CHAPTER 5

State based Class Testing using Software Contracts

5.1 Introduction

State based testing is one of the most important testing techniques of object oriented software. It complements the traditional approaches of testing: black box and white box testing. It focuses on testing the state of an object which means whether a message or a sequence of messages puts an object of the class into a valid state or not. State based testing focuses on testing the state of an object throughout its life cycle. State based unit testing focuses on behavior of a single object and state based integration testing focuses on the interaction among the objects. This chapter explains the proposed state based class testing (unit level) technique using the software contracts. The state based testing is used to test the implementation of the class to find the state based errors, which arise due to the execution of a particular sequence of methods of a class. Since this technique uses software contracts, which have been used to improve the testability of the software. Hence, this technique improves the testability of the software by reducing the testing effort.

A number of techniques have been developed for object oriented state testing which are discussed in the literature survey in chapter 2. Chow [36] gave a state testing method using automata, which verifies the control structures at the design level. Kung [42] gave an object state test model for testing objects, which is based on the design specification of a class. Turner [33] developed a new technique to test the interaction between the methods and the state of an object.
using finite state automata. Tse [131] generates test cases through formal object
test model which is based on the predicate/transition nets. Robert Binder [22] has
developed a testable FREE (Flattened Regular Expression) state model of the class
behaviour. The Free state model focuses on the object state at method activation
and deactivation. Tsai [135] has developed an automated approach, which is
derived from the state testing method. Our technique makes use of software
contracts for state based testing of a class.

5.2 State Based Testing Technique

The steps of our proposed technique are as follows:

1. Given a class specification in terms of software contracts, identify the
   states from the preconditions, postconditions, and class invariants.
2. Identify the transitions among all the states obtained in step 2 and draw the
   state test model based on the states and transitions obtained.
3. Identify the test cases (test cases are the method sequences) based upon the
   state testing criteria.

All the above steps are elaborated throughout this chapter by taking a much
known example of CoinBox class of vending machine from [42]. This
problem is as explained in chapter 4. The C++ code for this class (as given by
Kung) with an implementation error is as shown in figure 5.1. The error in
this class is that the method ‘ReturnQtr()’ has a missing statement
‘AllowVend=0’. Due to this error the message sequence ‘CoinBox(),
AddQtr(), AddQtr(), ReturnQtr(), Vend()’ becomes valid, which is otherwise
invalid. Because, through this sequence, even after returning the coins the
person gets the drink. Our testing technique eliminates this error.
CoinBox{
CoinBox()
{
AllowVend=0;
CurQtrs=0;
TotalQtrs=0;
}
AddQtr()
{
CurQtrs = CurQtrs + 1;
If(CurQtrs >1)
then AllowVend = 1;
else AllowVend = 0;
}

Figure 5.1: C++ Implementation of CoinBox Class

Step1: Identification of the States

The state of an object is defined as a particular member of a class’s combinational value set or the current contents of an object [22]. The preconditions and the postconditions for a given method determine what should be true before the method execution and what should be true after the method execution respectively. Hence, we can say that the precondition of a method specifies the state of the object allowable for the execution of a method and postcondition specifies the state allowable after the execution of a method. But that makes the number of states = 2 * number of methods in the
class. Hence, we need to minimize that because certain preconditions and postconditions can be common and can be covered among others. Hence, we find the states of our state model using the following steps.

1. Take the contract based specification of the class i.e. the preconditions and postconditions of each method, and the class invariant. The contract based specification of the CoinBox class is as shown in figure 5.2.

2. Write all the possible values of all the state determining data members of the class from the preconditions and postconditions of methods of the class. The state determining data members are the ones whose change in value causes change in the state of the object [42]. In this example we have two state determining data members CurQtr and AllowVend. For this example we can write all the possible values of these data members from the CoinBox class shown in Figure 5.2. The first value belongs to CurQtr and the second to AllowVend attribute of the CoinBox class.

