Automated Test Code Generation from Class State Models

Dianxiang Xu  
Department of Computer Science  
North Dakota State University  
Fargo, ND 58105, USA  
dianxiang.xu@ndsu.edu

Weifeng Xu  
Computer and Information Science Department  
Gannon University  
109 University Square  
Erie, PA 16541  
frank.w.xu@gmail.com

W. Eric Wong  
Department of Computer Science  
University of Texas at Dallas  
Richardson, TX 75083, USA  
ewong@utdallas.edu

Abstract

This paper presents an approach to automated generation of executable test code from class models represented by the UML protocol state machines. It supports several coverage criteria for state models, including state coverage, transition coverage, and basic and extended round-trip coverage. It allows the tester to add and modify detailed test parameters (e.g., actual arguments for method invocations and implementation-specific environments) if necessary. When the state model is modified due to requirements change, the hand-crafted test parameters, if still valid, are automatically reused. This reduces the working load for regeneration of tests for modified models. In addition to test code, we also automatically generate state wrapper aspects in AspectJ, which facilitates comparing actual object states to expected states during test execution. This enables the automated verdict of pass/failure for the test cases without need to modify the source code of the class under test. We present two examples for which the executable test code is generated. They demonstrate the reuse of test parameters and testing of object interactions, respectively.

Keywords: Software testing, UML, protocol state machines, test generation, object-oriented programming.
1. Introduction

Model-based testing is an appealing paradigm of conformance testing [19]. The modeling activity in the testing process helps clarify requirements and enhances communication between developers and testers. The testing process can be automated or partially automated. Model-based testing makes use of explicit models of the system under test’s (SUT) intended behavior for generating test cases and verifying conformance between the SUT and its models [16]. A behavior model is an abstract, simplified description of the SUT’s intended behavior. Traces of the model are selected to constitute test cases for the SUT. The key aspects of model-based testing include the behavior model, the test generation algorithm, and supporting infrastructure for the test execution (e.g., test harness). Several empirical studies have demonstrated that model-based testing improved the fault detection capability and reduced testing costs [2][6][20].

This paper focuses on test code generation from finite state models for classes. Finite state machines are widely used to document the design of object-oriented systems [17]. 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. Although generation of test sequences in existing state-based testing methods can be automated, manual transformation of the test sequences into executable test code tends to be time-consuming and error-prone. It can be a daunting task when a large number of test sequences are generated from a complex model. It is highly desirable to automatically generate executable test code. In the context of testing object-oriented programs, objects of one class can be associated or composed with objects of other classes. Typically, a state model captures the state transitions of an object for the reception of events or method invocations (in this paper, we will use events and methods interchangeably). A method invocation not only updates the state of the callee object for which the method is defined, but may affect the states of other interacting objects. To detect interaction faults, an automated test generation method is expected to be able to create tests that cover and inspect such interactions.

In general, state-based testing requires some level of human intervention in order to produce executable tests. For example, tests generated from state models often require actual arguments for method invocations in
the tests. Such actual arguments can be objects that need to be constructed. Some tests even require additional code for setting up an environment (e.g., database or network connection) before test execution and then cleaning up the environment after test execution. Such data and code that need to be hand-crafted by the tester are referred to as test parameters. As system requirements change from time to time during development, a major problem with automated test generation is the extent to which the hand-crafted test parameters can be reused from one development version to the next. This is similar to regression testing [21][22][24], which involves three issues: (1) selecting from the current test suite those tests that remain valid for the modified program; (2) removing from the current test suite those tests that are obsolete for the modified program; (3) identifying additional tests for the modified program. Issues (1) and (2) together are also called the regression test selection problem, whereas issue (3) is called the coverage identification problem [21]. Although each of them is significant, existing regression testing techniques, by and large, are limited to the test selection problem. For automated model-based test generation, all tests can be generated from a model no matter whether the model is original or modified. This implies that, for a modified model, new tests are automatically added and obsolete tests are automatically excluded. Nevertheless, the tester may need to provide certain test parameters for each version of the models. In this context, carrying hand-crafted test parameters from one development version to the next becomes critical.

As an expanded version of our SEKE’07 paper [25], this article presents an approach to automated generation of executable test code from class models represented by the UML protocol state machines. The approach is fully supported by a tool, MACT\(^1\). It first generates a transition tree from a state model for the chosen coverage criterion, such as state coverage, transition coverage, and basic and extended round-trip coverage (the extended round-trip generalizes the basic round-trip for testing object interactions by using postconditions as part of the termination condition of test generation). The tester can add and modify detailed test parameters if necessary. When the state model is modified due to requirements change, the hand-crafted test parameters, if still valid, are automatically reused. Based on the transition tree and concrete test

\(^1\) MACT (Model-based Aspect/Class Testing) can generate test cases from state models for both classes and aspects. Aspect testing, however, is beyond the scope of this paper.
parameters, MACT generates a test class that contains a method for each of the test sequences in the tree. From the corresponding state model, MACT also generates a state wrapper aspect in AspectJ\(^2\), which provides a mapping from concrete object states in the implementation to model-level states. It is used to verify actual object states against the expected model-level states during test execution. This enables the automated verdict of pass/failure for the test cases without need to modify the source code of the class under test. We present two case studies for demonstrating reuse of test parameters from one development version to the next and testing of object interactions. The case studies show that our approach is effective in determining reusable test parameters and that among the four coverage criteria the extended round-trip is the most effective for detecting program faults with regard to object interactions.

