An Automatic Test Case Generator Derived from State-Based Testing

Bor-Yuan Tsai*, Simon Stobart, Norman Parrington, and Ian Mitchell
School of Computing and Information Systems,
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

Abstract
This paper describes an automated approach to generating test cases for an object-oriented class. The approach is derived from state-based testing methods and refers to a state machine from which a threaded multi-way tree (duplicating the behaviour of the state machine) is produced. All possible sequential test cases can then be automatically created, when the test case generator parses the tree.

1: Introduction

A test case generating approach based on functional testing is discussed within this paper. From a testing standpoint, it is possible to test all member functions of a class individually with structural tests, but at some point we need to test them all together. In a state-based testing method for an object-oriented class, the object class is the unit under test by test cases.

In functional testing, test cases must be derived from a functional specification, such as a finite state machine (FSM), which can be used to generate test cases following the requirements described by such a machine. Having the state machine of a class under test at hand, the next step to generate test cases is to design a state-transition tree. This paper demonstrates a threaded multi-way tree to completely duplicate the behaviour of a state machine (thereby improving the reported weakness of the state-transition tree [2] [12]). Test cases can be automatically produced to test the class implemented by programmers. The tree also contains the expected state of each transition, with which we can examine whether the state of an object under test changes correctly. This is a component of the automatic class testing MACT (the Method for Automatic Class Testing) [13].

Chow’s testing method using the FSMs, Turner’s state-based testing approach, and Hoffman’s class testing with testgraphs are introduced in section 2. In section 3, we discuss some testing concepts and terms, which are used in this paper. The MACT will be demonstrated using a graph in section 4. An example of test cases generated automatically will be shown in section 5. Here, a Vending Machine class will be used as an example to explain the generation of test cases from the state machine of the class to the test driver produced. Two algorithms to build a test case tree and to generate test cases using the tree are given. The conclusion is presented in section 6.

2: Related work

Chow [2] presented a testing strategy called “automata theoretic”, which deals with the verification of control structures at the design level. Chow showed that automata theoretic testing could find operation errors, extra states, missing states, and transition errors in a FSM [2]. However, this method can only be used if the control structures are represented by FSMs.

Turner [12] proposed a testing method, called state-based testing, which places emphasis on the state determined by the values (state values) stored in the variables, and the transitions between states. These can be described with FSMs. The next state of an object is determined by the current state and the next transition is then executed.

Hoffman [3], in the ClassBench methodology for automated class testing, used a testgraph, which is a partial model of the state-transition graph machine of the class under test. With the framework, testers need to develop the testgraph, an oracle class, and a driver class. The nodes and arcs in the testgraph are correlative to the state-
transition machine of the class, but the graph is smaller than the machine. The test cases are generated by repeatedly traversing the testgraph beginning at the state node and are derived from the paths that are followed.

However, we propose a novel method to generate test cases by following a state-based testing method and traversing a threaded multi-way tree which completely maps onto the shape of the state machine of the class.

3: Background

A specification describes functions that map inputs to outputs. The implementation gives a mapping from inputs to outputs, which may differ from the specification, indicating an error. Specification-based testing attempts to determine whether a component’s implementation provides the behaviour described in its specification. Techniques for developing test cases rely on the specification as the source for test cases without regard to the implementation.

A state machine offers a specification of the dynamic behaviour of the class of an object. The machine defines a set of states and a set of transitions, which describe the changes in states. Therefore, the state machine can be used as an aid in state-based testing. Various extensions have been promoted which can be used in testing approaches for object-oriented programs [1][3][4][6][12]. In practice, an examination of the state machine can result in different versions of implemented classes being produced by different programmers. Therefore, it is necessary to use another diagram to show the behaviour and states in the implemented class.

State-based testing is a form of implementation class testing which uses state machines (or FSMs). State machines represent the specification and are used to demonstrate the behaviour of systems or objects. They can also be derived from the code of the program under test [6]. The main point of state-based testing 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. The changing states rely on the values that are changed by the transition.

