A Novel Hybrid Object-Oriented Class Testing Method


Bor-Yuan Tsai
Department of Information Management
Aletheia University, Matou Campus
70-11, Pei-Shih Liao, Matou, Tainan, Taiwan.
Fax: +886 6 5700545 Tel:+886 6 5702500  E-mail: by.tsai@mail1.mt.au.edu.tw

Abstract: A hybrid testing method, discussed in this research, combines state-based testing with data flow analysis to test classes at the intra-class level. Employing state-based testing, the method can generate all possible test message sequences. Using data flow analysis, the method can also detect whether data anomalies exist in those generated sequences, and then to compute data flow test cases which can be applied to the classes under test. The hybrid testing method is appraised through three case studies selected from research articles. The author’s research suggests that the success of the hybrid methodology is the best demonstrated by testing object-oriented classes.

Key Words: Class Testing, Intra-class testing, State-Base Testing, Data Flow Testing, Data Flow Anomaly, Test Case Tree

1. INTRODUCTION

At the class testing level, state-based and data flow testing techniques have been employed. However, the former may not detect errors occurring in variables, which do not define the object’s states. Data flow testing has been applied to generate test cases for testing classes using data flow criteria, but this is a difficult task. Moreover, some of the test cases thus generated may be unworkable.

In this paper, a hybrid object-oriented class testing method is proposed. Although this study concentrates on intra-class testing, an approach based on data flow analysis is suggested to perform the testing at the intra-method and inter-method levels. Therefore, data occurrence information in individual methods (member functions) obtained at these testing levels will be able to be reused during intra-class testing.

The overall aim of this research is to adapt and improve the existing testing techniques in order to propose a novel object-oriented class testing technique. Some of the testing techniques used to test conventional programs (e.g. state-based testing and data flow testing) have been adopted or adapted for testing object-oriented programs (Turner, 1994) (Harrold and Rothermel, 1994) (Binder, 1996) (Huang et al., 1996). State-based testing is one of the most popular techniques for testing object-oriented classes (Binder, 1996) (Kung et al., 1996). Data flow testing is also employed in the testing of object-oriented classes (Harrold and Rothermel, 1994) (Parrish et al., 1993) (Hong et al., 1995). However, the former ignores the program code, and could miss the detection of data members that do not define the states of the object class. Moreover, this technique might not generate all possible sequences of test cases based on the state-transition trees in (Kung et al., 1996) (Chow, 1978). The latter is not suitable for generating the sequences of messages for testing a class at the intra-class level, because such messages could be in an arbitrary order. Moreover, the data flow test cases cannot be completely generated if any data flow anomalies exist in the class.

The paper is organised as follows. The concepts of state-based testing, data flow testing and other testing techniques that are necessary introduced first are presented in section 2. The reason of detecting data anomaly before generating data flow test cases is explained in section 3. In section 4, the motivation for developing the hybrid class testing method is given. The data anomaly detection technique of the method is described in section 5. After that, the limitations of this research are given. Examining the hybrid testing method using three selected classes is evaluated in section 7. Our future work and conclusions are introduced in section 8.

2. LITERATURE REVIEW

2.1 Class Testing

In the object-oriented world, it is very common that object-oriented programs are structured with three levels: the cluster level, the class level and the function level. In an object-oriented language like C++ a class contains both member functions and data members, exhibiting a strong degree of internal coupling. Private members are encapsulated entirely within the class and are invisible from
outside of the class. Therefore, a class is a basic component to be tested in the unit testing of object-oriented software. In this research the main focus is at this class level. The class testing can be further distributed into three testing levels: *intra-method testing*, *inter-method testing*, and *intra-class testing*. The hybrid class testing presented in this research is limited at this intra-class testing level.

### 2.2 State-Based Testing and Data Flow Testing

The main concept of state-based testing is to examine the interactions within an object by monitoring the changes of states, and errors are detected by testing states of an object. The advantage of using state-based testing over white-box control flow testing is that it provides a much better and more productive level of coverage, such as state coverage, transition coverage, event coverage.