The rest of this paper is organized as follows. Section 2 describes how protocol state machines are used for class modeling and presents the automated process of test generation. Section 3 presents generation of transition trees from state machines for each of the coverage criteria. Section 4 discusses automated reuse of test data for modified state models. Section 5 discusses verification of object interactions. Section 6 presents two examples for demonstrating our work. Section 7 reviews related work. Section 8 concludes the paper.

2. **MACT: An Automated Test Generation Tool**

In this section, we first discuss how protocol state machines are used for modeling object behaviors. Then we present the automated test generation process, including generation of test code and state wrapper aspects. The facilities for test editing and management are briefly introduced.

2.1 UML Protocol State Machines for Class Modeling

In UML, a protocol state machine specifies which operations can be called in which state and under which conditions, thus specifying the allowed call sequences of the operations. A protocol state machine presents the possible and permitted transitions on the objects, together with the operations that trigger the transitions. In this manner, “an object lifecycle can be created, by specifying the order in which the operations can be

activated and the states through which an instance progresses during its existence” [23]. While this paper focuses on protocol state machines, our approach is applicable to flattened behavioral state machines or Statecharts. 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 in Statecharts). Actions in Statecharts often specify the object behaviors (or a procedural process) when an event occurs. These are typically encoded as a procedural process in a class implementation. As a black box testing strategy, model-based testing is concerned with the effect of the transition, rather than the procedural process. We specify the effect of the actions as part of the transition postcondition, which can be used to verify if the actions are implemented correctly.

In this paper, 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 $T$. A 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 (called) in the state $s_i \in S$, when $p$ holds, then the state $s_j \in S$ must be reached under the condition $q$. For the state model of a given class, $S$, $E$, and $T$ represent the possible states of objects, public constructors/methods, and functionality implemented by the constructors/methods, respectively. For test generation purposes, our approach also relies on a mapping from model-level states to implementation-level object states. This mapping is part of the description of the state model. For example, the state OPEN in the BankAccount state model in Figure 1 is corresponding to $\neg \text{getClosed()} \land \text{getBalance()} \geq 0 \land \neg \text{getFrozen()}$. Such a state mapping provides a link between the model and the implementation. For example, verifying whether or not a bankAccount object is in the OPEN state defined in the state model is equivalent to evaluating whether or not the implementation satisfies $\neg \text{bankAccount.getClosed()} \land \text{bankAccount.getBalance()} \geq 0 \land \neg \text{bankAccount.getFrozen()}$.

A pre- or post-condition is a logical formula constructed by using constants, instance variables, and query functions (methods with return values, but without side effects). A transition $(s_i, e[q], s_j)$ without precondition means that the transition is unconditional: event $e$ under state $s_i$ always results in state $s_j$ and $q$ (if $q$ exists). For
convenience, we use $\alpha$ to denote the state before an object is created (as in [1]) and the new event to represent the constructor. Usually, a class model includes $\alpha$ in $S$ and new in $E$. The object creation transition, $(\alpha, \text{new}[p, q], s_0) \in T$, if condition $p$ holds, constructs an object with the initial state $s_0$ and achieves the postcondition $q$. We allow different constructors that create objects with distinct initial states. For a state $s$, $s$ is an initial state if there is an object creation transition $(\alpha, \text{new}[p, q], s)$. Given a state $s$, if there is no transition that transforms it to any other state, then $s$ is a termination state. Figure 1 shows the state model of a BankAccount class. The model consists of three state invariants: OPEN, FROZEN and CLOSED. The events are deposit, withdraw, getBalance, close, freeze, and unfreeze.

![State Model of BankAccount Class](image)

**Figure 1. The state model of a BankAccount class**

A test sequence is a sequence of transitions $(\alpha, \text{new}[p_0, q_0], s_0), (s_0, e_1[p_1, q_1], s_1), \ldots, (s_{n-1}, e_n[p_n, q_n], s_n)$. 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. The sequence of constructor and method invocations new, $e_1, \ldots, e_n$ whose actual arguments satisfy respective preconditions $p_i$ ($0 \leq i \leq n$) is the test input. $s_0, q_0, s_1, q_1, \ldots, s_n, q_n$ specify conditions of the expected states (oracle values) that a correct implementation should produce when executing the test.

### 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. Section 3 will elaborate...
on the coverage criteria and the algorithms for generating transition trees that achieve the criteria. For a given state model and coverage criterion, MACT automatically 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. For a transition tree, MACT can automatically generate a test class that includes all the test sequences in the tree - each test sequence becomes a method in the class. From the corresponding state model, MACT can also generate state wrapper aspects that serve as a mechanism of runtime code instrumentation for the execution of generated test code. The SUT, generated test code, generated state wrapper aspects, and support class and interface form an executable program. The support class and interface as a test harness access runtime object states and determine whether test cases pass or fail. This will be detailed in subsections 2.2.1 and 2.2.2.

