Automated class testing using threaded multi-way trees to represent the behaviour of state machines

Bor-Yuan Tsai [a,b], Simon Stobart [a], Norman Parrington [a] and Ian Mitchell [a]

[a] *School of Computing, Engineering and Technology, University of Sunderland, St. Peter's Way, Sunderland SR6 0DD, UK*
E-mail: bor-yuan.tsai@sunderland.ac.uk
[b] *Department of Information Management, Aletheia University, Taipei, Taiwan*
E-mail: david@jupiter.touc.edu.tw

Extensive test data is required to demonstrate that "few" errors exist in software. If the process of software testing could be carried out automatically, testing efficiency would increase and the cost of software development would be significantly reduced. In this paper, a tool for detecting errors in object oriented classes is proposed. The approach uses a state-based testing method. The method utilises state machines in order to produce threaded multi-way trees, which are referred to as inspection trees. Inspection trees can be used to generate test cases and parse test results files. This allows us to determine whether the classes under test contain errors. The algorithms for the creation of inspection trees and the examination of the test results file using an inspection tree are described in the paper.

1. Introduction

Software testing is a considerable expense in the development of software systems, with estimates ranging between 20% and 50% of development costs [Glass 1990; Myers 1979; Norman 1993]. It is a very expensive, time consuming and tedious activity to demonstrate that "few" errors exist and requires a large amount of test data to accomplish successfully. If the process of software testing could be carried out automatically, then testing efficiency would increase and the cost of software development can be reduced.

The method of automated class testing which is introduced in this paper is a part of a larger software testing development project. In brief, the method duplicates the behaviour of a state machine into a threaded multi-way tree in order to generate test cases. The threaded multi-way tree (also called an inspection tree), the process of test result inspection and inspection tree generation are described.

Class testing, state-based testing and state machines are first explained in section 2 using a bounded queue example. In section 3, the structure of the threaded multi-way tree is demonstrated and a graph used to illustrate the relationship of the threaded multi-way tree to a state machine. An algorithm for creating the tree is also proposed. The detection of errors using the tree is discussed in section 4. A demonstration of using inspection trees for complicated state machines, such as concurrency, hierarchy and nesting, is given in section 5. A graph showing an approach which can automatically generate inspection trees for various state machines is included in section 6. Finally, section 7 contains our conclusions.

2. Class testing

During class testing, testers analyse and test the interactions of functions within the classes. There are several methods which can be used, such as structure testing, specification testing, state-based testing and/or data flow testing.

The method described within this paper is state-based testing, which may be performed as either specification testing or structural testing [Jacobson *et al.* 1992]. The following subsections give some basic concepts of state-based testing and state machines. In addition, a template and a state machine of a bounded queue are illustrated to explain automated class testing.

2.1. State-based testing and state machines

State-based testing is a form of implementation class testing. The main point of state-based testing is to examine the values which have been stored in an 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. State machines can be used to illustrate the relationships between transitions and states. Moreover, test cases, in state-based testing, can be generated from state machines [Binder 1995; Tsai *et al.* 1997a; Turner and Robson 1995]. Therefore the state machine is an aid to performing state-based testing.

State machines specify the state sequence caused by a transition sequence in a modular fashion. A state machine is a graph whose nodes are states and whose directed arcs are transitions. The directed transition arcs are drawn from the receiving state to the target state. A state has a unique name and denotes the status of an object at a particular point in time. A transition arc is also uniquely named to show the behaviour of the object and it also depicts the current state changing to the next state. Figure 2 illustrates a state machine of a bounded queue class in figure 1. A statechart, proposed by Harel [1987], is an extended form of a state machine. Hierarchy, concurrency, and broadcast communication are the three basic extensions to state machines.

The communication between integrated classes can be tested using integration testing, but this will not be discussed in this paper. To aid understanding, the example, presented in sections 3 and 4, consists of simple state nodes and transitions. Inspection trees can also be used to represent complicated scenarios, such as hierarchical, nested, or concurrency. These will be discussed in section 5.

2.2. A queue class example

This example illustrates how a threaded multi-way tree can be derived from a state machine. In further subsections, a bounded queue class is used, the C++ template of which is given in figure 1. Suppose that the queue class, called *Queue*, has limited spaces in which the *Queue* object can only store five units of data. When a *Queue* object is created, it is in an *empty* state. Once some data is stored in the object, then its state is changed from *empty* to *not full*. As the *Queue* object has been filled with five items of data, its state is *full*. A *Queue* object thus has *empty*, *not full* and *full* states, which are represented by three nodes in the *Queue* class state machine shown in figure 2.

Empty state could be a member of *NotFull* state set, but we assume the *NotFull* state set does not include the *Empty* state, so the *NotFull* state is a kind of *PartlyFull* state in this paper. Moreover, if a queue has only space to store a single data item, the queue will change from *Empty* state to *Full* state when the *addrear* transition is executed. In this paper, we assume that the state of the queue object cannot be directly changed from *Empty* to *Full*. If the assumptions are excluded here, then the queue has only *empty* and *nonempty* states.

In figure 1, *qlist[]* is declared as an array for the *Queue* to store data. The *f_index* and *r_index* are two array indices, used to indicate the position from/in which the next data will be deleted/added. Moreover, the *count* data member is used to calculate the amount of data in a *Queue* object. For example, a *Queue* object is an empty queue when *count* is zero, and it is full once *count* is equal to *Size*. The three condition expressions, $count == 0$ (the object is at empty status), $0 < count < Size$ (the object is at not full status), and $count == Size$ (the object is at *full* status), are also known as the state information of the *Queue* object. The *Queue* class state machine shows the state of a *Queue* object is no longer at *empty* and *full* states once the post-condition, after executing the *addrear/deletefront* function, is $0 < count < Size$.

```
#include <iostream.h>
#include <stdlib.h>

const int Size = 5;

class Queue {
protected:
   char   qlist[Size];          // bounded queue
   int    f_index;              // Front index
   int    r_index;              // Rear index
   int    count;                // cumulator of the queue
public:
   queue( );                    // default constructor
   int    is_empty( );          // queue is empty or not
   int    addrear(int c,
             char new_data);// add new data into the queue
   char   deletefront(int);     // remove from the Queue
   ~stack_queue( );             // destructor
};
```

Figure 1. The template of a bounded queue.

