To appear at QW '99, May 1999, San Jose, USA

A State-Based Testing Approach Providing Data Flow Coverage in Object-Oriented Class Testing

Bor-Yuan Tsai*, Simon Stobart, Norman Parrington, and Ian Mitchell
School of Computing, Engineering and Technology
University of Sunderland,
St. Peter's Way, Sunderland SR6 0DD UK
Tel: +44 191 5153201
E-mail: {cs0byt, cs0sst, cs0npa, cs0imi}@isis.sunderland.ac.uk

*Also of Department of Information Management,
Tamsui Oxford University College,
32 Jen-Lee Street, Taipei County 251, Taiwan

Abstract

A novel Object-Oriented class testing approach, proposed in this paper, combines functional with structural testing techniques. Based on state-based testing, test cases generated from the MACT (Method for Automatic Class Testing) tool can be used to execute functional testing. The definition-use information of data members, occur in public member functions of a class under test, is generated from MACT to facilitate data flow analysis. Testers can compute definition-use path with the information in order to ensure that the class is also satisfied with data flow coverage at intra-class level. The discussion with a queue class example to reveal that using a hybrid testing technique benefits class testing.

1 Introduction

Most computer applications can be tested in one of two ways: (1) functional (black-box) testing, and (2) structural (white-box) testing. The supporters of structural testing argue that functional techniques may not provide sufficient coverage of the code. Their opponents contend that structural approaches do not consider the requirements of specifications at all, since test cases are entirely generated from the implementation [1] [2] [3].

The new class testing approach for Object-Oriented classes, discussed in this paper, is adopted by MACT [4] [5] [6] [7]. In which the test case tree, created from the state machine of a class under test, is used to produce test messages for functional testing as well as the intra-class definition-use information for structural testing. In functional testing, each class is tested as a unit based on state-based testing techniques, to verify the behaviour of an object, such as state changes. Data flow techniques are used to perform this structural testing that enables us to detect whether every data variable (data member, or parameter in public methods) has been defined prior to being used, and whether all defined data variables have been used in the class.

The functional testing approach in MACT needs a (an implementation) state machine as a specification. Following the state machine, the test case tree generator builds a threaded mutli-way tree. After tracing the tree (also called test case tree), all possible test cases are generated. The tree completely duplicates the behaviour of the state machine of the class under test and it comprises all possible expected states of all transitions of the class [3] [7] [8]. Therefore the tree is also used as a test oracle, with which the test results are inspected by the test result inspector of MACT. Eventually, the pass/error messages generated by the MACT tool show whether the class under test is satisfied with the state-based coverage.

In the MACT tool, the various associations of data members with definitions and uses across the public methods in the class are taken into account, in order to compute intra-class definition-use pairs. The definition-use information across public functions is examined whether or not any data anomalies exist in the sequence public methods called by clients. Inter-method definition-use pairs can also be computed by MACT, if the interaction of private and public functions can be described as a sub-state machine and embedded in the super state machine that the public methods are depicted in. However this is excluded from this paper. In data flow coverage, other pairs of definition-use (e.g. parameters, alias and local variables) also need to be concerned in the class under test. At the intra-class level, however, only the occurrences of data members within all possible sequences of function calls, which can be executed by the class under test, are considered.

The framework of MACT is illustrated in section three. A *queue* class example and the technique of test case generation are introduced in section four and five respectively. The concept of data flow testing and the approach of MACT, which is used as a structural test tool, are described in section six and seven. In which the *queue* example is also used to show how the definition and usage of data members are generated by MACT. The rest functions of MACT are briefly discussed in section eight. Following that is our future work and conclusions.

2. Background

The main point of *state-based testing* [9] [10] [11] is to examine the values, which have been stored in the object at a particular time. Those particular values represent the state of the object. State-based testing also validates the interactions that occur between the transitions and the state of an object. The changing states rely on the values that are changed by the transition. After executing a transition, it validates the final state that has been achieved by the object.

A *state machine* defines the set of states and transitions, and depicts the dynamic behaviour and the state changes of an object. Therefore the state machine can be used as an aid in state-based testing [12]. Programmers refer to the state machine of a class as a specification to code an implemented class, and the expected results are the requirements that should be followed. Therefore the states of the implementation object should be the same as the states that are shown in the state machine. Nevertheless, a transition name in the state machine could be different from its member function name in the implemented class. If any differences exist, then it is necessary to use an *implementation state machine* to reveal the behaviour and state changes of the implemented class.

Data flow testing techniques [13] [14] [15] are based on data flow analysis, and require that the test data (cases) exercise paths from definitions to uses. These techniques are also code-based testing techniques, and they can be used in code optimisation, anomaly detection, and test data (cases) generation [2] [15]. Extending the techniques to test object-oriented classes, the test cases can be classified into intra-method, inter-method and intra-class three categories [2] [16]. Intra-class testing examines the definition-use paths of the class's variables which across public methods when they are called in various sequences.

Object-oriented classes are usually designed as an independent small (sub) system. They respond to service requested from outside. If a whole class is tested as a unit, then it can be similarly performed as an application system test. Systems can be tested using *unit (intra-method*), integration (*inter-method*), and system (*intra-class*) testing techniques. The state-based testing technique covers the entire object rather than just individual methods and so is much more appropriate for testing objects. The *test cases* generated with state-based testing methods will be used to execute intra-class testing.

A test case can create a starting state, the expected action causing transition to the next state, and the expected next state [17]. The test cases considered in this paper are defined for class testing by sending messages to objects and estimating the results. The test message generator in

MACT only produces the expected messages using state machines. Those messages are directly sent to the object class under test.

3 The Framework of MACT

Based on the new object-oriented class testing approach, an automatic object-oriented class test tool, called MACT, is built. Which consists of five components: Test Case Tree Generator, Test Message Generator, Test Driver Generator, Test Result Inspector and Def-Use Info Generator. The first four are used to achieve state-based testing, and the last is a mechanism for data flow testing. The framework of MACT is shown in Figure 1.