Data flow analysis examines where/how variables are defined with values and where/how the values of the variables are used (Rapps and Weyuker, 1985). A test path is formed from definitions to uses in a program. A definition-use pair (def-use pair) is an ordered pair \((d, u)\), where a statement called \(d\) contains a definition of a variable \(v\), which is used in a statement \(u\) in a program.

Uses of a variable are further divided into two classes, as either computation uses \((c\text{-use})\) or predicate uses \((p\text{-use})\). A c-use occurs when the value of a variable is used in a computation or output statement, and a p-use occurs when the value is used in a condition (predicate) statement. Key data flow testing concepts and criteria can be found in (Frankl and Weyuker, 1988).

The data flow testing technique of this hybrid testing method is concerned with tracing the flow of data members among class member functions in the class, rather than local variables within an individual function.

When employing data flow testing on object-oriented classes, three types of def-use pairs should be tested (Harrold and Rothermel, 1994): (1) Intra-method def-use pairs (2) Inter-method def-use pairs and (3) Intra-class def-use pairs. Only the intra-class def-use pairs are concerned in this research.

At the intra-class testing level, the flow of data members (global variables in a class) between member functions is traced. Adapting the global definition and use, a **global use** of data member \(x\) is when \(x\) has been defined in member function \(i\), and can be used (c-used or p-used) in some member functions other than function \(i\). A member function has a **global definition** of a data member \(x\), only if the member function has a definition of \(x\) and there is a def-clear path with respect to \(x\) from the member function (e.g. function \(i\)) to some member functions (e.g. function \(j\)) containing a global use of \(x\).

### 2.3 Data Flow Anomalies

Data flow analysis is often used in code optimisation and program reliability. Many researchers (Fosdick and Osterweil, 1976) (Chan and Chenl, 1987) (Livadas and Tsao, 1994) have used it to detect programming faults known as “data flow anomalies”. When the pattern of use of variables is abnormal, there is an anomaly in the data flow (Jachner and Agarwal, 1984), such as misspelling and confusion of variable names, omission of statements, incorrect parameter usage, and so on. For instance, variables having the “referencing killed variable anomaly” are usually due to a misspelling (Fosdick and Osterweil, 1976).

The important types of data flow anomalies stated in (Fosdick and Osterweil, 1976) (Spillner, 1992) (Beizer, 1990) consist of: (1) defining a variable twice with no intervening use, (2) using a variable that is killed, and (3) releasing variables that are defined but not used.

<table>
<thead>
<tr>
<th>Actions on a variable</th>
<th>Anomaly</th>
</tr>
</thead>
<tbody>
<tr>
<td>(dd)</td>
<td>A define action is followed by a define action. Probably a harmless anomaly but strange.</td>
</tr>
<tr>
<td>(ku)</td>
<td>A kill (undefined) action is followed by a use (reference) action. A harmful anomaly; the value is released before reference.</td>
</tr>
<tr>
<td>(dk)</td>
<td>A define action is followed by a kill action. Why was the variable defined but not used? Probably an anomaly.</td>
</tr>
<tr>
<td>(d-)</td>
<td>A variable was defined without usage. Probably an anomaly, but this could be a global definition.</td>
</tr>
<tr>
<td>(-u)</td>
<td>A variable is used without definition. Probably an anomaly, but the variable may have been previously defined.</td>
</tr>
</tbody>
</table>

**Table 2-1** The actions of data flow anomaly
Let defined, used and killed be three types of data occurrences defined on a variable. A variable is defined (d) when a value is assigned to it, and it is used (u) when its value is obtained from memory. A variable is killed (k) when its value is released or it contains no known value. “Killed” (or undefined) is that the instance of a variable is killed. The anomalies that are often used in this research are listed in Table 2-1 after (Beizer, 1990).

