Generating Test Cases for Overloaded Object-Oriented Programs using EGS (Extended General State-charts)

Bor-Yuan Tsai*, Simon Stobart and Norman Parrington
School of Computing and Information Systems, University of Sunderland,
St. Peter’s Way, Sunderland. SR6 0DD
+44 191 5153201
E-mail: {cs0byt, cs0sst, cs0npa}@isis.sunderland.ac.uk

Abstract

Overloading and polymorphism are two important aspects in object-oriented programming languages. When an class has either of these two specialties and is depicted with a state-chart which characterizes the behaviour of the class, problems arise. Such as different transition arcs are labelled with the same name but may represent similar duties on that state-chart. General state-charts (GS) can be used as patterns to model the overloading and polymorphism aspects of C++ programs. Furthermore, method functions can be mapped to a general state-chart to form an extended general state-chart (EGS). From these, state-transition trees can be generated.

Errors, which occurs when overloaded/polymorphism functions are invoked against the wrong object, are hard to detect by state-based testing. This is because the resultant states after executing the messages could be the same as the expected states. In this class (unit) testing paper, we try to use EGS to produce legal and illegal test cases in order to test a class which has overloaded functions. The illegal test cases show all possible misusages of the overloaded functions.

1. Introduction

Overloading in object-oriented programming languages lets programmers use the same name for different functions in the same scope. However, testers must be concerned that the objects can not receive improper messages. For example, if an object receives messages with wrongly overloaded functions, undetected ambiguous errors could occur. Furthermore, no syntax errors may occur, and even the resultant state of the object may be correct. Therefore such errors can not be detected using state-based testing.

The common testing approach is to test a class under test with “correct” test cases, and then to inspect whether the output is correct or not. If the output is incorrect (or unexpected), some errors could still exist in the class. Another method is to test a class with “incorrect” test cases, and if the output looks correct, we can say the class still has errors. Similarly, if an object accepts inappropriate overloaded functions and works, this also suggests errors. A GS is also a state-chart in which a generalized transition arc can represent several transition arcs but only if they have same behaviour. An EGS (refined from a GS) can be used to depict an object class which has overloaded functions. A state-transition tree, generated from the EGS, will
uncover the illegal transition paths and show the legal transition paths. Therefore legal and illegal test cases can be automatically produced by a test case generator which can examine the structures of the trees.

The problem of testing overloaded functions in a class is explained in section two. Section three has a brief discussion about state-based testing and state-charts. Using GS and EGS to generate legal and illegal test cases is proposed, in addition, a solution of the problem is suggested in section four. Finally, section five presents our conclusion.

2. Problem description

The data access operation of a stack is first-in, last-out and first-in, first-out for a queue. The abstract data types of stacks and queues are similar. Both can have similar data structures to store data. Some of their operations are the same, such as the storage is empty or not; how many data items are in the storage; visiting all data items in the storage; and printing the first data in the storage without deleting it. Some operations are also very similar, for example add data into, and delete data from a data storage area.

```cpp
#define SIZE 10;

class stack_queue {
protected:
    char S-Q[SIZE]; // unbounded array
    int k; // Top/Front index of stack/queue
    int j; // Rear index of queue
    int count; // a counter to number items in array
public:
    stack_queue( ); // default constructor
    int is_empty( ); // empty storage or not?
    int add(char, int); // push data to the top of the array
    int add(int, char); // add data to the tail of the array
    char del_data(); // remove data from front of the array
    int sizes( ); // the number of data in the array
    ~stack_queue( ) // destructor
};

void main( ) {
    stack_queue S, Q; // define S(a stack), Q(a queue)
    S.add( ?1);
    S.add( ?2);
    S.del_data( );
    S.add(99,'c');
    Q.add(10, ?);
    Q.add(99, ?);
    Q.del_data(1);
    S.del_data( );
    Q.add( ?122);
    Q.del_data( );
    :}
```

Figure 1 Stack-queue class contains overloaded functions

The ability of overloaded functions can significantly reduce the number of functions required. For example programmers may design a class which can be used to define both stack objects and queue objects, such as the example class in Figure 1. This has overloaded functions, whose function definitions and its complete executable C++ program code can be referenced in elsewhere [1]. To access the data in the object, the functions add(char,int), del_data() are defined for the stack, and the functions add(int,char), del_data(int) are defined for the queue. These functions are overloaded, and the remaining functions, is_empty(), sizes(), constructor, and destructor are shared by both objects.

