Automated Test Code Generation from UML Protocol State Machines

Dianxiang Xu, Weifeng Xu

Department of Computer Science
North Dakota State University
Fargo, ND 58105, USA
{dianxiang.xu, weifeng.xu}@ndsu.edu

W. Eric Wong

Department of Computer Science
University of Texas at Dallas
Richardson, TX 75803, USA
ewong@utdallas.edu

Abstract

This paper presents a framework for automated generation of executable test code from UML 2.0 protocol state machines. It supports several coverage criteria for state models, including state coverage, transition coverage, and basic and extended round-trip coverage. It transforms the state invariants and transition postconditions of a state model into executable assertions to be verified against the actual object states by runtime code instrumentation. Hand-crafted test data are reused from one development version to the next due to the change of requirements. This reduces the working load for test regeneration of modified models. Our framework also reports the complexity of generated test suites, which can facilitate empirical evaluation of different coverage criteria for state models.

Keywords: Software testing, finite state machines, UML, object-oriented programming, coverage criterion.

1. Introduction

Finite state machines (e.g., UML Statecharts [10]) are widely used to document the design of object-oriented systems. Test generation from the state models of object behaviors has gained much attention in the past decade. Several coverage criteria (e.g., state coverage and transition coverage) have been proposed for state-based testing. Existing testing methods, however, often use them to measure how much of the state model is covered by a given test suite, rather than automatically generating executable test code for the criteria.

In general, model-based testing requires some level of human intervention in order to produce executable tests. A major problem is the extent to which the manual work can be reused from one development version to the next. This is similar to regression testing [9], which involves three issues: (1) selecting from the current test suite those tests that remain valid for the modified program; (2) removing obsolete tests from the current test suite; (3) identifying additional tests. Issues (1) and (2) together are also called the regression test selection problem, whereas issue (3) is called the coverage identification problem [9][11]. Although each of them is significant, code-based regression testing is by and large limited to the test selection problem. For model-based test generation, all tests are newly generated for the modified models. New (obsolete) tests are added (excluded) automatically. As such, hand-crafted test data need to be carried from one development version to the next.

This paper presents a framework for automated generation of executable test code from UML 2.0 protocol state machines. It is fully implemented in the MACT (Model-based Aspect/Class Testing) toolkit. MACT first generates a transition tree from a state model for the chosen coverage criterion. The tester can edit detailed test parameters if necessary. When the state model is modified due to requirements change, the hand-crafted test parameters are automatically reused. Once the required parameters are provided, MACT can generate executable test code, including test methods and state wrapper classes. The state wrapper classes provide a mapping from the state invariants in a state model to the concrete object states. They are used by runtime code instrumentation for verifying actual object states against expected states.

The rest of this paper is organized as follows. Section 2 describes how UML protocol state machines are used for class modeling and provides an overview of the automated testing process. Section 3 presents test generation from state machines. Section 4 discusses automated reuse of test data for modified models. Section 5 presents test metrics and our experiments. Section 6 reviews related work. Section 7 concludes the paper.

2. MACT: An Automated Test Framework

2.1 UML Protocol State Machines for Class Modeling
In UML 2.0, a protocol state machine specifies which operations can be called in which state and under which conditions, thus specifying the allowed call sequences on the operations. A main difference between protocol state machines and Statecharts is that transitions in a protocol state machine are associated with a precondition (guard) and postcondition, but not actions. As a blackbox testing strategy, model-based testing is concerned with the effect of the transition, rather than the procedural process. Therefore, protocol state machines provide an appropriate level of abstraction for test generation from state models.

We exploit protocol state machines to capture intra-object behaviors and inter-object effects. A protocol state model \( M \) consists of states \( S \), events \( E \), and transitions \( T \). Transition \( (s_i, e[p, q], s_j) \in T \) (precondition \( p \) and postcondition \( q \) are optional) means that, when event (method) \( e \in E \) is triggered in the state \( s_i \in S \), when \( p \) holds, the state \( s_j \in S \) must be reached under \( q \). For a class state model, \( S, E, \) and \( T \) represent the possible states of objects, public constructors/methods, and functionality implemented by the constructors/methods, respectively.