   Preconditions are : [(0, 1), 0], [1,0][2,1],[2,1]

   Postconditions are :[0,0],[1,0],[2,1], [(0,1),0],[0,0]

3. Now, we remove the duplicates if any. In the above example, after removing duplicates we get:

   [(0,1),0],[1,0][2,1], [0,0]

4. Next we try to further minimize the states by merging the values. For example, we have [0,0] and [1,0] as the states so we can remove [(0,1),0] as it is covered among the states[0,0] and [1,0]. We can not remove [0,0] and [1,0] to keep [(0, 1], 0] as our state because we have a method in our class whose postcondition is [0, 0]
CoinBox
{
    int CurQtrs, AllowVend, TotalQtrs;
    CoinBox ()
    {
        Precoinbox(True);              //Precondition of Constructor
        PostCoinbox(AllowVend==0 & & CurQtrs==0);    // Postcondition of constructor
    }
    AddQtr()
    {
        PreAdd (0<=CurQtrs<=1, AllowVend ==0);     //Precondition of the method AddQtr
        PostAdd(CurQtrs=1 & & AllowVend==0!!CurQtrs=2 & & AllowVend==1);
                                                               //Postcondition of the method AddQtr
    }
    ReturnQtr()
    {
        PreReturn ( CurQtrs ==1 & & AllowVend== 0 !!CurQtrs ==2 & & AllowVend ==1);
        //Precondition of the method ReturnQtr
        PostReturn(0<=CurQtrs<=1, AllowVend ==0);
        Invariant(0<=CurQtrs<=2, 0<=AllowVend <=1)
    }
}

Figure 5.2: Contract based Specification of CoinBox Class

5. Check the minimized set of values that we get in step 3 against the class invariant. If it does not violate the class invariant then these are the states of our
state test model. For the above example the states \([0, 0], [1, 0]\) and \([2, 1]\) are state1, state2, and state3 respectively.

**Step 2: Identification of Transitions**

A transition occurs from one state to another upon the execution of a method of the class. A transition is composed of: (1) method of the class, and (2) optional guard conditions. The transitions are identified as follows:

1. The first transition is labeled as the constructor method of the class to the state where its postcondition is true. There is only one transition labeled constructor. In case of the above example, there is a transition to state1 \((\text{CurQtrs}=0, \text{AllowVend}=0)\) by execution of the constructor method “CoinBox()”. This transition is called the initial transition.

2. Now, from state1 the only possible transitions are the ones which methods have state1 as their precondition. This transition is between state1 and the state which satisfies the postcondition of the concerned method. For example, the only possible transition from state1 in the above example above is AddQtr(). Since the preconditions of Vend() and ReturnQtr() are not satisfied by state1. There can be one or more transitions out of a given state depending upon the value of the data members. If in a state a method is invoked which can lead it to two different states then this transition has a guard condition. Repeat the above process for all the states to find all the possible transitions between all the states. The resulting state test model is shown in figure 5.3.
Step 3. Generation of Test Cases:

The test cases are generated using the state test model obtained in the last section.

The state testing criterion used is:

![State Test Model for CoinBox Class](image)

**Figure 5.3: State test model for CoinBox class**

- All states should be traversed at least once.
- All transitions should be traversed at least once.
- The generation of test cases must adhere to the following rules:

**Rules:**

- Every test case must start with the initial state.
- A test case is sequence of methods.
- Constructor is the first method of any test case.

Some of the possible test cases for the above example are:

Test case 1: CoinBox(), AddQtr(), AddQtr(), Vend()

Test case 2: CoinBox(), AddQtr(), ReturnQtr()

Test case 3: CoinBox(), AddQtr(), ReturnQtr(), AddQtr(), AddQtr(), Vend()

Test case 4: CoinBox(), AddQtr(), ReturnQtr(), AddQtr(), AddQtr(), AddQtr(), ReturnQtr(), ReturnQtr()
Through this testing technique we have eliminated the sequence CoinBox(), AddQtr(), AddQtr(), ReturnQtr(), Vend(), which is an invalid method sequence. Also, the implementation error in the method “ReturnQtr()” can be diagnosed through this technique. When we run this test case then after executing the method “ReturnQtr()” the object state should be state1 but it does not come out to be the same as the implementation has a flaw which is the statement “AllowVend= 0” is missing.

5.3 Case Study

Here we illustrate our technique using an example of class stack. The C++ code for stack class is given in figure 5.4 and the contract based specification of the stack class is shown in figure 5.5.