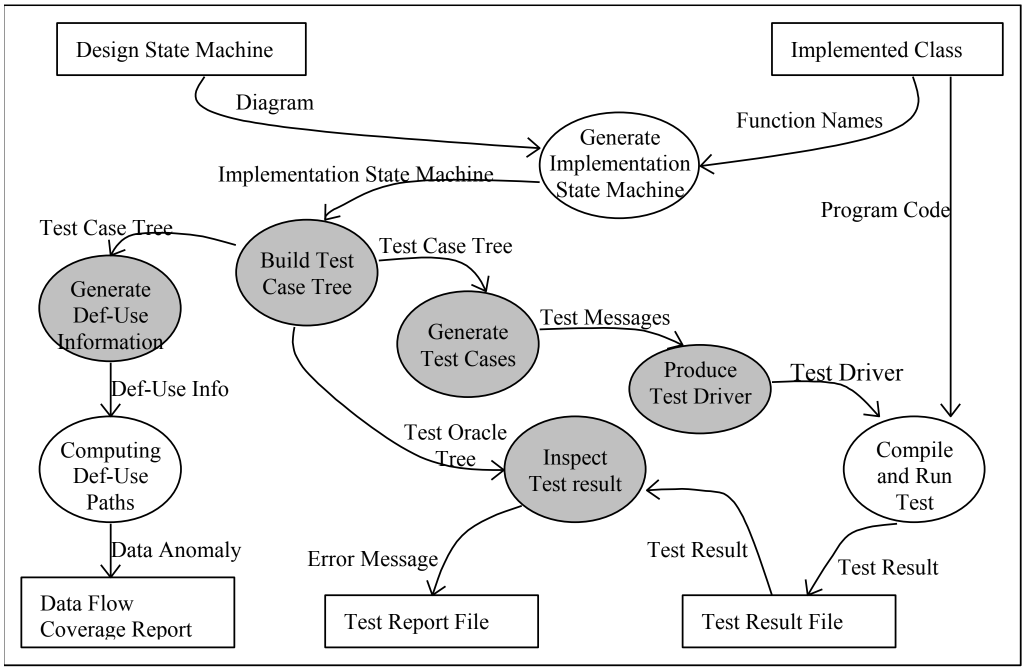


Figure 1 *The components of the automated class testing framework (MACT)*

Designing test cases for a unit test, testers usually require the specification and source code of the unit [18]. Programmers follow the specification (e.g. state machine) to implement the program code of the class. Testers also need to review the specification and implementation of the class to design test cases, in order to test the implemented class with the test messages directly. Therefore, the implementation state machine (e.g. in Figure 4) may be required in MACT. The *test case tree generator* can generate various test case trees according to the various implementation state machines. Traversing the test case tree, test messages for the class under test are automatically produced by the *test message generator* of MACT. The *test driver generator* in MACT receives test messages and generates a C++ *main()* format function as a test driver in a C++ program code. The class under test is included as a user-defined class and test messages are listed in the *main()* function. An example of the test driver is illustrated in Figure 12. In a driver, a test result file is declared as an output file, in order to record the test result (state) values while the program is executing. The tree also contains test oracles, which are based on state-based testing, consists of function names and state information. Hence the test results are inspected by the *test result inspector*, which parses the test result records (containing messages and resultant states of the messages) one by one using the test oracle tree. The *def-use info generator* also traverses the same tree, which contains the occurrences of data members in each member function, to produce definition-use information. This intra-class level information facilitates computing definition-use paths and detecting data anomaly.

4 Example: A Queue Class

Assume a circular bounded *queue* class that can only store five units of data. An object of the *queue* has six state spaces, which are the empty state, contains 1 unit, contains 2 units, ... , to the full state. The state of the *queue* object is defined by the value of the *count* data member. We can classify the states into *Empty, NotFull,* and *Full* sub-states. The *Que* object is at the start (*Empty*) state when it is defined and the constructor function sets its *count* to *0*. Moreover the state of *Que* will change to *NotFull*, when a piece of data is added into it, and *addone* transition is executed. The behaviour of *Que* is depicted with the state machine in Figure 2. In which the transition *checksize* does not cause any state change. When *Que* is at the *NotFull*, *add* and *del_data* methods (transitions) may not cause any state change until *count = 5* or *count = 0*.

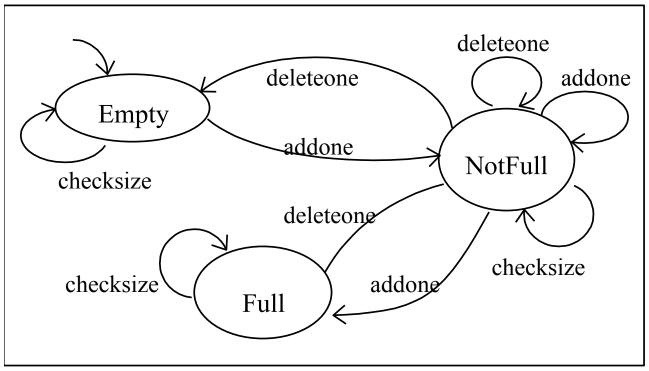


Figure 2 *The Design State Machine of the queue*

Suppose the *queue* class has *f*, *r*, *count* and *Q[SIZE]* four data members. The *Q* can only contain five units of data, and *queue* has a constructor, a destructor and four public member functions. The C++ code template of *queue* is given in Figure 3.

```
const int Size = 5;
class queue{
protected: char    Q[Size];    // bounded array
           int  f, r;           // Front/Rear index of queue
           int  count;          // a counter of the array
public: queue(void);          // default constructor
           void    is_empty(void);
           int  add(char);      // add data to the queue
           char    del_data();//delete data from the queue
           void    sizes(void);// get the size of the queue
           ~queue(void); // destructor
};
queue::queue(void){
     r = -1;  f = 0;
     count = 0;
     for (int i = 0; i < Size; i++){
        Q[i] = ' ';}
}
int queue::add(char data){
     if (count == Size){
        cout << "Not room for adding new data to stack\n";
        return (0);}
     r++;
     if(r==Size){
        r=0;}
     Q[r] = data;
     count++;
     return (1); }
```

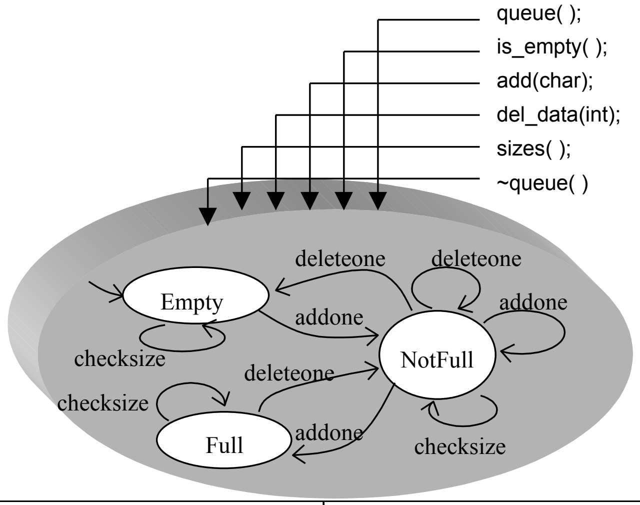
```
char queue::del_data(){
     char data;
     if (count == 0){
        cout << "can't delete data from an empty stack\n";
        return('0');}
     data = Q[f];
     f++;
     if(f==Size){
        f=0;}
     count--;
     return (data);
}
void queue::is_empty(){
     if (count == 0)
        cout<<"The queue is empty\n";
     else
        cout<<"The queue is non-empty\n";
}
void queue::sizes(){
     cout << "the size of the queue :" << count << "\n";
     return;
}
queue::~queue(){
     cout << "\nfinishing the class test\n";
}
```

Figure 3 *The implemented C++ code of the* queue *class*

4.1 Implementation State Machine of the *queue* Class

The *chechsize* transition in Figure 2 is performed by the *is_empty()* or *size()* functions in the implemented class, see Figure 3. Moreover, the detailed information of transitions and member functions, in which data members are accessed, are required to facilitate data flow analysis. Hence an *implementation state machine* of the *queue* is illustrated in Figure 4 to display the information of implemented class code and the behaviour of the object class.

The design state machine (Figure 2) is encapsulated as a class. The attached member functions show the possible messages, which can be sent to the objects by invoking these member functions declared inside the class. The table following the oval diagram describes the mapping between transitions and member functions. Function names with the | symbol in the table means OR.



