Applying Conventional Testing Techniques for Class Testing *

In -S Chung
Department of Computer Engineering
Hallym University
1 Okchun-Dong Chunchun 200-702 Korea

Malcolm Munro
Centre for Software Maintenance
University of Durham
Durham England DH1 3LE

W. K. Lee and Y. R. Kwon
Department of Computer Science
Korea Advanced Institute of Science and Technology
373-1 Kusung-Dong Yusung-Gu Taejon 305-701 Korea

Abstract

This paper discusses how conventional testing criteria such as branch coverage can be applied for the testing of member functions inside a class. To support such testing techniques we employ symbolic execution techniques and finite state machines(FSMs). Symbolic execution is performed on the code of a member function to identify states that are required to fulfill a given criterion. We use FSMs to generate a sequence of member functions leading to the identified states. Our technique is a mixture of code-based and specification-based testing techniques in the sense that it uses information derived from codes using symbolic execution together with information from specifications using FSMs for testing activities.

Keywords: software testing, object-oriented programs, classes, finite state machines, symbolic execution

1 Introduction

Recently there has been much research on testing techniques for a class which is often considered as the basic unit in an object-oriented program[3, 5, 8, 11, 12]. Most exsiting work in conducting class testing has focused on specification-based testing techniques which involve selecting sequences of member functions to test for defects in their interactions. One reason why test cases are made of sequences of member functions is that execution paths of each member function are determined by states of the object to be considered as well as its input parameter values. Usually, the values of data members that define the current state of an object depend on some other member functions within the class.

Suppose that a class STACK has public member functions PUSH(), POP() and a data member Top which is assumed to be accessible only by PUSH() and POP(). Then, for example, determination of which paths in the POP method are executed depends on the values of data member Top which are affected by member functions PUSH() and POP() itself. From this fact we can find that it is difficult to isolate a member function as the basic testing unit from its class.

In contrast, conventional (unit) testing techniques deal with the procedure, or the function, as its basic testing unit and involve selecting a collection of input data values as test cases. This is one important reason why it is difficult to apply conventional testing techniques to object-oriented programs directly. Moreover, since most existing techniques on class testing select test cases on the basis of specifications, they do not require coverage of particular code components. In order to obtain confidence, the notion of coverage which is analogous to that used in conventional code-based unit testing should be introduced into class testing as well.

In this paper, we present a testing method that combines specification-based testing and code-based testing for class testing. The testing method adapts conventional code-based unit testing techniques to test member functions of a given class. In order to apply the notion of coverage provided by conventional testing we consider each member function inside a class as the basic testing unit. Because the testing of member functions requires states of the object as discussed

*This work was supported in part by the Korea Science & Engineering Foundation under contract number 951-0908-022-2.

above, it is necessary to identify the states and then build up the identified states.

We use symbolic execution techniques to identify states which are required by each member function for satisfing a given coverage criterion. By performing symbolic execution on the paths of the member function that are not yet exercised, we can obtain expressions in terms of data members. The resultant expressions indicate the states to be required for the testing of member functions. The testing method proposed in this paper is also based on finite state machines(FSMs) in order to generate a sequence of member functions that leads to the identified states.

The remainder of this paper is organized as follows: Section 2 discusses how to employ conventional testing techniques for class testing. Section 3 describes a state model needed for generating test sequences. Section 4 presents our testing technique with an example. Section 5 describes related works to class testing and compares our work with them. Conclusion and future works are given in Section 6.

2 Adequate Testing of Member Functions in a Class

When building a test set for testing a member function inside a class it is not sufficient to consider only its input parameter values. We should consider one additional factor, states which the objects of the class may be in. A state of an object is defined in terms of data members of the object. For example, consider a stack object with its data member Top of type unsigned integer. Then, the stack object may be regarded as having two states, an initial state *Empty* characterized by Top=0 and a state *NotEmpty* characterized by Top>0. It is evident that the dynamics of a member function such as POP() depends on the current state of the object. Therefore, we must take into account the state of the object as well as the values of the input parameters of the member function. Consequently, we can represent a test case of a member function m as a pair $t_m = (v, s)$, where v is an input value vector corresponding to the input parameters of m and s is a state of the object which m is applicable to.