A state \( s \in S \) can be a concrete object state or a state invariant. For example, the state \( \text{OPEN} \) of a \( \text{BankAccount} \) object may refer to the following: \( \text{getClosed}() == \text{false} \&\& \text{getBalance}() >= 0 \&\& \text{getFrozen}() == \text{false} \). Such states are specified in a state model for the purposes of test generation. They imply a link between the SUT and the generation of executable test code. The use of state invariants provides a high level of abstraction of object behaviors. A pre- or post-condition is a logical formula constructed by using constants, instance variables, and functions. A transition \( (s_i, e[p, q], s_j) \) without precondition means that the transition is unconditional: event \( e \) under state \( s_i \) always results in state \( s_j \) and \( q \) (if it exists). For convenience, we use \( \alpha \) to denote the state before an object is created (as in [1]) and the \( n \) event to represent the constructor. Usually, a class model includes \( \alpha \) in \( S \) and \( \text{new} \in E \). The object creation transition, \( (\alpha, \text{new}[p, q], s_0) \in T \), if \( p \) holds, constructs an object with the initial state \( s_0 \) and achieves the postcondition \( q \). Figure 1 shows the state model of a \( \text{BankAccount} \) class, where, for simplicity, \( b \) denotes \( \text{getBalance}() \). The model consists of three state invariants: \( \text{OPEN} \), \( \text{FROZEN} \) and \( \text{CLOSED} \). The events are \( \text{deposit} \), \( \text{withdraw} \), \( \text{getBalance} \), \( \text{close} \), \( \text{freeze} \), and \( \text{unfreeze} \).

A test sequence is a sequence of transitions \( (\alpha, \text{new}[p_0, q_0], s_0), (s_0, e[p_1, q_1], s_1), \ldots, (s_{n-1}, e[n_p, q_n], s_n) \). It starts with object creation, invokes methods on the object, and leads the object and other interacting objects to the respective states. Such a test sequence exercises not only individual constructors and methods, but also interactions between them.

### 2.2 Automated Testing Process

The automated testing process is shown in Figure 2. It starts with the tester building the state models for the classes under test and selecting a coverage criterion for test generation. The supported coverage criteria include state coverage, transition coverage, basic round trip, and extended round trip. For a given state model and coverage criterion, MACT generates a transition tree: the root represents the \( \alpha \) state; each non-root node represents the resultant state and postcondition of a transition from the state in the parent node. As such, each path from the root to a leaf is a test sequence as described before. Section 3 will elaborate on the coverage criteria and the algorithms for generating tests that achieve the criteria.

![Figure 2. Automated testing process](image_url)

Figure 3 shows the generated transition tree for the basic round-trip coverage of the \( \text{BankAccount} \) model in Figure 1. Negative test sequences, whose leaf nodes are illegal transitions, are marked with “[-]”. For instance, the test sequence along the path \( 1 \rightarrow 1.7 \rightarrow 1.7.4 \) consists of object creation and method invocations \(<\text{new}, \text{freeze}, \text{withdraw}>\) (the test input of the test sequence) as well as a sequence of expected resultant states \(<\text{OPEN}, \text{FROZEN}, \text{FROZEN}>\) (the oracle values of the test sequence). It is a negative test because one cannot \text{withdraw} money from a
**FROZEN BankAccount.** This test is to check whether or not the BankAccount class implementation would actually prohibit such an operation.

Figure 3. A transition tree for the model in Figure 1

Actual parameters have to be assigned to new and withdraw before the above test sequence becomes an executable test case (i.e., without compile-time errors). MACT provides a user-friendly interface for the tester to define such parameters. Once the tester clicks on a leaf node, the whole path from the root to the leaf is presented as a list of tables for editing. Figure 4 shows an editing session for the aforementioned test sequence. The user first inputs a value 1000 and uses it as the actual parameter for new by checking the parameter checkbox. The method freeze needs no parameter. For the invocation to withdraw, the user first provides a Java statement defining a double variable amount with value 100 (in this example, the parameter checkbox is not checked), and then uses amount as the actual parameter of withdraw.

Figure 4. A sample editing session

The ability to insert Java statements makes it possible to define runtime context for a specific testing task and set up and clean up test fixtures (e.g., establish and close a database or network connection before/after object creation or method invocations). Because the details of business logic (e.g., for deposit and withdraw) are often abstracted away in the state models, the tester is responsible for the satisfaction of method preconditions (e.g., getBalance()-amt<=0) when presenting actual test parameters. This is a non-trivial task, though. To alleviate this challenge, our future work will consider adapting a constraint satisfaction solver. This would require an executable language in place for specifying the detailed business logic (e.g., how deposit and withdraw operate).