Transition Names	*Function Names*
checksize	is_empty() | size()

The implementation state machine of the class **queue** is defined with finite sets ISM = (**V**, **F**, **S**, **T**) where
V = {char Q, int f, r, count}
F = {queue(), is_empty(void), add(char), dele_data(void), sizes(void), ~queue()}
S = {(**Empty**, count=0), (**NotFull**, 0<count<5), (**Full**, count =5)}
T = { **S_0**, queue(), **Empty,** (count=0); {[count, r, f, Q]}
Empty, is_empty(), **Empty,** (count=0), {<count>}
Empty, sizes(), **Empty,** (count=0), {[count]}
Empty, add(data), **NotFull,** (0<count<5), {<count, r>, [r++, r=0, Q[r]=data, count++]}
NotFull, is_empty(), **NotFull,** (0<count<5), {<count>}
NotFull, sizes(),**NotFull,** (0<count<5), {[count]}
NotFull, add(data), **NotFull,** (0<count<5), {<count, r>, [r++, r=0, Q[r]=data, count++]}
NotFull, add(data), **Full,** (count=5), {<count, r> [r++, r=0, Q[r]=data, count++]}
NotFull, del_data(),**NotFull,** (0<count<5), {<count, f>, [data=Q[f], f++, f=0, count--]}
NotFull, del_data(), **Empty,** (count=0), {<count, f>, [data=Q[f], f++, f=0, count--]}
Full, is_empty(), **Full,** (count=5), {<count>}
Full, sizes(), **Full,** (count=5), {[count]}
Full, del_data(),**NotFull,** (0<count<5), {<count>, [data=Q[f], f++, count--]}
Empty|**NotFull**|**Full,** ~queue(), **S_t,** (Ø), {Ø}
}

Figure 4 *An Implementation State machine of the* ***queue*** *Class*

In the implementation state machine, V is a set of data members declared in *queue* and the parameters of the member functions; F implies a set of member functions declared in *queue*. S is a set of all states which an object of *queue* has, and T is a set of all transitions between states, and each element of T is a quintuplet = {*current*, *function, result*, *predication*, *data access*}. Where *current*, *result* ∈ S, are states having a transition outgoes and incomes respectively. *Function* ∈ F, which triggers the transition from current to result. The *predication* causes state change and it is the post/pre-condition of the current/next transition. A set of <predication use> and/or [definition or computation use], bracketed within {} in Figure 4, indicates the data access of the data members in the functions. For instance, the <*count, r*> exposes the *count* and *r* are used at condition statements (predication use), and*, [r++, r=0, Q[r]=data, count++]* indicates that the *r, Q and count* are defined in the *add()* function, as well as the values of *r* and *count* are referenced[*].

The state change of an object is determined by the execution of the next member function at the current state. The current state value is the pre-condition/post-condition of the next member function/previous member function will be/has been executed. For example the *Que* object is at *Empty* state when *count = 0* is true. This is unsatisfied with the pre-condition of *del_data()* member function, so that the *del_data()* function cannot be triggered when the object is at the *Empty* state. The transition "*Empty, add(data), NotFull, (0<count<5)*" described in Figure 4 is *Empty* and *NotFull* as a current state and a result state respectively. This transition occurs when an object is at the *Empty*, pre-condition *(count=0)* is true, and the member function *add()* is called. When the *add()* function is executed and *count++* is performed, then the state changes to *NotFull.* Hence, the pre-condition of this transition is *count=0* and its post-condition is *0<count<5*.

4.2. The test case tree of the queue class

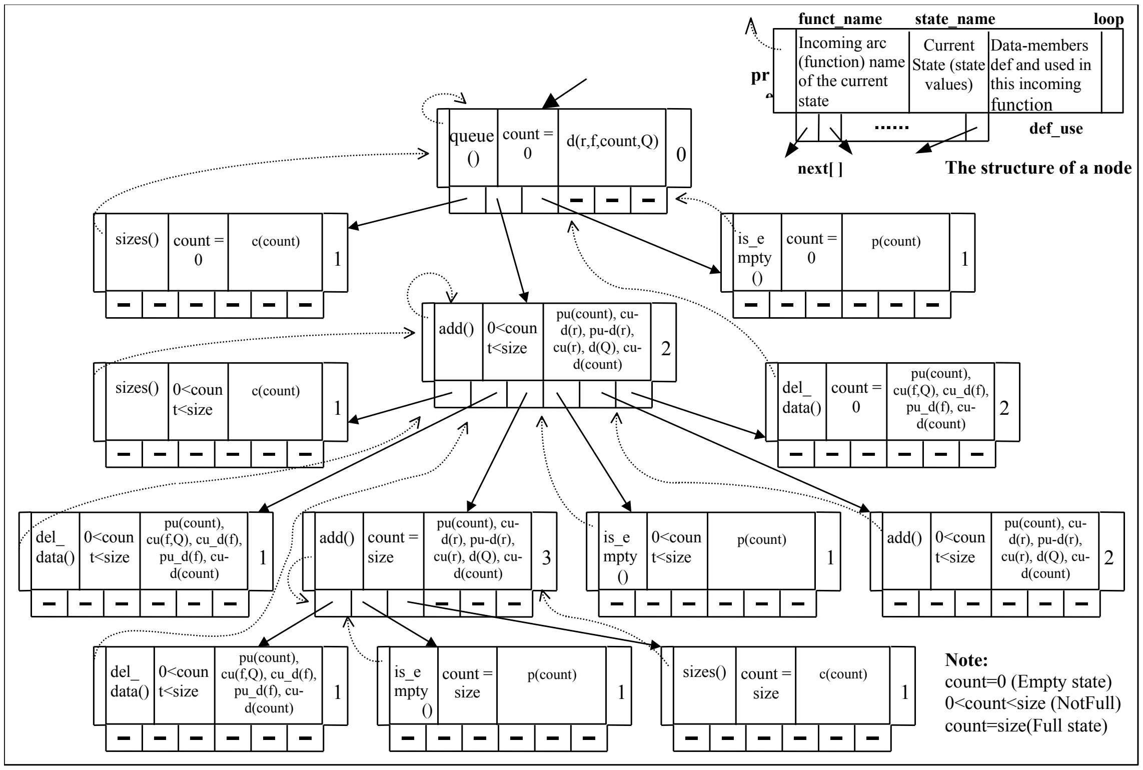


Figure 5 *A Tree Contains the Test Messages, Test Oracles, and Data Member Def-Use Information of the queue Class.*

Following the implementation state machine in Figure 4 and based on state-based testing, the *test case tree generator* (see Appendix A) of MACT produced the threaded multi-way tree in

[*] In C or C++, *int x;* is a definition statement and *x=10*; is an assignment statement. However, in data flow testing, the definition of a variable *x*, when it is at the left side of an assignment (e.g., *x=10*). The value of *x* is referenced, when it is at the right side of an assignment (e.g. *y=x+10;)*

Figure 5. The tree completely duplicates the graph of the state machine. The pointers and the threads in the tree can simulate any cyclic links in the state machine. Each (circular) path is a sequence of test cases. The core of MACT is the threaded multi-way tree, which simulates the behaviour of a state machine. Even more complicated state machines (featuring hierarchies, concurrency, and nested states) can also be represented by this approach [7]. The structure of each tree node is shown at right up corner in Figure 5.

5 Possible test cases of *queue* generated from MACT

After tracing the tree in Figure 5 with *Que*, the test cases of *Que* are produced by loops in the tree from the root node and shown in Figure 6. The condition in bold square brackets followed by messages is a pre-condition. The state value (post-condition) within bold square brackets following the messages means the expected resultant state, after the messages being executed. For the sake of simplicity, repeatedly executing a member function is represented with "*Oue.f(),...,Oue.f()*" form, and messages in a pair of arrow brackets represent the state is still at same after executing the messages. The *test message generator* (its algorithm shows in [5]) traverses the tree from the root node down to the leaves. A generated message file, consists of a sequences of function calls, for *Que* is produced, see Figure 7.