In the main program, the statement stack_queue S, Q; defines an object S as a stack and an object Q as a queue. Therefore, sending messages to S via the member functions which should do the same work as the operations for stack. The object Q can only accept the messages via the functions which can work as same as the operations for queue as well. However, a programmer may make a mistake and send messages to the wrong objects in the client class. For example S.del_data(1);, Q.del_data();, S.add(99,'c'); and Q.add(’z’,122); shown at the main program in Figure 1.
These sending message statements are accepted by the C++ compiler (without syntax errors), and can be executed. Moreover, the state of the object S (or Q) can correctly change to the next state from its current state. In state-based testing, a test case for a class under test is:

1. to create a starting state for an object, such as an empty state of object S when it is defined;
2. to invoke the function under test, for example, object S accepts a message `add('a',1)` to call `add(char,int)` function in the stack_queue class; and
3. to validate the resultant state after executing the function, the state of object S is not full after the `add('a',1)` message executed, the value of `count` data member is one [2].

If testers just validate the resultant states after executing test cases in the state-based testing, they can not detect these kind of errors. For example, consider the following statements:

- `S.delete(1);`  //delete(int), to delete data from a queue object, but S is a stack object.
- `Q.delete();`  //delete(), to delete data from a stack object, but Q is a queue object.
- `S.add(99,'c');`  //add(int,char), to add data into a queue object, but S is a stack object.
- `Q.add('z',122);`  //add(char,int), to add data item into a stack object, but required by Q.

These can be accepted and executed without syntax errors. In addition, the states are correctly changed to the next expected states after execution, see table 1. The `del_data(int)` member function is used to delete data from a queue, but object S is a stack. When the statement, `S.del_data();`, is executed, the data at the bottom (not at the top) of the stack is deleted and the state of the object S is also changed, due to the data in the storage being decreased by one. The `Q.del_data();` statement, if executed, causes the rear data to be deleted rather the front data, and the state of the object Q is changed. The remaining statements, `S.add(99,'c')` and `Q.add('z',122)`, have the same errors whereby objects receive inappropriate messages. Therefore, testers can not find the errors by simply inspecting whether the resultant states are correctly changed or not.

If the overloaded functions have been well designed, the illegal messages should be rejected by the overloaded functions. Therefore, the conclusion here is that testers need to generate legal test cases as well as illegal test cases to test a class containing overloaded functions.

Some programmers would argue that it is better to declare the stack_queue class in Figure 1 as an inheritance structure, such as the class hierarchy shown in Figure 2.

<table>
<thead>
<tr>
<th>State change of object S in the case</th>
<th>State change of object Q in the case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current State</td>
<td>Executed Function</td>
</tr>
<tr>
<td>Empty</td>
<td>stack queue()</td>
</tr>
<tr>
<td>NotFull</td>
<td>add( ?1)</td>
</tr>
<tr>
<td>NotFull</td>
<td>add( ?2)</td>
</tr>
<tr>
<td>NotFull</td>
<td>delete()</td>
</tr>
<tr>
<td>Empty</td>
<td>add(99, ?)</td>
</tr>
</tbody>
</table>

Table 1 State change of object S and Q in Figure 1
In which the list class is a superclass and it is inherited by both stack and queue subclasses. Tsai et al [1] have tried to use EGS to discover the possible misuse of overloaded/polymorphism functions in superclasses and subclasses.

![Diagram of class inheritance](image)

**Figure 2** Stack and queue are subclasses of a list class

In an automatic testing environment, testers can test the class under test with test tools rather than examine the internal code of the class. Therefore, the test case generation here is based on functional testing, state-based testing method, and Extended General State-charts.

