Employing Data Flow Testing on Object-Oriented Classes

Bor-Yuan Tsai\textsuperscript{1}, Simon Stobart\textsuperscript{2} and Norman Parrington\textsuperscript{2}

\textsuperscript{1}Department of Information Management
Aletheia University,
70-11 Pei-Shih Liao, Matou,
Tainan County, Taiwan
Fax: +886 6 5700545
Email: by.tsai@mail1.mt.au.edu.tw

\textsuperscript{2}School of Computing, Engineering and Technology
University of Sunderland
St. Peter’s Way, Sunderland
SR6 0DD UK
Fax: +44 191 515 2781
Email: simon.stobart@sunderland.ac.uk
Email: norman.parrington@sunderland.ac.uk

Abstract

At the class testing level, state-based testing and data flow testing techniques have been employed. However, while the former only involves the variables that have an effect on the behaviour of the object under test, it is possible for errors to occur in variables, which do not define an object’s state. 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 data flow test cases thus generated may be unworkable. Selecting data flow test cases based on sequences of specification messages is a way to reduce the effort of generating feasible intra-class data flow test cases. However, some test cases cannot be selected, if data flow anomalies exist within the sequences. The data flow testing technique in this research is divided into two stages; first detecting data flow anomalies and then computing data flow test cases.

1 Introduction

The basic unit of testing an object-oriented application is a class, and class testing work has mostly centred on functional testing [10]. A state transition diagram can model the dynamic behaviour of a single class object if the object has significant event-order behaviour [3]. After executing a sequence of methods, the final state that has been achieved by the object can be verified and thus object-oriented classes are well suited to state-based testing [2].
State-based testing mainly examines state change and behaviour rather than internal logic and thus data faults may be missed. Furthermore, data members that do not define an objects state are generally ignored when the classes are validated using state-based testing. Those unexamined data members need to be examined by some other technique in order to ensure the quality of the implemented classes. Data flow testing uses the data flow relations in a program to guide the selection of test cases [7, 13] and has been employed to generate data flow test cases for testing object-oriented classes [10, 11, 14]. However, the authors in [17] argue that selecting data flow test cases for testing classes at the intra-class level is difficult and expensive, as the selection is based on data flow criteria. In addition, some of the generated test cases may be infeasible.

To avoid any infeasible data flow test cases being computed, intra-class data flow test cases can be generated using the sequences of feasible messages rather than directly using the data members in the class [17]. These sequences of test cases (messages) are transformed from the sequences of feasible transitions (paths) described on the class transition diagram. However, those test cases may not be completed if any data anomalies exist within the class, as the anomalies may distort the definition-use pairs that are the basis of selecting data flow test cases.

To reduce the effort of test case generation and assure that feasible test cases are computed, the authors propose that data flow testing at the intra-class testing level should be accomplished in the following two stages: (1) detecting and removing data flow anomalies within the sequences of messages, and (2) generating intra-class test cases from the anomaly-free sequences of messages. This paper is mainly concerned with the first stage.

The concept of data flow testing, and related work, is presented in section 2. Section 3 introduces the types of data flow anomalies in object-oriented programs. The reason for performing data flow anomaly detection before computing intra-class data flow test cases is explained in section 4. The technique of detecting intra-class data flow anomalies that the authors proposed in [17] is described in section 5. Section 6 contains a case study, in which the difficulty of data flow test case generation, the anomaly detection technique and data flow
anomaly removal are demonstrated, in addition a case study is described. Future work and conclusions are given in section 7.

2 Data Flow Testing

In applying traditional testing techniques for testing object-oriented software, structural path testing can be considered. The more popular techniques for test path selection are control flow testing and data flow testing [16]. The studies in [8, 15] show that all-uses (a criterion of data flow testing) include all-edges (a criterion of control flow testing), and the former is stronger than the latter since it requires a path from every definition of a variable to every possible predicate-use (p-use) of that variable. Most data flow testing techniques [8, 9, 14, 15] are based on data flow analysis and require test data (cases) to exercise all the individual data definition and use relationships (definition-use paths, called def-use paths). Therefore, data flow testing can be used to find data faults on those data members that are not considered in state-based testing.

Def-use paths (also called du-paths) are formed from definitions (where values are assigned to variables) to uses (where the values are referenced without modification) 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.

In order to support data flow testing of the class, the intra-method, inter-method, and intra-class levels of def-use pairs for the class must be considered. However, only intra-class testing is considered in this paper.

