Chapter 4

Enhancing Data Flow Testing of Classes through Design by Contract

4.1 Introduction

Design by Contract (DbC) is an approach for constructing object-oriented software to improve the quality of the software [20]. DbC associates the contract specification (pre and postconditions of the methods of a class, and a class invariant) with the design of a class (DbC is described in Chapter 3). In this chapter, we combine the data flow testing and design by contract to develop a testing technique called "Data flow testing using Contract". This technique generates the test cases (message sequences) on the basis that: a method M2 can be invoked after method M1 only if postcondition of M1, precondition of M2 and class invariant are true. The execution of a test case (sequence of messages) in a DbC class and a conventional class is as shown in figure 4.1.

![Diagram showing test case execution for a DbC vs conventional class]

**Figure 4.1: Test case execution for a DbC Vs Conventional Class**

Let us take a test case containing the messages M1, M2, M3, M4, M5, M6. Now, suppose there is an error in the implementation of method M3. When we execute
this test case then that error is known after the execution of M3 in a DbC class because the postcondition of M3 comes out to be false. Since postcondition of M3 is false hence M4 cannot be executed and it tells that there is an error in the implementation of method M3. But for the conventional class (the one which is not having software contracts) all the methods from M1 to M6 are executed and the error is known only after the execution of M6. Also, we don’t know exactly which method is having the error. This shows that an error in a DbC class is known earlier than in a conventional class. An example of a queue class developed using DbC is shown in figure 4.2. DbC class contains preconditions and postconditions of methods, and a class invariant. Mathematically, DbC can be expressed as:

(I and Pre) M (Post and I)

Where I is the class invariant, Pre and Post are the precondition and postcondition of the method M of the class respectively. A method of a class is executed if its precondition and class invariant are satisfied. After the method execution the postcondition of the method and class invariant should be satisfied. Based on the above formula we can test the sequences of methods as follows:

(I and Pre1) M1 (Post1 and I and Pre2) M2 (Post2 and I and Pre3) .............. (I and PreN) MN (PostN and I)

In the above sequence any method Mi can only be invoked if its precondition, class invariant and postcondition of method Mi-1 are satisfied. Since the number of message sequences (test cases) can be infinite. Hence, to choose a set of quality test cases, we use data flow testing criterion.
class queue  
{
  int q:Array[SIZEx];
  int front;
  int rear;
  public:
  queue();
  void add(int i);
  int remove();
};
queue::queue() {  
  Pre(true);
  front=0; rear=0;
  Post(front==0 && rear==0)
}
void queue::add(int i) {  
  Pre (rear < SIZE);
  q:Array[rear] = i;
  rear = rear + 1 ;
  Post(q [rear-1] = i);
}
int queue::remove()  
{  Pre (front!=rear);
  element = q:Array[front];
  front = front +1;
  Post(! Is_Full ());
  return element;
}
bool queue_invariant() {return front >= 0 && rear >= 0  
&& front < SIZE && rear < SIZE;}

Figure 4.2: C++ code for queue class using DbC

Design by Contract is a well established way of producing quality software and
from the above we can deduce that this methodology produces quality test cases
because:

1. The test sequences (message sequences) are executed only if the software
contract (precondition, postcondition and class invariant) is satisfied.
2. Due to assertions (contracts) some of the p-uses are not generated which reduces the number of def-use pairs and further reduces the test cases and the testing effort. Hence, the testability of the software is improved.

3. Breaking of the precondition of a method directly tells about the infeasibility of the test sequence.

4. Breaking of the postcondition of a method tells about the implementation error in the currently executed method.

Software contracts have been used in a number of ways to improve the quality of the software. In this chapter, software contracts are used for testing an object oriented class using data flow testing criterion.

4.2 Data Flow testing using Software Contracts

In this section, we present a class testing technique called “Data flow Testing using Contract”. The implementation of a class is tested with respect to the contract specification of the class. Briefly, our technique consists of the following steps:

1. Given a class constructed through DbC, a class flow graph (CFG) is generated from its contract specification. The contract specification includes preconditions and postconditions for the methods of the class, and an invariant for the class.