Definition 2.1 Let $T_m = \{t_m^k | k = 1, 2, ..., n\}$ be a test set of a member function m, where $t_m^k = (v^k, s^k)$. Let A be a conventional test criterion. We say that T_m is A-adequate for m and write $A(m, T_m)$ if T_m contains test cases to satisfy the criterion A.

This form of a test case of a member function requires some means to identify the states as well as input value vectors in order to perform the adequate testing of the member function, thereby compounding the difficulty of test case selection. Furthermore, even though we can identify the desired states, testing a member function requires building up the states in such a way that the member function can be tested isolated from its surrounding class.

One way to resolve these problems is to remove all other code fragments except the member function under testing and then write test drivers and test stubs. The role of these driver and stubs is to adjust the values of data members to make the desired state and replaces the code removed. However, this approach has some difficulties in being applicable in practice. Especially, it is difficult to write test drivers and stubs when the majority of code in the member function is already provided in the form of other member functions in the class. Furthermore, the test drivers must supply to the member function the states which are required to fulfill a given test criterion. In general, test drivers with such facilities are likey to require as much testing effort as when testing all the member functions of the class.

Another approach is to use the member functions in the class to make the intended states without removing them. Suppose, for example, that we want to test member function POP() in the (bounded) STACK class of size n according to some test criterion. If POP() requires the bounded stack object being full to satisfy the criterion, then we can construct the required state by using a sequence of member functions, e.g., $< \overbrace{\text{PUSH}, \cdots, \text{PUSH}}^{n} >$. After selecting a possible sequence of member functions, we can apply POP() to the state resulting from execution of the selected sequence. Of course, there is no gurantee that such a sequence reaches the intended state because the member functions associated with the sequence may have defects. However, we know that testing is the process of finding defects rather than certifying correctness, in this sense, this approach is of value.

3 The State Model

We employ a state model to generate a sequence of member functions leading to the required state. State models such as finite state machines(FSMs) can be used in representing a set of states and the transitions between those states to capture the dynamic behavior of class instances, i.e., objects[1, 9, 10].

There are many candidates for representing the state model, including the model proposed by Rumbaugh et al[9] and that of Shaler and Mellor[10]. Such representations provide enough highly expressive power to use them as analysis and design tools. However, the state model needed for testing activities does not necessarily have to encompass all the features provided by the existing models. In this paper, we will use a very simple state model based on finite state machines.

Constructing the state model involves two steps: 1) building a basic state machine(BSM) for each data member which has the significant effect on the behavior of an object and 2) building a composite state machine(CSM) from the constructed BSMs. In the following subsections, we describe these steps in detail.

3.1 Constructing Basic State Machines

When constructing BSMs it is not likely to be useful to consider all data members; some data members may exist only for housekeeping and, therefore, they do not participate in the definition of the object's state because state changes of an object are independent of them. This leads to the reduction of the number of data members that need to be considered. A second consideration to build the (sub)state space for a data member is that the state space must be partitioned to equivalent classes each of which shares the properties of interest. For example, in an array-based implementation of a stack of size n, there may be from 0 to n elements on the stack as indicated by data member Top and, therefore, the state space for Top can be partitioned to n+1 states. From a modeling or testing perspective, we are usually only interested in the states characterized by Top=0, 0 <Top< n, and Top = n. Rather than considering n states we need only to consider three. The formal definition of BSM is as follows.