2.1 Related Work

Harrold and Rothermel in [10] used a SymbolTable class as an example to demonstrate their data flow technique. Harrold and Rothermel [10] also claim that the further advantage of their technique is to determine which sequence of methods should be executed to test a class and point out error sequences with examples that need not be run. However, their class example only has two public member functions so that the selected intra-class test cases (based on intra-
class def-use pair) are merely pairs of ordered functions. If a class has \( N \) public member functions, then there are \( N! \) sequences at the intra-class testing level [5]. In fact, it is very difficult to select all possible test cases and exclude all infeasible sequences from the \( N! \) sequences by referencing the functionality of the class [17].

Parrish et al. in [14] applied the testing criteria of conventional program flow graphs to define the sequences of methods based on the class flow graph. All definitions and uses are associated with class methods rather than statement blocks. This approach only considers the parameters of member functions, and ignores data members. When building a test set for testing a member function within a class, it is not sufficient to consider only its input parameter values.

In object-oriented class testing, Hong et al. in [11] presented the class state machine (CSM) to specify the behaviour of the class under test. The machine, is transformed to a flow graph, from which definition-use flows of each data member in the CSM can be explicitly identify. Finally, they apply conventional data flow testing methods [19] on the flow graph for generating test cases. Hong et al.’s [11] method is a specification-based technique based on the state transition diagram of the class under test. In fact, data flow testing is a code-based technique, but Hong et al. [11] did not consider the source code of the class. The def-use pairs selected using the specification may not be suitable for testing the implemented program, if the latter contains any data flow anomalies.

2.2 Overview of Data Flow Testing Criteria

Data flow analysis examines where/how variables are defined with values and where/how the values of the variables are used [13]. Uses of a variable are further divided into two types, as either computation uses (c-use) or predicate uses (p-use) [15]. 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. For instance, the if \( (x > 0) \) \( \{x = y + 10;\} \) statement contains p-use of \( x \) and c-use of \( y \), followed by a def of \( x \).
Table 1 Data Flow Testing Criteria

<table>
<thead>
<tr>
<th>Terms</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>du pair</td>
<td>A path between the definition and use of a particular variable.</td>
</tr>
<tr>
<td>def-clear</td>
<td>A def-use path is definition-clear (def-clear) with respect to a variable x if it has no variable redefinition of x within the path.</td>
</tr>
<tr>
<td>du-path</td>
<td>A def-use path (du-path) is any def-clear path that traverses both definition and use of the du pair.</td>
</tr>
<tr>
<td>all defs</td>
<td>The path of all defs is such that a test case can exercise a def-clear path from each definition to one of its uses (a c-use or a p-use).</td>
</tr>
<tr>
<td>all uses</td>
<td>Some test cases can exercise a def-clear path from every definition to every use, called all uses.</td>
</tr>
<tr>
<td>all du-path</td>
<td>The strongest criterion, all du-path, is if a test case can exercise every du-path from every definition to every use that is reached by the definition.</td>
</tr>
<tr>
<td>dcu(v,i)</td>
<td>Let i be any member function and v any variable such that v ∈ def(i). Hence, dcu(v,i) is the set of all functions j such that v ∈ c-use(j), and in this path there is a def-clear sub-path with respect to v from i to j.</td>
</tr>
<tr>
<td>dpu(v,i)</td>
<td>dpu(v,i) is the set of all functions j such that v ∈ p-use(j) and in this path there is a def-clear sub-path with respect to v from i to j.</td>
</tr>
<tr>
<td>global use</td>
<td>In data flow testing, a global use (c-use or p-use) of variable x is if and only if the definition of x preceding its use does not occur within the same block (function). Otherwise, it is a local use. A global definition of variable x is if and only if the last definition of the x occurs in block i and there is a def-clear path with respect to x from block i to another block j containing a global use of x.</td>
</tr>
<tr>
<td>global def</td>
<td></td>
</tr>
</tbody>
</table>

Key data flow testing concepts and criteria are listed in Table 1, and the remaining data flow testing criteria can be found in [7, 10, 13, 15]. The data flow testing technique in this paper is concerned with tracing the flow of data members among member functions in the class, rather than local variables within an individual function.

3 Data Flow Anomalies

Data flow analysis is often used in code optimisation and program reliability [1]. Many researchers [4, 6] 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, caused by misspelling and confusion of variable names, omission of statements, incorrect parameter usage, and so on.

There are three types of data occurrences that can be applied to a variable. Firstly, 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. A
A variable is also "Killed" when the instance of a variable is destroyed. The anomalies are used in this research are listed in Table 2.

<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>

### 3.1 False Data Flow Anomalies

The presence of a data anomaly does not imply that the execution of the program will definitely produce incorrect results. Some anomalies are probably harmless yet they are suspicious. The harmless data anomalies are also called false data anomalies. For instance, the dd data anomaly (or d-) may be a false data anomaly and probably be harmless. A data anomaly implies only that execution MAY produce incorrect results depending on the input data, or it may never produce incorrect results. Whether or not a data anomaly is false, the tester needs to check the source program. The point is that the presence of a data flow anomaly is at least a reason for concern because it often is a sign that an error exists. Obviously, if a program that contains data flow anomalies is less likely to be reliable than a program that does not contain them [6].