### 3. Background

#### 3.1 State-based testing

State-based testing is a form of implementation class testing. The particular state of an object results by combining values from all of the object’s data members at a particular point in time. The technique of state-based testing is to transform a class behaviour model into test case, which requires a behaviour model for the class under test [3], [4], [5]. States of an object are changed by the value of its attributes. If a class has no attributes, it can not be tested using state-based testing, as it has no states.

With state-based testing, Turner [5] listed an algorithm to generate test cases to test all of the member functions in the class under test at least once. Binder [3], [4] mentioned that the test cases generated from the transition tree of the class under test can find four kinds of “bugs”: missing transitions, incorrect transitions, incorrect output actions, and incorrect states which are not easily found by simply testing each method, transition, or state once.

The class testing approach of MACT [6] is to build up an inspection tree which copies the behaviour of the state machine of the object under test. From the tree, testers can inspect whether the state values, changed by executing transition(s) at the current state, cause the state to change to the next expected state or not. If the resultant state is not the expected state, then errors still exist in the class.

#### 3.2 State-charts

A state-chart is a graph whose nodes are states which are defined by the value of data attributes. Its directed arcs are transitions that are defined by its related member functions. The directed transition arcs are drawn from the receiving state to the target state. A state has an unique name that describes the status of an object at a particular point in time. A transition arc is also uniquely named to show a behaviour of the object. In addition, state-charts can specify the state sequence caused by a transition sequence in a modular fashion. After a transition, if the receiving state and the target state are the same state, then the transition is a loop transition [7], [8], [9]. The state-chart of the stack-queue class, in Figure 1, is shown in Figure 3.
The main point of state-based testing is to examine the values which have been stored in the object at a particular time. The changing states rely on the values which have been changed by the transition. Therefore the state-chart can be used as an aid in state-based testing to generate test cases [3], [4], [10], and [11].

4. The problem solved with state-based testing

The illegal messages, mentioned in section 2, quite often happen during system integration. Programmers misunderstand the functionality of overloaded functions or make mistakes by sending messages with inappropriate overloaded functions. This sort of errors could be found using structural testing. However, state-based testing is unlike structural testing, since testers find the errors by validating the resultant state after the test case(s) are executed. State-based testing method can detect these errors if a generator, based on state-base testing, can generate illegal test cases.

In fact, these kind of errors do not discover the weakness of state-based testing. Nevertheless, it shows that testers need to generate illegal test cases as well as legal test cases. But how should we use the state-based testing method to generate illegal test cases for testing? Binder [4] has used state/event matrix to find the illegal transitions. The overloaded (called twin) functions have the same (or similar) transitions, so the error in this case does not belong to the illegal transition. The error is caused by twins since they have the same transitions. In the following sections, we use a GS to generate legal test cases, and then demonstrate how an EGS can produce illegal test cases.

4.1 Test cases generated from GS and EGS

The two overloaded (twin) functions, add(char,int) and add(int,char) in Figure 1, have the same behaviour. Both functions increase the data size in the storage of a stack or queue. They could be loop transitions or non-loop transitions which change objects from NotFull state to Full state. The other pair of overloaded functions del_data() and del_data(int) have the same transition as well, see Figure 3.

This state-chart is complicated due to assigning each overloaded function an unique transition arc. Actually, transition arcs are only used to represent the change between states, and to describe the behaviour of an object. If overloaded (twins, triplets, or quadruplets,...) functions have the same behaviour in an object, they can share a transition arc in a state-chart. During generalizing, the same name and same behaviour but different signature functions (which are represented with different transition arcs in Figure 3) are combined into a transition arc, which is only labelled with a single function name without parameters (see Figure 4). In this case, the state-
chart can be simplified if a transition arc \textit{add} is shared by \textit{add(char,int)} and \textit{add(int,char)}; and another \textit{del_data} transition arc is shared by \textit{del_data()} and \textit{del_data(int)}. Obviously, the generalized state-chart is more clear and simpler than the normal state-chart, and the former can still describe the behaviour of stack objects and queue objects.

![Figure 4 The General Statechart of the stack_queue class](image)

A general state-chart is enclosed within an ellipse which can be imagined as a shell of an object, and represent the changing of states occurring inside of the object.

