Coupling-Based Testing Techniques for Object-Oriented Languages

Latika Kharb and Rajender Singh*

Research Scholar, Reader*
Department of Computer Science & Applications, M. D. University, Rohtak, Haryana, INDIA

ABSTRACT
In procedural languages, the unit of integration is module/procedures; but in object-oriented (OO) languages the integration mechanism used is inheritance; and to be more precise inheritance and polymorphism are the key characteristics that distinguish an OO language from procedural languages. An interesting area of investigation that remains open is whether or not coupling based testing techniques would be effective in the presence of multiple threads. Coupling based testing techniques can be extended to detect faults that result from the polymorphic relationships among components in an OO program. Further, these techniques prove to be effective testing strategy for OO programs that use inheritance and polymorphism. The paper is organized as follows. We’ll review the faults that are peculiar to OO programs because of inheritance and polymorphism and later on, we’ll present the concepts of coupling-based testing applied to object-oriented programs through CBAT, a tool that validates the coupling-based testing applied to object-oriented programs. In this paper, we’ve discussed about a Coupling-Based Analysis tool called CBAT that supports the coupling-based testing of OO programs and could also be used as a general-purpose research tool for conducting program analysis. We have proposed that there are a number of enhancements and modifications that need to be made in CBAT.

KEYWORDS
Integration, CBAT, polymorphic relationships, effective testing strategy, inheritance.

1. ISSUES IN OO TESTING
In procedural languages, the unit of integration is module/procedures; but in OO languages the integration mechanism used is inheritance; and to be more precise inheritance and polymorphism are the key characteristics that distinguish an OO language from procedural languages. Traditional testing techniques do not work effectively for OO software, at least in the sense that they are not capable of detecting the faults that programmer makes in OO programs.

1.1. INTRODUCTION
The emphasis in OO languages is on defining abstractions that model concepts relative to some problem and solution domain. These abstractions appear in the language as user defined types that have both state and behavior. Unfortunately, while the use of abstract types often results in a design of higher quality, the level of testing effort required to achieve a desired level of quality can increase. This is due to inherent complexity in the nature of the relationships found in OO languages. The compositional relationships of inheritance and aggregation, combined with the power of polymorphism, increase the difficulty in detecting faults that result from the integration of components to form new types. This is due to the differences in how component-integration occurs in OO languages.

1.2. TESTING ISSUES UNIQUE TO OBJECT-ORIENTED SOFTWARE
A number of issues associated with OO software that is not relevant in systems written using procedure-oriented languages and many researchers have made the assertion that contrary to many widely held beliefs, OO language features actually increase the effort required to achieve adequate testing and not all forms of traditional testing techniques are applicable or effective in testing OO software. Binder observes that inheritance and polymorphism present opportunities for the commission of errors that simply do not exist in procedural languages. There are a number of testing issues that are unique to object-oriented software. Harrold and Rothermel [1] define three levels of testing:

(1) intra-method testing, in which tests are constructed for individual methods;
(2) inter-method testing, in which multiple methods within a class are tested in concert; and
(3) intra-class testing, in which tests are constructed for a single class, usually as sequences of calls to methods within the class.

Gallagher and Offutt [2] added inter-class testing, in which more than one class is tested at the same time. So, much of the early research in object-oriented testing focused on the inter-method and intra-class levels. Later, researchers focused on the testing of interactions between single classes and their users [3] and system-level testing of OO software. Most research in OO testing has focused on one of two problems. One is the ordering in which classes should be integrated and tested [4] and the other is developing techniques and coverage criteria for selecting tests. But, the problems associated with the essential language features of inheritance and polymorphism cannot be addressed at the inter-method or intra-class levels. These require multiple classes that are coupled through inheritance and polymorphism, which can only be addressed via inter-class testing. In this paper, we’ll specifically focus on problems that
can arise because of the use of inheritance and polymorphism. The fundamental building block in object-oriented programming is the class, which is used to define new types. A class encapsulates state information in a collection of state variables, and has a collection of methods that operate on those state variables. A class defines a type that is used to create objects (instances).

1.3. COUPLING IN OBJECT-ORIENTED PROGRAMS
Coupling was originally proposed to measure design, and the original papers presented up to twelve types of coupling in lists that were ordered in terms of estimated severity. Only three unordered types are needed for testing:

- **parameter coupling**, occur whenever one procedure passes parameters to another.
- **shared data coupling**, occur when two procedures reference the same global variable. and
- **external device coupling**, occur when two procedures access the same external storage device.