False data anomalies may not be harmful, but they may prevent some intra-class data flow test cases from being computed. Therefore, any source code containing them is still worthy of the tester’s attention.

### 3.2 Data Flow Anomaly in Object-Oriented Programs

After static analysis of the stack’s source code in Figure 1, no harmful data anomalies occur with the top data member in both the push() and pop() functions. This kind of data flow anomaly detection can be done by optimising compilers based on information known at compile
time [1]. However, the static analysis approach cannot detect data anomalies within sequences of messages. These anomalies can be the reason that some intra-class data flow test cases cannot be produced from the feasible sequences of messages.

In object-oriented classes, every public member function can be sequentially executed in any order. Independent analysis of member functions for static data flow analysis may not give meaningful results. For instance, data members inside a destructor may be killed without any preceding definition. In a constructor, data members may be defined without any succeeding use within the constructor. However those defined/killed data members may be used/defined in the subsequent/previous member functions. This means that data anomalies occurring in individual functions may be false data anomalies, but they can still be useful in detecting sequences of member functions at the intra-class testing level.

4 Why Is Data Flow Anomaly Detection Needed?

If the class under test is implemented by following its state transition diagram, the paths of transition in the diagram reveal the feasible sequences of member functions of the implemented object class. 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 data members be completely generated by using the above technique? The answer is not positive. Once any data anomalies of data members exist in the sequences of functions, some necessary data flow test cases cannot be selected from the sequences. For example, the simplified stack class (in Figure 1) has a constructor and push() and pop() member functions. It has the top data member to indicate the location for storing the data to be pushed and the a...
data member is used to temporarily store data. For simplicity, the internal data storage is omitted in this example.

Public member functions in the stack class can be sequentially executed by any order. For example the four pairs of sequence functions are: \(<\text{push()}\rightarrow\text{push()}; \text{push()}\rightarrow\text{pop()}; \text{pop()}\rightarrow\text{push()} \text{ and } \text{pop()}\rightarrow\text{pop()}\>\). Based on the global def-use pair selection technique [14], the sequences of \(\text{push()}\rightarrow\text{push()}\) and \(\text{pop()}\rightarrow\text{push()}\) cannot be selected as data flow test cases, because no global def-use pairs of the \(\text{top}\) data member exist in these two pairs of sequence member functions. For instance, the pair \(\text{pop()}\rightarrow\text{push()}\) cannot be generated because there are no intra-class pairs originating within the \(\text{pop}\) method and terminating within the \(\text{push}\) method.

Are they really not necessary as test cases? Is there a data anomaly in between? After analysing the data flow in the four functions pairs, it can be found that the \(\text{top}\) data member has a \(\text{dd}\) data anomaly in the \(\text{push()}\rightarrow\text{push()}\) and \(\text{pop()}\rightarrow\text{push()}\) sequences. Whether or not the programmer meant \(\text{cout}<<\text{top};\text{ or } \text{cin}>>\text{a};\) instead of \(\text{cin}>>\text{top};\) is unknown until the tester investigates. However, it is clear that there is an anomaly, and the anomaly is the reason that the test cases \(\text{push()}\rightarrow\text{push()}\) and \(\text{pop()}\rightarrow\text{push()}\) cannot be computed.

5 Data Flow Anomaly Detection

In data flow anomaly detection, static analysis detection has the advantage that all potential anomalies of a program can be uncovered without execution. Dynamic detection only analyses the execution path (it is input data dependent) during execution and only the anomalies that lie on that particular path are detected [4]. The purpose of data anomaly detection in this research is to assure that the intra-class data flow test cases can be selected from the sequences of feasible messages. These in turn can be derived from the state transition diagram of the class under test. Therefore, static anomaly detection is adopted in this research.

5.1 An Automatic Object-Oriented Class Testing Tool – MACT

To test a whole class as a unit, the authors in [17, 18] propose a hybrid testing approach that combines state-based testing with data flow anomaly detection to employ class testing at the
intra-class level. To achieve this, a Method for Automatic Class Testing (MACT) tool is designed with five components: a test case tree generator, a test message generator, a test driver, a test result inspector and a definition-use info generator. The first four components support functional testing, and the fifth enables testers to automatically execute data flow anomaly detection. Only the data flow anomaly detection of MACT is demonstrated in this paper, and the detection process will be introduced in the next section.

