Object-Oriented Software Testing — Some Research and Development

David C. Kung and Pei Hsia
Computer Science and Engineering Dept.
The Univ. of Texas at Arlington

Yasufumi Toyoshima, Cris Chen, Jerry Gao
Fujitsu Network Communication Systems

Abstract

It is widely accepted that the OO paradigm will significantly increase software reusability, extendibility, interoperability, and reliability. This is also true for high assurance systems engineering, provided that the systems are tested adequately. Software testing is an important software quality assurance activity to ensure that the benefits of OO programming will be realized. OO software testing has to deal with new problems introduced by the powerful new features of OO languages. The objective of this article is to review some of the existing researches in OO software testing, in particular, the research at the University of Texas at Arlington.

1 Introduction

OO software testing has to deal with new problems introduced by the powerful OO features such as encapsulation, inheritance, polymorphism, and dynamic binding. The objective of this article is to review some of the existing researches and applications of OO software testing. In particular, we will focus more on describing our research on OO software testing at the University of Texas at Arlington. Although none of the results was developed with high assurance systems in mind, many of them still can be applied to high assurance systems.

2 OO Testing Problems

D. E. Perry and G. E. Kaiser [19], discusses the problems from a theoretical point of view. They revisited some of the test adequacy axioms originally proposed by Weyuko [25] [26]: **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. **General Multiple Change** – When two programs are syntactically similar (i.e., one can be obtained from the other by changing constants and/or relational/arithmetic operators) they usually require different test sets. **Antidecomposition** – Testing a program 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. **Anticomposition** – Adequately testing each individual program component in isolation does not necessarily suffice to adequately test the the entire program. Composing two program components results in interaction that cannot arise in isolation.

They reached a surprising conclusion. That is, 1) when a subclass or superclass is added to a class, the inherited methods must be retested in this newly formed context; 2) even if the overriding and overridden methods are semantically similar, there is a need to retest the classes in the context of overriding and overridden methods; and 3) if the order of specification of superclasses of a subclass is changed, the subclass must be retested even though only syntactic changes are made.

Smith et al. [22] considered problems involved in testing classes, abstract classes, message passing, concurrency, inheritance, polymorphism and template classes. For example, how to test a class, an abstract class, or a template class. Another problem is the concept of control flow through a conventional program does not map readily to an OO program. Flow of control in OO programs may be thought of a message passing from one object to another, causing the receiving object to perform some operation, which may be an examination or alteration, of its state. In order to test such programs, when there is no conceptual input/process/output, it is probably more appropriate to specify how the object's state will change under

certain conditions. If one object sends two messages to two other objects, the two can respond to it concurrently. The complexity of testing such systems, considering the possible time-dependent interactions between objects, is potentially greater than that for normal sequential OO programs.

Wilde and Huit [27] addressed problems in maintaining an OO system and proposes potential solutions to these problems. These include problems of dynamic binding, object dependencies, dispersed program structure, control of polymorphism, high-level understanding, and detailed code understanding. In fact, all of these are problems in the testing and regression testing processes. For example, dynamic binding implies that the code that implements a given function is unknown until run time. Therefore, static analysis cannot be used to identify precisely the dependencies in the program. And hence, it is difficult for a tester to prepare test stubs as well as identifying the change impact in regression testing.

The dependencies occurring in conventional systems are:

1. data dependencies between variables;
2. calling dependencies between modules;
3. functional dependencies between a module and the variables it computes;
4. definitional dependencies between a variable and its type.

OO Systems have additional dependencies:

1. class to class dependencies;
2. class to method dependencies;
3. class to message dependencies;
4. class to variable dependencies;
5. method to variable dependencies;
6. method to message dependencies; and
7. method to method dependencies.

Environments for maintaining OOP's need to provide ways of browsing these different kinds of relationships. The multi-dimensional nature of interconnections will make it very difficult to use listing or text screen based systems for program understanding.

Kung et al. [15] summarized OO testing problems to be: 1) the understanding problem; 2) the complex interdependency problem; 3) the object state behavior testing problem; and 4) the tool support problem.