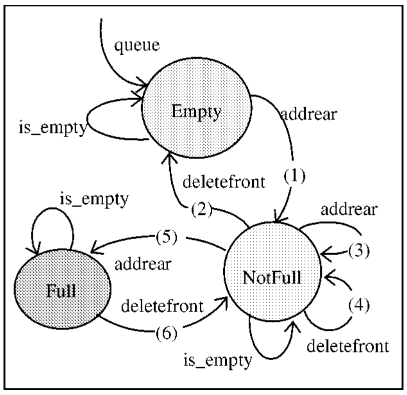

Figure 2. The statechart of the bounded queue.

To explain the template in figure 1, there are three member functions and a constructor and a destructor declared. The functions *addrear()* and *deletefront()* are discussed as an example in the following sections. When a *Queue* object is declared, the constructor *queue()* causes the object to be in the *empty* (initial) state, and the three data members *f_index*, *r_index* and *count* are initialised to zero.

When an *empty* queue object receives an "add new data" message, the *addrear* member function will be executed. After executing the function, the state of this object is changed to *NotFull* state from *Empty* state, see *addrear* arc labelled "1" in figure 2. The value of the *count* is increased to one, which is greater than zero and less than *Size*. If the next message is to delete the only data in the object, then the state of the object is at an *Empty* state again, as the *count* is once again zero. Furthermore, if the queue object is in a *NotFull* state, when it receives another message to add a new data item, the value of *count* is increased. The state of this object is still unchanged at *NotFull*. This transaction is depicted as a loop transition arc on the state machine. For instance, the transition arc, labelled "3" in figure 2, is a loop transition arc which comes out from and then goes back to the same *NotFull* state. That means the transition does not cause the state of object to change to another different state. Once the queue object is *full*, then its state is changed to *Full* state, into which no more data can be added. During this state, the queue object cannot receive any "add new data" messages. Similarly no data can be deleted when an object is in an empty state.

2.3. State machines in class testing

To design and test an application, which is still under development, sounds a hard task to achieve. This is because no matter how good the method used to test an application during development is, testers will still need to execute the application when it has been finished, with further test data again to prove no errors have been introduced at the final stages. However, during development testers can undertake some test work. This may include the design of test methods, test drivers, test stubs and generating oracles, test cases, test data, and expected results. These can be used to test the application once it has been fully developed. This means, of course, that the end of the design process is the beginning of test execution but not necessarily the beginning of test activity.

This testing approach should begin by following the development of the specification, which is also used as a reference document to develop the application, but not the program code. Therefore, a well defined specification is fundamentally important for designing and testing applications [Tsai *et al.* 1997b].

Our specification is represented as a state machine, which illustrates the association between object states and object transitions. In addition, the state machine has many advantages, including test case generation. The transition arc flow and state information aspects of a state machine can be applied to detect errors in a program. Moreover, the transition arcs in a state machine show the various behaviours which the object has at any particular time. The result of a transition having been executed, known as a post-condition, can be represented by a state node in the state machine. Based on this, a threaded multi-way tree can be developed to duplicate the behaviour of a state machine. The nodes of the tree contain the expected results of all the transitions of an object. A test results file can be parsed to determine whether the class under test has errors or not.

3. Threaded multi-way trees and state machines

A single node of the threaded multi-way tree works the same way as a state node of the state machine. Each tree node contains an incoming-transition field, a thread field, pointer fields and a state name (information) field. Those fields are discussed in detail in the following subsections.

3.1. *The nodes of the tree*

Each node of the threaded multi-way tree, which is also called an inspection tree, consists of four fields shown as the following fragment of C++ code.

```
typedef struct node {
  struct node *pre_state          //threaded to previous state
  char funct_name[15];            //incoming trans arc name
  char state_name[15];            //state information
  struct node *next_state[5];     //point to next states
}Treenode;
```

The structure of the node is shown in figure 3.

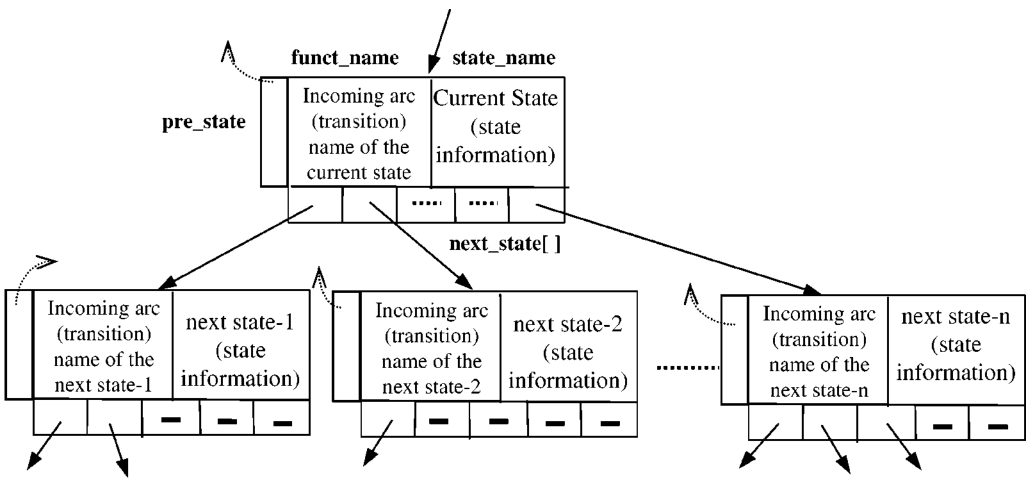
funct_name
state_name
pre_state
Incoming arc (transition) name of the current state
Current State (state information)
next_state[ ]
Incoming arc (transition) name of the next state-1
next state-1 (state information)
Incoming arc (transition) name of the next state-2
next state-2 (state information)
Incoming arc (transition) name of the next state-n
next state-n (state information)

Figure 3. The structure of a threaded multi-way tree is presented as a state machine.

The field named *funct_name* is used to store the function name that emerges from the previous state and, of course, the name is the same as the function name of the message. The *state_name* field contains the state information (post condition). The *state_name* field can be replaced by the variable items that store the value of the attributes, such as the *count* data member in the *Queue* class template example. If the state information is $0 < count < Size$, which is also the result of a function, then the object is in a *NotFull* state. In practice, testers can use this sort of expression instead of the real state name which is shown in the state machine. More importantly, the node in an inspection tree should be designed to detect the test results file, so that the node of the tree should be declared in order to be able to read through the test results file correctly.