5.2 Data Flow Testing Approach in the MACT

The transition paths in the state transition diagram show all feasible sequence methods of the required object. Based on the transition paths, the static analysis technique is adopted for detecting data anomalies on data members within sequences of methods. The data anomaly detection steps of the MACT tool are:

1. **Mapping implemented methods to state transitions on the state transition diagram.**
   The state transition depends on the current state and the message received. Therefore, a sequence of state transitions in the state transition diagram implies a sequence of messages sent from outside the object. MACT detects any data flow anomalies which exists in this sequence of messages.

2. **Revealing the global definitions and uses of data members within functions.**
   In object-oriented classes, the methods of an object can define, use or define-use its data members. Therefore, global definitions and uses of each data member in member functions can be extracted by static analysis of the data flow paths. For instance, in the implemented stack class (see Figure 1), the top data member is globally defined in \texttt{push()}, denoted \texttt{def(top)}. In the \texttt{pop()} function, the top data member is globally used and then defined again, which is denoted as \texttt{use-def(top)}.

3. **Generating sequences of occurrences of data members.**
   The sequences of def-use information can be generated by traversing the sequences of member functions. Such that \langle \texttt{def(top)}, \texttt{def(top)}, \texttt{use-def(top)}\rangle is produced during traversing from the \texttt{stack()→push()→pop()} sequence, see Figure 1. Other sequences are:
<def(top), def(top), def(top)> on the stack()→push()→push() sequence and <def(top), def(top), use-def(top), def(top)> on the stack()→push()→pop()→push() sequence, and so on.

(4) Detecting data anomalies of data members within the generated sequences.

In general, an anomaly on a data member in a sequence of data occurs if one or more of the dd, ku and dk data anomalies exists in the sequence [6]. For example, a dd data anomaly exists within stack()→push(), push()→push() and pop()→push() member function sequences respectively. This is because the top’s actions in these sequences are <def(top), def(top)>, <def(top), def(top)> and <use-def(top), def(top)>.

6 Case Study

In this section, the CCoinBox class in [12] is used to demonstrate the difficulty of test case generation, infeasible test messages being generated and necessary test messages being missed. The data-flow criteria are employed to generate the intra-class data flow test cases. To overcome these weaknesses of generating intra-class data flow test cases, the test cases can be selected from sequences of specification messages. Before the selection, it is necessary to detect if any data flow anomalies occurs within the sequences. How to detect intra-class data flow anomalies within the class, to remove the detected anomalies, and to produce feasible intra-class data flow test messages for the CCoinBox class are shown in the example.

6.1 A Coin Box Class Example

The CCoinBox class (a coin box component of a vending machine), see Figure 2, has very simple functionality. Only 25¢ coins are acceptable and vending is allowed when two 25¢ coins are received. There is an error in the implemented CCoinBox class. Kung et al. [12] detect the error using state-based testing. The data flow anomaly detection technique in section 5.2 can also detect the data anomalies and identify the locations of the errors in the CCoinBox class. These will be discussed in section 6.4.
class CCoinBox {
    unsigned totalQtrs; //total quarter collected
    unsigned curQtrs;  //current quarters collected
    unsigned allowVend; // 1 = vending is allowed

public:
    CCoinBox() {Reset();}
    void AddQtr(); //add a quarter
    void ReturnQtrs() {curQtrs = 0;} //return current quarters
    unsigned isAllowVend() {return allowVend;}
    void Reset() {totalQtrs = 0; allowVend = 0; curQtrs = 0;}
    void Vend(); //if allowed, update totalQtrs and curQtrs
};
void CCoinBox :: AddQtr() {
    curQtrs = curQtrs + 1; //add a quarter
    if (curQtrs > 1)       //if more than one quarter is collected
        allowVend = 1;    //then set allowVend
}
void CCoinBox :: Vend() {
    if (isAllowedVend()) { //if allowVend
        totalQtrs = totalQtrs + curQtrs; //update totalQtrs,
        curQtrs = 0;          //curQtrs, and
        allowVend = 0;       //allowVend,
    } else no action
}

6.2 The Def-Use Paths of the Example

Data flow testing techniques may require directed flow graphs, also called control flow graphs, that contain the definitions and uses of data variables. These show the data occurrences within programs and facilitate the computation of def-use pairs. These also help testers select test cases and to detect whether anomalies exist in the program under test.

To seek the global definitions/uses of data members, only the data occurrences within the public member functions of the class are analysed. To simplify the def-use presentation of each member function, each code statement is a unit in which the definition and/or use of data members can occur. In the graph of the AddQtr() function (see Figure 4), the c-use(curQtrs) at node 1 is concerned as to whether the curQtrs data member used in this function has been properly defined in the preceding functions. Moreover, it is necessary to examine: 1) whether the defined data members curQtrs and allowVend at nodes 1 and 3, in Figure 4, can be used in the succeeding functions.
void CCoinBox :: AddQtr() {
  1  curQtrs = curQtrs + 1;  //add a quarter
  2  if (curQtrs > 1)       //if more than one
      //quarter is collected
    3    allowVend = 1;  //then set allowVend
}

Figure 4 The C++ code AddQtr() member function on the left and its directed flow graph on the right. Nodes in the graph represents statements in the function; start and terminate nodes are added for analysis.