[count=0], Que.is_empty(), **[count=0].**
[count=0], Que.sizes(), **[count=0].**
[count=0], Que.add(), **[0<count<5]**, <Que.add(),...,Que.add()>, **[0<count<5]**, Que.sizes(), **[0<count<5].**
[count=0], <Que.add(),...,Que.add()>, **[0<count<5]**, Que.del_data(), **[0<count<5].**
[count=0], Que.add(), **[0<count<5]**, Que.del_data(), **[count=0].**
[count=0], <Que.add(),...,Que.add>, **[0<count<5]**, Que.add(), **[count=5]**, Que.sizes, **[count=5].**
:
[0<count<5], Que.add(), **[0<count<5]**, Que.add(),**[count=5]**, Que.is_empty(), **[count=5].**
[0<count<5], Que.add(), **[0<count<5]**, <Que.del_data(),…,Que.del_data()>, **[0<count<5].**
[0<count<5], <Que.add(),...,Que.add()>, **[0<count<5]**, Que.add(), **[count=5].**
[count=5], Que.del_data(), **[0<count<5].**
:

Figure 6 *Test cases of the Que object*

```
   :
Que.add();
Que.sizes();
Que.add();
Que.del_data();
Que.add();
   :
 Que.del_data();
Que.add();
Que.sizes();
Que.add();
   :
Que.sizes();
Que.is_empty();
   :
```

Figure 7 *A test message file for the Que object*

6 Test cases of *queue* generated based on data-flow criteria

A class is a basic unit of testing in an object-oriented program, and most of this test work has centred on black-box approaches. Harrold [2] developed a class control flow graph to connect all methods in the class, and adapted Pande's [19] data flow analysis algorithm to compute the data flow information required for data flow testing. Hong [20] demonstrated the Class State Machine (CSM), extended from Finite State Machine, to specify the behaviour of classes. The CSM was then transformed into a Class Flow Graph to show data flows of the state machine. Eventually, selected intra-class test cases using data flow testing techniques.

Harrold [2] and Hong [20] describe that intra-class def-use information can guide testers in the selection of sequences of methods (function calls). In the following, the *queue* class is used as example to discuss the difficulty of generating intra-class test cases for the *Que* based on data flow testing techniques. Moreover, the selection of sequences of function calls, which satisfy certain data flow coverage criteria, may also contain some unnecessary (ambiguous) method sequences. Testers need to filter them by referencing the functionality of the class.

6.1 Overview of Data Flow Testing Criteria

Data flow testing techniques need directed flow graphs to facilitate the computation of def-use pair information, the selection of test cases, and the detection of anomalies within the program under test. The indication of anomalies may include: (1) defining a variable twice with no intervening use, (2) referencing a variable that is undefined, and (3) not referencing the variables that are defined [1] [21].

Based on data flow testing, each instance of a variable in a program is classified as a **def**inition or a **use**. A definition of a variable is that a variable is assigned a value. A use of a variable is that the value of the variable is used (referenced). Uses of a variable are further divided into two classes as either **c**omputation **use**s (c-use) or **p**redicate **use**s (p-use) [22]. 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*.

The du path of each data member is from its definition to every use that is reached by the definition. Let *i* be any member function and *v* any variable such that $v \in$ def(*i*). Hence, *dcu(v,i)* is the set of all functions *j* such that $v \in$ c-use(*j*), and in this path there is a def-clear sub-path with respect to *v* from *i* to *j*. The *dpu(v,i)* indicates the set of all functions *j* such that $v \in$ p-use(*j*) and in this path there is a def-clear sub-path with respect to *v* from *i* to *j*.

Data members in a class are also global variables. In data flow testing at intra-class lever, a global use (c-use or p-use) of a data member *x* if and only if the definition of the *x* preceding its use does not occurs within the same member function. Otherwise, it is a local use. A global definition of a data member *x,* if and only if the last definition of the *x* occurs in a member function *i,* and there is a def-clear path with respect to the *x* from the *i* to another member function *j.* In the function *j*, there is a global use of the *x* [23].

The criteria of testing-path-selection are based on data flow, and focus on variables to be defined and used. When executing a test case, the test case is said to exercise a def-use (sub) path if the (sub) path is traversed. Therefore, tracing the flow of data members among member functions in the class rather local variables within an individual function is concerned in this paper. Some data flow testing concepts criteria can be referenced in [2] [14] [15] [22].

6.2 Def-use information in the *queue* example

Selecting intra-class test cases, the interest is only the data members in the public member functions of *queue*. In this example, moreover, each element in the data member *Q* array will be dealt with individually.

For simplicity of def-use presentation in each member function of *queue*, each code statement is a unit, in which the definition and/or use of data members occur. For example, in the data flow graph of *add()* function (see Figure 8), the **p-use(*count*)** and **c-use(*r*)** at the node 1 and 2 cause concerned as to whether or not the *count* and *r*, used in this function, have been properly defined in the preceding functions. Moreover, we also need to examine if the defined *count* and *Q*, **def(*count*)** and **def(*Q*)**, will be able to be used in the succeeding functions.

```
int que::add(char data){
1.   if (count == Size){
       cout << "Not room for new \n";
       return (0);
     }
2.   r++;
3.   if(r==Size){
4      r=0;}
5.   Q[r] = data;
6.   count++;
7.   return (1);
}
```

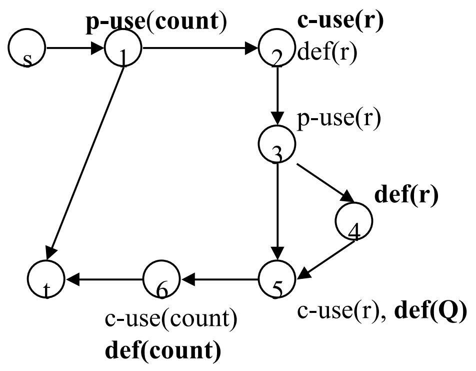


***Figure 8** The C++ code add() member function on the left and its directed flow graph on the right. Nodes in the graph represent statements in the function; **s**tart and **t**erminate nodes are added for analysis.*

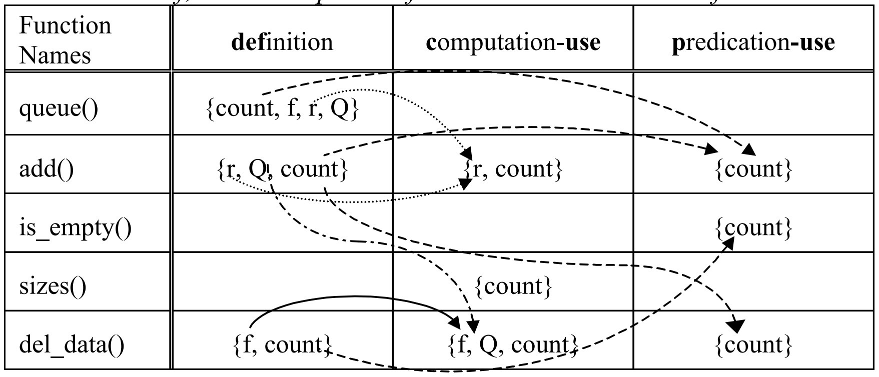
As in Figure 8 the bold def and use of data members should be computed, and also the non-bold data members. The dpu(*r*, 2)={3}, and dcu(*r*, 4)={5} have formed sub paths within the *add()*, and the definition of the *r* at statements 2 or 4 could be either used in the succeeding functions. Figure 9 illustrates the intra-class definition-use information of data members in *queue*.

```
functions (queue(), is_empty(), add(), del_data(), sizes()).

global_defs (queue(), [count, f, Q, r]).       global_defs (del_data(), [f, count]).
global_cuses (queue(), []).                    global_cuses (del_data(), [Q, f, count]).
global_puses (queue(), []).                    global_puses (del_data(), [count]).

global_defs (is_empty(), []).                  global_defs (sizes(), []).
global_cuses (is_empty(), []).                 global_cuses (sizes(), [count]).
global_puses (is_empty(), [count]).            global_puses (sizes(), []).

global_defs (add(), [r, Q, count]).
global_cuses (add(), [r, count]).
global_puses (add(), [count]).
```