An anomaly is indicated with a pair of sequence actions. For instance, two sequence statements \( \{X = 5; X = 10;\} \) mean that the X is defined twice without an intervening use. Fatal anomalies are those that correspond to actual programming errors. Some anomalies are probably harmless yet they are suspicious. The harmless data anomalies are also called false data anomalies. Whether or not a data anomaly is a false one, the tester needs to check the source program. For instance, after checking \( \{X = 5; X = 10;\} \), the tester finds the typing error and re-codes \( \{X = 5; Y = 10;\} \) instead, then this dd data anomaly in \( \{X = 5; X = 10\} \) is harmful. On the other hand, if it is not a mistake, then the dd is a false data anomaly. Nevertheless, this raises a question: why is a variable defined twice without any usage in between? The point is that the presence of a data flow anomaly is at least a reason for concern because it often is a sign of an error. “Certainly software containing data flow anomalies is less likely to be reliable than software which does not contain them” (Fosdick and Osterweil, 1976).

3. WHY IS DATA FLOW ANOMALY DETECTION NEEDED?

If the class under test is implemented by following the state model, which describes the behaviour of the class, then the paths of transition in the model reveal the feasible sequences of member functions of the implemented object class. This means the sequences of member functions (mapping to the paths of transitions) of the object are feasible. Therefore, data flow test cases can be selected from the sequences of member functions based on the conventional def-use pair technique.

Can intra-class data flow test cases for each data member be completely generated by using the above technique? The answer is not positive. Once any data anomalies of data members exist in sequences of functions, some necessary data flow test cases cannot be selected from the sequences.

The conclusion of the argument in (Tsai, 2000) reveals that data anomaly detection should be performed before generating intra-class data flow test cases.

4. A NOVEL HYBRID CLASS TESTING METHOD

The hybrid testing method proposed in this paper covers black-box and white-box testing. State-based testing and data flow anomaly detection techniques are adopted for the black- and white-box testing, respectively, in this method.

The main focus of this hybrid class testing method is at the intra-class testing level. Intra-class dynamic behaviour can be modelled using its state transition diagram. Method sequence specification (i.e. the correct sequences of messages which objects of the class must receive) for the class can be derived from its state diagram. The sequences of the methods of a class are the basis of the hybrid testing technique, which tests the class at the intra-class testing level. Therefore, the object-oriented classes can be validated by the hybrid testing method if their behaviour can be described using state diagrams.

The motivation for developing the hybrid class testing method that is the subject of this research is two-fold: state-based testing fails to detect some data members, and intra-class data flow test cases cannot be completely generated if any data anomalies are present in sequences of messages. Therefore the obvious approach is to combine state-based testing with data flow anomaly detection.

As discussed in (Tsai, 2001), to generate all possible test cases using data flow analysis is a difficult task, and some of the test cases may be infeasible. Even some necessary test cases cannot be computed if data flow anomalies exist in the class. For preventing this situation, the data flow anomalies within the sequences should be eliminated, because they may break the def-use pairs and influence the estimate of data flow associations. Hence, it is necessary to detect data flow anomalies within method sequences of a class before computing intra-class data flow test cases from the sequences for testing the class.

The author proposes that data flow testing at the intra-class testing level should be accomplished in two stages:

1. detecting and removing any data flow anomalies of the data members within the sequences of messages, and then
2. generating intra-class test cases from the anomaly-free sequences of messages.

This research is mainly concerned with the first stage. After excluding the anomalies, the data flow criteria can be applied to compute the data flow test cases. Additionally, the program code may be reconsidered and optimised by programmers during anomaly removal.