2.2.1 Automated Test Code Generation

---

3 Although transition trees are automatically generated, the tester is allowed to provide test parameters and make modifications. This is discussed in subsections 2.2.1 and 2.2.3.
MACT can automatically generate test code from a transition tree (generation of transition trees will be detailed in Section 3). For illustration purposes, Figure 3 shows the generated transition tree for the basic round-trip coverage of the BankAccount model in Figure 1 (the root representing the α state is hidden). Negative test sequences, whose leaf nodes are illegal transitions, are marked with “[-]”. For instance, the test sequence along the path $1 \rightarrow 1.7 \rightarrow 1.7.4$ (called sneak path in [1]) consists of a sequence of object creation and method invocations $\langle\text{new, freeze, withdraw}\rangle$ (the test input of the test sequence) as well as a sequence of expected resultant states $\langle\text{OPEN, FROZEN, FROZEN}\rangle$ (the oracle values of the test sequence). It is a negative test because one cannot 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

Before the above test sequence becomes an executable test case (i.e., without compile-time errors), actual parameters have to be assigned to new and withdraw according their signatures in the class interface. MACT provides a user-friendly interface for the tester to define and modify 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 case, the corresponding parameter checkbox is not checked), and then uses amount as the actual parameter of withdraw. For this test sequence, withdraw should not change the FROZEN state, otherwise the test fails. To reduce redundancy, the user-defined parameters and statements for a non-leaf node by default are shared by all descendents of the node. For example, all the test sequences in Figure 3 would share the value 1000 for new unless other operations on the tree are applied. The tester may choose a different value for some test sequences by first cloning a node or sub-tree (refer to subsection 2.2.3).

<table>
<thead>
<tr>
<th>1 new</th>
<th>→ OPEN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Java Expression or Code</td>
<td>Parameter</td>
</tr>
<tr>
<td>1000</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1.7 freeze</th>
<th>OPEN → FROZEN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Java Expression or Code</td>
<td>Parameter</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1.7.4 withdraw</th>
<th>FROZEN → FROZEN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Java Expression or Code</td>
<td>Parameter</td>
</tr>
<tr>
<td>double amount = 100;</td>
<td></td>
</tr>
<tr>
<td>amount</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Figure 4. A sample editing session**

The ability to insert Java statements makes it possible to define runtime context for specific test cases 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 new, 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. After the required test parameters and additional code are completed, MACT can generate executable test code as follows: a test method is created for each test sequence in a transition tree. The following shows the generated Java method for the aforementioned test sequence. The input value 1000 is used as the actual parameter for creating a `BankAccount` 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.

```java
private static void modelTest1_7_4()
{
    System.out.println("Test case 1_7_4");
    BankAccount bankaccount = new BankAccount(1000);
    assertState(bankaccount, BankAccount.OPEN);
    bankaccount.freeze();
    assertState(bankaccount, BankAccount.FROZEN);
    double amount = 100;
    bankaccount.withdraw(amount);
    assertState(bankaccount, BankAccount.FROZEN);
}
```

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, `assertState(bankAccount, BankAccount.OPEN)` verifies if the `BankAccount` object is in the `OPEN` state after it is created. The user-defined code (e.g., `double amount = 100;`) is inserted before the method invocation. Transition postconditions are also transformed into assertions after the method invocation. 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 each of the test methods (the main method, i.e., the test driver, is not needed if JUnit is used). This test class thus becomes a test suite that satisfies the selected coverage criterion.

### 2.2.2 Automated Generation of State Wrapper Aspects and Test Execution Infrastructure

To facilitate execution of the generated test code, MACT also automatically generates a state wrapper aspect for each class involved in a state model. The following code shows the state wrapper aspect for the `BankAccount` class. It is generated from the state definitions in the `BankAccount` state model, as discussed in section 2.1.
public aspect BankAccountModelState{

declare parents: BankAccount implements StateBasedClass;
	public static final String BankAccount.OPEN="OPEN";
	public static final String BankAccount.FROZEN="FROZEN";
	public static final String BankAccount.CLOSED="CLOSED";

public String BankAccount.getModelState(){
    if (!getClosed() && getBalance() >= 0 && !getFrozen()){
        return OPEN;
    } else if(getFrozen() == true){
        return FROZEN;
    } else if(getClosed() == true){
        return CLOSED;
    } else return "Wrong state";
}
}

In general, the state wrapper aspect for a class includes three parts: (1) The class under test is made to implement the interface StateBasedClass via the declare parents statement. The purpose of a declare parents statement in AspectJ is to introduce one or more new superclass or interface for a base class. This allows object states of the class to be compared through the assertState method in the generated test code. (2) The model-level states of the class are defined as constants via inter-type constant declarations. An inter-type declaration introduces a new constant, instance variable or method to a base class. The model-level states can thus be referenced by the assertState method in the generated test code, such as assertState(bankaccount, BankAccount.OPEN). (3) The getModelState method is defined for the class via an inter-type declaration. It maps concrete object states (e.g., !getClosed() && getBalance() >= 0 && !getFrozen()) to model-level states (e.g., OPEN). This makes it possible for the assertState method to compare concrete object states to the expected model-level states and report the verdict of pass or failure for test cases. As such, state wrapper aspects serve as a mechanism of runtime code instrumentation – they introduce new elements (constants and methods) to classes but do not modify the source code of classes. This is a significant benefit because if the Java program under test is modified, we would have to undo the modifications once testing is finished. In our
approach, we simply run the java program under test together with the generated test code in the AspectJ environment.