Figure 9** Global definition-use information of data members within **queue

The definitions and uses of the data members among the functions of *queue* are shown in Table 1. The test cases can be generated to cover associations between definitions and uses of each data member from the table, based on du-path criteria. The arcs in Table 1 show that some pairs of sequence methods are used for intra-class testing. For example, du-paths with respect to the *count* are in *queue()→add()*, *add()→add()*, *add()→del_data()*, and *del_data()→is_empty()*. There are d-cu paths of the *r* in *queue()→add(),* and *add()→add()*. A d-cu path of *Q* and *f* exists in *add()→del_data()* and *del_data()→del_data()* respectively*,* Furthermore, a sequence of messages *queue()→add()→add()→del_data()→del_data()* which has all defs coverage, can be established. It is time-consuming work to compute this sequence.

Table 1 *The def, c-use and p-use information associated with functions*



Function Names	**def**inition	**c**omputation-**use**	**p**redication-**use**
queue()	{count, f, r, Q}		
add()	{r, Q, count}	{r, count}	{count}
is_empty()			{count}
sizes()		{count}	
del_data()	{f, count}	{f, Q, count}	{count}

6.3 Generating Data Flow Test Cases

It is difficult to discover all possible test cases to achieve all defs, all uses, or all du-paths. For example, a class has N data members (denoted V_1, V_2, …, V_n) and M public member functions, in each of which every data member is defined (denoted M_d), computation used (denoted $M_{cu)}$, and predicate used (denoted M_{pu}). Then the maximum du-paths across only two member functions with respect to every data member are $V_1\times(M_d \times(M_{cu}+M_{pu})) + V_2\times(M_d \times(M_{cu}+M_{pu})) + \ldots +V_n\times(M_d \times(M_{cu}+M_{pu})) \Rightarrow N\times(M_d\times(M_{cu}+M_{pu})) \Rightarrow 2N\times M^2$. In this *queue* example, there are $3\times(3+3)+2\times(1+1)+2\times(1+1)+2\times(1+0)=28$ du-paths within two member functions in the sets with respect to data member, *count, r, f* and *Q* individually. Additionally, a pair of ordered functions may not only contain a data member. Such that *count* and *r* two data members occur in the *queue()→add()* test path. Selecting a test case, the du pairs of all possible data members in each test case ought to be considered.

6.4 The Complexity of Data Flow Criteria for Test Case Selection

Weyuker [24] [25] proposed that all du-paths require an exponential number of test cases in the worst case. If *d* is the number of (two way) decisions in the program, then the *all uses* data flow criterion requires $O(d^2)$ test cases, and *all du-paths* requires $O(2^d)$ in the worst case. For example, suppose a program (procedure) 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. However, this is simply the worst case and in practice both criteria would require only at most *d + 1* test cases [24]. That means many test cases of the worst case are not necessary.

In a class, each method has to be tested individually. Each class will then have N associated test cases when there are N methods in the class. Nevertheless, to check the validity of calling sequences within the class, such as intra class coupling, *N* methods implies the order *N!* test cases [26]. In those cases there could be some redundant sub sequences. For instance, the first element of the *Q* is defined in the first *add()* function (called F_{add1}), and the element is referenced in the first *del_data()* function (called F_{del1}). There can be several member functions with different performance orders in the interval of F_{add1} and F_{del1}. Such as *add()→del_data(); add()→is_empty()→del_data(); add()→ … →del_data();* and each of the test cases has a du-path with respect to *Q[0]*.

The result, found by Bieman in [27], shows that 80% of the procedures need to be tested by ten or fewer complete paths to satisfy all du-paths criteria. However, it is a tedious task to find the redundant test cases.

6.5 Inadequate and Ambiguous Test Cases

Test cases can be produced following the du-paths. Some of the paths could be inadequate. For example in the queue, a *queue()→del_data()→add()* can achieve the du-paths with respect to *count*, *f*, *r,* and *Q*. However, it is unreasonable to perform the *del_data()* member function following the constructor, *queue()*. In addition, there is a du-path of the *count* and *r* data members in *queue()→add()*, but a double definition anomaly occurs on *Q*. If we avoid the ambiguous test cases, then some of data members may not be tested, such as the *queue()→add()* which is needed to test *count* and *r*.

6.6 Feasible and Infeasible Test Paths

An *infeasible* path is that no input data exists which can cause such a path to be executed. However, a path is *feasible* if there are some input data, which will cause the path to be traversed during execution. The sequence methods calls from outside of the class under test can be specification infeasible or implementation infeasible. Infeasible sequence methods (subpaths) should not (or cannot) be executed according to the specification. For example, *queue()→del_data()* sequence should not be required in the specification, and an infeasible implementation is such as *queue()→add()→del_data()→del_data()* sequence. Because *queue* cannot accept the second delete data message from the client object when its state value is zero (i.e. *count* == *0*).

Harrold [2] and Hong [20] demonstrated their techniques are also useful for determining which sequences of methods should be executed to test a class, and pointed out error sequences with examples that need not be run. However, they did not discuss the technique to select infeasible sequences from *N!*. In fact, Weyuker [25] found that the non-executable (infeasible) path problem was the primary practical difficulty in using the all du-paths criterion, because there are many infeasible paths to contend with the criterion. In automatically generating sequences of calling methods to satisfy data flow criteria, the problem of generating infeasible sequences is impossible to avoid [28].

To gain all possible useful test cases, we need to remove redundant paths and to eliminate the infeasible (such as inadequate, ambiguous, or non-executive) test cases from the generated intra-class level test cases based on data flow testing criteria. However, in practice, it is not an easy task to detect them from *N!* test cases.

7 Data Flow Testing Approach in MACT

At the different point of view, if we already have test cases for all possible feasible paths of the class. We, then, use data flow analysis technique to detect whether or not these feasible sequences violate data flow criteria. On this assumption, intra-class test cases are generated by MACT based on state-based testing technique, and then the sequence paths within these test cases are analysed with data flow criteria to conclude whether these test cases are satisfied with data flow coverage.

The data flow testing approach of MACT is to firstly generate a test case tree of the class, which is derived from the design (or implementation) state machine. This is the same as the discussion in section 4.2. Secondly, by tracing the tree, intra-class definition-use information of data members can be produced for data flow testing. Finally, the def-use information of the data members is analysed to detect whether or not the class under test is satisfied with data flow coverage.

7.1 Generating Def-Use Information from MACT

Reviewing the implementation state machine, we then fill the occurrences of data members into the *def-use* field of the respective node in the test case tree, while the test case tree generator is executing. After the test case tress of the *queue* contains def-use information is produced, see Figure 5. Each node of the tree has the function name, current state value (name), and def-use information of data members. Based on the tree, we are

(1) to identify definitions and uses of each data member in the transition of the implementation state machine;

(2) to review these definitions, uses, and the sequences of test messages (generated by the MACT) to compute the sequences of definition-use pairs; and

(3) to analyse the information of these definition-use pairs to show what kinds of the data flow criteria those test messages can achieve and if any anomaly occurs.