The understanding problem is introduced by the encapsulation and information hiding features. The dependency problem was caused by the complex relationships that exist in an OO program. Objects have states and state dependent behaviors. That is, the effect of an operation on an object depends also on the state of the object and may change the state of the object. Thus, the combined effect of the operations must be tested [2] [8] [12].

3 OO Software Test Strategy

A test strategy can be defined as the order to unit testing and integration testing of the classes in an OO program. The test order problem for the classes in an OO program can be stated as finding an order to test the classes so that the effort required is minimum.

One pioneering work on OO testing strategy is due to Harrold et al. [5]. They described a class testing methodology that utilizes the hierarchical nature of classes related by their inheritance relationships to reduce test overhead. This is accomplished by reusing the test information for a parent class and incrementally updating it to guide the testing of the subclasses. Experiments show that significant amount of effort can be saved for well designed hierarchies with a large amount of functionality defined at the top level and modifications and additions are made at lower levels. Additional savings include the reuse of the parent class' test suites to test the methods of the subclasses.

The methodology consists of the following steps:

Step 1. Initially, base classes having no parents are chosen and a test suite is designed that tests each member function individually and also the interactions among member functions.

Step 2. A testing history associates each test case with the attributes it tests. In addition to inheriting attributes from its parents, a newly defined subclass "inherits" its parent's testing history. Just as a subclass is derived from its parent class, a subclass's testing history is derived from the testing history of its parent class.

Step 3. The inherited testing history is incrementally updated to reflect differences from the parent and the result is a testing history for the subclass. A subclass's testing history guides the execution of the test cases since it indicates which test cases must be run to test the subclass.

Step 4. With this technique, new attributes can be easily identified in the subclass that must be tested along with inherited attributes that must be retested.

Step 5. The inherited attributes are retested in the context of the subclass by identifying and testing their interactions with newly defined attributes in the subclass.

Step 6. The test cases in the parent class's test suite that can be reused to validate the subclass and attributes of the subclass which require new test cases can also be identified in the process.

The authors also used a variety of existing C++ class hierarchies as experiments to determine the savings in testing using their technique. The InterViews 2.6, a library of graphics interface classes, is used. The result shows that significant amount of effort would be saved with this technique. Since many methods that must be retested will reuse the test cases of their parent, additional saving can be obtained through test case reuse.

Another work on OO software test strategy was proposed by Kung et al. [14]. The test strategy is defined to be an order to test the classes such that the effort required to construct the test stubs is minimum. To compute the strategy, an Object Relation Diagram (ORD) is used. An ORD displays the various relationships among the object classes, including *inheritance, aggregation, association, instantiation (of a template class), nested*, and *use (to instantiate a template class)*. The ORD is generated by a reverse engineering tool which identifies these relationships from the source code of an OO program and displays the classes and their relationships diagramatically.

A solution to the test order problem must consider two cases:

1. The $ORD = (V, L, E)$ is an acyclic digraph, meaning that there exists no cycle in the digraph. A cycle is a directed path leading from one node, traversing directed edges, back to itself.

2. The $ORD = (V, L, E)$ is a cyclic digraph, meaning that there exists one or more cycles.

In the first case, the test order is simply the topological sorting [1] of the set of classes using the dependence relation defined by the inheritance, aggregation, and association relationships. The computational complexity is the number of classes in the OO program since each node needs to be visited only once. The effort required to construct the test stubs is zero and hence it is minimum.

The solution to case 2 is not so trivial since topological sorting cannot be applied to cyclic digraphs. A simple solution is to convert the cyclic digraph into an acyclic diagraph by treating each strongly connected component[1] as a composite vertex. Topological sorting can then be applied to the resulting acyclic digraph to yield a major test order for testing the unit vertices and composite vertices. Since each strongly connected component is a cyclic digraph, certain association edges are removed to convert it into an acyclic digraph so that topological sorting can be applied to obtain a minor test order to test the vertices in the strongly connected components. The edges to be removed are those that will result in minimum effort to construct the test stubs.

An experiment using the InterViews library indicated that 316 person-hours or 8 person-weeks would be required if a random order is used. In comparison, the optimal test order would result in 93% saving in terms of test effort.