The sequences of the member functions, generated from a state transition diagram, can also be used to detect any state-based faults in the objects of a class. Based on this theory, a novel hybrid object-oriented class testing method, which detects whether any state-based faults and/or data anomalies exist in the class under test, is proposed in this research.
5. THE DATA ANOMALY DETECTION TECHNIQUE

Some data flow anomalies can be detected by the compiler using the information known at compile time. However, the method sequences are unknown until run-time when the sequences of messages are sent from the client objects in an arbitrary order. The transition paths in the state transition diagram show all feasible sequence behaviours of the required object. Based on the transition paths, the data anomaly detection steps of this hybrid class test method are:

(1) **Mapping implemented methods to state transitions on its state transition diagram.**
   A sequence of state transitions in the state transition diagram of a class implies a sequence of messages sent from outside of the object. The hybrid method detects any data flow anomalies existing in this sequence of messages.

(2) **Revealing the global definitions and uses of data members within functions.**
   In object-oriented classes, global definitions and uses of each data member in member functions can be extracted by static analysis scanning the data flow paths.

(3) **Generating sequences of occurrences of data members.**
   The sequences of def-use information can be generated by traversing the paths of state transitions in the diagram.

(4) **Detecting data anomalies of data members within the generated sequences.**
   In general, an anomaly on a data member in a sequence of data occurrences occurs if one or more of the $dd$, $ku$ and $dk$ data anomalies exist in the sequence (Fosdick and Osterweil, 1976).
   These detected data anomalies are the reason that the test cases cannot be computed to examine data members. This was also argued in section 3.

6. LIMITATIONS

It must be stressed that a data flow anomaly is not necessarily an error. Intuitively, an anomaly shows that something is irregular. Therefore, the presence of a data anomaly only implies that the program may have errors. The main concern in this research is that the anomaly may prevent necessary intra-class data flow test cases from being computed. The purpose of anomaly removal in this research is therefore to remove the barriers of data flow test case selection.

In the hybrid method, the various associations of data members with definitions and uses across the public member functions in the class are taken into account. This is in order to detect whether any data anomalies exist in the sequences of member functions called by its clients. This is the pre-processing of computing intra-class data flow test cases using conventional data flow criteria. This test case computation is not included in this research, but automatic data flow test case generation is identified as a future task.

In data flow coverage, the definition-use pairs of parameters and local variables in a class need to be computed. They are classified into the intra-method and inter-method testing levels, which are excluded in this paper. At the intra-class testing level, however, only the occurrences of the data members within sequences of function calls from outside of the class are considered. Additionally, detecting intra-class data flow anomalies within private member functions invoked by public member functions can also be employed and demonstrated by the author in (Tsai, 2000).

In this research, the problems associated with pointers and recursion are not included, because they have already been widely discussed (Livadas and Tsao, 1994) (Gross and Steenkiste, 1990).

7. METHODOLOGY EVALUATION

This section examines the hybrid class testing method using three object-oriented classes (i.e. CCoinBox, SymbolTable and Door are selected from research articles). The examination reveals that the hybrid class testing method is a more effective technique than existing methods for validating object-oriented classes.

The state-based testing technique of the hybrid testing method will be evaluated using the CCoinBox and SymbolTable classes. The data flow technique of the hybrid method will be appraised using the classes CCoinBox, SymbolTable and Door. A discussion and evaluation of each test case is given in each case study, and an evaluation table comparing the different methods is included in section 7.5.

Table 7-1 summarises the four different kinds of object-oriented class testing techniques discussed in this section.

7.1 The Rationale

The purposes of the evaluation are to determine whether the hybrid class testing method proposed in this research improves testing performance, and to examine the effectiveness of the hybrid class testing method. This is done using three case studies.
In software testing, the evaluation cases selected to appraise a new proposed technique or theory can be obtained from three different sources. One of them is adopted and adapted from existing cases that have been used in support of other researchers’ methods.

The benefit of adopting and adapting existing cases from research work is that the cases provide an opportunity to carry out comparisons. Another major benefit of using existing case studies is that there is an understanding of the error domain under study. For these reasons, this method of selecting evaluation cases has been chosen.