Figure 5 shows the test execution infrastructure. When JUnit is used, we define StateBasedTest as a subclass of TestCase in JUnit. The StateBasedTest class provides the implementation of the assertState method, whereas the StateBasedClass interface defines the method getModelState. They are the support class and interface in Figure 2. The generated test class is defined as a subclass of StateBasedTest so as to inherit assertState. SUT is the class under test or the head class of a class cluster under test. The SUT, generated test code, generated state wrapper aspects, StateBasedTest and StateBasedClass form an executable system in the AspectJ running environment. The execution reports whether each test passes or fails.

![Figure 5. Test execution infrastructure](image)

### 2.2.3 Support for Test Case Management

MACT provides a number of utilities for test case management, such as:

1. Reusing test data due to changes of state models (to be discussed in Section 4).
2. Saving/importing test data, including the generated transition tree structures, test parameters, and code written by the tester. The generated tree structures can be modified by the user. Executable test code is generated from such test data. In general, it is important for the tester to save the test data, rather than the generated test code.
(3) Merging test data. This is to combine different test suites for the same class. It is also possible to merge tests produced by other test generation strategies. For example, model-checking can be used to generate property tests of a system from the traces of property violation (i.e., counterexamples). We can merge such property tests into a transition coverage test suite. MACT detects and removes redundant test cases automatically.

(4) Adding a node in a transition tree. This is useful when the tester wants to include more method invocations that follow a specific test sequence.

(5) Deleting a node/subtree from a transition tree. This can be used to rule out some particular tests.

(6) Cloning a test node or sub-tree. This is useful for two situations: (1) the tester can achieve stronger condition coverage for part of or the entire test tree, such as full-predicate [18]; (2) the tester can employ different test parameters in the common path of different test sequences.

3. Automated Generation of Transition Trees

This section describes the coverage criteria and presents the algorithms for generating transition trees.

3.1 Coverage Criteria for State Models

As mentioned earlier, 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 \( A > B \) represent that coverage criterion \( A \) subsumes coverage criterion \( B \) (i.e., a test suite that is adequate with respect to \( A \) also gives 100% adequacy for \( B \)). Then we have: extended round-trip > basic round-trip > transition coverage > state coverage. For example, the transition coverage subsumes the state
coverage because a test suite adequate with respect to the transition coverage must cover all the states (a state cannot be reached without transition firings). The round-trip transition tree of a state model includes all transitions in the model. Here we assume all states and transitions in a state model are reachable from the initial states (MACT can check a model for unreachable states and transitions). A state model with unreachable states or transitions usually implies some modeling problem, which needs to be resolved before test generation. 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 criterion is generated. The root of a transition tree always represents the $\alpha$ state. The transition tree generation starts with the root node 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 and contains a reference to the transition. This is similar for the other algorithms below. The state contained in the new node 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 the transition is not yet covered and its precondition can be satisfied. The new 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.

The extended round-trip coverage is similar to the basic one except, in step (2), where the new node is expanded only if the resultant state and transition postcondition as a whole are not contained by any other node in the tree. Suppose \( m \) and \( n \) are the number of states and transitions, respectively. In general, the size (number of nodes) of the transition tree for the state and transition coverage is \( O(m) \) and \( O(n) \), whereas the size for the round-trip is \( O(m^2 \times n) \) [1].

According to the above algorithms, branch coverage for transition preconditions is applied by default to the two round-trip criteria (i.e., negative tests are generated). The state and transition coverage criteria are not concerned with negative tests because their goals are to cover all the states and transitions. The tester can further achieve full-predicate coverage [18] for transition preconditions by using the test management operations introduced in subsection 2.2.3. This is beyond the scope of this paper, though.

4. **Reuse of Test Parameters 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 parameters must be carried from one development version to the next. Consider the `BankAccount` model in
Figure 1. It does not allow overdraft because the precondition of `withdraw, getBalance-amt>=0`, means that the current balance must be greater than or equal to the amount of withdrawal. Suppose a new banking policy allows an overdraft of up to $1,000. This requirements change is reflected in the modified `BankAccount` state model in Figure 6. The new `OVERDRAWN` state represents that the balance of a `BankAccount` object is in the range of (0, -1000]. New transitions with respect to `deposit, withdraw` and `getBalance` are introduced.

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 test parameters and it 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 test parameters and it subsumes a valid test sequence $ts$ in $TS$ (i.e., $ts$ is a sub-sequence of $ts'$). In this case, the user-defined test parameters for $ts$ are all valid for $ts'$.