The *pre_state* threaded field stores the previous node address, if a transition function is executed and the state is still the same, then the *pre_state* field points back to the previous node which contains the same state information as the current node does. The *next_state[]* field is a pointer array (or a linked list) in which each element of the array holds the address of a child node representing the next state node in the state machine. Of course, the size of the array is determined by the most number of next state nodes of the states in a state machine. In the state machine of the *Queue* class, the *Empty* state has two next states which are *NotFull* and itself. The *NotFull* state has five next states in which three of them are *NotFull* itself. The *Full* state has two next states *NotFull* and itself (see figure 2). The number of child nodes in this tree, duplicating the behaviour of the *Queue* state machine, is five. Therefore, the array is declared with five elements.

Figure 2 shows that the *NotFull* state has five transition arcs leaving it, but three of them loop back *NotFull* state itself. Therefore the next state of the *NotFull* state might be *Empty*, *Full* or *NotFull*. The next state of the *Full* state is either *NotFull* state or *Full*. Furthermore, the next state of the *Empty* state is either *NotFull* or *Empty*. Consequently, the state machine in figure 2 can be illustrated by a threaded multi-way tree shows in figure 4. Each node of the tree represents the corresponding node in the state machine.

3.2. *The expected results in the tree*

The reason why the inspection tree can be used as a tool, automatically inspecting errors in a class, is that the nodes of the tree contain the expected results for the executed functions in the class. After feeding a test data file to the class program, a test results file can be produced. Consequently, the test results file can be sent record by record to the inspection tree to match against the expected results. If they match, then the class under test passes the inspection, otherwise the class fails. The expected results are stored in the *state_name* field of the tree node. The actual inspection approach is described with examples in section 4.

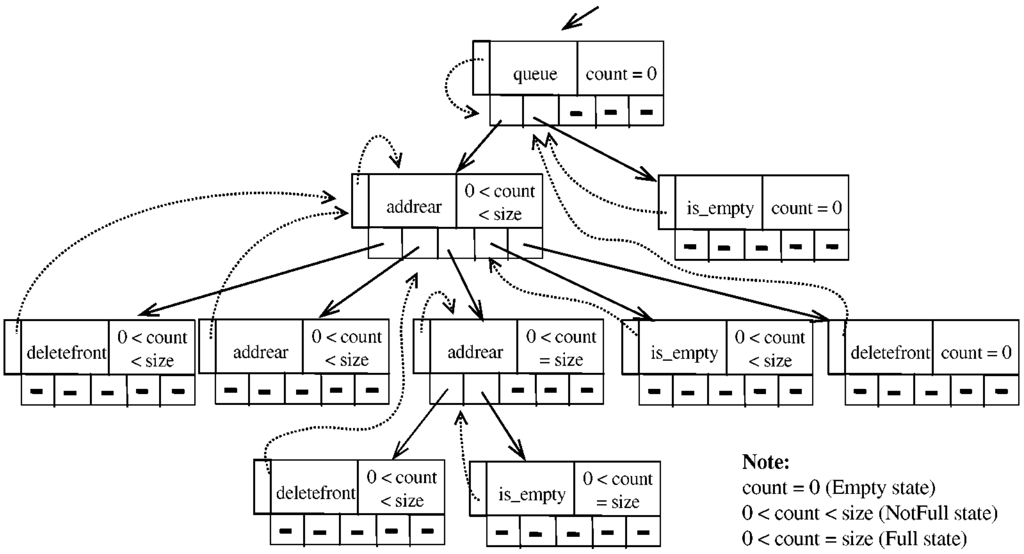

Figure 4. The inspection tree of the *Queue* class.

```
// Need a pointer queue to build up a tree level by level
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
Step 6        While the new node is not a child of the first node in the 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_state of the new node threads to the node in the queue
Step 11       Add the new node into the queue
```

Figure 5. The algorithm to build a threaded multi-way tree.

3.3. *The algorithm to create the tree*

The threaded multi-way inspection tree is created level by level using a queue. When a non-leaf node at level N is allocated, it will be added into the queue. It will be removed from the queue once all its children nodes at level $N + 1$ in the tree have been allocated and linked with it. This tree will grow level by level until all the nodes (leaves) at the lowest level are linked to the tree. The algorithm, which can create an inspection tree, is shown in figure 5. A complete C++ program, which follows the algorithm, can be found in [Tsai *et al.* 1998a].

3.4. The mapping between inspection tree and state machine

In this section, the mapping between the inspection tree and the state machine is presented. More specifically, it shows that the inspection tree can represent the state machine completely. For example, the state machine, in figure 2, of the *Queue* class is mapped to the inspection tree in figure 4; this is shown in figure 6.

When a function is executed successfully, either the state changes from the current state to next state, or does not change. The emerging function arcs of a state are stored in the children's nodes. For example, the outgoing arc, named *addrear*, of the *Empty* state is stored in its *NotFull* children nodes and the other outgoing arc, named *is_empty*, is stored in the other child node *Empty* (see figure 4). If a child node's state information is the same as that of the ancestor's, this means the state does not change after executing the function. Therefore, the *pre_state* field can be used to route the current pointer back to the ancestor node. An example of the *pre_state* field is the relationship of the root, *Empty,* node and its rightmost child node. That shows the *Empty* state of the *Queue* object is still unchanged after an *is_empty* transition having been executed. Another example is the *pre_state* field of the rightmost grandchild of the root, *Empty,* node. Here the *Queue* object loops back to the empty state again, see figures 4 and 6, because its state information has changed from $count == 0$ to $0 < count < Size$, and to $count == 0$. This shows that the state of the object has changed from *Empty* to *NotFull* and to *Empty* at last, after three functions *queue(), addrear(),* and *deletefront()* having been executed in sequence.

For the sake of clarity, numbers which have been marked on the arcs of the state machine in figure 2 are also marked in figure 6, and the threads in the *pre_state* fields shown in figure 4 are omitted in figure 6. In figure 6, the relationship between the *Queue* state machine and the threaded multi-way tree of the *Queue* can be easily seen. Thus, a threaded multi-way tree can duplicate the behaviour of a state machine.