2) Whether the allowVend, curQtrs and totalQtrs data members used in the Vend() member function have already been defined in the preceding functions. 3) Whether the allowVend, curQtrs and totalQtrs data members defined in the Vend() can be used in the succeeding functions. If they have not been/cannot be, a definition/using anomaly is present. The allowVend, curQtrs and totalQtrs data members are defined in the constructor, CCoinBox(), and the curQtrs data member is defined in the ReturnQtrs() function (see Figure 2).

6.3 The Complexity of Data Flow Test Case Selection

6.3.1 Generating Data Flow Test Cases

As with the discussion in [10, 11], the test cases generated to cover associations between definitions and uses of each data member can be yielded from the du-path criteria. The definitions and uses of data members among the functions of the CCoinBox class are shown in Table 3.

<table>
<thead>
<tr>
<th>Data members</th>
<th>curQtrs</th>
<th>allowVend</th>
<th>totalQtrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>definition</td>
<td>CCoinBox(), AddQtr(), ReturnQtrs(), Vend()</td>
<td>CcoinBox(), AddQtr(), Vend()</td>
<td>CcoinBox() Vend()</td>
</tr>
<tr>
<td>c-use</td>
<td>AddQtr(), Vend()</td>
<td>Vend()</td>
<td>Vend()</td>
</tr>
<tr>
<td>p-use</td>
<td>AddQtr()</td>
<td>Vend()</td>
<td>Vend()</td>
</tr>
</tbody>
</table>

Table 3 The definitions and uses of the data members in the member functions of the CcoinBox class
The arcs in Table 3 show eight pairs of sequence methods that are used for intra-class testing. For example, d-pu paths with respect to the curQtrs data member can be found in CCoinBox() → AddQtr(), AddQtr() → AddQtr() and Vend() → AddQtr(). There are d-cu paths of curQtrs in CCoinBox() → AddQtr() and ReturnQtrs() → Vend(). A d-cu path of the totalQtrs data member exists in CCoinBox() → Vend(). Two d-pu paths with respect to the allowVend are CCoinBox() → Vend() and AddQtr() → Vend(). Using Table 3, the sets of d-cu and d-pu of each data member of the CcoinBox class can be computed, see Table 4.

**Table 4 The d-cu and d-pu sets for the class CCoinbox**

<table>
<thead>
<tr>
<th></th>
<th>dcu(curQtrs, CCoinBox())={AddQtr(), Vend()}</th>
<th></th>
<th>dpu(curQtrs, CCoinBox())={AddQtr()}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>dpu(curQtrs, AddQtr())={AddQtr(), Vend()}</td>
<td>10</td>
<td>dpu(curQtrs, AddQtr())={AddQtr()}</td>
</tr>
<tr>
<td>2</td>
<td>dpu(curQtrs, ReturnQtrs())={AddQtr(), Vend()}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>dpu(curQtrs, Vend())={AddQtr(), Vend()}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>dpu(totalQtrs, CCoinBox())={Vend()}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>dpu(totalQtrs, Vend())={Vend()}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>dpu(allowVend, CCoinBox())=Ø</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>dpu(allowVend, AddQtr())=Ø</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>dpu(allowVend, Vend())=Ø</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>dpu(allowVend, Vend())=Ø</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A sequence of messages CCoinBox() → AddQtr() → AddQtr() → Vend(), which achieves the all definitions coverage, can be established. However, it is a time-consuming task to yield the sequence to achieve all defs or all uses coverage by computing the du-paths of every data member in the CCoinBox class.

### 6.3.2 The Difficulty of Test Case Generation

Weyuker [19] proposed that all du-paths require an exponential number of test cases in the worst case scenario. If \( d \) is the number of two way decisions in the program, then in the worst case all du-paths require \( O(2^d) \). For example, suppose a program comprises a sequence of \( d \) IF-THEN-ELSE statements, and each of them defines and uses a variable \( x \). All du-paths may then require \( 2^d \) test paths.

Suppose a class has \( M \) data members and \( N \) member functions, and that every data member is defined and used in every member function. The maximum def-use paths across two member
functions with respect to every data member are $M \times N^2$ [17, 18]. However, the number of permutations of $N$ functions taking two functions at a time is $P(N, 2)$. This indicates that there are $N!/(N-2)!$ ways of choosing two functions from $N$ functions, and means that many test cases, selected using def-use path criterion, are redundant. In the $CCoinBox$ class case, there are 12 (i.e. $P(4, 2)$) pairs of sequences of member functions. This shows that five of 17 pairs of the computed member functions (test messages) are redundant.

