Testing Based On Object Oriented Software

Reshma Ajitkumar Shinde[1]

Abstract: **This paper deals with design and development of an automated testing tool for Object Oriented Software. By an automated testing tool, it's mean a tool that automates a part of the testing process. It can include one or more of the following processes: test strategy generation, test case generation, test case execution, test data generation, reporting and logging results. By object-oriented software we mean a software designed using OO approach and implemented using a OO language. Testing of OO software is different from testing software created using procedural languages. Several new challenges are posed. In the past most of the methods for testing OO software were just a simple extension of existing methods for conventional software. However, they have been shown to be not very appropriate. Hence, new techniques have been developed. This thesis work has mainly focused on testing design specifications for OO software. As described later, there is a lack of specification-based testing tools for OO software. An advantage of testing software specifications as compared to program code is that specifications are generally correct whereas code is flawed. Moreover, with software engineering principles firmly established in the industry, most of the software developed nowadays follow all the steps of Software Development Life Cycle (SDLC). For this work, UML specifications created in Rational Rose are taken. UML has become the de-facto standard for analysis and design of OO software.**

Testing is conducted at 3 levels: Unit, Integration and System. At the system level there is no difference between the testing techniques used for OO software and other software created using a procedural language, and hence, conventional techniques can be used. This tool provides features for testing at Unit (Class) level as well as Integration level. Further a maintenance-level component has also been incorporated. Results of applying this tool to sample Rational Rose files have been incorporated, and have been found to be satisfactory.

Keywords: **Class, Object, SDLC, Object-oriented, Testing, Unit, Integration, System, UML, Control flow graph.**

---

1. *Asst. Professor, Sinhgad Institute of Computer Sciences, Pandharpur, India*
reshma.ajitshinde@gmail.com

1. Introduction

Software testing is a phase of SDLC that entails much effort, time and cost. Often, testing phase is the single largest contributor towards the whole development time. Testing can not only uncover bugs in the program, but also flaws in design of the software. To make the testing phase quicker, easier and more efficient, automated testing tools are being used. These tools help in test case generation, reporting results and variance from expected ones (if any), bugs in code and other flaws. Usage of these tools speeds up the testing process and also ensures reduction in the probability of a bug/error being uncovered later. However application of these automated testing tools in software testing has its own disadvantages, namely, learning the tool to use it, adapting it to your purpose, and also the tool may not provide specific functionality which you may desire. Object-oriented testing essentially means testing software developed using object-oriented methodology. The target users for the Testing Tool are mainly software testers and maintainers. As the tools would provide valuable insight into the program's structure and behavior plus automate the testing process to a certain extent, it would be highly useful for testers. Also the tool would be beneficial to maintainers who would like to study change impact (here they will be aided by the program's analysis done by the tool), and perform regression testing. The objectives of developing the Testing Tool for software testers and maintainers are:

(1) to help them understand the structures of, and relations between, the components of an OO program

(2) to give them a systematic method and guidance to perform OO testing and maintenance

(3) to assist them to find better test strategies to reduce their efforts

(4) to facilitate them to prepare test cases and test scenarios , and

(5) to generate test data and to aid them in setting up test harnesses to test specific components.

2. TESTING

By definition, a program is deemed to be *adequately tested* if it has been covered according to the selected criteria. The principle choice is between two divergent forms of test case coverage reported by specification-based and program-based testing.[3] *Specification-based* (or ‘‘black-box’’) testing is what most programmers have in mind when they set out to test their programs. The goal is to determine whether the program meets its functional and non-functional (for example, performance) specifications. The current state of the practice is informal specification, and thus informal determination of coverage of the specification is the norm. For example, tests can be cross-referenced with portions of the design document [2], and a test management tool can make sure that all parts of the design document are covered. Test adequacy determination has been formalized for only a few special cases of specification-based testing — most notably, mathematical subroutines [10].

In contrast to specification-based testing, *program-based* (or ''white-box'') testing implies inspection of the source code of the program and selection of test cases that together cover the program, as opposed to its specification. Various criteria have been proposed for determining whether the program has been covered — for example, whether all statements, branches, control flow paths or data flow paths have been executed. In practice, some intermediate measure such as essential branch coverage or feasible data flow path coverage is most likely to be used, since the number of possibilities might otherwise be infinite or at least infeasibly large.[1] The rationale here is that we should not be confident about the correctness of a program if (reachable) parts of it have never been executed. The two approaches are orthogonal and complimentary. Specification-based testing is weak with respect to formal adequacy criteria, while program-based testing has been extensively studied.[7] On the one hand, specification-based testing tells us how well it meets the specification, but tells us nothing about what part of the program is executed to meet each part of the specification. On the other hand, program-based testing tells us nothing about whether the program meets its intended functionality. Thus, if both approaches are used, program-based testing provides a level of confidence derived from the adequacy criteria that the program has been well tested whereas specification-based testing determines whether in fact the program does what it is supposed to do.