4. Inspecting test results with the inspection tree

Testers can manually inspect the test results file which has automatically generated after executing the class under test against a test data file. This manual inspection is slow, boring, and time-consuming. However, the inspection tree can be treated as a test tool to enable us to read through the test results file and detect errors within the file automatically. Before describing this approach, the structure of the test data file and the test results file are introduced.

A test data file consists of messages. Each message has a function name and parameters. The parameters are the test data (see figure 7). The class under test receives the messages one by one from the test data file, and then executes them. The executing result of each message is stored in a file which is called the test results file. In the file, each record corresponds to each test message. That means each record has the executing function name as well as the actual test results. For example, after executing the first message *addrear(a)*, the results in an object are $count == 1$, data a

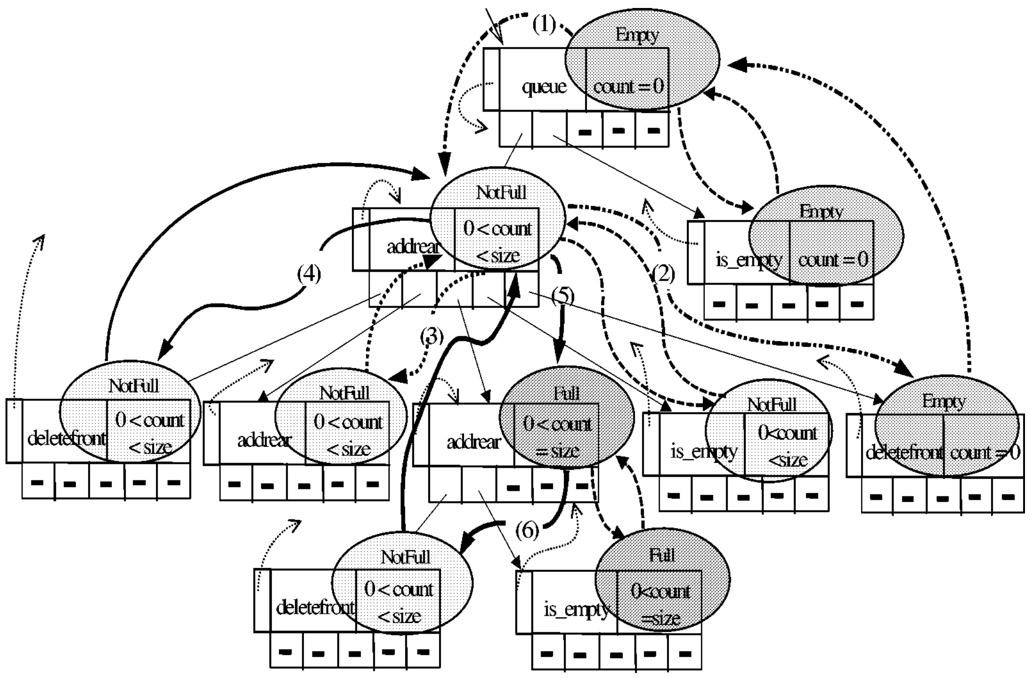

Figure 6. The inspection tree of the *Queue* class.

Test data file-1 (**Over flow testing**)	Test data file-2 (**Underflow testing**)
addrear(a)	addrear(a)
addrear(b)	addrear(b)
addrear(c)	deletefront(1)
deletefront(1)	addrear(c)
addrear(d)	deletefront(1)
deletefront(1)	deletefront(1)
addrear(e)	addrear(d)
addrear(f)	deletefront(1)
addrear(g)	deletefront(1)
addrear(h)	addrear(e)
deletefront(1)	deletefront(1)
deletefront(1)	addrear(f)
:	:
:	:

Figure 7. Testing data files.

Test result via test data file-1		Test result via test data file-2	
Function name	Value of Count	Function name	Value of Count
addrear	1	addrear	1
addrear	2	addrear	2
addrear	3	deletefront	1
deletefront	2	addrear	2
addrear	3	deletefront	1
deletefront	2	deletefront	0
addrear	3	addrear	1
addrear	4	deletefront	0
addrear	5	deletefront	-1
addrear	6	addrear	0
deletefront	5	deletefront	-1
:	:	:	:
:	:	:	:

Figure 8. Test results files.

is stored in, and *r_index* is increased by one. However, the most important item is the value of the data member *count*, which causes the state to change from *Empty* to *NotFull*. In other words, the state information (*count* = 1) shows the next state has been reached after executing the message. The significant executing results are stored in the test results file (see figure 8).

Following the state machine (specification), each node of the inspection tree is stored with a function name and an expected result. The inspection tree is used as an oracle to detect the test results file. A driver program, such as the algorithm in figure 9, is used to read the test results file record by record and sequentially send each record to the inspection tree to find out whether the test result is expected or not. If the test result is not as expected then an error has occurred.

Another advantage of inspection trees is that they can detect *missing-path* errors. This type of error occurs if a partial function (or path) is specified but not implemented in the program. To discover this error, an extra Boolean field (called *visited*) can be added to each node of the inspection tree, and is set to true when it has been visited during processing of the test results file. Of course, the test data file should be well designed to make sure that each function of the class under test is executed at least once. The algorithm in [Tsai *et al.* 1998b] can be adopted to generate the test data files.

4.1. The method for inspecting the test results file

Figure 9 illustrates the method used to process inspection trees. The following describes the use of this method to process the inspection tree in figure 6.

```
STEP 0  set the root of inspection tree to current
STEP 1  open test_result file
STEP 2  read the test_result record and store data to funct and state two variables
STEP 3  while not end of the test_result file
STEP 4      if all children nodes of the current node have been traced go to STEP 6
STEP 5        if a child node's funct_name = funct and the condition in state_name is true
                then the test_result record is correct;
                     assign the child node's pre-state to current and
                     go STEP 7
                else go to STEP 4
STEP 6      print "this test result record is error"
STEP 7      read the test_result record and store data to funct and state two variables
STEP 8  endwhile
STEP 9  close file and stop
```

Figure 9. The algorithm of detecting test results with inspection tree.