Table 3 illustrates that it is more difficult to generate all possible intra-class test cases (especially sequences of test cases) for a class, in order to achieve all defs/all uses coverage, if the class contains more data members and member functions.

In the $CCoinBox$ class, for instance, the $totalQtrs$ data member is defined in the $CCoinBox()$ function, and also used in the $Vend()$ function. There can be several member functions with different performance orders within the interval of $CCoinBox() \rightarrow Vend()$. Each of the following examples has a du path with respect to the $totalQtrs$ data member.

$$
CCoinBox() \rightarrow Vend();
CCoinBox() \rightarrow AddQtr() \rightarrow AddQtr() \rightarrow Vend();
CCoinBox() \rightarrow AddQtr() \rightarrow ReturnQtrs() \rightarrow AddQtr() \rightarrow AddQtr() \rightarrow Vend();
\vdots
CCoinBox() \rightarrow AddQtr() \rightarrow AddQtr() \rightarrow ReturnQtrs() \rightarrow AddQtr() \rightarrow AddQtr() \rightarrow AddQtr() \rightarrow Vend();
$$

The above sequence $CCoinBox() \rightarrow Vend()$ has a du-path with respect to the $totalQtrs$ data member, but it is an infeasible sequence based on the requirement.

6.3.3 Infeasible and Ambiguous Test Cases

Weyuker in [19] finds that the infeasible path problem is the primary practical difficulty in using the all du-path criterion, as there are many infeasible paths to contend with. In the automatic generation of methods to satisfy the data flow criteria, the problem of generating infeasible sequences is impossible to avoid [14].

A sequence of method calls from outside of the class can be specification infeasible or implementation infeasible. Infeasible sequence methods (subpaths) cannot be executed
according to the specification. For example, the $\text{CCoinBox()} \rightarrow \text{Vend()}$ sequence should not be required in the specification. The $\text{CCoinBox()} \rightarrow \text{AddQtr()} \rightarrow \text{ReturnQtrs()} \rightarrow \text{Vend()}$ sequence is an implementation infeasible example, because a $\text{CCoinBox}$ object cannot accept a vending message from the client object when the inserted coins have been returned.

### Table 5 The infeasible and feasible pairs of test messages of the CcoinBox

<table>
<thead>
<tr>
<th>Data members</th>
<th>Infeasible pairs of member functions</th>
<th>Feasible pairs of member functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>curQtrs</td>
<td>$\text{CCoinBox()} \rightarrow \text{Vend()}$  \text{ReturnQtrs()} \rightarrow \text{Vend()} \text{Vend()} \rightarrow \text{Vend()}</td>
<td>$\text{CCoinBox()} \rightarrow \text{AddQtr()}$  \text{AddQtrs()} \rightarrow \text{AddQtr()} \text{AddQtrs()} \rightarrow \text{Vend()} \text{ReturnQtrs()} \rightarrow \text{AddQtr()} \text{Vend()} \rightarrow \text{AddQtr()}$</td>
</tr>
<tr>
<td>totalQtrs</td>
<td>$\text{CCoinBox()} \rightarrow \text{Vend()}$  \text{Vend()} \rightarrow \text{Vend()}</td>
<td>$\text{AddQtrs()} \rightarrow \text{Vend()}$</td>
</tr>
<tr>
<td>allowVend</td>
<td>$\text{CCoinBox()} \rightarrow \text{Vend()}$  \text{Vend()} \rightarrow \text{Vend()}</td>
<td>$\text{AddQtrs()} \rightarrow \text{Vend()}$</td>
</tr>
</tbody>
</table>

Using du-paths criteria, pairs of member functions have been computed and shown in Table 4. Some of them are infeasible, see Table 5.

To obtain all possible valid test cases and to reduce the cost of testing, the redundant paths should be removed and the infeasible test cases should be eliminated from the test cases that are generated based on data flow testing criteria.

### 6.3.4 The Selection of Feasible Test Message Sequences

If the class under test is implemented by following the state transition diagram, then the paths of transition in the diagram reveal the feasible sequences of member functions of the implemented 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 def-use pair technique.

