Analysis of **ASTOOT** Approach for **O**bject-**O**riented **T**esting

ARPIT GATTANI

The Object-oriented paradigm is today the most favored software developing paradigm in the software industry. Object-oriented paradigm has many unprecedented benefits in the software development cycle but testing Object-oriented software is still a very much challenging task in software engineering community. Lately much required attention is devoted to this field of research. In this paper, we analyze ASTOOT, **A** **S**et of **T**ools for **O**bject-**O**riented **T**esting, approach to Object-oriented testing and discover some limitations in ASTOOT strategy. The paper proposes suggestions to overcome these hindrances and consequently tries to make the ASTOOT approach enhanced.

PAPER TYPE

Case Study

KEYWORDS

Software testing, Object-oriented testing, class under test (CUT), algebraic specification, automated testing, test driver, state charts

---

1 INTRODUCTION

Object oriented paradigm for software development came a long way in the last two decades. Though many Object-oriented analysis, designing and developing methods and techniques have been proposed, relatively little attention is given to Object-oriented testing and maintenance. Conventional testing strategies have been found inadequate for Object-oriented systems. This is because conventional functional testing strategies take only structural flow into the consideration and not the behavior of the objects in different states. The outcome of a method executed by an object often depends on the state of the object at the time of method invocation. It is therefore important for Object-oriented testing techniques to test class methods when the method's receiver is in different states. The state of an object at a particular time can be determined by the sequence of messages received and send by that object at that time. Thus, methods it is very important for object-oriented testing strategy that it should identify sequences of method invocations that are likely to uncover potential defects in the code under test. However, testing methods for conventional software do not provide this kind of information. ASTOOT is aimed to eliminate this defect in conventional testing methods and thus make it adaptable with Object-oriented approach. The paper analyzes a few implementations of the ASTOOT approach. This analysis finds differences in the implementation. The paper provides suggestions to improve the ASTOOT approach based on the study of the various tools for a better implementation.

2 THE ASTOOT CONCEPT

Testing is considered as the most tedious and uninterested part of the software engineering. It takes the maximum amount of time in the software development cycle which is by most considered as the work done with destructive approach. Testing is a NP hard problem. For complete testing, all objects have to be tested in all their different states. The ASTOOT approach interprets the problem as real-life scenario and creates the objects for all the important blocks of the problem. ASTOOT is based on the idea that the natural units to test are classes and testing at the class level in object oriented world can lead to better results. ASTOOT includes tools that allow the automation of the entire test process -- test generation, test driver generation, test execution and test result checking [5]. The format of a test case is a triplet (T_1, T_2, tag) where T_1 and T_2 are traces, and *tag* is "*equivalent*" if T_1 is equivalent to T_2 according to the specification, and "*non equivalent*" otherwise. A trace of a class is a description of a sequence of messages, applied to an object in an initial state. A test case executes by sending each sequence of messages to an object of the class under test, invoking a user-supplied equivalence-checking routine to check whether the objects are in the same abstract state, and then comparing the result of this check to the tag. The test generation tool requires the availability of an algebraic specification of the abstract data type being tested, but the test execution tool can be used without any need for formal specification [5]. Using the test execution tools, case studies involving execution of tens of thousands of test cases, with various sequence lengths, parameters, and combinations of operations can be performed.

2.1 The Approach

ASTOOT approach of testing has the four fundamental steps, a) generation of test cases, b) generation of the program that executes the test cases, c) execution of the test cases and recording the results and d) verification of the actual results with the expected results.

ASTOOT automatically checks if the test results are true or not by including a Boolean parameter in the test case, which is not dependent on the class under test. Test drivers can be automatically generated from class interfaces as they are for different classes. Test cases can be automatically generated by the algebraic expression. Algebraic expression if is not available, it can be generated manually by reasoning about informal specifications [6]. The use of method test sequences derived from sequence constraints is effective. It consists of generating pre-conditions, i.e., a set of requirements that must be met before method is used and post-conditions, specifying the expected property resulting from the method. The method sequence constraints are then derived to generate valid and invalid test sequences [4].

3 THE DOONG AND FRANKL TOOL

The tool designed by Doong and Frankl [5] handles test cases in the restricted format. It has three components: the driver generator, the compiler, and the simplifier. The driver generator takes as input the interface specifications of the class under test (CUT) and some related classes and outputs a test driver. This test driver, when executed, reads the test cases, checks their syntax, executes them, and checks the results. The compiler and simplifier together form an interactive tool for semi-automatically generating test cases from an algebraic specification.