Supposing, in *STEP 0*, that the current node is the root node, which is at an *Empty* status. In *STEP 2* a test result record is read from the test result file, and the recorded values (see test result file-1 in figure 8) are *addrear* and 1 (the value of *count*). This means a message, like *addrear(a)* in the test data file-1 of figure 7, has been passed to this object. After executing the message, the relation of the *count* and *Size* attributes is $0 < count < Size$, as $Size == 5$ and $count == 1$. In *STEP 4* a search for a function name is made from the leftmost child node of the current node to the right. The value of *funct_name* field in the first child node is *addrear*, which is the same as the data, *addrear*, of the test result record. Furthermore, the $count == 1$ satisfies the state information, $0 < count < size$, in the child node. This matches the *if* condition in *STEP 5* and the current pointer stays at the, *NotFull*, node according to the value of *pre_state* field in the node. Consequently, jumping to *STEP 7*, the next test result record will be read, and can be tested by following the steps until all records in the file have been tested.

4.2. Example programs

The *addrear* function in figure 10 is an operation to insert an item into a *Queue* object. An assumption is that an item is placed at the rear of list storage *qlist[]*. In this case, it is necessary to first test if the queue object is full and to terminate the operation if this is true. Otherwise, a new item is inserted at the rear of the list whose location is designed by *qlist[r_index]*. If the function misses the full condition, then an overflow error occurs. The *addrear* function body, shown in figure 10, does not contain a full condition statement on purpose. The inspection tree will find an overflow error when the test result file is processed with the tree.

```
// Add a new data into the queue
int Queue :: addrear(int rear, int & c, char new_data) {

      qlist[rear] = new_data;
      c++;
      rear++;
      if (rear == Size) {
          rear = rear % Size;
      }
      return(rear);
}
```

```
// Delete a data from the queue
char Queue :: deletefront(int &c, int &front) {
      char deleted_data;

      deleted_data = qlist[front];
      c--;
      front++;
      if (front == Size) {
          front = front % Size;
      }
      return(deleted_data);
}
```

Figure 10. The *addrear* and *deletefront* functions of the *Queue* class.

Error found in the test result file-1			Error found in the test result file-2		
Function name	Value of Count	Ok/Error	Function name	Value of Count	Ok/Error
addrear	1	Ok	addrear	1	Ok
addrear	2	Ok	addrear	2	Ok
addrear	3	Ok	deletefront	1	Ok
deletefront	2	Ok	addrear	2	Ok
addrear	3	Ok	deletefront	1	Ok
deletefront	2	Ok	deletefront	0	Ok
addrear	3	Ok	addrear	1	Ok
addrear	4	Ok	deletefront	0	Ok
addrear	5	Ok	deletefront	-1	Error
addrear	6	Error	addrear	0	Error
deletefront	5	Error	deletefront	-1	Error
:	:	:	:	:	:
.	.	.	.	.	.

Figure 11. Error found in the test result files by the inspection tree.

The other function in figure 10, *deletefront*, is an operation to delete a data item from a *Queue* object. Before deleting data from the queue object, it must be determined whether the queue object is empty and terminate the operation if the condition is true. In addition an underflow error could happen if the condition statement is missed in this function. However, the *deletefront* function body in figure 10 is defined without an underflow condition statement. If the test data file-2 in figure 7 is processed against the *deletefront* function, the test result file-2, in figure 8, will contain underflow errors.

4.3. *Test report*

After inspecting the test result file-1 and test result file-2 with the method described in figure 9, two test reports are generated with errors (see figure 11). The two test reports show that the *Queue* class whose *addrear* and *deletefront* functions have missing overflow and underflow conditions is highlighted as containing errors.

Following the test report, testers can correct the errors and recode the functions. They can also use the same inspection tree and driver program to re-inspect the corrected class. Thus, using the inspection tree for regression testing.

5. Complicated state machines

The example discussed in the previous sections is a simple state machine. Moreover, classes might be described with more complicated state machines. In this section, inspection trees are used to represent hierarchy, concurrent, and nested state machines. The design state machine of a class could be different from its implement state machine. That means a transition in the design state machine could be replaced with several member functions in the class program code. The inspection tree is also able to represent the implement state machine; this will be discussed in section 5.4.

5.1. Hierarchy state machine

The concept of hierarchy is presented in the form of substates. One state may be split into several substates. Moreover, only one of these independent substate represents the state of the system at any one time. Consider a simple queue called *Que*, whose state machine is in figure 13-a and its inspection tree in figure 12. *Que's NotEmpty* state can be divided into *NotFull* and *Full* substate, but only one of these can represent their superstate at a time, see figure 13-c. In this case, the former class is a superclass of the later class in which the subclass modifies the inherited *addrear* and *deletefront* transitions. In practice, the *addrear()* and *deletefront()* methods in the superclass are inherited by subclass (called bounded queue) with little changing.

McGregor and Dyer [1993] have demonstrated the mapping of inheritance to a state machine for a class from the state machine of its superclass. The inspection tree for a class can also be inherited by its subclass(es). The bounded queue, for example, only modifies the right subtree of the root in figure 12, and the inspection tree of the subclass (bounded queue) is the same as the tree in figure 4.

Another similar state diagram called a nested state machine will be discussed in section 5.3.

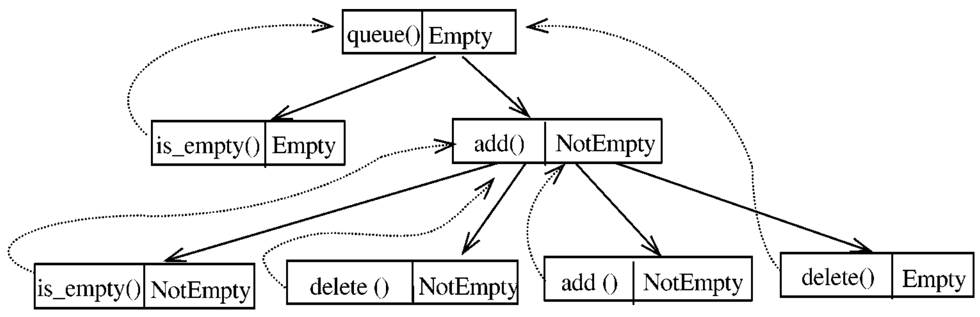

Figure 12. The inspection tree of *Que* class.

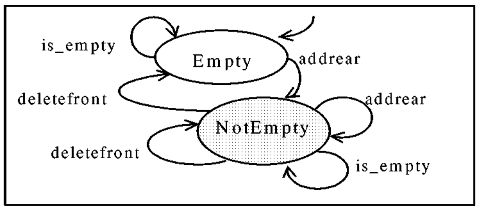

Figure 13-a. The statechart of a simple queue called Que.