After traversing the state transition diagram of the $\text{CCoinBox}$ class (in Figure 3), the sequences of member functions can be produced. Some of them are:
1. \texttt{CCoinBox()} → \texttt{AddQtr()};
2. \texttt{CCoinBox()} → \texttt{AddQtr()} → \texttt{ReturnQtrs()};
3. \texttt{CCoinBox()} → \texttt{AddQtr()} → \texttt{AddQtr()};
4. \texttt{CCoinBox()} → \texttt{AddQtr()} → \texttt{AddQtr()} → \texttt{ReturnQtrs()};
5. \texttt{CCoinBox()} → \texttt{AddQtr()} → \texttt{AddQtr()} → \texttt{AddQtr()} → \texttt{Vend()};
  
  6. \texttt{CCoinBox()} → \texttt{AddQtr()} → \texttt{ReturnQtrs()} → \texttt{AddQtr()} → \texttt{AddQtr()} → \texttt{Vend()};

The above sequences can be used as data flow test cases to examine the occurrences of the \texttt{curQtrs} data member. Only the above sequences 5 and 6 can be selected for examining the \texttt{totalQtrs} data member, as it is defined and used in the \texttt{CCoinBox()} and \texttt{Vend()} member functions respectively. The above six sequences of test messages cannot be utilised to examine the \texttt{allowVend} data member. This is because no global du-paths with respect to the \texttt{allowVend} data member are present within those sequences. This implies whether any data anomalies of the \texttt{allowVend} data member exist within the class, as the anomalies may distort the du-paths that are the basis of selecting the data flow test cases.

### 6.4 The Data Flow Anomalies in the Coin Box Class

The def-use information of the data members within the \texttt{CCoinBox()} → \texttt{AddQtr()} → \texttt{ReturnQtrs()}, \texttt{CCoinBox()} → \texttt{AddQtr()} → \texttt{AddQtr()} → \texttt{Vend()} and \texttt{CCoinBox()} → \texttt{AddQtr()} → \texttt{AddQtr()} → \texttt{ReturnQtrs()} sequences of test messages is chosen as an example to explain the data anomaly detection technique used in this research, see section 5.2. To assist explanation, the data occurrence list of the messages has been divided into three rows with different data members, as shown in Table 6. The def-use pairs of each data member in the lists are computed in order to find any data anomalies. The grey bold arcs in Table 6 show the pairs of definition-use data members. The black dotted arcs in the first row of Table 6 show that the \texttt{allowVend} and \texttt{totalQtrs} data members defined in \texttt{CCoinBox()} and the \texttt{curQtrs} data member defined in \texttt{ReturnQtrs()} are not used within the sequence of member functions. However, they may be used within other member functions such as \texttt{AddQtr()} → \texttt{AddQtr()} → \texttt{Vend()}. The \texttt{curQtrs} data member defined in \texttt{ReturnQtrs()} can be used in the first \texttt{AddQtr()} function, and the \texttt{totalQtrs} data member can be used in the \texttt{Vend()} function.
Two \textit{allowVend} data members are defined in \texttt{CCoinBox()} and the second \textit{AddQtr()} function respectively without any intervening use. This can be found in the second row of Table 6. The same data anomaly can be found in the third row of the table. This raises the questions: Is it necessary to define the \textit{allowVend} data member in the constructor? If it is necessary, should it be used before the second definition in the \textit{AddQtr()}? In the third row, an \textit{allowVend} data member (defined in the \textit{AddQtr()} function) and a \textit{curQtrs} data member (defined in the \textit{ReturnQtrs()} function) are not used.

As discussed previously, the defined \textit{curQtrs} data member can be used in the \textit{AddQtr()→AddQtr()→Vend()} sequence. In the preceding sequence, however, another defined \textit{allowVend} data member follows the unused \textit{allowVend} data member, as computed below.

The rows 2 and 3 of Table 6 show that there are no def-use paths with respect to the \textit{allowVend} data member within the message sequences. Therefore, these two message sequences cannot be selected as test cases to examine the \textit{allowVend} data member. An error, which the \textit{allowVend} data member is not reset to zero in \textit{ReturnQtrs()}, results in the vendee being provided with a
free drink; if the vending process follows the \textit{AddQtr} \rightarrow \textit{AddQtr} \rightarrow \textit{ReturnQtrs} \rightarrow \textit{Vend} sequence. Unfortunately, this sequence of test messages cannot be selected as a test case for examining the \textit{allowVend} data member based on the du-path criterion. The reason is that the \textit{allowVend} has \texttt{dd} and \texttt{d-} data anomalies within the sequence, and these anomalies prevent the sequence from being selected as an intra-class data flow test case.

This case shows that the error exists within the \texttt{CCoinBox} class cannot be detected using the data flow test cases (generated in section 6.3.4), because the data flow anomalies within the class break the def-use pairs and influence the estimate of data flow associations. Kung et al. [12] detected the error using state-based testing. However, the data flow anomaly detection technique of MACT (see section 5.2) can find the data anomalies as well as pinpoint the error location in the \textit{ReturnQtrs()} member function of the \texttt{CCoinBox} class.

To remove the data anomaly on the \textit{allowVend} data member, should the \textit{allowVend} data member defined in the \textit{AddQtr()} function be used in the following \textit{ReturnQtrs()} function? Should it be used as a predicate use or a computation use?

