Testing aspect-oriented programs with finite state machines

Dianxiang Xu1,*,†, Omar El-Ariss2, Weifeng Xu3 and Linzhang Wang4

1National Center for the Protection of the Financial Infrastructure, Dakota State University, Madison, SD 57042, U.S.A.
2Department of Computer Science, North Dakota State University, Fargo, ND 58105, U.S.A.
3Computer Science Department, Gannon University, Erie, PA 16541, U.S.A.
4State Key Laboratory of Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing 210093, People’s Republic of China

SUMMARY

Aspect-oriented programming yields new types of programming faults due to the introduction of new constructs for dealing with crosscutting concerns. To reveal aspect faults, this paper presents a framework for testing whether or not aspect-oriented programs conform to their state models. It supports two families of strategies (i.e. structure-oriented and property-oriented) for automated generation of aspect tests from aspect-oriented state models. A structure-oriented testing strategy derives tests and test code from an aspect-oriented state model to meet a given structural coverage criterion, such as state coverage, transition coverage, or round trip. A property-oriented testing strategy generates test code from the counterexamples of model checking. Two such strategies are checking an aspect-oriented state model against trap properties and checking mutants of aspect models against system properties. Mutation analysis of aspect-oriented programs is used to evaluate the effectiveness of these testing strategies. The experiments demonstrate that testing aspect-oriented programs against their state models can detect many aspect faults. The comparative evaluations also reveal that the structure-oriented and property-oriented testing strategies complement each other—some aspect faults were detected by the structure-oriented strategies, but not by the property-oriented strategies and vice versa. Copyright © 2010 John Wiley & Sons, Ltd.

1. INTRODUCTION

Aspect-oriented programming (AOP) [1] modularizes crosscutting concerns into aspects with the advice invoked at the specified points of program execution. Although the ability to modularize crosscutting concerns appears to improve the quality, aspect-oriented software development does not assure correctness by itself. For example, AOP supports a variety of composition strategies, ‘from the clearly acceptable to the questionable’ [2]. Aspects can be used in a harmful way that invalidates the desired properties [3, 4] and even destroys the conceptual integrity of programs [2]. The new AOP features and mechanisms can yield new types of programming faults, such as incorrect pointcuts, advice, or aspect precedence [5]. Therefore, new strategies, techniques, and practices are needed for aspect assurance.

*Correspondence to: Dianxiang Xu, National Center for the Protection of the Financial Infrastructure, Dakota State University, Madison, SD 57042, U.S.A.
†E-mail: dianxiang.xu@dsu.edu

Copyright © 2010 John Wiley & Sons, Ltd.
Finite state machines have long been used for modeling and testing object-oriented programs. For example, a number of publications [6–11] have demonstrated that finite state models of individual classes and clusters of classes are effective formalisms for class testing and even integration testing of object-oriented programs. As AOP languages are typically built upon object-oriented languages (e.g. AspectJ is based on Java), finite state machines are also expected to be useful for testing aspect-oriented programs. As such, the objective of the research in this paper is to investigate how finite state models can be used to test aspect-oriented programs.

This paper presents a framework, MACT (Model-based Aspect/class Checking and Testing), for testing whether or not aspect-oriented programs conform to their aspect-oriented state models. MACT provides explicit aspect-oriented notations (e.g. pointcut, advice, inter-model declarations, aspect and aspect precedence) for describing aspects at the abstraction level of UML protocol state machines [12]. MACT also provides two groups of strategies (i.e. structure-oriented and property-oriented) for generating aspect tests from aspect-oriented state models. A structure-oriented strategy derives tests to meet a given structural coverage criterion, such as state coverage, transition coverage, and round trip. A property-oriented strategy generates tests from the counterexamples of model checking. Two such strategies are checking an aspect-oriented state model against trap properties (i.e. negated system properties) [13] and checking mutants (i.e. incorrect variations) [14] of aspect models against the correct system properties. To check an aspect-oriented state model, MACT first composes the aspect models and class models by a weaving mechanism and then transforms the woven models into FSP (Finite State Processes), which are the input to model checker LTSA (Labeled Transition System Analyzer) [15].