Situation (2) addresses the test selection problem of regression testing. 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 (i.e., generation of new tests). They can adopt previous 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. For the test suite of the `BankAccount` model in Figure 1, we created nine input items for the test parameters of all method invocations and one assignment statement. This made the generated test code executable (the assignment statement is only for the purposes of demonstration; it is not required per se. In general, Java statements can be inserted to define runtime context for a specific testing task, as well as to set up and clean up test fixtures). Eight of the nine parameters and the statement are automatically copied into the
transition tree for the modified BankAccount model in Figure 6. One was left out because the truth value of the withdraw precondition getBalance-amt<-1000 of a newly generated negative test for the modified model could not be determined due to the lack of detailed business logic for new and withdraw. That is, the test parameter 3000 provided by the tester for the old test sequence <new(1000), withdraw(3000)> is not carried into the new test sequence <new, withdraw[getBalance-amt<-1000]>.

Figure 6. The modified BankAccount model

5. Verification of Object Interactions

Due to the composition relationship among classes, an object not only changes its own states upon invocations of its methods, but also affects the states of other interacting objects. Consider the Controller class in the cruise control simulation shown in Figure 7 (it is a modified version of the cruise control applet in [15]). Each Controller object has a SpeedControl object. The methods of Controller include engineOn, engineOff, accelerator, brake, on, resume, and off. An invocation to these methods (e.g., on) changes the state of a Controller object (e.g., from ACTIVE to CRUISING). It may also update the state of the composed SpeedControl object through such method invocations as recordSpeed and enableControl. The following code snippet from the Controller class reflects this issue:

private SpeedControl sc;
...
```java
synchronized void on(){
    if (controlState != INACTIVE){
        sc.recordSpeed();
        sc.enableControl();
        controlState=CRUISING;
    }
}
```

From the black box testing perspective, we should check if the Controller object reaches the CRUISING state after the on method is called when the car is running (not INACTIVE). We should also test if this would affect the SpeedControl object correctly. Here, the state model of Controller has to specify the impacts on both the Controller and SpeedControl objects. The former is described directly by the transitions between the Controller states whereas the latter is captured by on’s postcondition, sc.E1, which means getCruiseEnabled()==1 && getSetSpeed()>0 (it is the effect of invocations to recordSpeed and enableControl). The Controller state model, as shown in Figure 8 (where sc denotes SpeedControl), depicts this situation by the following transitions, which happen to have no preconditions:

\[(ACTIVE, \text{on}[\text{, sc.E1}], \text{CRUISING})\]

\[(STANDBY, \text{on}[\text{, sc.E1}], \text{CRUISING})\]

![Figure 7. The architecture of a cruise control simulation (Adapted from [15])](image-url)
There are two important issues pertaining to the use of postconditions for test generation:

1. How can test sequences be generated to cover the situations where the postconditions occur?
2. How can the postconditions be verified against the runtime object states?

We address the first issue by extending the basic round-trip test generation. That is, the generation of a test sequence is not terminated until the object’s state and the resultant postcondition have both been contained within a single node of some other test sequence. Obviously, missing calls to the `SpeedControl` methods in the previous code snippet would not be caught without the specification and verification of the postconditions. We address the second issue by transforming the postconditions into executable assertions. The assertions are then compared to the actual object states by the code instrumentation aspects. For example, the following is the generated code of test sequence `<new, engineOn, on, accelerator, resume>` for the `Controller` class.

```java
private static void modelTest1_1_5_2_7()
{
    System.out.println("Test case 1_1_5_2_7");
    Controller controller = new Controller();
    assertState(controller, Controller.INACTIVE);
    assertState(controller.getSpeedControl(), SpeedControl.D0);
    controller.engineOn();
    assertState(controller, Controller.ACTIVE);
    assertState(controller.getSpeedControl(), SpeedControl.D0);
    (D0, "getCruiseEnabled()==0 &\& getSetSpeed()<=0")
    (D1, "getCruiseEnabled()==0 &\& getSetSpeed()>0")
    (E0, "getCruiseEnabled()==1 &\& getSetSpeed()<=0")
    (E1, "getCruiseEnabled()==1 &\& getSetSpeed()>0")
}
```

**Figure 8. The state model of Controller**

D0, "getCruiseEnabled()==0 &\& getSetSpeed()<=0")
D1, "getCruiseEnabled()==0 &\& getSetSpeed()>0")
E0, "getCruiseEnabled()==1 &\& getSetSpeed()<=0")
E1, "getCruiseEnabled()==1 &\& getSetSpeed()>0")
controller.on();
assertState(controller, Controller.CRUISING);
assertState(controller.getSpeedControl(), SpeedControl.E1);
controller.accelerator();
assertState(controller, Controller.STANDBY);
assertState(controller.getSpeedControl(), SpeedControl.D1);
controller.resume();
assertState(controller, Controller.CRUISING);
assertState(controller.getSpeedControl(), SpeedControl.E1);
}

After the object creation (new) and each method invocation (engineOn, on, accelerator, resume), the states of both Controller and SpeedControl objects are verified.