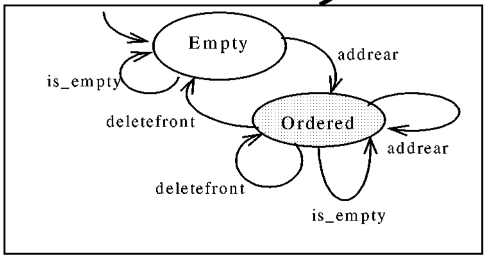

Figure 13-b. The statchart of a priority queue.

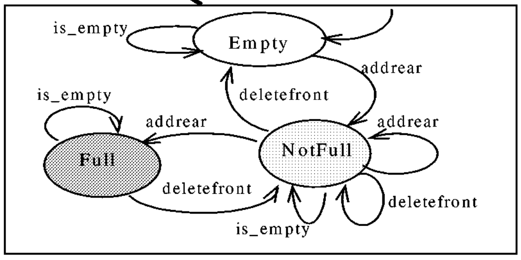

Figure 13-c. The statechart of a bounded queue.

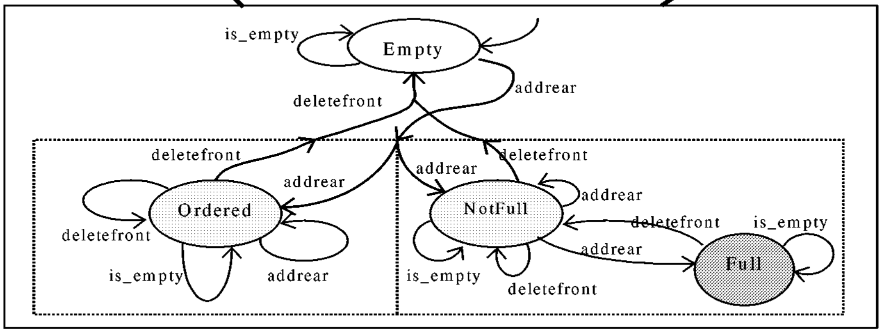

Figure 13-d. The Statechart of a bounded priority queue.

Figure 13. State machines using single inheritance and multiple inheritance.

5.2. Concurrency state machine

Multiple inheritance is the use of multiple superclasses inherited as the basis for a new class. As discussed in the previous subsection, the state machine of superclass can also be inherited by its subclasses. Assuming that the inherited superclasses represent independent concepts, then the state machines of the superclasses do not have any overlap to show the state machine of the subclass [McGregor and Dyer 1993]. In figure 13, the state machines of the bounded queue and the priority queue are drawn from the state machine of the simple queue, *Que*, in figure 13-a. The state machine of the bounded priority queue, figure 13-d, is constructed by inheriting from the state machines of the priority queue and the bounded queue (in figures 13-b and 13-c). Concurrent states do not always occur in multiple inheritance, however, in this bounded priority queue example, the *NotFull* state and *Ordered* state concurrently exist, as well as the *Full* state and *Ordered* state.

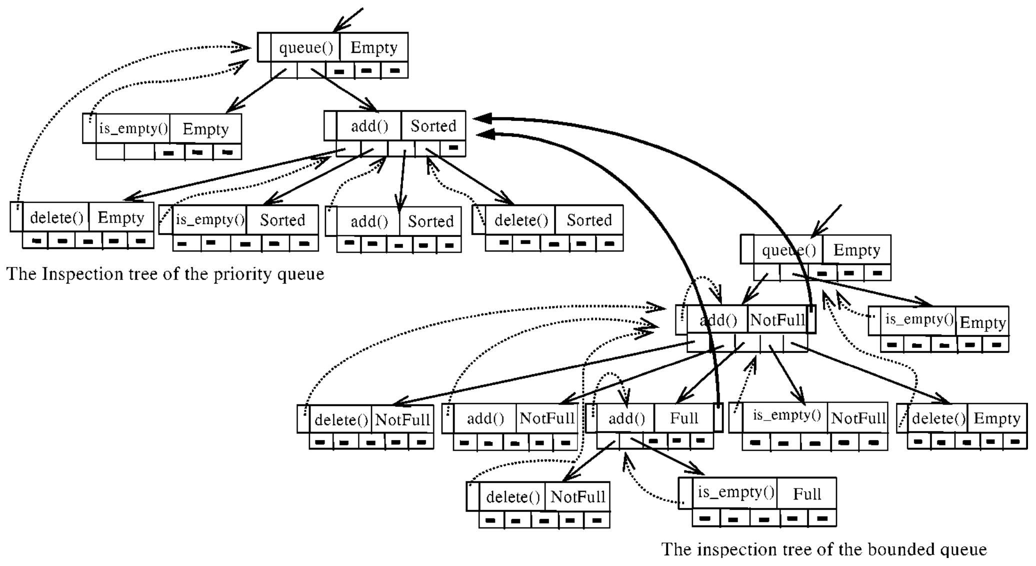

Figure 14-a. The inspection tree of the priority and bounded queue.

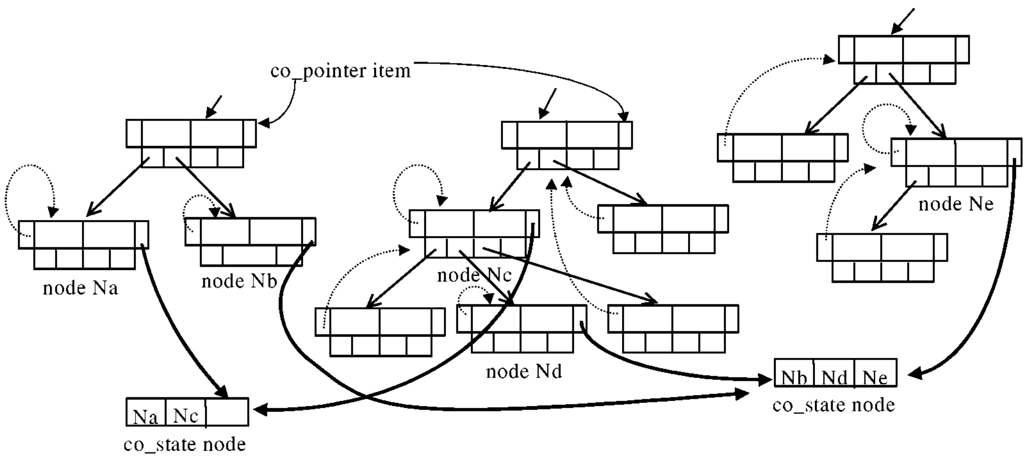

Figure 14-b. The *co_state* nodes in a concurrent inspection tree.

