}
```

(a)

Class: vBaseItem

Nomenclature: *Parent Families P P* U**

Feature Properties

Attributes:

[2] Protected Family P
`Widget vBaseItem::_vHandle`
`int vBaseItem::_copied`

*[1] Protected Family P**
`char* vBaseItem::_name`

Routines:

*[1] Has_Polymorphic Non−Virtual Public Family U**
`vBaseItem::vBaseItem(const vBaseItem&)`

[1] Virtual Public Family NA
`vBaseItem::~vBaseItem()`

[2] Non−Virtual Public Family NA
`Widget vBaseItem::vHandle()`
`char vBaseItem::name()`

*[1] Non−Virtual Protected Family P**
`vBaseItem::vBaseItem(char*)`

Feature Classification:

None

(b)

Fig. 1. (a) C++ code for class vBaseItem from the V GUI library. (b) Cataloged entry for class vBaseItem.

the component entries in the Nomenclature, Attributes and Routines components respectively. The descriptors are assigned names that intuitively reflect the characteristic they capture. For example, a class is cataloged as *Inheritance-free* (Column 1, Row 8 of Table 1) if it is not a part of an inheritance hierarchy. The add-on descriptors for the C++ language are shown enclosed in parentheses in Columns 1 and 3 of Table 1. Column 4 shows the types families used in the Nomenclature, Attributes and Routines component entries. A detail explanation of the descriptors and type families are provided in reference [?].

3.2. *Illustrative Example*

Figure 1(a) shows the C++ code for the class vBaseItem and Figure 1(b) the cataloged entry for vBaseItem. The class vBaseItem, Figure 1(a), is taken from the V GUI Library [?] a multi-platform C++ graphical interface framework to facilitate construction of GUI applications. Class vBaseItem declares three attributes and five routines. The cataloged entry in Figure 1(b) presents a succinct summary of all