The test case paths (see Figure 10), generated by tracing the tree, are used to detect whether they cover def-use paths. For example, the "*constructor, **Empty**, sizes(), **Empty***" test case has a sequence of member functions, *queue()→sizes()*, and we can analyse the def-use information to find if the def-use path with respect to data members occur in this test case, and thus detect if data anomalies exist.

7.2 Computing DU Paths

The def-use information of data members in each member functions of the several test cases, generated by the MACT, is given in Figure 10.

d(r,f,count,Q), cu(count). *// constructor, **empty**, sizes(), **empty**.*

d(r,f,count,Q), *pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count),* pu(count),cu(f,Q), cu-d(f), pu-d(f), cu-d(count).
*// constructor, **empty**, add(), **notfull**, del_data(),**empty**.*

d(r,f,count,Q), *pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count)*, *pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count)*,…, *pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count)*, pu(count),cu(f,Q), cu-d(f), pu-d(f), cu-d(count), pu(count).
*// constructor, **empty**, add(), **notfull**, add(), **notfull**, ...,add(), **notfull**, del_data(), **notfull**, is_empty(), **notfull**.*

:

d(r,f,count,Q), *pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count)*, *pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count)*, …, *pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count)*, pu(count),cu(f,Q), cu-d(f), pu-d(f), cu-d(count), cu(count).
*// constructor, **empty**, add(), **notfull**, add(), **notfull**, ..., add(),**full**, del_data(), **notfull**, sizes(), **notfull**.*

:

:

Note:

d(x):	x is defined
cu(x):	x used for computation
pu(x):	x used in predicate
cu-d(x):	cu(x) and then d(x)
pu-d(x):	pu(x) and then d(x)

Figure 10 *Data members definition-use pairs in the **queue** Class*

The definition-use of the data members in the "*queue(), **Empty**, add(), **NotFull**, del_data(), **Empty***" is shown as an expression in Figure 10. For the sake of explanation, the expression has been divided into four rows with different data members and they are shown in Table 2. In the first row, for example, the du paths can be computed with respect to the data member *r*. The *r* is defined in the *queue()* and *add()* functions, and then is used in the *add()* function. This shows no anomaly occurred on the *r* in this test case.

Table 2 *The du paths with respect to each data member in* ***queue***

Functions / Data Members	**queue()**	**add()**	**del_data()**
r	def(r)	cu-d(r), pu-d(r), cu(r)	
f	def(f)		cu(f), cu-d(f), pu-d(f)
count	def(count)	pu(count), cu-d(count)	pu(count), cu-d(count)
Q	def(Q_1,…,Q_5)	d(Q_1)	cu(Q_1)

Note: The arcs emerging from table show the defined data member could be used in succeeding functions

7.3 Code Optimisation

There are several du paths with respect to *f* and *count* in the second and third rows, and the last definition of *count* and *f* in this case is not against the du path criteria. Because the *count* and *f* are global definitions in the *del_data()*, they may be used in the succeeding messages, such as *is_empty()*, *sizes()*, or *del_data()*. However, if the all test cases generated from the test case tree (see Figure 10) are carefully reviewed, we can find out the last defined *f* will never be used. The sequence member functions following the *del_data()*, in which a definition of the *f* occurs, do not reference the defined *f*, except another following *del_data()*. The following *del_data()* references the *f* defined in the previous *del_data()*, will define the *f* again. This shows that the last defined *f* within the last *del_dat()* function in every sequence test case does not be used. That *del_dat()* has a "not referencing the variable that are defined" anomaly.

The *f* data member is used as an index when the front element in the Q*[SIZE]* array is removed. The last defined *f* can only be used in the *sizes()*, *is_empty()* or *~queue()* functions. In the *Que***,** the *count* is used to reply to the messages of the *sizes()* and *is_empty()* methods. However, the *count* can be replaced with *f* and *r*. By calculating *f* and *r*, the result can indicate whether the *Que* is at the empty, not-full or full state, even show how many units of data exist in the *Que*. This is a code optimisation problem and will not be discussed further in this paper.

7.4 Anomaly Detection

An anomaly of data member *Q* occurs in the *queue()→add()* test case is found in the last row of Table 2. In fact, programmers are used to initialise variables as they are defined in the constructors of a class. In this case, it is necessary to initialise the *count*, *r* and *f*, but it is not necessary to empty the *Q* in the *queue()*. The test cases generated based on state-based testing could not detect this kind of anomaly.

Moreover, theses anomaly problem, the *Q* has double definition and the last defined *f* may not be referenced, can never be detected in the intra/inter method testing. Because intra-method testing is performed to test each function in the class individually and inter-method testing is executed to test a function together with other functions (in the same class) that are called directly or indirectly.

8 Other Functions of MACT

8.1 Test Driver Generator of MACT

The *test message generator* writes the function names into a test message file, which is part of the test driver. The *test driver generator* in MACT reviews the message file, and then generates a C++ *main()* function as a test driver. The driver will include the class under test, and then test the class with the messages. After executing the test driver, the test results can be recorded into a test result file.

The class, *queue*, under test is included in the program as a user-defined class (e.g. *#include "queue.h"*) [29], and it is inherited by a subclass called Queue. Moreover, an additional member function, called *amt()*, is declared in the subclass in order to access the state values (the type is private) stored in data member *count,* which is also inherited from the *queue* class. Obviously, the *amt ()* can be declared as a friend function. Therefore, the implemented *queue* class (from the developers) is insulated from change and will not be modified at all.

A test result file is declared as an output file in this main function (test driver), so that it records the result of each test message execution. The driver is to store the test result of each message into the test result file, while the program is executing.

Test messages are embedded in the main function in the test driver example (see Figure 12). Alternatively, a test message file, containing those test messages, can be declared as an input file in the main function, and each test message is accessed as an input record, while the file is read. Test message files are ASCII text type files and each test message is stored in the files as a string. Therefore, a program code generator is utilised to read the test message file as an input file, and then produces the source code of a test driver program.

A test driver program, in which a class under test is included and a test message file and a test result file are declared, can be manually/automatically produced by a *program editor* or a *program source code generator*. If a source code generator can play as a test driver generator. A complete C++ test driver program can be automatically built by answering the prompts asked by the generator, see Figure 11. The prompts may ask to enter the names of (1) a class under test (e.g. *queue.h* in Figure 12), (2) a test message file, (3) a test result file, and (4) a function to reveal state value, e.g. *amt()* in Figure 12.