Figure 5. A generated executable test method

After the required test parameters and additional code are completed, the test code generated by MACT is executable. If no method needs actual parameters, the generated test code is immediately executable. The general idea of code generation is as follows: a test method is created for each test sequence in a transition tree. Figure 5 shows the generated Java method for the aforementioned test sequence. The input value 1000 is used as the actual parameter for creating a bank account object: BankAccount bankaccount = new BankAccount(1000); The user-defined statement double amount = 100; is inserted before the call to withdraw, and amount is used as the actual parameter of the call.

After each object creation and method invocation, an assertion is created to verify if the class under test has reached the expected state. For example, is the bankaccount object in the FROZEN state after the invocation to freeze? The user-defined code (e.g., double amount = 100;) is inserted either before or after the method invocation, depending on the order in which it occurs. Transition postconditions are also transformed into assertions. Once all test methods for the entire transition tree are created, MACT wraps them up into a test class and defines a main method that invokes all the corresponding test methods. This test class thus becomes a test suite that satisfies the selected coverage criterion.

MACT also generates a state wrapper class for each class involved in a state model. It consists of constants representing the state invariants in a state model and a getModelState method evaluating when the runtime object states achieve these state invariants. It thus builds a bridge between state invariants and runtime object states and facilitates determining whether tests pass or fail. The
following code shows the state wrapper class for the BankAccount class. It is generated from the state definitions in the BankAccount state model. For example, the state invariant OPEN is defined as a constant. It represents a bankaccount object state that satisfies getClosed()==false && getBalance() >= 0 && getFrozen()==false.

```java
public class BankAccountModelState{
    public static final String CLOSED="CLOSED";
    public static final String FROZEN="FROZEN";
    public static final String OPEN="OPEN";

    public class BankAccountModelState{
        public final String getModelState( BankAccount bankaccount){
            if (bankaccount.getClosed()==false && bankaccount.getBalance() >= 0)
                return OPEN;
            else if(bankaccount.getFrozen()==true){
                return FROZEN;
            } else if(bankaccount.getClosed()==false
                && bankaccount.getFrozen()==false)
                return CLOSED;
            else return "Wrong state";
        }
    }
}
```

The test execution infrastructure is supported by a collection of AspectJ aspects that instrument additional code to the class under test at runtime. AspectJ [6] is a Java-based aspect-oriented language. The aspects monitor runtime object states and compare them with the expected states. In brief, the generated test class and state wrapper classes, the code instrumentation aspects, and the SUT together form an executable system under the AspectJ running environment. Due to the limited space, this paper will not elaborate on the code instrumentation aspects. Interested readers can contact the authors for more details.

In addition, MACT provides a number of utilities for test management, such as saving/importing/merging test data and adding/modifying/deleting/cloning a node in a transition tree.

3. Automated Test Generation

3.1 Coverage Criteria for State Models

Our approach supports the state coverage, transition coverage, basic and extended round-trip coverage for automated test generation from state models. A test suite is said to achieve the state (or the transition) coverage if it covers each of the states (or the transitions) at least once. The basic round trip coverage refers to the Binder’s round-trip path testing [1]. A basic round-trip test suite consists of a set of test sequences such that the resultant object state of each sequence has occurred at least once in some other sequence. An extended round-trip test suite consists of a set of test sequences such that the resultant object state and postcondition of each sequence is present at least once in some other sequence.

Let $A \Rightarrow B$ represent that coverage criterion $A$ subsumes coverage criterion $B$ (i.e., a test suite that achieves $A$ also achieves $B$). Then we have: extended round-trip $> \text{basic round-trip} > \text{transition coverage} > \text{state coverage}$. For example, the transition coverage subsumes the state coverage because a test suite of the transition coverage must cover all the states. The extended and basic round-trip coverage criteria are equivalent for a state model where no transitions have postconditions.

3.2 Test Generation Algorithms

Now we describe how the transition tree for a given coverage is generated. The root of a transition tree always represents the $\alpha$ state. The transition tree generation starts with the root and expands it.

The transition tree generation algorithm for the state coverage expands a node as follows: (1) find the transitions that start with the state represented by the current node (they are the object creation transitions if the current node is the root); (2) for each of these transitions, create a child node of the current node if its precondition can be satisfied and its resultant state is not yet traversed. The new child node represents the resultant state of the transition (it also contains a reference to the transition. This is similar for the other algorithms below). This state is marked as traversed; and (3) expand the new node.