4.1.1 Test cases generated from GS

Both object S and object Q, in Figure 1, are defined from the same stack_queue class. S is a stack object if it operates with stack functions and Q is a queue object if it operates with queue functions. The GS works as a pattern state-chart which can be used to describe the behaviours of twin objects, such as S and Q in this case. Moreover, a pattern state-transition tree, in Figure 5, can be produced by following the GS. Each path from the root node to the leaf node represents a sequence of transitions from the beginning state to the end state of an object. By tracing each branch (or path) of the tree, testers can generate test cases for different objects by passing different object parameters into the test case generator.

![Figure 5 State-transition tree generated from GS](image)

![Figure 6 Test cases for Stack objects](image)

Each square node represents a state and each out-edge (also called a branch) represents a transition on the GS. The transition names in the GS are labelled at the top of state nodes. Each node represents a resulting state after executing a transition. This is repeated for each node until (1) the node already appears in upper level, or (2) the node is a final state of the GS. A full path, combined with a sequence of transition from the root node, is drawn to a leaf node which represents the final state after the sequential transitions being executed. Finally, each branch (or a sequence of branches) in the tree is a test case.
For example, after tracing the tree with object S and its member functions, the test cases for the object S are shown in Figure 6. The state value (post-condition) in bold square brackets shows the expected resultant state, when the test cases, prefixed to it, are executed. Repeatedly executing a member function is represented by “O.f(),...,O.f()”, and test case(s) in a pair of arrow brackets represent(s) the state is the same after executing the test case(s).

The automatic test case generator, consists (1) tree builder and (2) test case maker two parts in [6][12], can copy the structure of the state-transition tree to build up a threaded multi-way tree, such as in Figure 7. In which each node has a threaded pointer and a repeater item. Of course, nodes also need some room for function name, state name and pointers which link child nodes. The test case maker, traces the threaded multi-way tree in Figure 7, can generate test cases for the objects of the stack_queue class.

4.1.2 Test cases generated from EGS

In C++, messages are sent to objects by invoking their member functions. The behaviour of an object is coded into an algorithm known as a method. A method, in C++, is simply the body of a member function. When a message is received at the interface of an object, an appropriate method is invoked. For example, if a stack object S is declared, and an add message invoked to add a new data into the stack storage. This is represented by the C++ code:

```cpp
stack_queue S;
S.add('a',1);
```

When the object S receives the message `add('a',1)`, it matches this message to the `add(char,int)` member function (see Figure 1). Each message should have an appropriate and related member function to map to, if it has no syntax errors. Because, as discussed earlier, the state of an object changes from time to time, the behaviour of its methods can also be changed [13]. Therefore related objects are linked by messages (called a message packet) which contains member functions.
In Figure 8, the stack_queue class is linked with the client class by a message packet that can be imagined as a cable. Inside the cable there are several "wires" treated as methods that carry messages from a client object to the stack_queue object.

The GS of a class is a pattern illustrating the changing state of the class. The methods in the class’s message packet are declared as member functions in the server class. We can combine the two diagrams, the GS of the stack_queue class in Figure 4 and the methods of the class in Figure 8, to produce an extended general state-chart by completing the following steps [14]:

1) Unpacking the message packet, and attaching the member functions of the ellipse. The transition arcs, which exist in the state nodes of the enclosed GS, represent the transition of the states. The method arcs, unpacked from the message packet, point to the enclosed GS (see Figure 4). Finally, all of the overloaded functions are shown on the EGS in Figure 9.

2) Filling the state information onto related nodes, as discussed before, states consist of a set of data members and once the values of data members are changed, the object’s state changes simultaneously. Moreover, the state change is achieved by the post-condition of the executed transaction(s). This means that the post-condition corresponds to a particular resultant state. Therefore, the state information is left in the state node rather than suffixed with the name of transition arcs [3].

3) Prefixing method arc names with pre-condition, because the transition arcs in the GS are shared by several overloaded functions. The GS, encapsulated in an ellipse, is invisible to the outside world. Therefore we prefix the pre-conditions with the name of method arcs.