It should be noted however that, during the development of the hybrid testing method the authors did create specific cases of this type to ensure the method worked as expected. For example, the authors designed a bank account class, a vending machine CoinSlotPart class and a queue class containing errors, in order to evaluate the state-based technique of the hybrid method in (Tsai et al., 1999a) (Tsai et al., 1998) respectively. A bounded queue class that was coded by the authors illustrated the hybrid testing technique in (Tsai et al., 1999b) (Tsai et al., 1999c).

<table>
<thead>
<tr>
<th>Test model</th>
<th>The hybrid class testing technique</th>
<th>Kung et al.’s technique</th>
<th>Harrold and Rothermel’s technique</th>
<th>Hong et al.’s technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Units</td>
<td>Class</td>
<td>Class</td>
<td>Class</td>
<td>Class</td>
</tr>
<tr>
<td>State view</td>
<td>Combination of attribute values</td>
<td>Combination of attribute values</td>
<td>Combination of attribute values</td>
<td></td>
</tr>
<tr>
<td>Testing Technique</td>
<td>State-based fault detection and data anomaly analysis</td>
<td>State-based fault analysis</td>
<td>Data flow testing</td>
<td>Data flow testing</td>
</tr>
<tr>
<td>Test cases generation</td>
<td>Test case tree, derived from state model</td>
<td>From state model</td>
<td>Computing def-use pairs</td>
<td>From state model</td>
</tr>
<tr>
<td>Test Case Format</td>
<td>Sequence of member functions</td>
<td>Sequence of member functions</td>
<td>Sequence of member functions</td>
<td>Sequence of member functions</td>
</tr>
<tr>
<td>Measure of Quality</td>
<td>All states and transitions coverage, all definition coverage at intra-class level</td>
<td>All states and transitions coverage,</td>
<td>All-use coverage</td>
<td>All definition coverage</td>
</tr>
<tr>
<td>Other</td>
<td>Tool support (MACT)</td>
<td>Tool support</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7-1 Summary of the four class testing techniques

7.2 Case Study One – A Coin Box Class

The coin box class, called CCoinBox, presented by Kung et al. in (1996) demonstrates their object state testing technique for finding state-based errors. In using this class the author is seeking to establish that the hybrid testing technique can find the same state-based errors as the technique of Kung et al., as well as other errors such as data anomalies.

Summary of Evaluation: As the test results, the error in the CCoinBox class can be detected using sequences of test messages, which are generated using the hybrid class testing method. In addition, the demonstration of the data flow anomaly detection shows that the data-flow testing technique of the hybrid method is able to detect the data anomalies of data member allowVend and identify the locations of the errors in the CCoinBox class. In this case study it has been clearly shown that the hybrid testing technique offers more support for the testing process than that offered by Kung et al. A sequence of test messages, generated by the hybrid testing method, detected a state-based error in the CCoinBox class. The data anomalies within the class were also detected by the data flow technique of the hybrid testing method. The suggestion of re-coding the ReturnQtrs() and AddQtr() functions can remove the state-based fault and eliminate the data anomalies.
7.3 Case Study Two – A Symbol Table Class

Harrold and Rothermel (1994) employ their data flow testing technique on a symbol table class (called SymbolTable) and show how to generate test cases for inter-method and intra-class testing. The private member functions of the SymbolTable class contain many intra-class definition-use pairs. This class therefore provides an opportunity to determine whether the hybrid method has the ability to detect data flow anomalies, i.e. not only those within public functions but also those within private functions.

Summary of Evaluation: No known state-based errors or data anomalies were found in the SymbolTable class. However, the state-based technique of the hybrid method can produce more test cases than Harrold and Rothermel’s method for intra-class testing. This case study also indicates that program-based testing may not generate enough test cases and the requirements of specifications should also be considered. The hybrid testing method serves these two approaches, to improve class testing.