The generation algorithm for the transition coverage expands a node as follows: (1) find the transitions that start with the state represented by the current node; (2) for each of these transitions, create a child node of the current node if its precondition can be satisfied and its resultant state is not yet traversed. The new child node represents the resultant state of the transition. The transition is marked as traversed; and (3) expand the new node.

The generation algorithm for the basic round-trip coverage expands a node as follows: (1) for each event, find the transitions that start with the state represented by the current node; (2) for each of the found transitions for the given event, create a child node of the current node if its precondition can be satisfied. The new child node represents the resultant state of the transition. Expand the new node if the resultant state has not appeared anywhere in the tree; (3) if no transition for the given event is found in the step (1) or the disjunction of the transition preconditions in step (2) is not a tautology (always true), create a new child node for the event (the event is illegal at the current state). The state of the new node is set to the state of the current node under expansion (i.e., an illegal event does not change object state). The precondition of the transition referenced by the new node is either null or the negation of the disjunction. Therefore the new node indicates a negative test. For example, the node 1.3 deposit ![amt>=0] $\Rightarrow$ OPEN ![\neg] in Figure 3 is a negative node. It is generated because deposit at the OPEN state is legal only when $\neg$.

The extended round-trip coverage is similar to the basic one except for,
in Step (2), expanding the new node if the resultant state and the transition postcondition are not contained by any node in the tree.

4. Reuse of Test Data for Modified Models

Frequent requirements change has been a norm in software development. To deal with requirements change, the design and implementation have to be modified. In the context of automated test generation, hand-crafted test data must be carried from one development version to the next. Consider the BankAccount model in Figure 1. Suppose a new banking policy allows overdrafts of up to $1,000. This requirements change is reflected in the modified BankAccount state model in Figure 6. The new OVERDRAWN state represents the balance of a BankAccount object is in the range of (0, -1000]. New transitions with respect to deposit, withdraw and getBalance are introduced.

![Figure 6. The modified BankAccount model](image)

Let $M$ and $M'$ denote the models before and after modification, $TS$ and $TS'$ are their test suites, respectively. Each test sequence, $ts'$, in $TS'$ belongs to one of the following situations:

1. $ts'$ needs no test parameters. In this case, its executable code can be generated immediately;
2. $ts'$ needs test parameters and is also a valid test sequence $ts$ in $TS$. In this case, the user-defined test parameters for $ts$ are all valid for $ts'$.
3. $ts'$ needs parameters and is also part of a valid test sequence $ts$ in $TS$. In this case, the user-defined test parameters for the common part are all valid for $ts'$.
4. $ts'$ needs parameters and it subsumes a valid test sequence $ts$ in $TS$ (i.e., $ts$ is a subsequence of $ts'$). In this case, the user-defined test parameters for $ts$ are all valid for $ts'$.

The situation (2) addresses the test selection problem of regression testing. It is not concerned about whether the tests in $TS$ are valid or invalid. Obsolete tests in $TS$ are not used in (i.e. automatically excluded from) the new test suite. Situations (3) and (4) deal with the coverage identification problem. They adopt existing test data, even if the test sequences containing these test data in $TS$ have become obsolete. MACT offers an efficient algorithm for carrying test data from the test suite of one model to the next. Instead of comparing individual test sequences, it works directly on the two transition trees and associated test parameters.

5. Test Metrics and Experiments

MACT provides the following statistical information on the complexity of generated test suites:

- test methods (#M),
- test methods with negative tests (#N),
- constructor and method calls (#CM),
- assertions in the test methods (#A),
- parameters used in the tests (#P),
- parameters inputted by the tester (#PI),
- statements used in the tests (#S),
- statements provided by the tester (#SI).

We have applied MACT to the generation of executable test code for several applications. Due to the limited space, here we only report the test metrics for the two BankAccount models in Figure 1 (denoted by $BA1$) and Figure 6 (denoted by $BA2$). Let $a$, $b$, $c$ and $d$ be the extended round trip, basic round-trip, transition coverage, and state coverage, respectively. As neither of $BA1$ and $BA2$ has postconditions, there is no difference between the extended and basic round-trip. The number of assertions (#A) is also the same as the number of constructor and method calls (#CM).

Table 1 shows the metrics of the executable test suites for $BA1$. For the round-trip coverage, there are 18 test methods (#M); 14 of them contain negative tests (#N); a total of 26 parameters (#P) are used in the test methods; and only nine (#PI) are direct inputs by the tester. Table 2 shows the test metrics of the (non-executable) test suites when they are first generated by reusing the test parameters for $BA1$. For the round-trip coverage, eight inputs (#PI) for $BA1$ are carried into the test suite of $BA2$ for 32 total parameters (#P, See Table 2). Table 3 shows the metrics of the executable test suites for $BA2$. For the round-trip coverage, a total of 13 tester-input parameters (#PI) are expected. The tester needs to provide five more parameter inputs after reuse. Similarly, for the transition/state coverage, the tester needs to input three and one more parameters, respectively.