The test cases considered in this paper are defined for unit (class) testing by sending messages to objects and estimating the result. The test cases, here, are designed to examine transitions between states. A test case can create a starting state, the expected action causing transition to the next state, and the expected next state [5]. However, the test case generator in the MACT only produces the expected messages using implementation state machines. These messages are directly sent to the object class under test. The expected next states for the expected messages are stored in the test oracle tree (another component in the MACT) [13].

4: The automated class testing (MACT) framework

Designing test cases for a unit test requires a specification for the unit and the source code of the unit [8]. Programmers follow the specification (state machine) to implement class code, while testers also need to review the specification and implementation of the class to design test cases. In order directly to test the implemented class with the test cases, testers should test what they are given. Therefore, the implementation state machine may be required for test design. In MACT, the test case generator produces test cases based on the test case tree, which in turn is built from the implementation state machine. The test oracle tree is also created by the test case tree generator. The four components in the environment of the MACT are shown as a diagram in Figure 1.

![Figure 1 The components of the automated class testing framework (MACT)](image)

Test messages for a class under test are automatically produced by the test case generator, which traverses the test case tree. The tree is built from the implementation state machine. The test case tree generator can build various test oracle trees depending on the specific implementation state machine. The test driver generator receives test messages, which contain object and member function names, and then uses these to produce test drivers. The test results are inspected by the test result inspector, which parses the test result records (the messages and the resultant states of the messages) one by one with the test oracle tree. The tree contains an expected state for each expected message.
5: The test case generator

5.1: Coke Vending Machine example

The Coke Vending Machine example, assumes five components (classes) - CoinSlotPart, Light, Change, Stock, and TotalCounter. The behaviour of each class object is depicted with the state machines in Figure 2.

In an automated test environment, the messages (functions) in the generated test cases can be used directly to detect the class under test. Therefore, the test cases should invoke the member functions of the class under test instead of the transition names of its design state machine. Sometimes a transition between two states in a state machine is finished by several actions that may not only influence the local state of the object for which it is applied, but also the states of connected objects. For example, the Sold transition from \(\text{Amt} \geq 45\) state to \(\text{Amt} = 0\) state in the main component, CoinSlotPart class (see Figure 2), is completed by several steps, in which four other objects are involved. When the inserted amount is greater than or equal to 45 pence, a sequence of messages will be sent to Light, Change, TotalCounter, and StockCounter to finish the Sold transition.

Moreover, the WithdrawAll transition in the state machine is decomposed into three sub-transitions (member functions) overpaid(), switchlight(), and resetamt(). These three functions can be performed in any order, and only the resetamt() function causes the state of the CoinSlotPart object to change to the \(\text{Amt} = 0\) state. The overpaid() function will send a return inserted coin (returnamt) message to the Change object and the switchlight() function will send a switch light off message to the Light object. Neither of these member functions causes the state of the CoinSlotPart object to change. Therefore, no state exists between the \(\text{Amt} = 0\) and \(0 < \text{Amt} < 45\) states or \(\text{Amt} = 0\) and \(\text{Amt} \geq 45\) states in the CoinSlotPart state machine.

In practice, an examination of the state machine can result in different versions of implemented classes being produced by different programmers. For instance a programmer, who follows the CoinSlotPart state machine in Figure 2, could code the implemented class as per Figure 3. Of course other implementations could be produced which would also be correct.

5.2: Implementation state machine of the CoinSlotPart class

Programmers refer to the state machine as a specification for coding the class, and the expected results are a requirement that should be followed. Therefore, the states of the implementation object should be the same as the states that appear in its state machine. Nevertheless, the transition name in the state machine could be different from the member function name in the implemented class. For instance, using a state-based testing method, the test case for testing the Sold transition in the state machine is "...[\(\text{Amt} \geq 45\)], Sold, [\(\text{Amt} = 0\)]." However the test cases, "... \([\text{Amt} \geq 45]\), switchlight(), overpaid(), addamt(), oneinsold(), switchlight(), resetamt(), [\(\text{Amt} = 0\)]", will be used to test the implemented class, in Figure 3. Hence it is necessary to use another diagram to show the behaviour and state changes for the implemented class, if any difference exists.