The inspection tree of the bounded priority queue, which shows how an inspection tree might be used to represent concurrent states in a state machine, is given in figure 14-a. Here, an additional *co_pointer* field is needed in the tree nodes. Both the inspection trees of the superclasses (priority class and bounded class) are reused in this state machine. The method to link the existing inspection trees into a concurrent inspection tree is simply to find concurrent nodes and then store the address of the nodes into the *co_pointer* field of the related nodes.

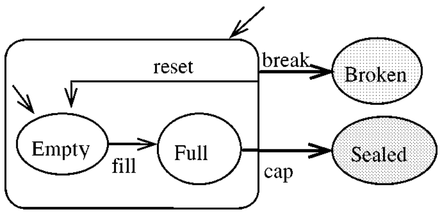

Figure 15-a. Nested bottle filling state machine.

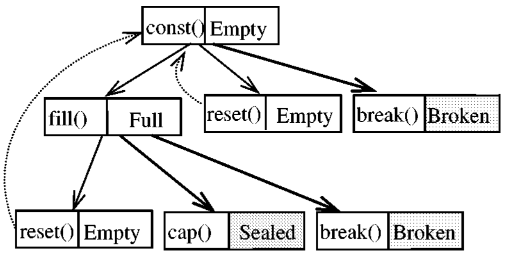

Figure 15-b. The inspection tree of the nested state machine.

The test result file of the bounded priority class program is processed with the inspection tree in figure 14-a. Assuming that three data have been added into the object of the priority bounded queue, and the current visited node is at the *NotFull* node. The *co_pointer* of this *NotFull* node points to the *Sorted* node at the left up tree in figure 14-a. Under the state information stored in the *state_name* field of the node, the queue object will be checked whether it is sorted or not. If it is not sorted then errors existed.

Another sort of inspection tree, see figure 14-b, needs to create a different tree node, called *co_states* node, in which is contained the node address of all the concurrent state nodes. This is pointed to by each of all the concurrent nodes in the linked trees. When the current visited node is whose *co_pointer* field is not null, then the *co_state* node will be visited next. From the *co_state* node, all the concurrent state nodes among the trees will be visited to check whether the object satisfies all the concurrent states.

5.3. *Nested state machine*

In the reuse of state machine, designers can produce a nested state machine for a new state machine. The nested state machine, in figure 15-a derived from [Cook and Daniels 1994], describes the behaviour of filling a bottle. If the bottle is broken, the bottle is in a broken state, no matter whether it is empty or full. The reset transition causes the bottle to be an empty state again, although it has already been sealed. A bottle is at sealed state if it is filled and capped. An inspection tree can also be used to represent the whole behaviour of the nested state machine, and its simplified inspection tree is shown in figure 15-b. In fact, the nested state machine can be redrawn as a normal state machine, if the outside/inside transitions on the square boundary line are mapped to each state which is at the inside/outside of the boundary.

5.4. *Methods comprised in a transition*

The state machines of classes, a sort of module specification, are the output of the detail design level in a software development life cycle [Tsai *et al.* 1997b]. A single transition in a design state machine, from the designers' view, could be finished by several sequential (methods) functions. From the discussion in previous sections it was shown that the nodes of inspection trees contain member function names rather than

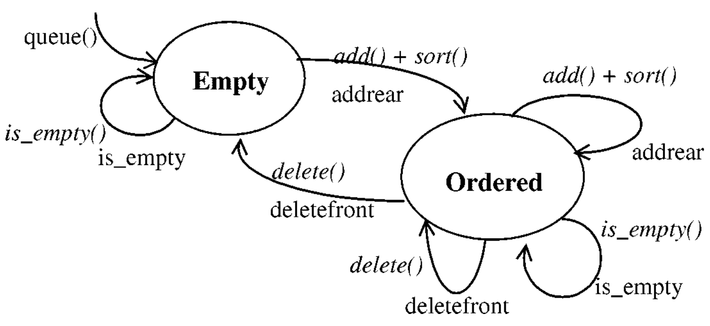

Figure 16. A transition comprises more than one methods in the program code.

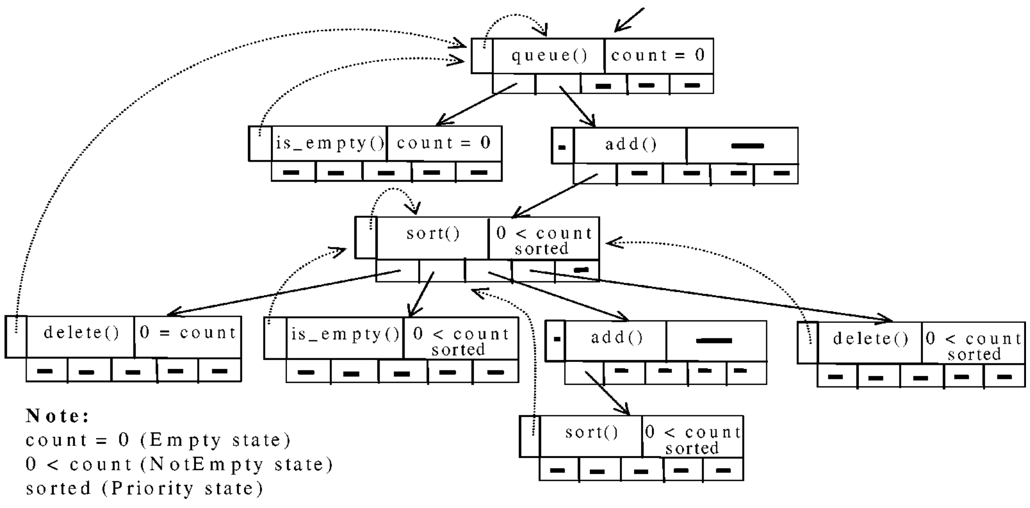

Figure 17. The inspection tree of a priority queue class in figure 16.

transition name to parse the test result. For example, the state machine of a priority class, shown in figure 16, contains one *addrear* transition which could be replaced with the *add()* and *sort()* member functions. The state of the class will change to the ordered queue state after the *addrear* transition (both *add()* and *sort()* functions) is executed. Therefore, these two functions should be stored in the inspection tree, and an *add()* followed *sort()* test case is required to test this class.