The ORD and test order are not only useful for conducting class unit testing but also provide a detailed road map for integration testing. That is, after unit testing, the classes are integrated according to the test order. In this way, the effort required to construct test stubs and test drivers will be reduced to a minimum. With the InterViews library their experiment showed that 96% saving can be achieved.

4 Unit Testing and Integration Testing

Although there are many research results addressing unit test problems, most of them focused on function-oriented/procedure-oriented software. These unit test methods can be classified into two types, a) specification-based test method, and b) structure-based (or program-based) test methods. Black-box test methods focus on verifying the functions and behaviors of a software component in terms of an external view. White-box test methods focus on checking the internal logic structures and behaviors of a software component, for example, basis-path testing based on control-flow graphs.

As stated earlier, object-oriented programs brought out some new testing problems to testers. Since classes are the major components in an object-oriented program, testers have to find the answers to the following questions:

- Whether the existing unit testing techniques can be applied on classes and a class cluster with a number of related classes?

- What test models, test generation methods, and test criteria can be used in unit tests for object-oriented software?

[1] A strongly connected component is a subgraph (of the original digraph) in which every vertex is reachable from any other vertex.

- How to perform unit tests for an object-oriented program in a systematic way?

Therefore, new test generation methods, test models, test coverage criteria for classes are needed in unit tests. Recently, there are a number of research papers addressing class unit tests. Several of them provide specification-based test methods for abstract data types.

Parrish, Borie, and Cordes [18] proposed a method for applying conventional flow graph-based testing strategies to classes. Based on the conventional flow-graph model, they proposed a general class graph model by extending the basic modeling concept to represent classes. Each class graph consists of: 1) nodes that is the set of class operations, 2) feasible edges, that are operation interactions from one to another according to the given specifications, 3) definitions and uses, each consist of pairs of (operation, type) , and 4) infeasible sub-paths according to the given specifications. Using this conceptual model as the test model, many existing flow graph-based techniques can be applied to classes in both specification-based unit testing and program-based unit testing. The authors provided their insights about how to use this class graph to define a new set of test coverage criteria for class unit testing, including node coverage, branch coverage, definition coverage, use coverage, and du-path coverage. Another important result of this framework is that some systematic test generation techniques are given based on class implementation. This makes the results more applicable to provide a systematic solution in class tests for today's industry practice.

Heechem Kim and Chisu Wu [9] focused on data bindings in class testing. In their approach, class testing consists of three steps. In the first step, testing each method, in which the existing functional and structural test methods can be used. The second step is testing actual data bindings, in which the major focus of testing is the data bindings between the methods in a class. They use an actual data binding to represent a data flow between two methods, so that data bindings between the methods of a class can be used as the basis for measuring inter-method interactions in its class. To reduce the complexity of the state-based testing, they apply state testing only to each simple MM-Path which is a sequence of a pair of methods represented by the actual data bindings. The final step is testing of sequences of methods. In this step, the class graph model for a class in [18] is sliced into a set of slices based on data members in the class. Each slice is a sub-graph of the class flow-graph. Based on this model, different flow-graph test generation methods given in [18] can be used to achieve various flow graph-based test criteria. According to the authors, this approach has the advantage on reducing the complexity of state-based testing of class objects, and simplifying test generations.

When software components (or parts) are separately tested, they are integrated together to check if they can work together properly to accomplish the specified functions. The major testing focus here is their interfaces, integrated functions, and integrated behaviors. In the past two decades, a number of software integration testing approaches have been used to perform software integration testing, such as top-down, bottom-up, sandwich, and "big bang". Since all of them were designed for integrating components in a traditional program, they might not be applicable to object-oriented programs due to the differences in their structures and behaviors.

The first is the structural differences between an object-oriented program and a traditional program. For example, a conventional program consists of three levels of components: a) functions (or procedures), b) modules, and c) subsystems. The structures of these components can be represented (or modeled) as call graphs, data-flow and/or control-flow graphs. However, an object-oriented program consists of four levels of components: a) function members defined in a class, b) classes, c) groups of classes, and d) subsystems. The conventional data-flow graph and control-flow graph can be used to represent the structure of a class function member. A class flow-based graph [18] can be used to model the interactions between functions defined in a class. A class relation diagram [15] can be used to model various relationships between classes, including inheritance, aggregation, and association relations.