We propose to extend the state machine (also called the design state machine) of the CoinSlotPart, in Figure 2, to an implementation state machine. This is shown in Figure 4, which shows all of the member functions of the implemented class in the state machine. The design state machine can be treated as a general state machine (GS) and the extended general state machine (EGS) can be used to represent an implementation state machine [10][11]. The EGS of the CoinSlotPart is given as Figure 4, and its GS is shown in Figure 2.
The table in Figure 4 shows that a transition in the GS maps to more than one member function in its implemented class. Function names with the ⇒ symbol mean that the functions are executed in sequence. The functions at the left-hand side of the symbol are performed before the functions following the symbol. The symbol + shows the combination of the functions without sequencing.

The style of the implementation state machine in Figure 4 seems easy, simple, and clear and encapsulates the design state machine. The table helps testers to understand the mapping between member functions and transitions, although the table is unnecessary for a simple mapping.

After mapping the member function names (outside of the encapsulated GS) to the transition name in the GS, testers can easily find the relationship between Sold transition and the other five member functions in the CoinSlotPart class. Additionally, the WithdrawAll transition maps to the overpaid( ), switchlight( ), and resetamt( ) member functions. This diagram can be completed by programmers, when they have coded the class, or by testers who review the member functions in the implemented class.
5.3: Test case generating (threaded multi-way) tree

5.3.1: State-transition tree. Test cases produced from state transition sequences have been discussed by Chow [2]. The first step in generating test cases is to prepare the state-transition tree, which is derived from the state machine. The initial state in the state machine becomes the root node in the tree, and each transition out of the initial state is a branch drawn to a node that represents the resultant state in the state machine. Therefore, the next possible states of the initial state in the state machine are the child nodes of the root node in the tree. This is repeated for each next state until (1) the node has already appeared in a previous level or (2) the node corresponds to a final state in the state machine [1][2][12][13]. Finally, each path from the root of the tree comprises a possible transition sequence from the beginning state to the end state in the state machine. A path also shows a sequence of test cases to detect whether the state changes correctly or not. The state-transition tree derived from the general state machine of the CoinSlotPart class is in Figure 5.

Tracing each full or partial path in the tree can yield a sequence or a single test case. Following the bold branches in Figure 5, a sequential test case example is shown below.

constructor(), [Amt=0], insertcoin(), [0<Amt<45], insertcoin(), [0<Amt<45],

\[
\begin{array}{c}
\text{ShowAmtDue} \\
\text{Amt=0} \\
\text{InsertCoin} \\
\text{0<Amt<45} \\
\text{WithdrawAll} \\
\text{Amt=0} \\
\text{sold} \\
\text{Amt=0} \\
\end{array}
\]

Figure 5 The state-transition tree derived from the GS of the CoinSlotPart

From the tree, we cannot use a sequence test case to test the state change from 0<Amt<45 to Amt≥45 when inserting three 20p coins sequentially, because of the first rule, discussed as above, for constructing a state-transition tree. The rule also prohibits us from having a sequence test case STC. Constructor [Amt=0], InsertCoin [0<Amt<45], InsertCoin [0<Amt<45], InsertCoin [Amt≥45], WithdrawAll [Amt=0], to test if 60 pence can be completely returned.

A transition in the state machine may be replaced by several member functions in its implementation code (as discussed previously). Additionally, test cases are automatically generated to reduce test effort. These reasons encourage us to propose a test case generating tree, which extends the inspection tree [13] and can create all possible test cases.