3. METHODOLOGY

For carrying out this paper, following methodology has been adopted:

1. Literature Survey: This involves study of existing testing techniques and strategies, with special emphasis on object-oriented testing.
2. Analysis of Problem: This incorporates analyzing the problem. Out of the literature survey emerged, the right techniques and tactics for object-oriented software testing. Also existing methods have been modified upon where ever necessary.
3. Software tool development: Since the ultimate objective of this paper is to develop an automated testing tool,
   all the steps of software development have been followed:
   (i) Analysis
   (ii) Design
   (iii) Implementation
   (iv) Testing
   (v) Iterative process

- EXISTING TESTING TECHNIQUES SURVEYED:
  1. Black Box Testing
  (i) Random Testing
  (ii) Equivalence Partitioning
  (iii) Boundary Value Analysis
  (iv) State Transition-based Testing
  2. White Box Testing
  (i) Basis Path Testing
  (ii) Loop Testing
  (iii) Mutation Testing
  (iv) Data flow-based Testing

- TESTING TECHNIQUES FOR OBJECT-RIENTED SOFTWARE:
  Certain subset of the testing techniques covered in the study can be favorably applied to object-oriented programs. At various levels of testing of object oriented software, techniques which can be applied are [Pressman]:
  1. Unit Testing
  2. Method Testing
  3. Class Testing
  4. Integration Testing
  5. System Testing

4. CHALLENGES TO TESTING OBJECT-ORIENTED SYSTEMS

A main problem with testing object-oriented systems is that standard testing methodologies may not be useful. Smith and Robson [4] say that current IEEE testing definitions and guidelines cannot be applied blindly to OO testing, because they follow the Von Neuman model of processing. This model describes a passive store with an active processor acting upon the store. It requires that there be an oracle to determine whether or not the program has functioned as required, with comparison of performance against a defined specification." They also present the following definition of the testing process: "The process of exercising the routines provided by an object with the goal of uncovering errors in the implementation of the routines or the state of the object or both." Smith and Robson say that the process of testing OO software is more difficult than the traditional approach, since programs are not executed in a sequential manner. OO components can be combined in an arbitrary order; thus defining test cases becomes a search for the order of routines that will cause an error. Siepmann and Newton agree that the state-based nature of OO systems can have a negative effect on testing.

Siepmann and Newton [2] state that the iterative nature of developing OO systems requires regression testing between iterations. Smith and Robson state that inheritance is problematic, since the only way to test a subclass is to flatten it by collapsing the inheritance structure until it appears to be a single class. When this is done, the testing effort for the super class is not utilized; therefore, duplicated testing takes place.

5. AXIOMS OF TEST DATA ADEQUACY

Weyuker in ‘‘Axiomatizing Software Test Data Adequacy’’[10] developed a general axiomatic theory of test data adequacy and considers various adequacy criteria in the light of these axioms. Recently, in ‘‘The Evaluation of Program-Based Software Test Data Adequacy Criteria’’, Weyuker revises and expands the original set of eight axioms to eleven. The goal of the first paper was to demonstrate that the original axioms are useful in exposing weaknesses in several well-known program-based adequacy criteria. The point of the second paper is to demonstrate the *insufficiency* of the current set of axioms, that is, there are adequacy criteria that meet all eleven axioms but clearly are irrelevant to detecting errors in programs. The contribution of our paper is that, by applying these axioms to object-oriented programming, we expose weaknesses in the common intuition that programs using inherited code require less testing than those written using other paradigms.
The first four axioms state:
· **Applicability**. *For every program, there exists an adequate test set.*
· **Non-Exhaustive Applicability**. *There is a program P and test set T such that P is adequately tested by T, and T is not an exhaustive test set.*
· **Monotonicity**. *If T is adequate for P, and T is a subset of T’ then T’ is adequate for P.*
· **Inadequate Empty Set**. *The empty set is not an adequate test set for any program.*

These (intuitively obvious) axioms apply to all programs independent of which programming language or paradigm is used for implementation, and apply equally to program-based and specification-based testing.

Weyuker’s three new axioms are also intuitively obvious.
· **Renaming**. *Let P be a renaming of Q; then T is adequate for P if and only if T is*
adequate for Q.
· **Complexity**. *For every n, there is a program P, such that P is adequately tested by a size n test set, but not by any size n-1 test set.*
· **Statement Coverage**. *If T is adequate for P, then T causes every executable statement of P to be executed.*

A program P is a *renaming* of Q if P is identical to Q except that all instances of an identifier x of Q have been replaced in P by an identifier y, where y does not appear in Q, or if there is a set of such renamed identifiers. The first two axioms are applicable to both forms of testing; the third applies only to program-based testing. The concepts of renaming, size of test set, and statement depend on the language paradigm, but this is outside the scope of this article.