The postcondition of a transition can be conditional. Consider the CarSimulator class in the cruise control application as an example. The method accelerate under state ON10 (which means that engine is on, throttle() is not zero, and brakepedal() is zero) preserves the state of a CarSimulator object. However, its effect on the Controller object is conditional: it updates the Controller object to the STANDBY state only when the Controller object is at the state CRUISING. MACT can evaluate such conditional effects during test generation.

6. Examples

This section presents two examples for demonstrating our work. For both examples, we were able to generate executable test code. In the following, we discuss the reuse of test parameters and the use of postconditions for testing object interactions in MACT.

6.1 Test Reuse

To demonstrate reuse of test parameters, we created four versions of the BankAccount class. Table 1 summarizes the four state models together with the changes from one version to the next. The changes have covered additions of states, events (methods), transitions, and modifications of transitions and guard conditions. In fact, BA2 and BA3 are corresponding to the BankAccount models in Figure 1 and Figure 6, respectively.
Tables 2, 3, and 4 show the metrics of the executable test suites for the basic and extended round-trip, transition and state coverage, respectively. In this study, there is no difference between the basic and extended round-trip because the BankAccount models involve no postconditions. This section is not concerned with the use of postconditions. The number of test methods refers to the number of test sequences or test cases in the corresponding transition tree. The number of calls includes calls to constructors and methods. The total number of test parameters refers to the number of parameters used by all object constructions and method invocations. The required inputs refer to the distinct inputs that must be provided in order to make the test cases completely executable without syntax errors. Reused inputs are those among the required inputs that are found from the test suite for the previous version, whereas actual inputs are those among the required inputs that are provided by the tester. The total number of test parameters used in the test methods is often much greater than provided by the tester. The reason is that each parameter can be shared by many test sequences, which could save the tester a lot of time and effort in producing executable test code.

For the first version, BA1, the number of reused inputs is always 0 and the number of actual inputs is the same as the number of required inputs because there is no reuse. Consider BA2 in Table 2. Among the 9 required inputs, 5 of them are from the test suite of BA1 and the rest 4 inputs are provided by the tester. For BA3, 8 of these 9 inputs are reused as discussed in Section 4. As of BA3, the total number of required inputs is 13. For BA4, only 8 of 13 are reused; the remaining 5 inputs are no longer valid due to the change of transition preconditions. In summary, this empirical study shows that MACT is very effective in determining reusable test parameters – the test parameters in the test suite of current state model can be carried into the next version if they are still valid.

Table 1. BankAccount state models

<table>
<thead>
<tr>
<th>Version</th>
<th># of States</th>
<th># of Events</th>
<th># of Transitions</th>
<th>Functional Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BA1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>N/A</td>
</tr>
<tr>
<td>BA2</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>Add new states, events, transitions</td>
</tr>
<tr>
<td>BA3</td>
<td>4</td>
<td>6</td>
<td>12</td>
<td>Add new transitions</td>
</tr>
<tr>
<td>BA4</td>
<td>4</td>
<td>6</td>
<td>12</td>
<td>Modify guard conditions</td>
</tr>
</tbody>
</table>
Table 2. BankAccount test reuse for the extended and basic round trip (RT) coverage

<table>
<thead>
<tr>
<th>Version</th>
<th># of Test Methods</th>
<th># of Calls</th>
<th># of Test Parameters/Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>BA1</td>
<td>5</td>
<td>15</td>
<td>9</td>
</tr>
<tr>
<td>BA2</td>
<td>18</td>
<td>48</td>
<td>26</td>
</tr>
<tr>
<td>BA3</td>
<td>25</td>
<td>94</td>
<td>43</td>
</tr>
<tr>
<td>BA4</td>
<td>25</td>
<td>94</td>
<td>43</td>
</tr>
</tbody>
</table>

Table 3. BankAccount test reuse for the transition coverage

<table>
<thead>
<tr>
<th>Version</th>
<th># of Test Methods</th>
<th># of Calls</th>
<th># of Test Parameters/Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>BA1</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>BA2</td>
<td>2</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>BA3</td>
<td>3</td>
<td>21</td>
<td>12</td>
</tr>
<tr>
<td>BA4</td>
<td>3</td>
<td>21</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 4. BankAccount test reuse for the state coverage

<table>
<thead>
<tr>
<th>Version</th>
<th># of Test Methods</th>
<th># of Calls</th>
<th># of Test Parameters/Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>BA1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>BA2</td>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>BA3</td>
<td>3</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>BA4</td>
<td>3</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

6.2 Testing Object Interactions

The cruise control application is used to demonstrate the impact of postconditions on testing of object interactions. From the testing perspective, the application has three clusters of classes: \{SpeedControl, CarSpeed, CruiseDisplay\}, \{Controller, SpeedControl, CarSpeed, CruiseDisplay\}, and \{CarSimulator, Controller, SpeedControl, CarSpeed, CruiseDisplay\}. These class clusters are tested through the state models of classes, SpeedControl, Controller, and CarSimulator, respectively. Table 5 summarizes the state models in the empirical study. The state model of SpeedControl involves no postconditions, 10 of the 12 transitions in
the model of Controller have postconditions about the states of the SpeedControl object and 15 of the 24 transitions in the model of CarSimulator have postconditions about the states of the Controller object.