2. The definitions and uses (p-use and c-use) of each data member of the class are identified and infeasible def-use pairs are eliminated based on the contract specification.

3. The test cases are generated from the CFG using the conventional data flow testing criteria.

The above steps are described in the following subsections.
4.2.1 Conversion of the Contract Specification to Class flow graph

Given a class constructed through DbC, a class flow graph is generated from its contract specification. A conventional flow graph represents the program's control structure and contains the nodes and the edges. The nodes of the program are represented by the program statements and edges represent the flow of control between the statements. In this technique, we make a Class Flow Graph (CFG) from the contract specification of the class. A Class Flow Graph is defined as follows:

**Definition:** A class flow graph (CFG) is a directed graph $G = (N, E)$ where

- $N$ is the set of nodes. Nodes represent the methods of the class. An ellipse in the diagram shows this type of node.
- $E$ is the set of edges. Method nodes are connected by edges. Whether an edge exists between two method nodes or not is decided by the contract specification of the class. An edge is drawn from method node $M_1$ to $M_2$ if the postcondition of $M_1$ satisfies the precondition of $M_2$ and invariant of the class is true.

There are few rules to draw the class flow graph.

1. The constructor method has only outgoing edges and no incoming edges. Because, the constructor is the method which creates an object and rest of the methods are the ones which are applied on this object.

2. Every method node is connected to itself through an edge except the node of the constructor method. Figure 4.3 shows the class flow graph for queue class shown in figure 4.2.
4.2.2 Generation of the Test cases

In this step, we generate the test cases using the data flow testing criteria, which selects the test cases according to the definitions and the uses of the data members of the class. A test case is a sequence of methods of a class.

<table>
<thead>
<tr>
<th>Data member</th>
<th>front</th>
<th>rear</th>
<th>q_Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>def</td>
<td>queue, remove</td>
<td>queue, add</td>
<td>Add</td>
</tr>
<tr>
<td>c_use</td>
<td>remove</td>
<td>add</td>
<td>remove</td>
</tr>
</tbody>
</table>

Table 4.1: Definitions and Uses of data members of the queue class

Each data member of a class is classified as being defined, computation used (c-use) or predicate used (p-use) and this is identified by the following rules.

Rules: Let G be a CFG of a class C and d be a data member of C.

a) d is said to be defined at method node if this method assigns a value to d.

b) d is said to be c-used at method node if this method references d.

c) d is said to be p-used at method node if it is used in a condition in the method.
<table>
<thead>
<tr>
<th>Feasible def-use pairs</th>
<th>Infeasible def-use pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(remove(), remove())</td>
<td>(queue(), remove())</td>
</tr>
<tr>
<td>(queue(), add())</td>
<td>(add(), add())</td>
</tr>
<tr>
<td>(add(), remove())</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Feasible and Infeasible def-use pairs of class queue

Following rules are followed to form the test cases.

1. Constructor method is the first method of any test case.

2. If there is an edge from M1 to M2 in the class flow graph then after the execution of the method M1, the postcondition of the method M1 and the invariant of the class are evaluated. This evaluation leads to the following cases:
   - Case 1: If the postcondition of M1 is true and satisfies the precondition of M2 and the class invariant is true then we follow the edge <M1, M2> and execute M2.
   - Case 2: If the postcondition of M1 is true but does not satisfy the precondition of M2. We don’t follow the edge <M1, M2> as it shows that this sequence is infeasible. Hence, M2 cannot be executed.
   - Case 3: If the postcondition of M1 is false then we don’t follow the edge <M1, M2> as it means that there is an error in the implementation of the method M1.