3.1 The test driver generation.

The ASTOOT approach leads to relatively simple test drivers, which operate by reading in test cases of the form (T_1, T_2, tag), one at a time, checking that the sequences are syntactically valid, sending sequences T_1 and T_2 to objects O_1 and O_2 of the CUT, comparing the returned objects of T_1 and T_2 with *EQN*, and checking whether the value returned by *EQN* agrees with *tag*. Drivers are complicated enough that writing them manually is a tedious and error-prone task. Since drivers for different classes are structurally quite similar, it is feasible to write a tool that can generate the test drivers for different classes automatically. This driver generator is a special-purpose parser generator, which, based on the syntax described in the class interfaces, generates test drivers that parse test cases, and also executes and checks them.

3.2 The test generation.

The test generation component of ASTOOT consists of two parts - the compiler and the simplifier. These two are based on ADT tree. The compiler reads the LOBAS specification, performs a syntactic and semantic check on the specification, and then translates each axiom into a pair of ADT tree. The nodes of the ADT tree represent operations along with their arguments. A possible state is represented by each path from root to leaf in ADT tree.

Simplifier searches the ADT tree to find an axiom with a left-hand side that matches some partial path of the ADT tree. If such an axiom is found, then the right-hand side of the axiom replaces the partial branch. The above operations are repeated till there is a matching axiom.

Simplifier works on a property that it is essential for the set of axioms in the specification to be convergent. This means that the axioms must have the properties of finite and unique termination. This ensures that process of simplification will not go into infinite loop and two terminating sequences starting from the same operation sequence have the same results [2].

4 Automated Testing of Classes

4.1 State-based testing

The Method of Automatic Class Testing (MACT) is a user's state-based testing method. It has four main phases - test data generation, inspection tree generation, test execution, and test results inspection [10].

1. **Test Data Generation**: A test data generator is a tool, which assists testers in the generation of test data for software. The user selects an execution paths or input data for the classes under test. In MACT, the test cases are manually

produced according to the state / transition tree. The test data generator automatically generates a test data file by reading the test cases written.

2. **Inspection Tree Generation**: The inspection tree generator in MACT generates various multi-way inspection trees according to the various state / transition trees. The inspection tree generator is designed to modify the source code of the inspect tree pattern into a program which the testers use to duplicate the behavior of a state chart.
3. **Test Execution**: Test execution feeds the test data to test the program and collects the test result data in MACT. Each test result record in a test result file corresponds to the each record in the relating test data file.
4. **Inspection and Certification**: Test results inspection process certify whether the class under test are fault-free or not by parsing the test result record one by one. This can be observed by comparing the test result with the expected result.

4.2 Based on Edison Design

This technique is developed to found state dependent failures, i.e. failures that can be clear only when an instance is in a certain state before executing a method. A sequence of message calls that bring the object under test under different states is than executed. State independent faults are found by using the same instance variables for the same pair of message calls. The identified sequences represent the test cases for the target class. Thus statements involving non-scalar instance variables can be easily represented in terms of definitions and uses of such variables; most execution conditions can be solved with existing automated constraint solvers is described in [1]. A CCFG, Class Control Flow Graph, generator parses the source code of a class and generates the corresponding CCFG. All constructs but exception handling are also represented in the corresponding CCFG. Three main phases used in the paper for automated testing of the class are;

1. **Data Flow Analysis**: The data flow analyzer identifies du-pairs for instance variables of the CUT starting from the CCFG output by the CCFG Generator. A du-pair consists of two nodes where both nodes contained in the CUT.
2. **Symbolic Execution**: The symbolic executor computes conditions for path execution and variable definitions. In particular, it computes the conditions associated with the execution of paths within a method, the relationship between inputs and outputs of a method, the set of variables defined along each path, and the conditions associated with the execution of paths leading to definitions and uses within a method.
3. **Sequence Generation**: This tool will generate method sequences using automated reasoning. It is based on the solution of the constraints generated with symbolic execution on paths defined by data. It is also possible to design tools based on the model-based ideas rather than algebraic specifications. One such strategy involves generating a flow graph from a component's specification and then applying white-box techniques to the graph. By using pre-condition, post-condition and invariant checks wrapped around CUT, fault detection ratios compared with white-box techniques can be achieved [1].