7.4 Case Study Three – A Door Class

Hong et al. (1995) propose the class flow graph (CFG) of the Door class to demonstrate their data flow testing technique for generating all definition coverage test cases. In adopting this class the author is seeking to reproduce these results using the hybrid testing method. In addition, the data flow technique of the hybrid method can further be used to detect whether any data anomalies exist in the generated test cases.

Summary of Evaluation: In the Door class example, Hong et al. (1995) do not provide the source code for the class, but offer a class state machine (an extended FSM) for generating specification-based test cases. Hong et al. (1995) generated test cases by computing the definition-use pairs of data members within the public member functions of a class. Those test cases can also be generated by the hybrid testing method. Three data anomalies within the sequences of test cases, generated by the hybrid method, are detected using the data flow technique of the method. Although they are not fatal anomalies, they prevent two necessary intra-class data flow test cases from being selected. The error removal suggestion also solves the anomaly problem. The testing technique of Hong et al. (1995) is a test case generation approach, and data anomaly detection is not discussed.

<table>
<thead>
<tr>
<th>Testing Method</th>
<th>Class CCoinBox</th>
<th>Technique</th>
<th>Error Removal Suggestion</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Hybrid Method</td>
<td>An error found</td>
<td>Two errors found</td>
<td>ReturnQtrs() and AddQtr() modified by using data flow analysis</td>
</tr>
<tr>
<td>Kung et al.’s Method</td>
<td>An error found</td>
<td>No execution</td>
<td>ReturnQtrs() modified based on object state diagram analysis</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Testing Method</th>
<th>Class SymbolTable</th>
<th>Technique</th>
<th>Error Removal Suggestion</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Hybrid Method</td>
<td>Not found</td>
<td>Not found</td>
<td>More test cases generated for intra-class testing by using the state-based technique</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data anomaly detection</td>
</tr>
<tr>
<td>H &amp; R’s Method</td>
<td>No execution</td>
<td>No execution</td>
<td>Test case generation based on data flow criteria</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Testing Method</th>
<th>Class Door</th>
<th>Technique</th>
<th>Error Removal Suggestion</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Hybrid Method</td>
<td>No source code</td>
<td>Three errors found</td>
<td>Test cases generated by using the state-based technique</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data anomaly detection (public and private functions coverage)</td>
</tr>
<tr>
<td>Hong et al.’s Method</td>
<td>No source code</td>
<td>No execution</td>
<td>Test cases generated by computing definition-use pairs (public functions coverage)</td>
</tr>
</tbody>
</table>

Table 7-2 The evaluation of the hybrid class testing method
7.5 Summarising the Evaluations of the Hybrid Testing Method

Table 7-2 summarises the evaluations of the hybrid class testing method using the three case studies. The table also shows the advantages of the hybrid class testing method over existing class testing approaches.

8. CONCLUSIONS AND FUTURE WORK

State-based testing transforms a class behaviour model into test cases. Errors within non-state defining data members may not be found by state-based testing. Some data members within the sequences of test messages generated using state-based methods may violate data flow criteria. An example of this would be a data anomaly.

Data flow testing generates test cases based on the patterns of definitions and uses of the variables in a class. Although data flow techniques have been proposed as a means of generating sequences of test cases for the class under test, some selected test cases may be infeasible, and filtering these out is a further problem. Furthermore, some necessary test cases cannot be computed if data anomalies exist in the class. Based on a combination of the state-based testing and data flow analysis methods, the hybrid class testing approach is proposed as a means of utilising the strengths of both techniques.

In section 7, an appraisal of the hybrid testing method was achieved through its application to three case studies, which were chosen from previously published research projects in the field of object-oriented class testing.

Based on this hybrid testing method, the prototype of an automatic class testing tool called MACT has been proposed in (Tsai et al., 1999d). A semi-automatic test tool developed from the prototype has been discussed in detail and demonstrated in (Tsai, 2000). To develop and to commercialise MACT into a full automatic tool is one of future tasks.

9. REFERENCES