- ANTIEXTENSIONALITY, GENERAL MULTIPLE CHANGE, ANTIDECOMPOSITION, AND ANTICOMPOSITION AXIOMS

Here there are four remaining axioms: the antiextensionality, general multiple change, antidecomposition and anticomposition axioms. [2] These axioms are concerned with testing various parts of a program in relationship to the whole and *vice versa*, and certain of them apply only to program-based and not to specification based adequacy criteria. They are, in some sense, negative axioms in that they expose inadequacy rather than guarantee adequacy.
- Antiextensionality. If two programs compute the same function (that is, they are *semantically close*), a test set adequate for one is not necessarily adequate for the other. *There are programs P and Q such that P ° Q,* [test set] *T is adequate for P, but T is not adequate for Q.*
This is probably the most surprising of the axioms, partly because our intuition of what it means to adequately test a program is rooted in specification-based testing.[12] In specification-based testing, adequate testing is a function of covering the specification. Since equivalent programs have, by definition, the same specification [12], any test set that is adequate for one must be adequate for the other. However, in program-based testing, adequate testing is a function of covering the source code. Since equivalent programs may have radically different implementations, there is no reason to expect a test set that, for example, executes all the statements of one implementation will execute all the statements of another implementation.

- General Multiple Change. When two programs are syntactically similar (that is, they have the *same shape*), they usually require different test sets.
There are programs P and Q which are the same shape, and a test set T such that T isadequate for P, but T is not adequate for Q.
Weyuker states: ‘‘Two programs are of the *same shape* if one can be transformed into the other by applying the following rules any number of times:
(a) Replace relational operator r1 in a predicate with relational operator r2.
(b) Replace constant c1 in a predicate or assignment statement with constant c2.
(c) Replace arithmetic operator a1 in an assignment statement with arithmetic operator a2.’’

Since an adequate test set for program-based testing may be selected, for example, to force execution of both branches of each conditional statement, new relational operators and/or constants in the predicates may require a different test set to maintain branch coverage. Although this axiom is clearly concerned with the implementation, not the specification, of a program, we could postulate a similar axiom about the syntactic similarity of specifications, as opposed to source code.

- Antidecomposition. Testing a program component in the context of an enclosing program may be adequate with respect to that enclosing program but not necessarily adequate for other uses of the component. *There exists a program P and component Q such that T is adequate for P, T’ is the set of vectors of values that variables can assume on entrance to Q for some t of T, and T’ is not adequate for Q.*

This axiom characterizes a property of adequacy as well as an interesting property of testing — that is, a program can be adequately tested even though it contains unreachable code. But the unreachable code remains untested, adequately or otherwise. The degenerate example is that in which Q is unreachable in P and T’ is the null set. By the Inadequate Empty Set axiom of the previous section, T’ cannot be adequate for Q. In the more typical case, some part of Q is not reachable in P but is reachable in other contexts; hence, T’ will not adequately test Q. While this axiom is written in program-based terms, it is equally applicable to specification-based testing. In particular, the enclosing program P may not utilize all the functionality defined by the specification of Q and thus could not possibly test Q adequately.

- Anticomposition. Adequately testing each individual program component in isolation does not necessarily suffice to adequately test the entire program. Composing two program components results in interactions that cannot arise in isolation.

There exist programs P and Q, and test set T, such that T is adequate for P, and the set of vectors of values that variables can assume on entrance to Q for inputs in T is adequate for Q, but T is not adequate for P;Q. [P;Q is the composition of P and Q.]

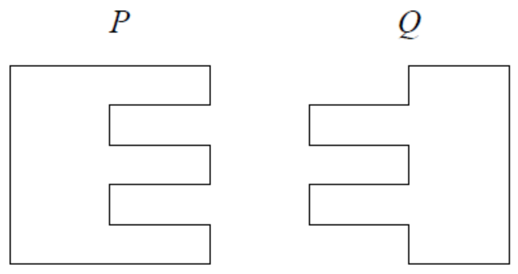

FIG. 1

This axiom is counter-intuitive if we limit our thinking to *sequential* composition of P and Q. Consider instead the composition illustrated in figure 1, which can be interpreted as either P calls Q multiple times or P and Q are mutually recursive. In either case, one has the opportunity to modify the context seen by the other in a more complex manner than could be done using stubs during testing of individual components in isolation. If the composition of P and Q is in fact sequential, then the axiom is still true — just less useful. The proof is by a simple combinatorics argument: If p is the set of paths through P and q is the set of paths through Q, then the set of paths through P;Q may be as large as p ´ q, depending on the form of composition and on reachability as considered by the previous axiom. However, T applied to P;Q generates at most p paths. A larger test set may be needed to induce the full set of paths. This is an issue for specification-based as well as program-based testing when the specification captures only what the program is supposed to do, not including what it is not supposed to do.