Definition 3.1 A basic state machine(BSM) of a class C is a tuple (d, F, S, T) where

- d is an element in the set of data members, D, of C i.e., $d \in D$.
- F is a finite subset of member functions of C that can modify d.
- S is a finite set of states, i.e., S = $\{s \mid s = (def)\}$ where def is a predicate on data member d.
- T is a finite set of transitions, i.e., T = $\{(s_i, t, s_j) \mid s_i \xrightarrow{t} s_j,\ s_i, s_j \in S \text{ and } t \in F\}$

Figure 1 shows a class specification for an unbounded integer queue, QUEUE, using C++ notations. Class QUEUE has member functions QUEUE(), ~QUEUE(), ADDQ(), DELQ(), FRONT(), ISEMPTY() and data members *front*, *rear* of type NodePtr. QUEUE(), called a constructor, creates a new queue, ~QUEUE(), called a destructor, destroys a queue object. We will not describe the functionalities of the other member functions and the role of data members because the QUEUE specification is well-known.

```
class QUEUE {
public:
      QUEUE();
      ~QUEUE();
      Boolean ISEMPTY() const;
      int FRONT() const;
      void ADDQ(int newItem);
      void DELQ();
private:
      NodePtr* front;
      NodePtr* rear;
}
```

Figure 1: A class specification for unbounded queue, QUEUE

In order to construct BSMs for class QUEUE, we firstly must consider which data members of QUEUE influence the definition of queue objects. Because, in this example, both data members *front*, and *rear* affects the class behavior, it is required to construct one BSM for each of the data members.

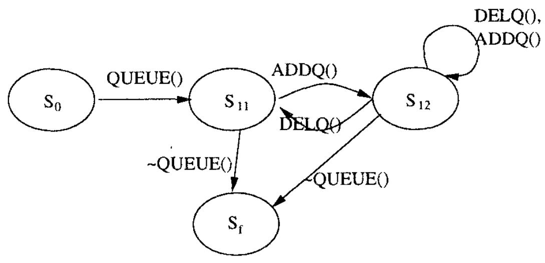

Figure 2: A BSM for data member *front*

Figure 2 shows a BSM for data member *front* of QUEUE. As can be seen in Figure 2, the domain of data member *front* is partitioned into two equivalent classes s_{11}, s_{12}. The states are associated with the values of *front* as follows:

$s_{11}.def$: $front = NULL$
$s_{12}.def$: $front \neq NULL$

Also, there are two specially designated states in BSM, s_0 and s_f:

- s_0 is an initial state representing the period before an object is created, i.e., a constructor of an object is called. $s_0.def$ is undefined because the data member being considered have not yet been created.

- s_f is a final state representing the period after an object is destroyed, i.e., a destructor of an object is called. $s_f.def$ is undefined because the data member being considered are killed in s_f.

The transitions between the states of a data member v consists of pre-state, post-state, and a member function. A data member changes its state if its value is changed by execution of a member function. Thus, a state transition from a pre-state to a post-state can occur if the pre-condition of the member function can be satisfied by the pre-state. The final value expression of the data member can be satisfied by the post-state. For example, the transition, tagged with member function ADDQ(), from s_{11} to s_{12} can occur if the precodition of ADDQ() can satisfied by state s_{11}, i.e., $front = NULL$. When the transition occurs, the state is changed into s_{12} that represents the state, $front \neq NULL$. Similarily, we can construct a BSM for data member *rear*. Figure 3 shows a BSM that partitions the domain of data member into two states, s_{21} and s_{22}; s_{21} represents the state $rear{=}NULL$ and s_{22} represents the state $rear \neq NULL$.

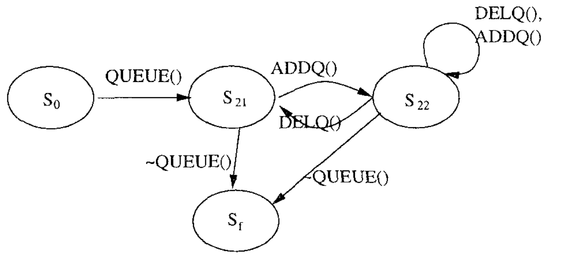

Figure 3: A BSM for data member *rear*

3.2 Constructing Composite State Machines

After constructing BSMs for each data member of a class, we are able to derive a composite state machine(CSM) for the class. In this section, we describe how to construct a CSM from BSMs. We only consider the case in which a CSM is constructed from two given BSMs because it is straightforward to construct a CSM for more general cases. Suppose that a class contains data members, d_1, d_2. Then, let us denote BSMs for the data members by BSM_1 and BSM_2, where $BSM_i (i = 1, 2)$ is a tuple (d_i, F_i, S_i, T_i). Then, the definition of CSM is as follows:

Definition 3.2 A CSM of a class C is a tuple (D, F, S, T) such that:

- D = $\{d_1\} \cup \{d_2\}$.
- F = $F_1 \cup F_2$.
- S $\subset S_1 \times S_2$ and for $s \in S$, $s.def$ is the conjunction of the predicates of its substates, i.e., if $s = (p, q)$, $s.def$ is defined by $p.def \wedge q.def$.
- For s_p = (p, q), s_q = $(p', q') \in S$, T is a subset of:

$$\left\{ (s_p, t, s_q) \;\middle|\; \begin{array}{l} if\ q = q', \wedge p \xrightarrow{t} p' \in T_1, \\ or\ p = p', \wedge q \xrightarrow{t} q' \in T_2, \\ or\ p \xrightarrow{t} p' \in T_1 \wedge q \xrightarrow{t} q' \in T_2 \end{array} \right\}$$

Note that the set of states in CSM is defined as a finite *subset* of the cartesian product of states in BSM_i. This means that there may exist some invalid composite states in S. Invaild states are produced because states in each BSM are combined without any regard to their meaning and how they might interact. Especially, only a few of composite states associated with initial or final states of BSM_i are valid. This is due to the following implicit assumptions: let us denote the initial state and the final state of BSM_i by s_0^i and s_f^i, respectively.

- We assume that when an object is created, the storage spaces for all data members constituting the object are allocated. Under this assumption, $(s_0^1, .., s_0^n)$ becomes the initial state for CSM. The other states such as $(s_{1j}^1, .., s_{kj}^k, .., s_{nj}^n)$ and s_{kj}^k is equal to s_0^k are considered as invaild.
- We assume that when an object is destroyed, all data members constituting the object are killed. Under this assumption, $(s_f^1, .., s_f^n)$ becomes the final state for CSM. The other states such as $(s_{1j}^1, .., s_{kj}^k, .., s_{nj}^n)$ and s_{kj}^k is equal to s_f^k are considered as invaild.

Let's consider a CSM for class QUEUE. We have already constructed the BSMs for data members *front* and *rear*. Because the BSMs, shown in Figure 2 and Figure 3 contain four states respectively, there may exist 16 composite states in the CSM. However, from the above assumptions, we only consider six states, i.e., (s_0, s_0), (s_f, s_f), (s_{11}, s_{21}), (s_{11}, s_{22}), (s_{12}, s_{21}), (s_{12}, s_{22}), as valid. By eliminating the other ten states and their associated transitions, we can obtain the CSM for QUEUE as depicted in Figure 4.

However, the CSM in Figure 4 is not the final version because it still contains so-called *contradictory* states. A composite state is contradictory if the predicates associated with some of its substates are inconsistent with the specification of a given class. The contradictory states in Figure 4 are (s_{12}, s_{21}) and (s_{11}, s_{22}) that are enclosed in the dotted region. The predicates of these states are as follows:

- $(s_{12}, s_{21}).def = (front \neq NULL \wedge rear = NULL)$

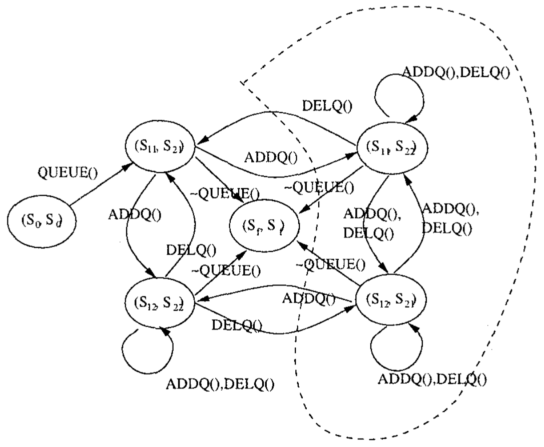

Figure 4: A CSM for QUEUE before eliminating contradictory states

- $(s_{11}, s_{22}).def = (front = NULL \wedge rear \neq NULL)$

The predicate of (s_{12}, s_{21}) implies that at least one item exists in the queue, but the rear of queue does not have an item. Obviously, this is contradictory to the QUEUE class specification because *rear* is required to point to the rear item (i.e., $front = rear \wedge rear \neq NULL$), if it exists, in the queue. Likewise, we can regard (s_{11}, s_{22}) as a contradictory state. Figure 5 shows the CSM for the QUEUE class after eliminating the contradictory states and their associated transitions.

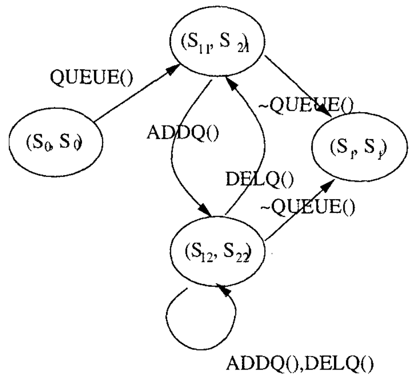

Figure 5: A CSM for class QUEUE

4 Testing Process

The testing technique proposed in this paper consists of the following steps:

1. Establish a test criterion and instrument each member function inside a class module under test in accordance with the criterion.
2. Perform random testing on the class module.
3. For each member function, investigate the unexercised blocks for test cases generated in Step 2 and determine the paths to be required for satisfying the given criterion.
4. Perform symbolic execution on the paths produced in the previous step and produce the path constraints in terms of data members.
5. Generate test cases from CSM with respect to the path constraints.
6. Execute the generated test cases on each member function that has the paths produced in Step 3.

The following subsections describe theses steps in detail.

4.1 Symbolic execution on member functions

This subsection is mainly devoted to illustratining Steps 3 and 4. Before proceeding, we briefly mention Steps 1 and 2.

A class module under test is instrumented in Step 1 for the purpose of collecting coverage information for the generated test cases and evaluating if more testing effort is needed for the chosen criterion. In Step 2, we generate sequences of member functions arbitrarily and use them as test cases for the class. In fact, we can employ the existing techniques for this purpose. One of the techniques that we can consider in this step is the technique proposed by Jalote et al[6]. Their technique uses the syntactic information of each member function to generate sequences of member functions. The reason why we firstly perform random testing on a class is that it can reduce the computational effort required to develop new test cases for satisfying the testing criterion.

The coverage statistics after random testing is used to determine whether or not additional test cases are necessary. If the intendend coverage level is not achieved, we produce path constraints for the paths required for satisfying the criterion through symbolic execution. This task is carried out in Steps 3 and 4. For example, consider member function ADDQ() in the QUEUE class. Figure 6 shows the code fragment of ADDQ() that contains numbered statements.

```
    void QUEUE::ADDQ(int newItem) {
1:      NodePtr newPtr = new NodeType;
2:      newPtr→data = newItem;
3:      newPtr→link = NULL;
4:      if (front == NULL)
5:          front=rear=newPtr;
        else {
6:          rear→link = newPtr;
7:          rear = newPtr;
        }
    }
```

Figure 6: The member function ADDQ() in the QUEUE class

There are two execution paths in ADDQ(); one is (1,2,3,4,5), denoted by p_1, and the other is (1,2,3,4,6,7), denoted by p_2. Suppose that testing of member function ADDQ() requires branch coverage. We should develop test cases to exercise these two paths if the paths have not been traversed by test cases generated during random testing. In order to develop test cases for the paths, we use symbolic execution techniques to find the path contraints. The following table shows the resulting path constraints for p_1 and p_2:

path constraint	data member		parameter
	front	rear	newItem
$C[p_1]$	NULL	don't care	don't care.
$C[p_2]$	not NULL	don't care	don't care.