\section*{6.5 Removing the Data Flow Anomalies and Code Optimisation}

The basic techniques of removing data anomalies can be: to add an intervening use into the double defined variable, to exclude the used variable without definition or to avoid the defined variable without use. There are several ways to solve the data anomalies of the \textit{allowVend} data member in functions \texttt{CCoinBox()} and \textit{AddQtr()}. One method is to reference the \textit{allowVend} data member as a predicate-use based on data flow analysis. This is shown in the following \textit{ReturnQtrs()} and \textit{AddQtr()} functions. The former replaces the \textit{ReturnQtrs()} function, which only contains a single statement \texttt{curQtrs = 0;} in Figure 2. Thus the changes are:

\begin{verbatim}
void ReturnQtrs() {
    if (curQtrs > 0) {curQtrs = 0;}
    if (allowVend > 0) {allowVend = 0;}
}
\end{verbatim}

\begin{verbatim}
void CCoinBox::AddQtr(){
    curQtrs = curQtrs + 1;
    if (curQtrs > 1 && allowVend == 0){
        allowVend = 1; }
}
\end{verbatim}
This replacement not only eliminates the data anomaly of the `allowVend` data member but also removes the error of *providing with a free drink*, discussed in section 6.4. If the `ReturnQtrs()` function in Figure 2 was coded as above and the sequence of member functions `CCoinBox() → AddQtr() → AddQtr() → ReturnQtrs()` were executed, then the data members’ values would be `curQtrs == 0` and `allowVend == 0`, rather than the error result `curQtrs == 0` and `allowVend == 1`. This indicates that the above correction solves the data anomaly and removes the error.

Kung et al. in [12] have eliminated the error by changing the `ReturnQtrs()` function to:

```c
void ReturnQtrs() { curQtrs = 0; allowVend = 0; }.
```

However, this solution does not exclude the data anomaly of the `allowVend` data member. Additionally, it raises another argument: should the `allowVend` and `curQtrs` data members be reset to zero in the function if their values are already zero? For example, if the sequence of functions `CCoinBox() → AddQtr() → ReturnQtrs() → ReturnQtrs()` are applied to a `CCoinBox` object, this can be compared to switching off a light when it is already off.

### 6.6 The Intra-Class Data Flow Test Cases for the Coin Box Class

After removing the data flow anomalies, discussed in section 6.5, the occurrences of the three data members (i.e. `curQtrs`, `allowVend` and `totalQtrs`) within the feasible sequences of messages are listed in Table 7. For instance, the data flow anomalies of the `allowVend` data member (marked with black dotted arcs in Table 6) are removed in Table 7. The def-use paths within the sequences of messages (marked with grey bold arcs) in Table 7 show that those message sequences can be computed as intra-class data flow test cases to examine the `curQtrs`, `allowVend` and `totalQtrs` data members.
**Table 7** The definitions - uses of the data members within sequences of the modified member functions

<table>
<thead>
<tr>
<th></th>
<th>CCoinBox()</th>
<th>AddQtr()</th>
<th>AddQtr()</th>
<th>ReturnQtrs()</th>
<th>Vend()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>def(curQtrs), def(allowVend), def(totalQtrs)</td>
<td>cu(curQtrs), def(curQtrs), pu(curQtrs), pu(allowVend)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pu(allowVend), def(curQtrs), pu(allowVend)</td>
<td></td>
</tr>
</tbody>
</table>

7 Conclusions and Future Work

For reducing the effort of producing intra-class data flow test cases for a class under test, the test cases can be computed from the feasible sequences of messages, which are derived from the state transition diagram of the class. However, some of test cases may be missed if any intra-class data flow anomalies exist within the sequences of messages. This has been discussed in sections 4 and 6.4. Using the data flow anomaly detection technique (discussed in section 5), intra-class data flow anomalies can be detected and the locations of the errors in the CCoinBox class can be identified. Moreover, from the discussion using a stack class in section 4, the authors recommend that intra-class data flow anomaly detection should be performed before intra-class data flow test case generation. If no data flow anomalies exist in the class, then data flow test cases can be generated straight away. Otherwise, the data flow anomalies should be excluded first. Moreover, the program code can be optimised during data flow anomaly removal, see section 6.5.

Sequences of messages derived from the state transition diagram of a class are used to detect data flow anomalies at the intra-class level. Thus, this data flow anomaly detection method is able to validate classes that can be modelled with a state transition diagram. The authors in [17,
18] proposed a threaded multi-way tree to transform the graph of the state transition diagram. The tree contains the member functions of the class under test and the global definition and use of data members in the member functions. Future work on how to automatically detect data flow anomalies and generate intra-class data flow test cases is currently being addressed.
References