The other major difference between an object-oriented program and a conventional program is their behaviors. In a dynamic view, a conventional program is made a number of active processes. Each of them has its control flow. They interact with one and another through data communications. An object-oriented program consists of a collection of active objects that communicate with one and another to complete the specified functions. In a multiple-thread program, there are a number of object message flows executing at the same time.

These differences bring out some new problems in integrating different components for an object-oriented program. Paul C. Jorgensen and Carl Erickson [8] proposed a method for integration testing based on their experience of an automatic teller machine

(ATM). They suggested five distinct levels of object-oriented testing, including a method, message quiescence, event quiescence, thread testing, and thread interaction testing. Their basic idea is to model the behaviors of an object-oriented program using an object network, in which nodes (rectangles) are methods and edges (dashed lines) are messages. Each object is a cluster (or a collection) of methods. Based on this model, integration tests are constructed based on different MM-Paths (which is a sequence of method executions linked by messages) in the object network.

5 Object State Testing

Object state testing is an important aspect of object oriented software testing. It is different from the conventional control flow testing and data flow testing methods. In control flow testing, the focus is testing the program according to the control structures (i.e., sequencing, branching, and iteration). In data flow testing, the focus is testing the correctness of individual data define-and-use. Object state testing focuses on testing the state dependent behaviors of objects. Finite state machines are most often used to model object state dependent behaviors. From these models test cases are generated to test the implementation.

C. D. Turner and D.J. Robson, described a black-box state-based testing method for testing the interactions between the features of an object and the object's state [24]. The features of an object are usually implemented as the object's operations or methods. This approach takes into account the random order in which the features can be invoked.

A state of an object is defined as the combination of the attribute values of the object. This view is commonly taken by researchers of the object-oriented paradigm. Since the number of all possible combinations is large, two concepts are introduced to reduce the complexity: 1) specific state values; and 2) general state values.

A specific state value of an attribute of an object has specific significance in the application on hand. For example, the quantities-on-hand of a certain product may drop below a certain thread level which would trigger a replenishment event. Another example is the NULL value of a linked list head pointer which signifies that the list is empty. Note that an attribute may have more than one specific state value. For example, the attribute used to denote the top of a stack may have "full" and "empty" as its specific values. The general state values of an attribute include all the values that are not specific. Using these concepts, the states of an object are identified according to the specification or design of the object class. States for invalid or exceptional situations are also identified.

The features of a class, when executed, cause state transitions from an "input state" to an "output state". The input states and output states of each feature are identified according to the specification or design. Test cases then are generated to test and validate each state transition. To simplify the process, the authors proposed to use substates, each of which is a combination of values of a subset of the attributes of a class, instead of states (which considers all the attributes) in the test case generation process. The steps for using this approach are: 1) identifying states which are combinations of attribute values; 2) identifying state transitions from input states to output states resulting in a finite state machine (FSM); and 3) generate test cases from the FSM.

D. Hoffman and P. Strooper [6] proposed a black-box methodology and support tools for testing object state dependent behaviors. The methodology consists of three main steps: 1) the tester prepares a testgraph which is essentially a state transition diagram that illustrates the expected state dependent behavior of the class under test (CUT); 2) according to the testgraph the tester implements an oracle class which has exactly the same methods as the CUT; 3) the tester implements test drivers to initialize the oracle class and the CUT; 4) the tester generates test cases according to the state transitions in the testgraph to execute the CUT and the oracle class; and 5) the tester determines whether the CUT implement the desired behavior by comparing the execution results of the CUT and the oracle class.

The approach developed at the University of Texas at Arlington [12] [16] is a reverse engineering approach. The authors first showed that certain object state behavior errors could not be readily detected by conventional testing methods like control flow testing or data flow testing. They then described an object state test method consisting of an object state model, a reverse engineering tool, and an object state test generation tool. The object state test model is an aggregation of hierarchical, concurrent, communicating state machines envisioned mainly for object state testing. A state machine may be an atomic object state diagram (AOSD) or a composite object state diagram (COSD).