Table 5. CruiseControl state models

<table>
<thead>
<tr>
<th>Class</th>
<th># of States</th>
<th># of Events</th>
<th># of Transitions</th>
<th># of Transitions with Postconditions</th>
<th>Affected Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpeedControl</td>
<td>4</td>
<td>4</td>
<td>17</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>Controller</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>10</td>
<td>SpeedControl</td>
</tr>
<tr>
<td>CarSimulator</td>
<td>4</td>
<td>7</td>
<td>24</td>
<td>15</td>
<td>Controller</td>
</tr>
</tbody>
</table>

Table 6. Metrics of generated executable tests for SpeedControl

<table>
<thead>
<tr>
<th>Coverage</th>
<th># of Test Methods</th>
<th># of Calls</th>
<th># of Assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extend Round Trip</td>
<td>13</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>Basic Round Trip</td>
<td>13</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>Transition</td>
<td>3</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>State</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 6 shows the metrics of generated executable tests for SpeedControl for each of the coverage criteria. The number of assertions refers to the total number of assertState statements in the generated test class. As no transition in the state model involves postconditions, the number of assertions is equal to the number of calls to the constructor and methods, i.e., there is exactly one assertState statement after each call. It is worth pointing out that the number of calls and assertions can better characterize the complexity of a given coverage criterion than the number of test methods [1]. For example, the number of test methods depends on the search strategy (depth-first search versus breadth-first search) and the order in which the states, events, and transitions are specified. The depth-first search algorithm may produce fewer but longer test methods compared to the breadth-first search algorithm. MACT uses the breadth-first search for all coverage criteria.

To demonstrate how transition postconditions affect testing, two versions of state models for each of Controller and CarSimulator have been used to generate tests: one without the postconditions and the other with the postconditions. Table 7 presents the metrics of executable tests for both versions. Each entry in the columns “# of Test Methods”, “# of Calls”, and “# of Assertions” consists of a pair of numbers with respect to
the two versions, respectively. For example, the extended round-trip tests for the two versions of Controller have 114 and 184 assertions, respectively, although they have the same number of calls. The assertions are increased by 61% due to the use of transition postconditions in test generation. For Controller, the extended and basic round trip criteria have the same number of test methods, calls, and assertions. This is not the case for CarSimulator. The use of postconditions in CarSimulator has imposed a significant impact on test generation: the test methods of the extended round-trip test suite have increased from 25 to 43, the calls increased from 108 to 249 by 131%, and the assertions increased from 108 to 390 by 261%.

Table 7. Metrics of generated executable tests for CarSimulator and Controller

<table>
<thead>
<tr>
<th>Class</th>
<th>Coverage Criteria</th>
<th># of Test Methods</th>
<th># of Calls</th>
<th># of Assertions</th>
<th>% Increased Calls</th>
<th>% Increased Assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller</td>
<td>Extended Round Trip</td>
<td>25/25</td>
<td>114/114</td>
<td>114/184</td>
<td>0%</td>
<td>61%</td>
</tr>
<tr>
<td></td>
<td>Basic Round Trip</td>
<td>25/25</td>
<td>114/114</td>
<td>114/184</td>
<td>0%</td>
<td>61%</td>
</tr>
<tr>
<td></td>
<td>Transition</td>
<td>4/4</td>
<td>29/29</td>
<td>29/52</td>
<td>0%</td>
<td>79%</td>
</tr>
<tr>
<td></td>
<td>State</td>
<td>1/1</td>
<td>5/5</td>
<td>5/9</td>
<td>0%</td>
<td>80%</td>
</tr>
<tr>
<td>CarSimulator</td>
<td>Extended Round Trip</td>
<td>25/43</td>
<td>108/249</td>
<td>108/390</td>
<td>131%</td>
<td>261%</td>
</tr>
<tr>
<td></td>
<td>Basic Round Trip</td>
<td>25/25</td>
<td>108/108</td>
<td>108/159</td>
<td>0%</td>
<td>38%</td>
</tr>
<tr>
<td></td>
<td>Transition</td>
<td>7/5</td>
<td>80/67</td>
<td>80/87</td>
<td>-16%</td>
<td>9%</td>
</tr>
<tr>
<td></td>
<td>State</td>
<td>2/2</td>
<td>8/8</td>
<td>8/12</td>
<td>0%</td>
<td>50%</td>
</tr>
</tbody>
</table>

We use mutation analysis to demonstrate the fault detection capability of the generated test cases. We have created a total of 19 mutants of the cruise control application (specifically the CarSimulator and Controller classes). The fault in each of the mutants is related to the object interaction. Again, let us consider the code snippet in Section 5. Mutants of this code would be either missing sc.recordSpeed(), sc.enableControl() or substituting sc.enableControl() for sc.disableControl(). A mutant is said to be killed by the test set of a coverage criterion if at least one test reports a failure. Our experiments have confirmed that none of the mutants were killed by any of the coverage criteria if no assertions were generated for the postconditions.