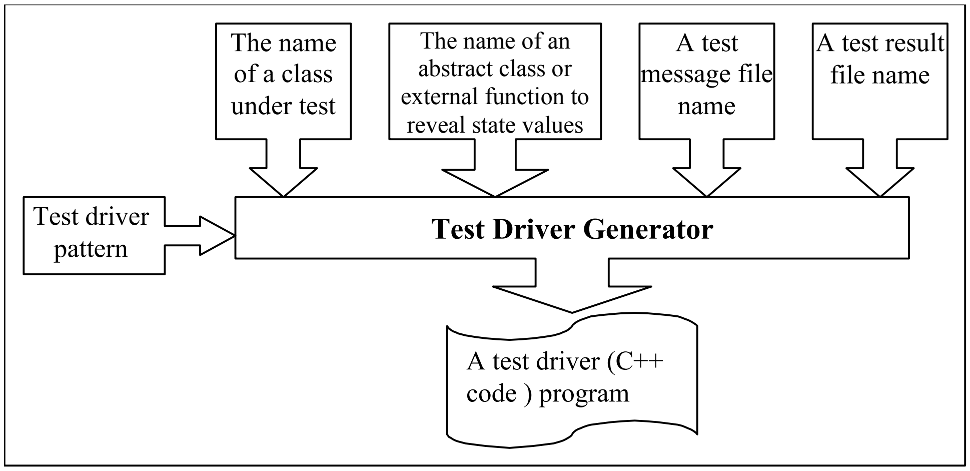

The name of a class under test
The name of an abstract class or external function to reveal state values
A test message file name
A test result file name
Test driver pattern
Test Driver Generator
A test driver (C++ code) program

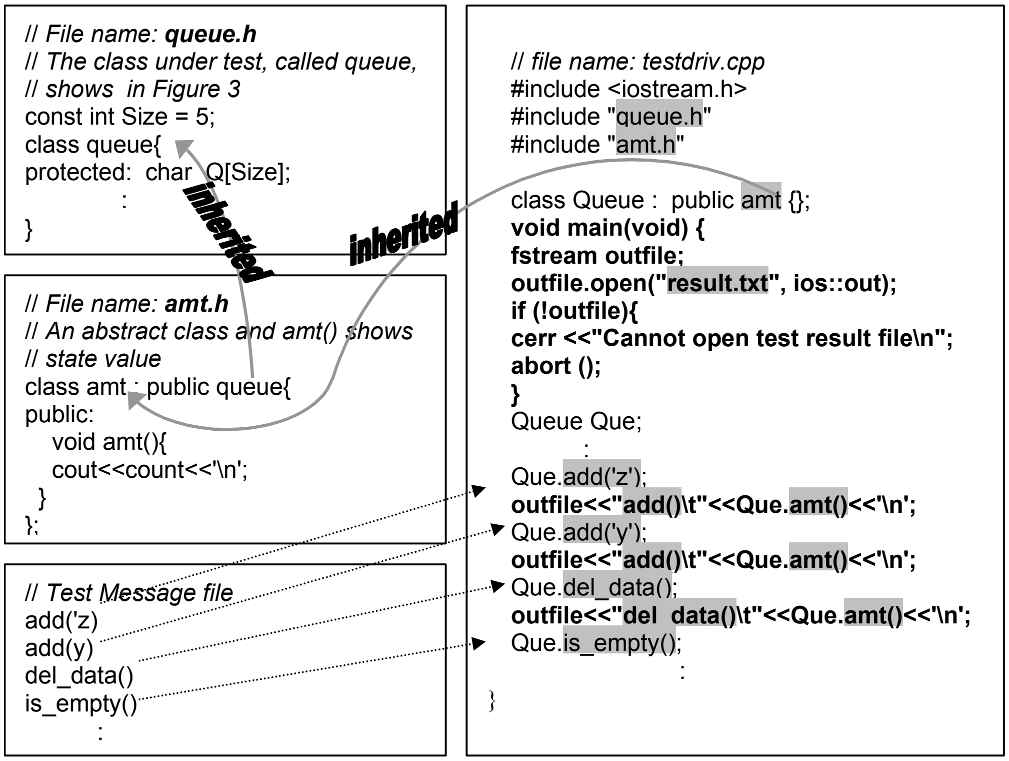
Figure 11 *Possible requirement to generate test driver in MACT*

Of course, the class under test and the function to show state values should be developed at first. The class is stored as a user-defined class with '*.h*' sub-program name in proper directory, such as *'c:\borlandc\include\'* sub-directory in Borland C++·. Then, the class is included in the generated test driver program and the function (friend function) is linked as an external function [29] with the program.

Another way is that the function can be defined in an abstract class [29], which may also be stored with '*.h*' sub-program name in *'c:\borlandc\include\'* sub-directory. In order that the class

· Borland C++ for Windows, Version 5.0, Borland International, Inc.

can be included and inherited by the test driver program and subclass in the test driver program respectively, see Figure 12. In which, the right hand side program is a test driver program example. The parts high lighted in grey are inlayed when testers enter proper names in response to the prompts of the test driver generator. The rest parts without high light is the test driver pattern, see Figure 11.



```
// File name: queue.h
// The class under test, called queue,
// shows  in Figure 3
const int Size = 5;
class queue{
protected:  char  Q[Size];
            :
}
```

```
// File name: amt.h
// An abstract class and amt() shows
// state value
class amt : public queue{
public:
    void amt(){
    cout<<count<<'\n';
  }
};
```

```
// Test Message file
add('z)
add(y)
del_data()
is_empty()
         :
```

```
// file name: testdriv.cpp
#include <iostream.h>
#include "queue.h"
#include "amt.h"

class Queue :  public amt {};
void main(void) {
fstream outfile;
outfile.open("result.txt", ios::out);
if (!outfile){
cerr <<"Cannot open test result file\n";
abort ();
}
Queue Que;
          :
Que.add('z');
outfile<<"add()\t"<<Que.amt()<<'\n';
Que.add('y');
outfile<<"add()\t"<<Que.amt()<<'\n';
Que.del_data();
outfile<<"del  data()\t"<<Que.amt()<<'\n';
Que.is_empty();
                    :
}
```

Figure 12 *The example of test driver generation.*

The diagrams from left top to down in Figure 12 are the class under test that is stored with *queue.h* file name, an abstract class which inherits the class under test and contains a function to show state value, and the test message file respectively.

8.2 Test result inspector

The *test result inspector* in MACT will parse the test result records (the messages and the resultant states of the messages) one by one using the test oracles in the tree. The oracles of *queue* (Figure 5) contain the expected state for each expected message. The inspector opens the test result file as an input file, and it can sequentially read each record, and then compare the function name and state value in the record with the oracles in the tree. If the state value in a record mismatches the expected state value in the tree, then an error is deemed to have occurred. The algorithm of inspecting the test result file is given in Figure 13.

```
STEP 0 set root to current
STEP 1 while not end of test_result file
STEP 2   read the test_result record and store data to funct and state two variables
STEP 3   if all children nodes of the current node have been traced go to STEP 5
STEP 4     if a child node's funct_name = funct and state is satisfied with its state condition
               then the test_result record is correct;
                     assign the child node's pre to current and
                     go STEP 6
               else go to STEP 3
STEP 5   print "this test result record is in error"
STEP 6   go to STEP 1
```

***Figure 13** The algorithm to inspect the test results with the oracle tree*

For example, when the first record (function name = "*add()*", and count = "1") of the test result file has been read at *STEP 2*. The inspector, at *STEP 3*, will look for a child node of the root (it is also the *current* node at the moment) in Figure 5 which contains the *add()* function name and its state condition is satisfied with the state value (count = 1). The middle child node of the root matches the requirement. Therefore, the *current* node moves to the middle node of the root when the statements in *STEP 4* have been executed.

Following the above algorithm, the Test Result Inspector can detect errors when state values in the result file cannot be matched with the state conditions in the oracle tree. For example, if count = 6 after executing an *add()* function, then an error message should be reported by the inspector. Because the state value 6 will not be satisfied with any state condition in the oracle tree, see Figure 5. If the inspector can not respond this with an error message, then the error existed in the *Que*. That intends the *Que* may not have ability to handle the over flow problem, or the *count* data member in the *Que* may be accumulated improperly.