An AOSD is a Mealy type state transition diagram defined for a single attribute of the class under testing. It represents the state dependent behavior of the attribute. Since only some of the attributes of a class have state dependent behaviors, only these attributes require an AOSD. For example, the title of a book in a library never changes, therefore, it does not re-

quire an AOSD. A COSD is an aggregate of AOSD's and COSD's. This recursive definition allows the test model to support inheritance and aggregation in OO programming. That is, the state dependent behavior of a derived class or subclass is a COSD consisting of the AOSD's for its own attributes and the COSD that represents the state dependent behavior of its parent class. Similarly, the behavior of an aggregate class is a COSD consisting of the AOSD's for its own attributes and the COSD's representing the behaviors of the component classes.

The reverse engineering tool produces an object state model from any C++ program[2]. The object state test generation tool analyzes the object state behaviors and generates object state test cases. Another important feature is state-based fault analysis which identifies the sequence(s) of method invocations that led to a faulty state. The steps to conduct object state testing can be summarized as follows: 1) selecting the class(es) to be tested; 2) generating the AOSD's for the class(es), note the COSD's are simply the aggregates of AOSD's and COSD's; 3) generating object state test cases from the AOSD's and COSD's; 4) generating test data for the test cases; 5) executing the test cases; 6) analyzing the test results to identify bugs; and perhaps 7) conducting state-based fault analysis to identify possible causes.

6 Regression Testing

The main concern in regression testing is how to effectively and efficiently identify the changes and their impact so that testing can be focused to the changed and affected components. Another consideration in regression testing is re-use of existing test cases and test suites.

Kung et al. [13] proposed an approach based on a test model called Object Relation Diagram (ORD). It captures the classes and their dependencies, like inheritance, aggregation and association. A reverse engineering method has been developed to extract the classes and their dependencies from an OO program and present them in the ORD. Changes to an OO program are automatically identified by comparing two different versions of program source code. The changes then are used to determine which of the classes are changed. Through the dependencies among the classes the impact of the changes are identified. Regression testing then can focus on testing the changed and affected classes.

After identifying the changed and affected classes, a tester will encounter the problem of determining in which order the classes should be tested. This is because in an OO program the classes may be dependent on each other. Therefore when testing a class which invokes methods of other untested classes, test stubs need to be constructed to simulate the untested methods. In some cases the test stub construction may be very costly. The authors also provided a method that computes the minimum cost test order for testing the classes.

Rothermel and Harrold [21] extended their results on selecting retest suites for conventional programs to OO programs. The problem addressed is which of the existing test cases must be rerun to determine whether the modified program still fulfills its functionality. One possible way is to select test cases that will produce different results. To do this, control dependencies and data dependencies among the statements and methods of an OO program and its modified version are identified and represented in two dependency graphs. From these graphs an algorithm identifies the statements in the modified program that will produce different results. Test cases, which traverse through these statements, need to be rerun to test the modified program. Additional test cases are needed to test the new functionalities.

Hsia et al. [7] addressed the same problem addressed in [21]. However, Hsia et al.'s presents a simpler but totally different approach. Unlike Rothermel and Harrold's approach, which analyzes the source codes of the OO program and its modified version, Hsia et al.'s approach inserts software probes into the source program. These probes record which test cases touch which of the classes. This information then can be used to establish a relationship between the test cases and the classes. Thus, when the changed and affected classes are identified the test cases that relate to these classes are the ones that must be rerun to test the modified program. Again, additional test cases are needed to test new functionalities.

In summary, Kung et al.'s approach uses only the relationships among the classes to identify the change impact. Therefore the approach is simpler but the retest effort may not be the minimum. The second approach goes down to the statement level and identifies and analyzes all the control dependencies and data dependencies among the statements and methods. This detailed analysis may substantially reduce the regression test effort but the complexity is high. The approach proposed by Hsia et al. is simple but it requires that the changed and affected classes must be known. Again, the retest effort may not be the minimum.

[2]The tool is being extended to process Java programs.