5.3.2: Test case generating tree. After adding a thread arc in a node to point back to the node with the same state name in the previous level, the threaded tree can produce the STC test case. This cannot be produced from the state-transition tree in Figure 5.

A test case consists of 3-tuples (S_i, t_i...t_n, S_2), where S_i is the current state and S_2 denotes the next (or final) state which will be reached after executing a transition t (or a sequence of messages, t_i...t_n, passed to the class object under test.) A pre-condition should be true in S_1 to trigger the transition t_i, and the execution result of the transition (also called the post-condition) should also be true to prove the state is being changed successfully. This is the reason Amt≥45, 0<Amt<45, and Amt=0 rather than “ReadyToSell”, “PayInsufficient” and “Standby” are used as state names in the state machines and the trees in this paper. These post-conditions will be placed in the state_name field of the test case generation tree node (see Figure 6 and Figure 7).

Each node of the test case generating tree consists of five fields as shown in Figure 6. The pre pointer field stores the previous node address. If a function is executed and the state is still the same as the previous node’s, then pre leads back to the previous one.

The next[] field is a pointer array in which the elements of the array hold the child nodes address. Of course, the size of the array is determined by the maximum number of transitions outgoing from a state in the state machine.

\[
\begin{array}{c}
\text{funct_name} \\
\text{state_name} \\
\end{array}
\]

Figure 6 The structure of a node in the test case generating tree

The loop field is used as a flag to terminate circular paths in the tree, when a series of test cases is being produced by loops in the tree from the root node.

The root node of the test case generating tree represents the initial state in the state machine. There are

---

* Vending machines, in UK, usually only accept 5, 10, 20, 50 pence or pound coins.
no incoming arcs to the root node (initial state of the state machine). Nevertheless, the initial state is the first step of an object’s lifetime, and in an implemented class this is done when the object is created. Therefore, the `functName` of the root node can store an object declaration statement instead of an empty string.

5.3.3: The algorithm to build the tree. The test case generating tree of the `CoinSlotPart` object class is shown in Figure 7, in which the function name and state name of each node are abbreviated due to limited space. The `stateName` fields in some of the nodes are empty which means the execution of the member functions in the nodes does not cause the state of the `CoinSlotPart` object to change. For example in the left-most path from the root node to the left-most leaf in the tree, the state change of the object starts from `Amt=0`, to `Amt≥45` and then back to `Amt=0` at the leaf node. This means that the state will change from `Amt≥45` to `Amt=0`, if these three member functions `Overpaid()`, `SwitchLight()`, and `ResetAmt()`, have been completely executed. Therefore, the `stateName` fields should be empty except for the last (leaf) node.

The threaded multi-way 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 its all child nodes at level N+1 are allocated and linked with it. This tree will grow level by level until all nodes (leaves) at the lowest level are linked to the tree. A complete C++ program following the algorithm (in Figure 8) can be referenced elsewhere [9].

![Figure 7 Test case generating tree of the CoinSlotPart](image)

**Figure 7 Test case generating tree of the CoinSlotPart**

![Figure 8 The algorithm to build a test case generating (threaded multi-way) tree](image)

**Figure 8 The algorithm to build a test case generating (threaded multi-way) tree**

![Figure 9 Test message file generated from the tree in Figure 7](image)

**Figure 9 Test message file generated from the tree in Figure 7**
5.4 Test cases and test message files

In a state-based testing method, messages are sent to the objects under test to examine whether the state values have changed correctly. Therefore, a test driver that directly tests the class will send messages. A message in C++ consists of an object name, a member function name, and parameters (or test data). The five object classes in this example are declared as CoinSlotPart csp; Light L; Change Chg; TotalCount Tcnt; StockCounter Scnt; and are used to form the test message file in Figure 9 and test driver in Figure 10.