9 Future Work and Conclusions

The core of MACT is a threaded multi-way tree (test case tree) which simulates the behaviour of a state machine. Test cases can be generated from the tree, which contains the test oracle of the class under test. A key feature of the *test case generator* is its ability to generate test cases even when the state machine has circular references (see Figure 5). More complicated state machines (featuring hierarchies, concurrency, and nested states) can also be represented by this approach [7], and provides a topic for future study. The def-use information of each data member in the test case tree nodes can be listed in a data file by the same test message generator (discussed in section 5.) Using stacks to automatically detect the data anomaly is another topic. Moreover, it is also worth studying if this du path information can support test data selection for the test cases, generated from MACT.

The MACT tool can generate all possible intra-class test cases has been discussed, and we have also followed *data-flow* criteria to compute the du paths with respect to the data members in the intra-class test cases. In order to insure the class under test is satisfied with *data-flow* coverage. We suggest that the steps of object oriented class testing are:

(1) The *data-flow* testing is employed at intra-method and inter-method levels.

(2) At intra-class level, each class is tested as a unit with test cases based on state-based criteria.

(3) The definitions and references of the data members within the sequences of test cases need to be computed and detected.

References

1. Beizer, Boris "Software Testing Techniques," Van Nostrand Reinhold, 1990.

2. Harrold, Mary J. and Rothermel, G. "Performing Data Flow Testing on Classes," 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering, Dec. 1994, 154-163.
3. Siegel, Shel 1996, "Object Oriented Software Testing-A Hierarchical Approach," John Wiley & Sons, Inc. 1996.
4. Tsai, Bor-Yuan; Stobart, Simon and Parrington, Norman; "A Method for Automatic Class Testing Object-Oriented Programs Using A State-Based Testing Method," 5th European Conference Software Testing Analysis and Review 1997, (EuroSTAR '97,) Edinburgh, UK, 403-415.
5. Tsai, Bor-Yuan; Stobart, Simon; Parrington, Norman and Mitchell, Ian "An automatic Test Case Generator Derived from State-Based Testing," 5th Asia-Pacific Software Engineering Conference 1998, (APSEC '98,) Taipei, Taiwan, 270-277.
6. Tsai, Bor-Yuan; Stobart, Simon; Parrington, Norman and Mitchell, Ian "A Hybrid Object-Oriented Class Testing Method - Based on State-Based and Data-Flow Testing," accepted for inclusion in 7th Annual International Conference on Software Quality Management (SQM '99), March, 1999, Southampton UK.
7. Tsai, Bor-Yuan; Stobart, Simon; Parrington, Norman and Mitchell, Ian "Automated Class Testing Using Threaded Multi-way Trees to Represent the Behaviour of State Machine," to appear at Special Volume on Software Reliability, Testing, and Maturity, Annals of Software Engineering, Vol. 8, 1999.
8. Tsai, Bor-Yuan; Stobart, Simon; Parrington, Norman and Mitchell, Ian "Performing State-based Testing and Data-flow Testing on Object-Oriented Classes," accepted for inclusion in The 15th International Conference on Advanced Science and Technology (ICAST '99), April, 1999, Argonne, Illinois
9. Chow, T. S. "Testing software design modeled by finite state machines", IEEE Transactions on Software Engineering SE-4(3): 1978, 178-187.
10. Turner, C. D. and Robson D. J. "A Suite of Tools for the State-Based Testing of Object-Oriented Programs," Tech. Report: TR-14/92, U. of Durham 1992.
11. Binder, R. V. "State-based testing", Object Magazine, July-Aug 1995, 75-78.
12. Turner, C. D. and Robson, D. J. "The State-based Testing of Object-Oriented Programs," Conference on Software Maintenance 1993, 302-310.
13. Harrold, Mary J. and Soffa, Mary L. "Interprocedural Data Flow Testing," 3rd Testing, Analysis and Verification SYMP (SIGSUFT89), 1989, 158-167.
14. Laski, J. and Korel, B. "A Data Flow Oriented Program Testing Strategy," IEEE Transactions on Software Engineering, SE-9(3), May 1983, 347-354.
15. Rapps, S. and Weyuker, E. J. "Selecting Software Test Data Using Data Flow Information," IEEE Transactions on Software Engineering, SE-11 (4), April 1985, 367-375.
16. Tsai, Bor-Yuan; Stobart, Simon; Parrington, Norman and Mitchell, Ian "Selecting Intra-class Test Cases Based on State-Based Testing and Data Flow Testing Methods," Occasional Paper, CIS-4-98, School of CIS, University of Sunderland, UK, 1998.
17. Kit, Edward "Software Testing in the Real World," Addison-Wesley, 1995.
18. Myers, Glenford J. "The Art of Software Testing," John Wiley & Sons, 1979.
19. Pande, H., Landi, W. and Ryder, B. "Interprocedural def-use associations for C systems with single level pointers," IEEE Transactions on Software Engineering, 20(5); 1994, 385-403.
20. Hong, H. S., Kwon, Y. R. and Cha, S. D. "Testing of Object-Oriented Programs Based on Finite State Machines," Asia-Pacific Software Engineering Conference '95, Australia, Dec. 1995, 234-241.

21. Spillner, Andreas "Control Flow and Data Flow Oriented Integration Testing Methods", Software Testing, Verification and Reliability, Vol. 2, 1992, 83-98.
22. Frankl, Phyllis G. and Weyuker, Elaine J. "An Applicable Family of Data Flow Testing Criteria," IEEE Transactions on Software Engineering, 14(10), 1988, 1483-1498.
23. Rapps, S. and Weyuker, Elaine J. "Data Flow Analysis Techniques for Test Data Selection," 6th International Conference on Software Engineering, 1982, pp 272-78.
24. Weyuker, Elaine J. "The Complexity of Data Flow Criteria for Test Data Selection," Information Processing Letters, 19(2), 1984, 103-109.
25. Weyuker, Elaine J. "The Cost of Data Flow Testing: An Empirical Study," IEEE Transactions on Software Engineering, 16(2), February 1990, 121-128.
26. Duncan, I. and Doake, J. 'The Use of OO Design Metrics to Indicate Adequate Testing and Maintenance Costs," TR-9801, Dept. of Computer Science, Anglia Polytechnic University, UK, 1998
27. Bieman, J. M., Schultz, J. L.: "Estimating the Number of Test Cases Required to Satisfy the All-du-paths," TAV3-SIGSOFT '89, 1989, 179-186.
28. Parrish, A. S., Borie, R. B. and Cordes, D.W.: 'Automated Flow Graph-Based Testing of Object-Oriented Software Modules,' Journal of Systems and Software, 1993, 23, 95-109.
29. Lafore, Robert " Object-Oriented Programming in C++," Waite Group Press, ISBN: 157169160X, 1998.
30. Tsai, Bor-Yuan; Stobart, Simon; Parrington, Norman "An automatic Test Case Generator Derived from State-Based Testing," Occasional Paper, CIS-1-98, School of CIS, University of Sunderland, UK, 1998.

Appendix A

```
Step 1    Create a pointer queue which can temporarily store tree nodes
Step 2    Create the head node of the tree
Step 3    Add the head node into the queue
Step 4    While not stop building the tree
Step 5        Create a new node and fill the required information in each field
Step 6        While the new node is not a child of the first node in queue
Step 7            Delete the first node from the queue
Step 8        Link the new node as a child of the first node in the queue
Step 9        If the state_name of a node in the queue is the same as the new node's
Step 10           the pre of the new node threads to the node in the queue
Step 11       Add the new node into the queue
```

This algorithm can build test case trees, such as a tree illustrated in Figure 5. A complete C++ program, coded following the algorithm, is shown in [30].