Procedure-oriented programs have couplings that occur between procedures in terms of parameters explicitly passed as arguments or through shared global data [5]. Object-oriented programs contain coupling paths that originate at last-declarations in an antecedent method and that terminate at first-uses in a consequent method. There are two general cases in which coupling paths can occur.

- When there is no possibility of polymorphic behavior at the call sites. In this case, the declared type of the context variable specifies the methods that execute.
- When there is a possibility of polymorphic behavior at the call sites. Polymorphic behavior means it is not possible to statically determine which methods will execute. However, it is possible to statically determine all of the possible methods that can execute.

2. CBAT - COUPLING-BASED ANALYSIS TOOL
Testing is the process of executing a component or system on a set of test cases and comparing the actual results with the expected results [6]. CBAT is a research proof of concept tool developed for the purpose of demonstrating the practicality of the coupling-based testing approach for typed OO languages.

2.1. CBAT SUPPORT FOR COUPLING-BASED TESTING OF OO PROGRAMS
CBAT satisfies a number of objectives.

2.1.1. CBAT supports the coupling-based testing of OO programs. CBAT includes representations, algorithms, and utilities for producing programs that are instrumented for collecting coverage information for the coupling-based testing criterion.

2.1.2. CBAT is intended to be a general-purpose research tool to support research activities involved in coupling-based testing.

CBAT is intended to provide a general-purpose analysis platform for conducting program analysis and collection of OO metrics. Jin and Offutt's approach, called coupling-based testing (CBT), is an application of data flow testing to the integration level. It requires that programs execute from definitions of a variable in a caller to a call site, and then to uses of the corresponding formal arguments in the called procedure. The execution path from the definition to the use must be definition-clear, that is, the variable must not be redefined along the path.

2.2. COMPONENTS OF CBAT
The principal components include the CBAT core, Analysis Engine, and the Instrumentation Engine.

2.2.1. CBAT CORE
The CBAT core exists of the Parse Tree Generator, and the Class And Method Graph Generator. Each of these components is described below:

2.2.1.1. Parse Tree Generator transforms Java compilation units expressed in source form into a parse tree that is based on the language grammar.

2.2.1.2. Class and method Graph Generator is implemented as a pair of visitor-based utilities that process the parse tree and generate a corresponding Class Graph and Method Graph.

The class graph generator (CGG) walks the produced by the parser and creates instances of Package, class element, interface, variable, and method as the corresponding syntactic elements are encountered. When a method body is found, the method graph generator (MGG) creates the corresponding control tree that represents the individual statements of the method.

2.2.2. ANALYSIS ENGINE
Analysis Engine is a collection of static analysis tools that utilize the information represented in class and method graphs to produce or refine representations to generate analysis information.

2.2.3. INSTRUMENTATION ENGINE
The final component of CBAT is the instrumentation engine that is a collection of utilities that are used to generate instrumented source code.

2.3. REQUIRED ENHANCEMENTS IN CBAT
There are a number of enhancements and modifications that need to be made to CBAT. In particular, these include:

2.3.1. Graphical User Interface
At present, CBAT is driven by a command line interface along with a set of properties maintained in a separate file. Using CBAT for a particular analysis problem requires that a number of steps must be carried out manually. This is both time-consuming and tedious. A graphical user interface would vastly improve the usability of CBAT.
2.3.2. Adding a database backend
CBAT collects an enormous amount of information during an analysis, all of which is held in memory and discarded when the analysis is complete. Storing this information in database would increase the net performance of CBAT and increase the size of problem that can be handled.

2.3.3. Support for Additional Languages
CBAT was developed for analyzing programs written in Java. To support a new language requires that a new language - specific front-end be provided.

CONCLUSION
In this paper, our approach is to state the effectiveness of test programs that make use of inheritance and polymorphism by exploring the aspects of CBT approach in lieu of object oriented software development. With this approach, professional testers will have a verifiable means of testing the work products produced by developers and a means of targeting specific types of faults peculiar to OO software. Consumers of software-based and software embedded products will also benefit by receiving products that are of higher quality. To conclude, we aim to provide knowledge about a way to testers and developers for judging the test requirements in context of CBAT.

FUTURE SCOPE
With this approach, professional testers will have a verifiable means of testing the work products produced by developers and a means of targeting specific types of faults peculiar to OO software. Consumers of software-based and software embedded products will also benefit by receiving products that are of higher quality. This discussion will surely benefit the testers and developers for providing a specific set of patterns to be used as a guide in production of test cases and thus provide a foundation for future researchers in understanding of how failures are manifested in object oriented programs.

REFERENCES