By following the general transition arcs and mapping them with method arcs. A state-transition tree will be produced as shown in Figure 10. For example, the Empty state, also the initial state in Figure 9, has no a previous state. Therefore, it becomes the root node of the state-transition tree. The add transition from the Empty state is mapped by method arcs add(char,int) and add(int,char) in the EGS. The corresponding arcs are drawn to the child, NotFull, nodes in the state-transition tree.
By tracing the tree, the illegal function calls by object S can be easily found. Those are marked with dotted lines. Consequently, the paths, from the root node to leaf nodes, contain dot out-edge line(s) are called illegal paths. This tree can be divided into two subtrees, one contains all the solid line paths and the other subtree has dotted illegal lines. The structures of both trees are as same as the state-transition tree discussed in section 4.1.1. Both subtrees can be copied to the threaded multi-way trees by the tree builder of the test case generator to produce legal and illegal test cases for the stack_queue class.

To check whether the overloaded functions in a server object have been well designed to reject services required by inappropriate client objects, testers can send all illegal test cases to the class under test. This approach is easier than getting the class to a particular state and sending each illegal test case [4]. The expected result has two parts: (1) The rejected message shows each illegal test case, then the class passes the test. (2) The state of the object is changed without rejected message, then errors exit in the class.

### 4.2 Problem-solving

How to avoid illegal messages sent to a class which contains overloaded functions? Illegal messages should be checked by the server class, before it offers its service. On the other hand, some counterparts argue that clients have responsibility to use servers correctly [4]. However, it is quite hard for clients to check what messages they can send out, especially with overloaded functions. For example, programmers could design a condition before sending message from client classes, but the overloaded functions (called twins, triplets, etc.) may be put under the same condition. It will reduce effort to set condition statement(s) in a server class instead of in each of clients. The overloaded functions `add(char,int)`, `add(int,char)`, `del_data()`, and `del_data(int)` could be modified with a condition which checks whether the calling objects are allowed or not, the example C++ code shown in [1].

### 5. Conclusion

The EGS can be used as a testing model to test complicated C++ programs that have been coded with a variety of overloaded functions. The legal test cases and...
illegal test cases can be automatically generated by the test case generator. Therefore
the errors that overloaded functions are required by wrong objects can be detected
without using structure (white-box) testing method.

The EGS can be used as a normal state-chart if the class under test has not
overloaded functions. In this case, the name of each method arc outside of the GS is
unique. Therefore EGS can be used to model classes, whether they have overloaded
functions or not.

Polymorphism occurs once a subclass inherits the interface of a superclass, and
the subclass could have overloaded functions. The EGS can be used to depict the
changing states of the superclass as well as the subclass[1]. In addition, testers can
also use the EGS of the subclass to generate legal and illegal test cases to test the
subclass before it is pieced together with superclasses or other related classes.

6. References

[1] Tsai, Bor-yuan; Stobart, Simon; Parrington, Norman “Generating Test Case for
Overloaded/Polymorphism Functions in Object-Oriented Programs”, Occasional
Report No.: TR 1/93, (SECS), University of Durham.
75-78.
Conference on Software Maintenance 1993, pp. 302-310.
[6] Tsai, Bor-Yuan; Stobart, Simon; Parrington, Norman “A Method for Automatic
Class Testing (MACT) Object-Oriented Programs --- Using a State-based Testing
(Addison-Wesley 1992) revised fourth printing.
[9] Turner, C. D.; Robson D. J. “A State-Based Approach to the Testing of Class-
[10] Gao, Jerry Z.; Kung, David; Hsia Pei. et al “Object State Testing For Object-
Oriented Programs”, International Computer Software & Application Conference,
pp. 472-482.
[12] Tsai, Bor-Yuan; Stobart, Simon; Parrington, Norman “Test case generator using
state-based testing”, OPS: CIS-1-98, School of CIS, University of Sunderland, 1998.
[14] Tsai, Bor-Yuan; Stobart, Simon; Parrington, Norman “Using Extended General
Statecharts in Object-Oriented Program Testing: A Case Study”, 24th Technology of
Object-Oriented Languages and Systems, TOOLS Asia ‘97, Sept. 1997, Beijing,
China, pp. 129-134.