Step1: Identification of states

The state of the stack object is determined by the value of the data member Top of the stack class. The states are as calculated below:

Preconditions ; [0, Stack_Size-1], [1, Stack_Size]

PostConditions [0], [1, Stack_Size], [0, Stack_Size -1]

After removing the duplicates we get: [0], [0, Stack_Size -1], [1, Stack_Size],

Merging the states: we cannot merge [0] as it is one of the postconditions (postcondition of constructor of the class) hence state1 is [0]

[0, Stack_Size -1] covers the value [0] hence we get a state [1, Stack_Size -1] out of [0, Stack_Size -1]. Now [1, Stack_Size -1] can be merged with [1, Stack_Size] to get two states : [1, Stack_Size -1] and [Stack_Size]

Hence, we get three states : [0], [1, Stack_Size -1], [ Stack_Size]

Hence, in this case the states are:
Top = 0, (2) 0< Top< Stack_Size (3) Top = Stack_Size

Step2: Identification of transitions

In case of the stack example the only transition possible from state1 is method “push” and it takes the stack object from state1 to state2 as it satisfies the postcondition of the method “push()”. We cannot have a transition named “pop” from state1 as its precondition is not satisfied by state1. There are guard conditions in this example. Guard condition always comes into picture whenever the object can go to two different states upon the execution of a method. The stack object in state2 after invoking push operation goes to state3 or to state2 depending upon the guard condition. To determine what would be the guard condition, the following rules are applied.

Rule 1: If there are two transitions in the state model from state A to state B and value of data member increases in going from state A to state A or to State B upon the execution of a method. Then the guard conditions are:

For the transition A to B is: data member = value of data member in state B – change in data member upon the execution of the method.

For the transition A to A is : data member< value of data member in state B – change in data member upon the execution of the method.

For example there are two transitions on method push: from state2 to state3 and state2 to state2. The value of ‘Top’ increases while going from state2 to state3 hence the guard condition is [Top = Stack_Size -1] for the transition push from state2 to state3 and the guard condition is [Top<Stack_Size -1] for the transition push from state2 to state2.
class Stack
{
    int *s;
    int Stack_Size;
    int Top;
    public:
    Stack( int Size)
    {
        Top=0;
        s = new int[Size];
        Stack_Size = Size;
    }
    Push(int element)
    {
        s[Top] = element;
        Top = Top + 1;
    }
    int Pop()
    {
        element = s[Top];
        Top--;
        return element;
    }

Figure 5.4: C++ Code for Stack Class
class Stack
{
    int *s;
    int Stack_Size;
    int Top;

    Stack () // Constructor of the class
    {
    PreStack (true); // Precondition of constructor
    PostStack(Top ==0) // Postcondition of the constructor
    }

    void Push (int item)
    {
    Pre Push (Top<Stack_Size ); // Precondition of Push method
    PostPush(Top <= Stack_Size and Top >0); // Postcondition of the Push method
    }

    void Pop()
    {
    PrePop(Top>0); // Precondition of pop method
    PostPop(Top< Stack_Size); // Postcondition of the pop method
    }

    invariant(Top >=0 & & Top <= Stack_Size); // Invariant of the Stack class
}

Figure 5.5: Contract based specification for Stack Class
Rule 2: If there are two transitions in the state test model from state A to state B and the value of data member decreases in going from state A to state A or to State B upon the execution of a method. Then the guard conditions are:

- For the transition A to B: data member = value of data member in state B + change in data member upon execution of operation.
- For the transition A to A: data member > value of data member in state B + change in data member upon execution of operation.

For example, there are two transitions on method pop: from state2 to state2 and state2 to state1. The value of 'Top' decreases while going from state2 to state1 hence the guard condition is [Top = 1] for the transition pop from state2 to state1 and the guard condition is [Top>1] for the transition pop from state2 to state2.

Step 3: Generation of test cases: The test cases are generated from the state test model. The state test model of stack is shown in figure 5.6. Some of the test cases are:

Stack(),Push(),Pop()
Stack(), Push(), Push(), Pop(), Pop()
Stack(), Push(), Push(), Pop()

In this example also a sequence like "Stack(), Push(), Pop(), Pop()" is invalid because after the execution of first pop the resulting state is state1 and there is no transition from state1 labeling pop. Hence, we come to know that this is an infeasible sequence.

5.4 Conclusions

Testing the state of an object is one of the most important issues in object oriented testing. In this Chapter, we have proposed a state test model whose states and transitions are derived from the preconditions, postconditions, and invariants. Through the use of contracts the state based errors are known earlier. We have shown that how implementation errors can be removed through this technique. This technique has been validated using a number of examples.