6. THE TEST MODEL AND ITS CAPABILITIES

The tools for automated testing is based upon certain models of software/programs and algorithms. This mathematically defined test model, consists of following types of diagrams:
1. the class diagram (object relation diagram)
2. the control flow graph (of a method), and
3. the state transition diagram (of a class)

1. CLASS DIAGRAM

A class diagram or an object relation diagram (ORD) represents the relationships between the various classes and its type. Types of relationships are mainly: inheritance, aggregation, and association. In object oriented programs there are three different relationships between classes They are inheritance, aggregation and association.

2. CONTROL FLOW GRAPH

A control flow graph represents the control structure of a member function and its interface to other member functions so that a tester will know which data is used and/or updated and which other functions are invoked by the member function.

3. STATE TRANSITION DIAGRAM

A STD or an Object State Diagram (OSD) represents the state behavior of an object class. Now the state of a class is embodied in its member variables which are shared among its methods. The OSD shows the various states of a class (various member variable values), and transitions between them (method invocations).

4. BASED ON SOFTWARE DESIGN/SPECIFICATION

These diagrams are taken from the design models prepared as part of Software Development process. UML (Unified Modeling Language) has become the defacto standard for object-oriented analysis and design (OOAD). UML provides features for specifying all the above types of diagrams. Rational Rose Suite is the most widely used.

• COMPONENTS OF THE OO TESTING TOOL

The tool for automated testing of OO programs has the following components/features:

1. GUI
2. Import File Feature
3. Change Impact Identifier for classes
4. Maintenance Tools
5. Logging results
6. Diagram Displayer
7. Class Diagram
8. State Transition Diagram
9. Control Flow Graph
10. Test Tools:
    (i) Test Order generator for testing of classes at integration level
    (ii) Test Case generator for testing classes
11. Basis Path generator for member functions/methods

7. CONCLUSION

This paper dealt with Design and Development of an Automated Testing Tool for OO software. The tool mainly focuses on testing design specifications for OO software. An advantage of testing software specifications as compared to program code is that specifications are generally correct whereas code is flawed. Moreover, with software engineering principles firmly established in the industry, nowadays, while developing software all the steps of Software Development Life Cycle (SDLC) are adhered to. For this work, UML specifications are considered. UML has become the defacto standard for analysis and design of OO software. UML designs created in Rational Rose are used by the tool as input. The main components of this tool are:

1. Test Order Generator for classes
2. Test Case Generator for State-based class testing
3. Change Impact Identification for Classes

Inheritance is one of the primary strengths of object-oriented programming. However, it is precisely because of inheritance that we find problems arising with respect to testing.

· Encapsulation together with inheritance, which intuitively ought to bring a reduction in testing problems, compounds them instead.

· Where non-inheritance languages make the effects of changes explicit, inheritance languages tend to make these effects implicit and dependent on the various underlying, and complicated, inheritance models.

8. REFERENCES

[1] The Journal Of Object Oriented Programming January/February 1990

[2] Jitendra S. Kushwah et al./ Indian Journal of Computer Science and Engineering (IJCSE) ISSN : 0976-5166 Vol.2.

[3] William Howden. Software Engineering and Technology: Functional Program

[4] Frankl, Elaine Weyuker ”An applicable family of data flow testing criteria” IEEE Transactions on Software Engineering, Vol. 14.

[5] Mary Jean Harrold, Gregg Rothermel “Performing data flow testing on classes” December 1994 ACM SIGSOFT Software Engineering Notes , Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering, Volume 19 Issue 5

[6] Takeshi Chusho. ‘‘Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing’’, IEEE Transactions on Software Engineering SE-13:5 (May 1987). pp

[7] Phyllis G. Frankel and Elaine J. Weyuker. ‘‘Data Flow Testing in the Presence of Unexecutable Paths’’, in Workshop on Software Testing, Banff, Canada, July 1987.

[8] David Gelperin and Bill Hetzel. ‘‘The Growth of Software Testing’’, Communications of the ACM 31:6 (June 1988).

[9] Roger S. Pressman. “Software Engineering A Practitioner’s Approach (7th International Edition) 2010.

[10] Elaine J. Weyuker. ‘‘Axiomatizing Software Test Data Adequacy’’, IEEE Transactions on Software Engineering SE-12:12 (December 1986).

[11] Elaine J. Weyuker. ‘‘The Evaluation of Program-Based Software Test Data Adequacy Criteria’’, Communications of the ACM 31:6 (June 1988).

[12] Object-Oriented programs and Testing Dewayne E. Perry Gail E. Kaiser*The Journal Of Object Oriented Programming January/February 1990