---

4 Our focus is on object interactions. Evaluating the fault detection capability of each coverage criterion with respect to general program faults is beyond the scope of this paper.
Table 8. Results of mutation analysis for object interaction

<table>
<thead>
<tr>
<th>Coverage Criteria</th>
<th># of Killed Mutants</th>
<th>Percentage of Killed Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extended Round-Trip</td>
<td>19</td>
<td>100%</td>
</tr>
<tr>
<td>Basic Round-Trip</td>
<td>15</td>
<td>79%</td>
</tr>
<tr>
<td>Transition</td>
<td>12</td>
<td>63%</td>
</tr>
<tr>
<td>State</td>
<td>7</td>
<td>37%</td>
</tr>
</tbody>
</table>

Table 8 presents the results of our mutation analysis. The test set generated for the extended round-trip coverage killed all the mutants. The test sets generated for the basic round trip, transition and state coverage killed 79%, 63%, and 37% of the mutants, respectively. The mutation analysis shows that (1) the use of postconditions as assertions in test cases can detect object interaction faults, and (2) the use of postconditions as part of the termination condition of test generation for the round-trip coverage can improve fault detection capability, but it also increases the complexity of test suites in terms of the number of calls and assertions.

7. Related Work

Finite state machines have long been in use as models of software and hardware systems. Significant research effort has been directed at the generation of test sequences from state models [16]. For example, the W-method [5] and Wp-method [8] construct a transition tree, traverse the transition tree so that each path is covered by the test cases, and append a state characterization or identification sequence to check the state that is reached. In addition to design models, 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 [16]. El-Fakih et al. [7] have recently adapted four of the well-known methods (W, Wp, UIOv, and HIS) for generating tests that would test only the modified parts of an evolving specification. However, none of these methods 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 are also widely used to document the design of object-oriented systems [17]. They provide a rigorous definition of the expected class behavior. 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. [3] 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, a classical functional testing strategy. Hong et al. [9] provide a way to derive extended state machines from Statecharts to devise test criteria based on control and data flow analysis. Offutt et al. [18] provide definitions for such test criteria as all transitions, all transition pairs, and full-predicate for guard conditions. While the state-based modeling and coverage criteria of state models in this paper are based on the above existing work, this paper focuses on generation of executable test code. Our aspect-oriented code instrumentation mechanism requires no modification to the source code of the SUT. Reuse of test parameters and use of postconditions for testing object interactions are also open issues.

Finally, it is worth pointing out that our work is different from such unit testing frameworks as JUnit and NUnit. They generate from the source code of given classes the skeleton of test classes and test drivers, not concrete test cases. Nevertheless, our work takes advantage of JUnit: the generation of test driver is not needed when JUnit is used.

8. Conclusions

We have presented an approach, supported by a tool, to automated generation of tests from class state models. The main features of our approach are as follows: (1) automated generation of executable test code for several coverage criteria of state models. Aspect-oriented programming is exploited as a mechanism of runtime code instrumentation for evaluating pass/failure of test cases. The aspect code is also automatically generated from state models; (2) reuse of hand-crafted test parameters for subsequently modified models. This is especially
important for incremental development and testing to deal with frequent requirements changes; and (3) automated verification of the effects of object interactions. Transition postconditions are not only transformed into assertions in test cases, but also exploited for extending the traditional round-trip testing strategy. This is useful for detecting interaction faults. Currently, we are building the state models and generating test code for the MACT tool itself, which has more than 20,000 lines of code.

Due to the support of automated generation of executable test code, we plan to further evaluate cost-effectiveness (e.g., correlation of fault detection capability and testing costs) of different coverage criteria for test generation from state models. Such evaluation by hand would be tedious and error-prone without tool support. We expect to incorporate more structure-oriented test generation methods for such coverage criteria as transition pairs and full-predicate [18] and develop property-oriented test generation methods through model-checking. For automated generation of test parameters, we plan to integrate a rigorous constraint language into our current approach for specifying preconditions and postconditions of transitions so that constraint problem solvers and test data generation tools can be applied. Moreover, a UML design model may depict object interactions with sequence diagrams while the internal state transitions of objects are captured by a state machine. It is worth investigating how to generate test code from protocol state machines, together with sequence diagrams.

Also of interest is the test prioritization problem for model-based regression testing – how generated tests of modified models should be selected and prioritized. It is often very expensive to run all the tests after each round of modifications. The generated tests for a modified system should be arranged in a way that can reveal regression faults as early as possible. In general, the new tests for a modified version should be executed first, but the order of those tests that have already occurred in the previous version need to be prioritized carefully. The existing test prioritization techniques mainly fall into two categories: code-based and model-based. Code-based test prioritization orders tests by exploiting the difference in modified and original programs [21][22][14]. Model-based test prioritization prioritizes the tests by running them on the modified model [12][13] or by analyzing the dependencies of modifications [4]. These techniques do not involve automated
test generation with respect to various coverage criteria of state models. In MACT, tests for both original and modified models are generated automatically for a given coverage criterion. It is worth investigating whether and how the coverage-oriented test generation algorithms in MACT can benefit from the existing model-based test prioritization techniques [4][12][13].

9. Acknowledgement

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

References