6 LIMITATIONS OF ASTOOT AND SUGGESTIONS FOR FURTHUR ENHANCEMENTS

This section summarizes the findings from the above analysis and proposes a few suggestions for improvements of ASTOOT approach.

6.1 Language Dependence

The biggest limitation of the ASTOOT approach is its language dependence. Tool proposed by Frankl *at el. works* with Eiffel. Daistish implementation was on C++ and Eiffel. Though it is been stated by most that the approach can be extended to other object-oriented languages as well, no strong way to achieve this is described. One efficient way to achieve this is to design a tool, which can generate the test driver, test cases and test result analyzer differently for different languages. This tool can be written in any language but the output would be either a working code or a pseudo-code for the tools to test classes in other languages. Another approach to this problem can be to generate a tool, which can convert the generated test driver and test cases in Eiffel or C++ to the required language. Input for the tool would be generated test driver and test cases in this approach.

6.2 Inappropriate interactions errors.

In Object-oriented testing strategies like ASTOOT, which is based on the message exchange between objects, errors like inappropriate interactions are not fully avoidable. It is possible that though classes individually interact with each other successfully but in a system as whole they don't perform as needed. One way to overcome this situation is to test classes in cluster, where cluster consists of all the classes involve in an interaction [9]. Automated test executor based on the scripts can be used for the object–oriented languages, which includes the unexpected interaction due to inheritance [9]. Another technique which can be applied to overcome this problem is after testing class individually, a state-chart based testing can be performed on the class those involves in more than one interactions.

6.3 Inability to define correct current state.

In ASTOOT approach, two messages sequences are send to the object of the class under test to check whether a method ends at a correct state or not. It doesn't define the current correct state of that object while under test. This can be achieved if we can define the correct state of the object by the parameters the sequence of messages takes. A well-formed definition of the correct state thus can be formed. Hence this approach can be practically implemented to any situation.

6.4 Need for the definition of algebraic specification.

Algebraic specifications of class under test play a big role in successful testing by ASTOOT. Tester need to write his own specifications if there aren't any, and this can prove to be an very inefficient as well as highly time consuming to write those specification by hand. To over come this limitation, ASTOOT can be extended with a specification tool generator, which can generate the algebraic specifications if there aren't any specified already.

6.5 Inefficiency with the ADT trees.

ASTOOT stores the operations and the attributes for the test class as the node of the ADT tree. Any path from the root to leaf represents a possible state for the object of the CUT. Using ADTs to store the operations and attributes can sometimes lead to misleading representations of the state for the CUT as it is hard to get the node at given instance in between root and leaf while the testing. Also it makes difficult the identification of the method that the CUT sends or receives in a particular state.

6.6 Axiom-based test case selection to guarantee effectiveness.

Tool proposed by the Frankl *at el* works on the concept that given an algebraic specification, the equivalent terms should give observably equivalent objects, and offer general heuristics on the selection of equivalent terms for testing. Only limited empirical results support this approach and this approach doesn't have strong theoretical basis [2]. It provides no guarantee of effectiveness. So it would be better to adapt the use of this approach to include axiom-based specifications which is much more efficient.

7 CONCLUSION

This paper explains the ASTOOT approach for object-oriented testing. It also describes the three popular methods by which ASTOOT can be applied to an Object-oriented system. Our study shows that ASTOOT proves to be a very efficient and effective approach for most cases except where algebraic specification are hard to specify. Paper also proposes some suggestions to enhance the performance for ASTOOT approach for existing as well as new Object-oriented languages. Future works can concentrate for making ASTOOT approach capable of testing even semi-object-oriented systems and for systems without algebraic specifications.

REFERENCES

[1] BUY, U., ORSO, A., AND PEZZE, M. 2000. Automated Testing of Classes. *In Proc. of the International Symposium on Software Testing and Analysis,* Portland, Oregon, August 2000, 39-48.

The paper generates the sequence of methods calls for the class under test by the data flow analysis, symbolic execution, and automated deduction. Method proposed in this paper automatically generates information relevant to testing even when there isn’t complete symbolic execution and automated deduction information.

[2] CHAN F.T., CHEN, H.Y., AND TSE, T.H. 1995. An Axiom-Based Test Case Selection Strategy for Object-Oriented Programs. *In Software Quality and*

Productivity: Theory, Practice, Education, and Training, M. LEE, B.-Z. BARTA, AND P. JULIFF, Eds. Chapman and Hall, London, 107-114.