<table>
<thead>
<tr>
<th>Table 1. Metrics of the executable tests for $BA1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>#M/N</td>
</tr>
<tr>
<td>BA1-a/b</td>
</tr>
<tr>
<td>BA1-c</td>
</tr>
<tr>
<td>BA1-d</td>
</tr>
</tbody>
</table>
Table 2. Metrics of BA2 non-executable test suites generated by reusing BA1 test parameters

<table>
<thead>
<tr>
<th></th>
<th>#M/N</th>
<th>#CM</th>
<th>#P/PI</th>
<th>#S/SI</th>
</tr>
</thead>
<tbody>
<tr>
<td>BA2-a/b</td>
<td>25/17</td>
<td>94</td>
<td>32/8</td>
<td>1/1</td>
</tr>
<tr>
<td>BA2-c</td>
<td>3/0</td>
<td>21</td>
<td>7/3</td>
<td>0/0</td>
</tr>
<tr>
<td>BA2-d</td>
<td>3/0</td>
<td>9</td>
<td>3/1</td>
<td>0/0</td>
</tr>
</tbody>
</table>

Table 3. Metrics of the executable tests for BA2

<table>
<thead>
<tr>
<th></th>
<th>#M/N</th>
<th>#CM</th>
<th>#P/PI</th>
<th>#S/SI</th>
</tr>
</thead>
<tbody>
<tr>
<td>BA2-a/b</td>
<td>25/17</td>
<td>94</td>
<td>43/13</td>
<td>1/1</td>
</tr>
<tr>
<td>BA2-c</td>
<td>3/0</td>
<td>21</td>
<td>11/6</td>
<td>0/0</td>
</tr>
<tr>
<td>BA2-d</td>
<td>3/0</td>
<td>9</td>
<td>4/2</td>
<td>0/0</td>
</tr>
</tbody>
</table>

6. Related Work

Significant research effort has been directed at the generation of test sequences from state models [6]. For example, the W-method [3] and Wp-method [4] construct a transition tree and traverse the transition tree so that each path is covered by the test cases. Many state-based test generation methods also use a state model to represent the SUT and then test whether or not the implementation and design models conform to each other. These methods have been extensively studied in the context of protocol testing [6]. However, none of them targets the testing of object-oriented programs. For example, events in the state machines are different from parameterized methods in object-oriented programming.

State models have also been used for model-based testing of object-oriented systems. The round-trip path testing [1] as the most referenced and applied technique is an adaptation of the W-method for deriving tests from a FREE state model (i.e., flattened Statechart) that describes the behavior of a single class or a cluster of classes. It replaces the identification sequence with a call to a state invariant checking method and requires the SUT to have a trusted ability to report the resultant states. Briand et al. [2] have recently conducted a series of controlled experiments evaluating the cost-effectiveness of the round-trip path testing, and they have showed that it can be enhanced by category partition. Hong et al. [5] provide a way to derive extended state machines from Statecharts to devise test criteria based on control and data flow analysis. Offutt et al. [8] provide definitions for such test criteria as all transitions, all transition pairs, and full-predicate. These criteria are used to evaluate how much of the state model is covered by a given test suite. Our approach uses the coverage criteria to drive test generation, i.e., generate tests that satisfy the criteria.

7. Conclusions

We have presented the framework for automated generation of executable test code from protocol state models. It supports four test coverage criteria. Reuse of hand-crafted test data for subsequently modified models can reduce the workload of creating new tests. MACT can also facilitate empirical evaluation of the cost-effectiveness (e.g., correlation of fault detection capability and testing costs) of various coverage criteria for test generation from state models. Such evaluation by hand would be tedious and error-prone without tool support.

Our future work will integrate a rigorous constraint language for specifying pre- and post-conditions in state models and a constraint problem solver for generation of test parameters. We will also investigate the test selection problem for model-based regression testing – how test sequences of modified models should be selected or prioritized.

8. Acknowledgement

This work was supported in part by the ND EPSCoR IIP-SG via NSF Grant EPS-047679 and the NASA OSMA/SARP Center Initiatives Program.

9. References