To save space, the parameters are omitted in the test message file. In order to detect the states of the related (connected) object, some messages to review the state value of the objects are added in the test driver shown in Figure 10. Attribute encapsulation in a class is one of the features of object-oriented languages. The data member values can only be accessed by its member functions. However, in testing, we may add friend functions (which are not member functions) in the class to expose the state values. The classes in the implemented class of the Coke Vending Machine, in Figure 3, have member functions, which can display their state values. Therefore, they can be composed with object names to become messages in Figure 10.

The parameters in the test driver will be filled, based on the signatures of the member functions in the implemented class (see Figure 3). We can use different numbers and types of arguments to test the overloading problem, however we assume the problems of the member functions in the class under test have been solved before we re-test the class by state-based testing.

5.5 The algorithm to generate test case files

Travelling along the circular paths with pointers, threads and nodes from the root of the test case tree (see Figure 7), and mapping the paths with transitions and states in the state machine (see Figure 4), we can find that the tree fully represents the state machine graph. The depth-first traversing of a graph is similar to the pre-order traversing of a tree, but visiting every node by this traversal can only produce parts of test cases. For example, a left sub-tree can not be revisited when the current visiting node is at a right sub-tree, such that the csp.insertcoin( ), csp.insertcoin( ), csp.overpaid( ), csp.switchlight( ), and csp.resetamt( ) sequence test messages can not be generated by the depth-first traversal. Therefore we propose an algorithm to generate all possible sequence test cases in Figure 11.

If a node (called A) has been visited and it is a leaf node or its child nodes (sub-trees) have also been visited, then the pointer which points to the node A and resides in the parent node of the node A will be cut off. For example, in Figure 7, when the nodes on the left-most path has been visited, those pointers labelled with 2, 3, and 4 are cut off. When visiting the middle child (sub-tree) of the root node, the pointers on the left sub-tree of the root and the pointer named with 5 have all been cut off.

If a node, A, is re-visited by another node’s pre pointer and A’s pointers which point to its child nodes (sub-trees) have been cut off, these pointers will be reset. For example, if the root’s left child node is re-visited via a pre pointer, the pointers (named with 2, 3, 4, 5, and 6) and others on the sub-tree will be reset, so the whole sub-tree can be traversed again. The algorithm, which traces the tree in Figure 7 to generate test cases (or test messages) listed in Figure 9, is given in Figure 11 and its C++ code program is shown in [9].

6. Conclusions

The test case file generated from the test case generator does not contain input data (parameters), which can be filled by referring to the signature of each member function. It is generally impossible to test an operation, a function, or a program for all possible input data. However, equivalence partitioning and boundary-value analysis [10] are typical methods that can be used to divide up the input data space into test domains from which test cases can be drawn. In this example, the integer type Amt has 0 to N input data space, but the test data for the CoinSlotPart can be divided into zero, bigger than zero.
and less than 45 pence, and no less than 45p domains. The third domain, however, can be limited to 45=Amst100 if the units of inserted coins are 5p, 10p, 20p, 50p, or 100p (a pound). Therefore, the input test data for the CoinSlotPart object is \{0, 5, 10, 15, \ldots, 75, 80, 85, 90, 100\}. How to generate appropriate input data as parameters in the test case file is a topic for further research.

In practice, a class can have another embedded class or have related classes by inheritance or communication. The embedded and related communication classes should have been tested before the class under test. An abstract class is never used to declare class objects, so it cannot be tested with this technique. However, we can test its derived class. The operations to trigger the action of the embedded classes or the communication with outside classes are included in the member functions (transition) of the class under test. Therefore, based on the state-transition (state-based) testing method, those classes will be involved when the member functions are executed. The responses of those related objects could affect the behaviour of the object under test. That means that the state value of the class may consist of values from the related classes. If a complicated class can be defined with a state machine, the test cases can also be generated with this test case generator. How to minimize and model proper states in the state machine of the class under test is another issue for designers and testers.

![Figure 10 The algorithm to generate a test case file](image)

7. References