We use these path constraints to check if there exist states in CSM to satisfy them. In this example, we can find that the following implications between the path constraints and the states in the CSM given in Figure 5 holds:

$$(s_{11}, s_{21}).def \Rightarrow C[p_1], (s_{12}, s_{22}).def \Rightarrow C[p_2].$$

From this, we know that (s_{11}, s_{21}) and (s_{12}, s_{22}) are the states required to achieve branch coverage of the ADDQ() member function.

We must also consider the case where there exists a path constraint, $C[p_k]$, such that $C[p_k]$ cannot be satisfied by any states in a given CSM. In this case, it is necessary to transform the CSM into the one that contains the states to satisfy the path constraint. If we assume that the path constraint is legal(i.e, there exist sequences of member functions leading to the states the path constraint requires), it is possible to derive a CSM with such states from a given CSM as described below:

(1) Select a state S_i that intersects with $C[p_k]$.

(2) Form states S_p, S_q in such a way that $S_p.def = S_i.def - (S_i.def \cap C[p_k])$ and $S_q.def = S_i.def \cap C[p_k]$.

(3) Replace S_i by S_p and S_q.

(4) Repeat the above steps until no intersecting states exist in the CSM.

Following the above procedure, the resulting CSM contains at least one state S_r holding

$$S_r \Rightarrow C[p_k].$$

We can show the existence of S_r by proving that there exists a state intersecting with $C[p_k]$ in the original CSM. It is straightforward to prove the fact. Because the states of the CSM are mutually exclusive and $C[p_k]$ is assumed to be legal, there exists a sequence of member funtions the execution of which results in a state S such that $S \cap C[p_k] \neq \emptyset$. That is, $S \cap C[p_k] = S_r$ and $S_r \Rightarrow C[p_k]$.

4.2 Test generation from CSMs

Once the required states are identified, we must generate test cases the execution of which can form the states. For branch coverage of ADDQ, we can form the state (s_{11}, s_{12}) by executing the sequence (QUEUE,ADDQ,DELQ,DELQ). Also, we can observe that executing (QUEUE,ADDQ,DELQ,ADDQ) results in the state (s_{12}, s_{22}). So, the testing of ADDQ() may involve executing the following sequences:

- (QUEUE,ADDQ,DELQ,DELQ,ADDQ)
- (QUEUE,ADDQ,DELQ,ADDQ,ADDQ)

In fact, there may exist infinite number of sequences that play a role as preambles to a member function in cases where CSMs have loops. Because we cannot test a member function for all possible sequences in practice, it is necessary to introduce some sequence selection rules that specify which of all possible sequences can be preambles to the member function under test. To this ends, we firstly generate a test tree from a CSM by following the steps given below:

(1) Starting from the initial state in CSM, the root of the test tree is constructed.

(2) We now examine the nodes in the test tree one by one. Let the node being examined be labeled by S_i.

(3) If S_i has already occurred at a higher level in the tree, then the node becomes a leaf node and will not be examined. Go to Step (2) and examine the next node; otherwise go to Step (4).

(4) If there exists a transition t such that $S_i \xrightarrow{t} S_j$ in the CSM, then we attach a branch and a successor node to S_i, the branch is labeled t and the successor node is labeled S_j.

(5) This step is repeated until no transition can be triggered. Then go to Step (3) unless no expansion is possible.

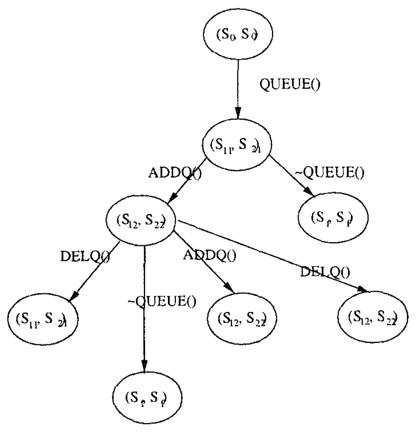

Figure 7: A test tree for class QUEUE

Conider the CSM in Figure 5. A test tree for this CSM is shown in Figure 7. From the test tree, we can select sequences of member functions which serve as preambles to a member function satisfying a given testing criterion according to the rules in Figure 8.