7 Test Tools

The major theme of this section is to explore software test automation for object-oriented software in terms of systematic approaches, adapted test models, test generation methods, as well as applications of test tools in software testing phase.

Kung et al. [15] reported their development work on an object-oriented testing and maintenance environment, called OOTME. In this system, a reverse engineering approach is used as a systematic way to recover the design of an object-oriented program (or a class library) into a collection of test models (called object-oriented test models) based on its source code (such as C++ code). These test models are classified into three types: 1) Object relation diagrams, called ORD, which represent the relationships between different classes. 2) Object state diagrams, called OSD, which depict the object state behavior for a class object. 3) Block branch diagrams, call BBD, which provide the control flow as well as the interface of a function member in a class.

Different tools are developed to assist testers in testing and regression testing. A tool, called Test Order Generator, can be used to compute an optimal test order as described in section 3. Another tool called Class Firewall Generator can be used to identify the changed and affected classes after various class changes. In addition, two types of test can be generated for testing a method. Based on an object state diagram, test cases can be generated for checking object behaviors of a class object.

In [2], Roong-Ko Doong and Phyllis G. Frankl reported their systematic approach to unit testing of object-oriented programs and a set of test tools, called ASTOOT. The major focus of this approach is how to automate the unit testing of abstract data types (ADTs) in object-oriented programs in test data generation, test execution, and test checking. ASTOOT system consists of a set of tools, including the driver generator, two test generation tools, called the compiler, and the simplifier. The driver generator takes as input the interface specifications of the class under test (CUT) and of some related classes, and outputs a test driver. This test driver, when executed, reads test cases, checks their syntax, executes them, and verifies the results. The compiler and simplifier together form an interactive tool for semi-automatically generating test cases from an algebraic specification, called LOBAS. The compiler reads in a specification written in LOBAS, and does some syntactic and semantic checking on the specification, then translates each axiom into a pair of ADT trees (in which nodes represent operations of an abstract data type along with their arguments). Each path from the root of a leaf of an ADT tree represents a possible state of the ADT. The simplifier inputs an operation sequence, supplied by the user, translates it into an ADT tree, and applies the transformations to obtain equivalent operation sequences.

In [20], Robert M. Poston, described how to reuse common object models (stored in repositories) for automated testing with a minimum of work and expense. Step by step through an example system, this article shows how to use an integrated set of software tools to perform the specification-based testing based on automated models during the development of object-oriented programs. This tool set includes three different tools, including a model-drawing tool, called StP/OMT, a test case generator called T, and a test execution tool, called XRunner. StP/OMT uses James Rumbaughs Object Modeling Technique (OMT) as its model to record the design information of object-oriented programs. To make the model to be test-ready, testers need to prepare the instances of data items, events, and states which cause logical conditions to be true and false and actions to be performed. Test case generator T provides the capability of generating test cases automatically. Xrunner , an execution or capture-replay tool, can be used to exercised or run a given object-oriented program on the integration level as well as on the system level. Rober M. Poston pointed out a very important fact, that is, when the OMT life cycle is supported by automated testing tools, the work of defining, designing, and writing test cases can be performed in parallel with the work of defining, designing, and implementing objects. Thus, the life cycle of an object-oriented software product is shortened significantly.

8 Acknowledgment

The material presented in this paper is based on work supported by the Texas Advanced Technology Program and many private companies.

9 References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, "Data Structures and Algorithms," Addison-Wesley Publ. Comp., 1983.

[2] R. Doong and P. Frankl, "The ASTOOT approach to testing object-oriented programs," ACM Transactions on Software Engineering and Methodology, Vol. 3, No. 2, pp. 101 - 130, April 1994.

[3] M. D. Fraser, K. Kumar, and V. Vaishnavi, "Strategies for incorporating formal specifications," CACM Vol. 37, No. 10, pp. 74 – 86, 1994.

[4] M. J. Harrold and J. D. McGregor, "Toward a testing methodology for object-oriented software systems," Department of Computer Science, Clemson University.

[5] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick, "Incremental testing of object-oriented class structure", Proc. of 14th International Conf. on Software Engineering, pp. 68 - 80, 1992.