In order to apply the data flow testing criteria to generate the test cases for a class we need to find the definitions and uses of every data member of the class. The definitions and uses of data members of the queue class are shown in Table 4.1. From the Table 4.1 we can derive the def-use pairs. Some of the def-use pairs are infeasible according to the contract specification of the class. The feasible and
infeasible def-use pairs of queue class are shown in table 4.2. A number of data flow testing criteria exist in literature [110]. The test cases can be generated using various criterion of data flow testing. Some of the test cases are as shown below.

1. queue, add, add, remove, remove
2. queue, add, remove, add, remove
3. queue, add, add, add, remove
4. queue, add, add, add, remove, remove

Since (queue, remove) is an infeasible def-use pair hence, no test case can have this pair. This pair was eliminated as a result of the static analysis of the contract specification of the class. There are certain test cases which are infeasible and can only be detected by the dynamic analysis (i.e. at the run time). For example the test case "queue, add, remove, remove" is infeasible and can not be known through the static analysis of the contract specification. This can only be known to be infeasible when we execute this test case. In this sequence we can not execute the second remove method as its precondition comes out to be false. Hence, this technique has an added advantage of the executable specification, through which we can know about the infeasible sequence at the run time.

4.3 Comparison with the conventional programming

Conventional programming contains a lot of checks to ensure that everything is right. Hence, due to these extra conditions, the number of def-use pairs is more in the conventional object oriented class as compared to the DbC version of the class. The reason is that, in the conventional class there're certain conditions which become the preconditions in a DbC class. Hence, the number of p-uses is reduced in a DbC class. The C++ code for the queue class, using the conventional
style of programming is shown in figure 4.4. The definitions and uses for the conventional queue class are shown in table 4.3.

```c++
class queue
{
    int q_Array[SIZE];
    int front;
    int rear;
    public:
    queue(){
        front=0;
        rear=0;
    }
    void add(int element){
        if(isfull()){
            cout<<"queue is FULL ";
            q_Array[rear] = element;
            rear++;;
        }
    }
    int remove()
    {
        if(isempty())
            {cout<<"Queue Empty ";}
        return(q_Array[front++]);
    }
    int queue::isempty()
    {
        if(front == rear) return 1;
        else return 0;
    }
    int queue::isfull()
    {
        if(rear == SIZE) return 1;
        else return 0;
    }
};
```

**Figure 4.4: Conventional queue class**

Table 4.4 shows the feasible and infeasible def-use pairs for the conventional queue class. We can see that the number of def-use pairs in the conventional class is more than in a DbC class. Some of the test cases are:
<table>
<thead>
<tr>
<th>Data Member</th>
<th>front</th>
<th>rear</th>
<th>q_Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Def</td>
<td>queue(), remove()</td>
<td>queue(), add()</td>
<td>add()</td>
</tr>
<tr>
<td>c_use</td>
<td>remove()</td>
<td>add()</td>
<td>remove()</td>
</tr>
<tr>
<td>p-use</td>
<td>remove()</td>
<td>remove(), add()</td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.3: Definitions and uses in conventional queue class**

- queue, add, add, remove, remove, add, add
- queue, add, remove, add, remove
- queue, add, add, add, remove
- queue, add, add, remove
- queue, add, add, remove, add, remove

<table>
<thead>
<tr>
<th>Feasible def-use pairs</th>
<th>Infeasible def-use pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(remove(), remove())</td>
<td>(queue(), remove())</td>
</tr>
<tr>
<td>(remove(), remove())</td>
<td>(queue(), remove())</td>
</tr>
<tr>
<td>(queue(), add())</td>
<td>(queue(), remove())</td>
</tr>
<tr>
<td>(queue(), add())</td>
<td>(queue(), remove())</td>
</tr>
<tr>
<td>(add(), add())</td>
<td>(queue(), remove())</td>
</tr>
<tr>
<td>(add(), add())</td>
<td>(queue(), remove())</td>
</tr>
<tr>
<td>(add(), remove())</td>
<td>(queue(), remove())</td>
</tr>
</tbody>
</table>

**Table 4.4: feasible and infeasible def-use pairs in the conventional queue class**

There are three benefits of using the contract specification for class testing.