Let $S = \{P_i \mid i = 1,..,r\}$ be a set of all paths from the initial state to the required states in the test tree. Each P_i is represented by $(S_0, M_0, S_1, M_1, \cdots, M_{k-1}, S_k)$ where S_0 is the initial state, S_k is one of the required states, and M_j is a member function that is involved with a transition from S_{j-1} to S_j. Then, for each P_i, generate a sequence of member functions T_i such that

(R1) each $M_j (j = 0,..,k-1)$ must participate in T_i at least once and

(R2) execution of T_i must reveal the states $S_0,..,S_k$ at least once and the order of state occurrences must be preserved.

Figure 8: Rules for selecting preambles

In the case of ADDQ, there are five paths from the initial state to the required states (s_{11}, s_{21}), (s_{12}, s_{22}):

- P_1 :$((s_0, s_0)$,QUEUE,$(s_{11}, s_{21}))$
- P_2 :$((s_0, s_0)$,QUEUE,(s_{11}, s_{21}),ADDQ,(s_{12}, s_{22}),DELQ, $(s_{11}, s_{21}))$
- P_3 :$((s_0, s_0)$,QUEUE,(s_{11}, s_{21}),ADDQ,$(s_{12}, s_{22}))$
- P_4 :$((s_0, s_0)$,QUEUE,(s_{11}, s_{21}),ADDQ,(s_{12}, s_{22}),ADDQ, $(s_{12}, s_{22}))$
- P_5 :$((s_0, s_0)$,QUEUE,(s_{11}, s_{21}),ADDQ,(s_{12}, s_{22}),DELQ, $(s_{12}, s_{22}))$

Using the rules (R1) and (R2) as guidelines, the following set of sequences can be derived for ADDQ:

- (QUEUE) for p_1
- (QUEUE,ADDQ,DELQ) for p_2
- (QUEUE,ADDQ) for p_3
- (QUEUE,ADDQ,ADDQ) for p_4
- (QUEUE,ADDQ,DELQ,ADDQ) for p_5

For example, consider the path p_5. (R1) requires that a sequence derived for the path must contain at least one occurrence for each of the member functions QUEUE, ADDQ, and DELQ. (R2) also requires that during execution of the derived sequence, the states $((s_0, s_0)$, (s_{11}, s_{21}), (s_{12}, s_{22}), $(s_{12}, s_{22}))$ must be revealed in the order appearing in the path. We can easily observe that the sequence (QUEUE,ADDQ,DELQ,ADDQ) satisfies (R1) and (R2) because:

(R1): It contains the occurences required for the memebr functions QUEUE(),ADDQ(),DELQ().

(R2): By executing QUEUE(), we can change the state (s_0, s_0) to (s_{11}, s_{21}) and by executing (QUEUE,ADDQ) we can form (s_{12}, s_{22}) and by executing (QUEUE,ADDQ,DELQ,ADDQ), we can form (s_{12}, s_{22}).

Finally, test cases are generated by using the above sequences as the preambles of ADDQ. The resulting test sequences are:

- (QUEUE,ADDQ)
- (QUEUE,ADDQ,DELQ,ADDQ)
- (QUEUE,ADDQ,ADDQ)
- (QUEUE,ADDQ,ADDQ,ADDQ)
- (QUEUE,ADDQ,DELQ,ADDQ,ADDQ)

5 Related Works

Recently, a FSM-based class testing technique has been proposed by Turner and Robson[11]. Their technique determines input states and output states for each member function from the design of the class and they also proposed test selection guidelines based on FSMs. However, this technique is based on FSMs derived from design specifications and so can not gurantee coverage of particular code components.

Kung et al.[7] proposed a class testing technique based on object state model (OSD) which is similar to statecharts[4]. They extract OSD from source code and generate test cases by constructing a spanning tree from OSD which is similar to our approach. In contrast, our technique constructs FSMs from specifications for generating sequences of member functions that serve as preambles for certain coverage of a member function being tested.