Each node of the inspection tree in figure 4 contains a function name, a state information and pointers, but in this example, there is no state information that can be stored with the *add()* function in a node. That means the node cannot represent the *Ordered* state node in the state machine, see figure 16. Therefore the *state_name* field in the node will be filled with an empty string, see figure 17. The algorithm in figure 5 can still be used to build this sort of inspection trees. The inspection algorithm, given in figure 9, can also be used with an additional condition statement.

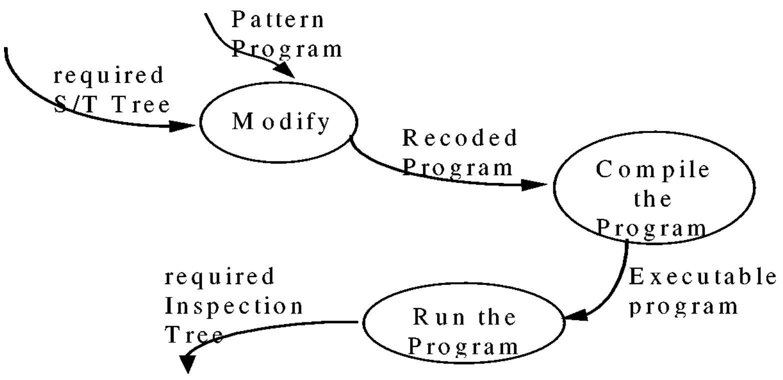

Figure 18. The process of creating an inspection tree.

6. The inspection tree generator

An object-oriented application can be aggregated by several classes. To facilitate design and maintenance, the classes may be designed as small as possible. Therefore, state machines are quite capable of modelling the states and transitions of the classes. In addition, testers can use threaded multi-way trees to represent each of these state machines.

If testers wish to create inspection trees for another state machine, all they need to do is to change the structure of the tree node. The algorithm to create inspection trees for various state machines is the same. Therefore, it is possible to design an inspection tree generator that can generate various inspection trees for various state machines. The inspection tree generator can be designed to modify the source code of the inspection tree pattern into another source code, in order to build a new tree.

The testers can follow the state machine to modify the structure of the node in the pattern, and then compile the modified code. Next, testers need to type the state names (or called state information) and function names in accordance with the state machine. Finally, the required inspection tree is created to enable inspection of the test results file. Testers do not need to code a new program to create a new inspection tree for another state machine. The process of modifying the pattern tree is shown in figure 18.

7. Conclusion

State-based testing focuses on the question of whether a message or a sequence of messages puts an object of the class under test into the correct state. The threaded multi-way inspection tree imitates state-based testing with a state machine, to determine whether the resultant state is correct or not, after the message(s) are passed to an object. Binder [1996] defined four kinds of classes for testing: *nonmodal*, *unimodal*, *quasimodal* and *modal*. This paper has shown the example of the class queue, which is *quasimodal*. A bank account class, a kind of modal class, has also been implemented as an example with this automated class testing approach by Tsai *et al.* [1997c].

The advantages of the inspection approach, discussed as above, are that it is not complicated. Furthermore, testers can produce various inspection trees from the state machines. Additionally, regression testing can be performed since the inspection trees can be reused. Thus using this approach with a class test tool can help to reduce the overall cost of testing.

References

Binder, R.V. (1995), "State-Based Testing: Sneak Paths and Conditional Transitions," *Object Magazine*, October, 87–89.

Binder, R.V. (1996), "Modal Testing Strategies for OO Software," *IEEE Computer 29*, 11, 97–99.

Cook, S. and J. Daniels (1994), *Designing Object Systems: Object Oriented Modelling With Syntropy*, Prentice-Hall, Reading/London.

Glass, R.L. (1990), "Software Maintenance is Solution-Not a Problem," *Journal of Systems and Software 11*, 2, 77–78.

Harel, D. (1987), "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming 8*, 3, 231–274.

Jacobson, I., M. Christerson, P. Jonsson and G. Övergaard (1992), *Object-Oriented Software Engineering – A Use Case Driven Approach,* Revised 4th Edition, Addison-Wesley, Reading, MA.

McGregor, J.D. and D.M. Dyer (1993), "A Note on Inheritance and State Machines," Technical Report TR 93-114, 5th May, Department of Computer Science, Clemson University, SC.

Myers, G.J. (1979), *The Art of Software Testing*, Wiley, Reading/New York.

Norman, S. (1993), *Software Testing Tools,* Ovum Ltd., Reading/London.

Tsai, B.-Y., S. Stobart and N. Parrington (1997a), "Using Extended General Statecharts in Object-Oriented Program Testing: A Case Study," In *Proceedings of the 24th Technology of Object-Oriented Languages and Systems, TOOLS 24,* IEEE Computer Society Press, Los Alamitos, CA, pp. 96–103.

Tsai, B.-Y., S. Stobart and N. Parrington (1997b), "Iterative Design and Testing Within the Software Development Life Cycle," *Software Quality Journal 6*, 4, 295–310.

Tsai, B.-Y., S. Stobart and N. Parrington (1997c), "A Method for Automatic Class Testing (MACT) Object-Oriented Programs – Using A State-Based Testing Method," In *Proceedings of 5th European Conference Software Testing Analysis & Review, EuroSTAR 97,* EuroSTAR Administration, pp. 403–415.

Tsai, B.-Y., S. Stobart and N. Parrington (1998a), "An Automatic Test Case Generator Derived from State-Based Testing," Occasional Paper: CIS-1-98, School of Computing and Information Systems, University of Sunderland, UK.

Tsai, B.-Y., S. Stobart, N. Parrington and I. Mitchell (1998b) "An Automatic Test Case Generator Derived from State-Based Testing," In *Proceedings of 5th Asia Pacific Software Engineering Conference, ASPEC 98*, IEEE Computer Society Press, Los Alamitos, CA, pp. 270–277.

Turner, C.D. and D.J. Robson (1995), "A State-Based Approach to the Testing of Class-Based Programs," *Software Concepts & Tools 16*, 3, 106–112.