1) Determination of the implementation error in the method of a class.

2) Finding the infeasible sequence.

3) Reduced set of def-use pairs as compared to the conventional class.
While executing a test case if the postcondition of a method is not satisfied that shows the implementation error in that method. The error can be removed by comparing the specification with the implementation because the contract specification is consistent with the implementation of the class. For example in the “add” method of the queue example, if the statement \( q[\text{rear}] = \text{element}; \) is by mistake written as: \( q[\text{rear}-1] = \text{element}; \) then the sequence “queue, add, remove” will detect the error before the execution of the “remove” method, as the postcondition of the method “add” will come out to be false. Hence, we would be able to know that there is an error in the implementation of “add” method of the queue class.

A message sequence \(<A, B>\) is infeasible if the postcondition of A does not satisfy the precondition of B, where A and B are the methods of the class. For example, in the queue class, the following sequence is an infeasible sequence.

**Test Case:** “queue, add, add, remove, remove, remove”

In this sequence the precondition of last remove is not satisfied by the postcondition of second remove in the sequence. Hence, this sequence does not get generated. The above message sequence is infeasible, can only be known at the run time. A few infeasible sequences are removed by the contract specification only. For example “queue, remove”, is a sequence, which is shown to be infeasible by the contract specification itself. This is the reason that there is no edge from the node queue to the node remove.

The DbC approach makes the code for the method body small. As some of the condition checking which is a part of the method body in conventional programming approach becomes the part of the precondition of the method. Hence, p-uses are reduced, which leads to reduced def-use pairs and less testing.
cost. Our approach is different from other specification based approaches [1, 22, 97, 115, and 154] in two ways: 1) our approach uses contract specification, which is executable hence, a method in a test case is executed only if its precondition and class invariant are satisfied. If it is not satisfied then it leads to an infeasible sequence. Also, if after executing a method its postcondition is not satisfied then it leads to implementation error in the method. Hence, through the contract specification we produce the quality test cases. 2) Since our specification is executable, it helps in finding whether a test case (message sequence) should be generated or not.

This technique has been validated through a number of C++ classes. We have compared the testing of a DbC class with a conventional class. A DbC class is different from a conventional class because in a DbC class methods are embedded with pre and postconditions and has a class invariant. Through this technique we find that the numbers of def-use pairs are reduced in a DbC class because some of the conditions in conventional class become the part of the precondition in a DbC class.

4.4 Case Study

In this section, we take an example of the CoinBox class of vending machine (given by Kung [89]) to demonstrate the proposed technique. The functionality of the CoinBox class is as given below: The CoinBox accepts the quarters, 2 quarters are needed to vend (to get the drink). There are three attributes in this class: AllowVend, CurQtrs, and TotalQtrs. The number of methods in this class is four which are: CoinBox(), AddQtr(), ReturnQtr(), and Vend(). CoinBox() is the constructor of the class, AddQtr() is used for adding a quarter, ReturnQtr() is used
for returning the current quarters, Vend() gives the drink. Vending is enabled if the value of AllowVend is 1 otherwise vending is disabled, CurQtrs keeps track of the current quarters and TotalQtrs keeps track of the total quarters. The C++ code for the DbC version of Coinbox class is shown in figure 4.5.