The paper defines the concept of a fundamental pair as a pair of equivalent terms which are formed by replacing all the variables on both sides of an axiom by normal forms. The paper describes the importance of concentrating on just the fundamental pairs for testing classes and presents its importance over the equivalent terms based testing.

[3] CHEN, H.Y., CHEN, T.Y., AND TSE, T.H. 2001. TACCLE: A Methodology for Object-Oriented Software Testing at the Class and Cluster Levels. In *ACM Transactions on Software Engineering and Methodology*, Vol. 10, No. 4, January 2001, 56-109.

The paper deals with object oriented testing at class and cluster levels. Method proposed in the paper tests the classes both specification defined by behavioral equivalence and non-behavioral equivalence. This paper also discusses the black-box testing at the cluster level.

[4] DANIELS, F.J., TAI, K.C. 1999. Measuring the Effectiveness of Method Test Sequences Derived from Sequence Constraints. In *Proceedings of the TOOLS (Technology of Object-Oriented Languages and Systems)*, Santa Barbara, California, August 1999, 74-83.

The paper proposes an approach to intra-class testing by executing sequences of class methods that are derived from sequence constraints and evaluate results for correctness. Paper also presents an empirical evaluation of different method sequence generation approaches and their effectiveness in finding faults.

[5] DOONG, R.K., AND FRANKL, P.G. 1994. The ASTOOT Approach to Testing Object-Oriented Programs. In *ACM Transactions on Software Engineering and Methodology*, Vol. 3, No. 2, April 1994, 101-130.

The paper describes a new technique to test Object-oriented units. Paper also describes a couple of case studies based on the approach it proposes. It proposes three tools, test driver generator, compiler and simplifier, to perform the automated testing.

[6] EDWARDS, S. H. 2001. A Framework for Practical, Automated Black-Box Testing of Component-Based Software. *Software Testing, Verification and Reliability*, Vol. 11, Issue 2, 2001, 97-111.

The paper describes a strategy for automatic black-box testing of software components. The strategy includes automatic generation of component test drivers, automatic generation of black-box test data, and automatic or semi-automatic generation of component wrapper that serve as test oracles. The paper shows that automation of the testing is practically possible to certain level.

[7] HUGHES, M., AND STOTTS, D. 1996. Daistish Systematic Algebraic Testing for Object-oriented Programs in the Presence of Side effects. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, San Diego, California, January 1996, 53-61.

The paper describes the Daistish tool, a tool based approach to the Object-oriented testing. It is similar to the ASTOOT approach. It creates test drivers for programs in languages. It is quite effective for the software which is written in the languages which doesn't support the ADT trees efficiently. The implementation for the approach in this paper is done in both Eiffel and C++.

[8] MCLEAN, J. 1984. A Formal Method for the Abstract Specification of Software. *Journal of the Association for Computing Machinery*, Vol. 31, No. 3, July 1984, 600-627.

The paper defines the formal specification and its types and how these types gave rise to each approach of testing. Along with this, paper also explains the requirements of good specifications.

[9] MURPHY G.C. Automating Cluster and Class Test Execution: Useful for Smalltalk Applications?

The report describes the testing process based on the testing at class and cluster levels. Report used the concepts like code walkthroughs, compilation diagnostics, system testing, and automated support for class and cluster testing. It also describes the approach of script based testing on class and cluster.

[10] PARRINGTON, N., STOBART, S., AND TSAI, B.Y. 1997. A Method for Automatic Class Testing (MACT) Object-Oriented Programs Using A State-

Based Testing Tool. 5th *European Conference Software Testing Analysis & Review*, Edinburgh, November 1997.

The paper describes a method for automated class testing. It proposes a state-based testing which in turn generates various inspection trees. It also introduces test results storage on MACT and verification with expected results.

[11] ROBSON, D.J., AND TURNER, C.D. 1993. The Testing of Object-Oriented Programs. *Technical Report: TR13/92, Computer Science Division School of Engineering and Computer Science (SECS)*, University of Durham, England, February 1993.

The report analyses the traditional structural testing strategies with Object-oriented strategies for unit and integration testing. It explains the need for state-based testing for the Object-oriented systems which separately tests the data members and methods calls for a class.