[6] D. Hoffman and P. Strooper, "ClassBench: a framework for automated class testing," Software Practice and Experience, Vol. 27, No. 5, pp. 573 – 597, 1997.

[7] P. Hsia, X. Li, C. Hsu, D. Kung, "A Technique for Selective Revalidation of OO Software," Journal of Software Maintenance, to appear.

[8] P. Jorgensen and C. Erickson, "Object-oriented integration testing," CACM Vol. 37, No. 9, pp. 30 - 38, September 1994.

[9] H. Kim and C. Wu, "A class testing technique based on data bindings," Proceedings of the 1996 Asia-Pacific Software Engineering Conference, pp. 104-109, Soul, South Korea, December 1996.

[10] T. Korson and J. D. McGregor. "Understanding object- oriented: a unifying paradigm." CACM Vol. 33, No. 9, pp. 40 - 60, Sept. 1990.

[11] D. Kung, J. Gao, P. Hsia, J. Lin and Y. Toyoshima, "Design Recovery for Software Testing of Object-Oriented Programs," Proc. of the Working Conference on Reverse Engineering, pp. 202 - 211, Baltimore Maryland, May 21 - 23, IEEE Computer Society Press, 1993.

[12] D. Kung, N. Suchak, P. Hsia, Y. Toyoshima, and C. Chen, "On object state testing," Proc. of COMPSAC'94, pp. 222 – 227, IEEE Computer Society Press, 1994.

[13] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen, "Change impact identification in object oriented software maintenance," Proc. of IEEE International Conference on Software Maintenance, pp. 202 - 211, 1994.

[14] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "A test strategy for object-oriented systems," Proc. of Computer Software and Applications Conference, pp. 239 – 244, Dallas Texas, August 9–11, IEEE Computer Society, 1995.

[15] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y.S. Kim, and Y. Song, "Developing an object-oriented software testing and maintenance environment", Communications of the ACM, Vol. 38, No. 10, pp. 75 – 87, October 1995.

[16] D. Kung, Y. Lu, N. Venugopalan, P. Hsia, Y. Toyoshima, C. Chen, J. Gao, "Object state testing and fault analysis for reliable software systems," Proc. of 7th International Symposium on Software Reliability Engineering, White Plains, New York, Oct. 30 - Nov. 2, 1996.

[17] D.L. Parnas, "A technique for software module specification with examples," CACM May, 1972.

[18] Parrish, Allen S., Richard B. Borie and David W. Cordes, "Automated Flow Graph-Based Testing of Object-Oriented Software Modules," Journal of Systems Software, no. 23, 1993.

[19] D. E. Perry and G. E. Kaiser, "Adequate testing and object-oriented programming," Journal of Object-Oriented Programming, Vol. 2, pp. 13 - 19, January-February 1990.

[20] R. Poston, "Automated testing from object models," CACM Vol. 37, No. 9, pp. 48 - 58, September 1994.

[21] G. Rothermel and M. J. Harrold, "Selecting Regression Tests for Object-oriented Software,", Proc. of IEEE International Conference On Software Maintenance, pp. 14-25, 1994.

[22] M. D. Smith and D. J. Robson, "Object-oriented programming - the problems of validation," Proc. IEEE Conference on Software Maintenance, pp. 272 - 281, 1990.

[23] E. Soloway et al, "Designing documentation to compensate for delocalized plans," CACM Vol. 31, No. 11, pp. 1259 - 1267, Nov. 1988.

[24] C. D. Turner and D. J. Robson, "The state-based testing of object-oriented programs," Proc. of IEEE Conference on Software Maintenance, pp. 302 - 310, 1993.

[25] E. J. Weyuker, "Axiomatizing software test data adequacy," IEEE Trans. on Software Eng., Vol. SE-12, No. 12, pp. 1128 - 1138, 1986.

[26] E. J., Weyuker, "The evaluation of program-based software test data adequacy criteria," CACM, Vol. 31, No. 6, pp. 668 - 675, 1988.

[27] N. Wilde and R. Huitt, "Maintenance support for object-oriented programs," IEEE Trans. on Software Eng., Vol. 18, No. 12, pp. 1038 - 1044, December 1992.