There is an implementation error in this CoinBox class. The error is that in the method ReturnQtr(), the variable AllowVend should be made 0 to disable vending if the the method ReturnQtr() is executed. This statement is missing in the code which makes the message sequence CoinBox(), AddQtr(),AddQtr(), ReturnQtr(), Vend() as valid and a person gets drink even if the quarters are returned. Kung [89] has used state based testing technique to remove this error. We will detect this error using our proposed technique. The class flow graph for this class is shown in figure 4.6. There is no edge from CoinBox() to Vend() and from CoinBox() to ReturnQtr() because the preconditions of the methods Vend() and ReturnQtr() are not satisfied by the postcondition of the CoinBox() method. For the similar reason there is no edge from ReturnQtr() to Vend() and from Vend() to ReturnQtr(). The definitions, c-uses and p-uses of each data member of the CoinBox class are shown in table 4.5. The C++ code for the conventional CoinBox class is shown in figure 4.7. To compare the DbC class with the conventional CoinBox class we find the definitions, c-uses and p-uses of every data member of the conventional CoinBox class as shown in table 4.7. The feasible and the infeasible def-use pairs of the conventional CoinBox class are shown in table 4.8. The feasible and infeasible def-use pairs for the DbC version of the CoinBox class are shown in table 4.6. Some of the test cases generated are:
CoinBox{
    boolean AllowVend;
    int CurQtrs;
    int TotalQtrs;
    CoinBox ()
    {Pre(True); //Precondition of Constructor
     AllowVend=0;CurQtrs=0;TotalQtrs=0;
     Post(AllowVend==0 & & CurQtrs==0 & &
     TotalQtrs==0); //Postcondition of constructor}
    AddQtr()
    {
     Pre (0<=CurQtrs<=1, AllowVend ==0);
     //Precondition of the method AddQtr
     CurQtrs = CurQtrs + 1; If (CurQtrs >1)
     AllowVend = 1;
     Post (CurQtrs=1 & & AllowVend==0!!CurQtrs=2 & &
     AllowVend==1); //Postcondition of the method }
    ReturnQtr()
    {
     Pre(CurQtrs = =1 & & AllowVend= = 0!!CurQtrs ==2
     & & AllowVend == 1); //Precondition of the method
     CurQtrs =0;
     Post(0<=CurQtrs<=1, AllowVend ==0);
     //Postcondition of the method ReturnQtr}
    Vend ()
    {Pre(CurQtrs= =2 & & AllowVend= =1);
     //Precondition of the method Vend
     Totalqtrs = Totalqtrs + CurQtrs;CurQtrs= 0;
     AllowVend = 0;
     Post (CurQtrs== =0 & & AllowVend = = 0& &
     Totalqtrs = Totalqtrs + CurQtrs); }//Postcondition of
     //the method }
Inv ((0 <=Curqtr<=2)& &; 0<=AllowVend<=1// Invariant of the
CoinBox class}

Figure 4.5: DbC version of Coinbox class

1. CoinBox(), AddQtr(), AddQtr(), Vend(), AddQtr(), AddQtr(), Vend()
2. CoinBox(), AddQtr(), ReturnQtr(), AddQtr, AddQtr, Vend()
3. CoinBox(), AddQtr(), Vend()--Infeasible sequence.
Figure 4.6: CFG for the CoinBox Class

<table>
<thead>
<tr>
<th>Data member</th>
<th>AllowVend</th>
<th>CurQtrs</th>
<th>TotalQtrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition</td>
<td>CoinBox(),</td>
<td>CoinBox(),</td>
<td>CoinBox(),</td>
</tr>
<tr>
<td></td>
<td>AddQtr(), Vend()</td>
<td>AddQtr(),</td>
<td>AddQtr(),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ReturnQtr(),</td>
<td>ReturnQtr(),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vend()</td>
<td>Vend()</td>
</tr>
<tr>
<td>c_use</td>
<td>AddQtr(), Vend()</td>
<td></td>
<td>Vend()</td>
</tr>
<tr>
<td>p_use</td>
<td>AddQtr()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Definition and uses of data members of CoinBox class (DbC version)