Parrish et al.[8] proposed a testing strategy using a flow graph that is constructed from source codes. A node of a flow graph is a member function and there exists an edge between nodes N and M if it is permissible to invoke N followed by M. A node is said to contain a definition of type T if the node contains a formal parameter of type T. A def-use edge is defined as a triple involving a type T, a node in which T is defined, and a node T is used. By incoporating a def-use edges into a flow graph, dataflow-like coverage criteria can be applied.

However, this technique only considers the parameters of member functions, ignoring data members. In addition, this technique does not ensure that we will test from each definition of a varaible to each use of the variable because the technique considers types rather than varaibles. In order to address these problems, Harrold and Rothermel[5] proposed three levels of class testing:intra-method testing, inter-method testing, and intra-class testing. To support each data flow in the three levels, they construct a flow graph which represents every possible sequences of member functions from the class's code. Then they generate tests using inter-procedural data flow testing techniques.

It is easy to incorporate intra-method and inter-method testing techniques into our testing framework. After random testing of a class to be tested, we perform Steps 3-6 presented in Section 4 for the paths to contain uncovered intra-method (or inter-method) def-use pairs. However, our testing technique may miss some intra-class def-use pairs because we consider a member function as a basic testing unit. That is, there may be some intra-class def-use pairs which are not limited within single member functions or within the calling context of a single member function.

6 Future Works

Our work presented in this paper does not consider inter-class relationships. In general, there are three types of relationships between classes: association, aggregation and inheritance[9]. We can incoporate these inter-class relationships into construction of CSMs. By constructing BSMs firstly we are enable to derive the CSM incrementally. For example, consider the case in which a class has as its parts other classes. In that case, we can reuse the BSMs for the component classes, if they were already constructed, in order to derive the CSM for the enclosing class. Also, we have an advantage of deriving CSMs for subclasses from a CSM for its base class in certain circumstances. Consequently, we believe that the extension of the proposed testing technique to the inter-class level would not requre much time. In addition, Further research needs the empirical validation of our testing technique.

References

[1] G. Booch, *Object Oriented Design with Applications*, Benjamin Cummings, 1991.

[2] T. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Trans. on Soft. Eng.*, Vol. SE-4, No. 3, May 1978, pp. 178-187.

[3] R. Doong and P.G Frankl, "Case Studies on Testing Object-Oriented Programs," in *Proc. of the 4th Symposium on Software Testing, Analysis and Verification*, 1991, pp. 165-177.

[4] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, 8, 1987, pp. 231-274.

[5] M.J. Harrold and G. Rothermel, "Peforming Data Flow Testing on Classes," in *Proc. of the second ACM SIGSOFT Symp. on Foundations of Software Engineering*, Dec. 1994, pp. 154-163.

[6] P. Jalote and M. G. Caballero, "Automated Test Case Generation for Data Abstraction," in *Proc. of COMPSAC*, 1988, pp. 205-210.

[7] D. Kung, N. Suchak, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "On Object State Testing," in *Proc. of COMPSAC*, 1994, pp. 222-227.

[8] A.S. Parrish, R.B. Borie, and D.W. Cordes, "Automated Flow Graph-Based Testing of Object-Oriented Software Modules," *Journal of Systems and Software*, 23, 1993, pp. 95-109.

[9] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object Oriented Modeling and Design*, Prentice Hall, 1991.

[10] S. Shlaer and S.J. Mellor, *Object Lifecycles: Modeling the World in States*, Prentice Hall, 1992.

[11] C.D. Turner and D.J. Robson, "The State-based Testing of Object-Oriented Programs," in *Proc. of Conf. on Software Maintenance*, 1993, pp. 302-310.

[12] S. Zweben, W. Heym, and J. Kimich, "Systematic Testing of Data Abstractions Based on Software Specifications," *Journal of Software Testing, Verification, and Reliability*, 1, 1992, pp. 39-55.