<table>
<thead>
<tr>
<th>Feasible def-use pairs</th>
<th>Infeasible def-use pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(CoinBox(),AddQtr()),</td>
<td>(CoinBox(),Vend())</td>
</tr>
<tr>
<td>(AddQtr(),AddQtr())</td>
<td>(Vend(),Vend())</td>
</tr>
<tr>
<td>(ReturnQtr(),AddQtr())</td>
<td>(ReturnQtr(),Vend())</td>
</tr>
<tr>
<td>(Vend(),AddQtr())</td>
<td>(CoinBox(),Vend())</td>
</tr>
<tr>
<td>(AddQtr(),Vend())</td>
<td>(Vend(),Vend())</td>
</tr>
<tr>
<td>(CoinBox(),AddQtr())</td>
<td></td>
</tr>
<tr>
<td>(AddQtr(),AddQtr())</td>
<td></td>
</tr>
<tr>
<td>(ReturnQtr(),AddQtr())</td>
<td></td>
</tr>
<tr>
<td>(Vend(),AddQtr())</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6: Feasible and infeasible def-use pairs of CoinBox class (DbC version)
<table>
<thead>
<tr>
<th>Data member</th>
<th>AllowVend</th>
<th>CurQtrs</th>
<th>TotalQtrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition</td>
<td>CoinBox(), AddQtr(), Vend()</td>
<td>CoinBox(), AddQtr(), ReturnQtr(), Vend()</td>
<td>CoinBox(), Vend()</td>
</tr>
<tr>
<td>C_use</td>
<td></td>
<td>AddQtr(), Vend()</td>
<td>Vend()</td>
</tr>
<tr>
<td>P_use</td>
<td>Vend()</td>
<td>AddQtr()</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.7: Definition and uses for conventional CoinBox Class

Figure 4.7: Conventional CoinBox class

The test case 3, above is infeasible because the postcondition of AddQtr() does not satisfy the precondition of Vend() and it is only known at the run time.
<table>
<thead>
<tr>
<th>Feasible def-use pairs</th>
<th>Infeasible def-use pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(CoinBox(), AddQtr()), (AddQtr(), AddQtr()), (ReturnQtr(), AddQtr()), (Vend(), AddQtr()), (AddQtr(), Vend()), (CoinBox(), AddQtr()), (AddQtr(), AddQtr()), (ReturnQtr(), AddQtr()), (Vend(), AddQtr())</td>
<td>(CoinBox(), Vend()), (Vend(), Vend()), (ReturnQtr(), Vend()), (CoinBox(), Vend()), (Vend(), Vend()), (CoinBox(), Vend()), (Vend(), Vend()), (Vend(), Vend())</td>
</tr>
</tbody>
</table>

Table 4.8: Feasible and infeasible def-use pairs of CoinBox class

4. CoinBox(), AddQtr(), AddQtr(), ReturnQtr(), Vend()—Error. This error can be found in this test case because postcondition of ReturnQtr() does not come out to be true. Hence, this indicates an implementation error in the method “ReturnQtr()”. Now, to find the error in the method ReturnQtr() we compare the postcondition of ReturnQtr() with its implementation and find that according to postcondition, the value of the attribute ‘AllowVend’ should be 0 but it is missing from the implementation. Hence, the error is found and the correction can be made in the implementation of the class.

4.5 Related Work and Discussions

There are a number of other data flow based testing techniques for testing a class. Parrish [1] has developed a class testing technique in which test cases are generated from the implementation of a class. This is helpful in cases where there is no formal specification of the class. Tsai [8] has given a technique for testing object-oriented classes using data flow analysis. This technique is based on the
fact that the data members of a class, which do not participate in determination of
date of an object, can also give rise to error and must be tested.

In this chapter, we have proposed a technique for class testing using
Design by Contract and data flow testing. We use data flow testing criteria to test
the class. There are a number of other class testing techniques, which use flow
graphs and apply data flow testing criteria to generate the test cases. Our
technique is different from the above techniques in the following ways:
1. This technique uses flow graph, which is constructed from the contract based
   specification.
2. Implementation level errors can be found.
3. Infeasible message sequences are eliminated.
The technique proposed is a unit testing technique as it tests only a single class.
This technique has been validated on the classes like stack, account, tree, and heap
etc.