The fault detection capabilities of the testing strategies are evaluated through mutation analysis of aspect-oriented programs. Mutants of aspect-oriented programs are created from a comprehensive fault model of aspect programming. A program mutant is said to be killed by a test suite if the test execution reports a failure. The percentage of killed program mutants reflects the fault detection capability of a testing strategy. MACT has been applied to two non-trivial systems, generated tests from their aspect-oriented state models for each of the five testing strategies, and evaluated their fault detection capabilities. The empirical studies demonstrate that testing aspect-oriented programs against their state models have detected many aspect faults. The comparative evaluations also reveal that the structure-oriented and property-oriented strategies complement each other—one group of strategies killed some mutants that were not killed by the other group and vice versa.

The remainder of this paper is organized as follows. Section 2 gives a brief introduction to aspect-oriented modeling and checking with finite state machines in MACT. Section 3 describes the automated incremental process for testing aspect-oriented programs using their state models. Sections 4 and 5 present the structure- and property-oriented strategies for aspect test generation, respectively. Section 6 introduces the implementation of MACT. Section 7 reports the empirical study. Section 8 reviews the related work. Section 9 concludes the paper.

2. ASPECT-ORIENTED MODELING AND CHECKING WITH FINITE STATE MACHINES

Aspect-oriented modeling is an activity that involves identifying, representing, analyzing, and managing core concerns (classes) and crosscutting concerns (aspects). The aspect-oriented modeling approach in MACT [16] uses UML 2.0 protocol state machines [12] to capture the behaviors of classes and aspects. It provides a conceptual link between aspect-oriented modeling and implementation. The model-level notions for concerns that crosscut the state machines of classes include aspects, pointcuts, join points, advice, inter-model declarations, and aspect precedence. An aspect-oriented state model captures the behaviors of all classes and aspects of a system or subsystem. It consists of classes, aspects, and aspect precedence. Aspect precedence as a partial-order relation on the given set of aspects resolves the conflict between multiple aspects that share join points but provide conflicting advice.
TESTING AOP WITH FINITE STATE MACHINES

2.1. Class models

In MACT, the state model $M$ for a class 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, if event $e \in E$ is triggered in the state $S_i \in S$, when $p$ holds, then the state $S_j \in S$ must be reached under $q$. For a class state model, $S$, $E$, and $T$ represent the possible object states, public constructors/methods, and functionality of the constructors/methods, respectively.

Figure 1 shows the state model of CarSimulator class in an aspect-oriented cruise control application\(^1\), engineOn, engineOff, accelerate, and brake are events corresponding to the respective methods of the class. The state invariants OFF00, ON00, ON10, and ON01 are defined upon three state variables: whether ignition is on or off, whether or not throttle is 0, and whether or not brakepedal is 0.

$\text{(OFF00, "!getIgnition() && getThrottle()==0 && getBrakepedal()==0")}$  
$\text{(ON00, "getIgnition() && getThrottle()==0 && getBrakepedal()==0")}$  
$\text{(ON01, "getIgnition() && getThrottle()==0 && getBrakepedal()!=0")}$  
$\text{(ON10, "getIgnition() && getThrottle()!=0 && getBrakepedal()==0")}$

Such state specifications provide a mapping from the state models to the implementation under test (IUT). This mapping is used to generate executable test code (to be discussed in Section 3).

For convenience, let $x$ denote the state before an object is created and the $\text{new}$ event represent the constructor. An object construction transition $(x, \text{new}[p, q], S_0) \in T$ if condition $p$ holds, constructs an object with the initial state $S_0$ and achieves the postcondition $q$. Multiple object creation transitions are allowed if they have different preconditions and lead to distinct initial states. Thus the initial state(s) can be inferred from the object creation transition(s). To represent different classes or objects, $X.e$ or $X(e)$ denotes the element $e$ (event, state, or transition) in the state model of class or object $X$.

Consider a sequence of transitions $(x, \text{new}[p_0, q_0], S_0), (S_0, e_1[p_1, q_1], S_1), \ldots, (S_{n-1}, e_n[p_n, q_n], S_n)$ or simply $\text{new}[p_0, q_0], S_0, e_1[p_1, q_1], S_1, \ldots, S_{n-1}, e_n[p_n, q_n], S_n$. It starts with object creation, invokes the methods, leads the object to the resultant states, and achieves the postconditions. Such a sequence is called a concrete test sequence if the actual parameters of all object construction and method invocations are determined. Otherwise it is called an abstract test sequence. The sequence of constructor and method invocations $\text{new}, e_1, \ldots, e_n$ whose actual arguments satisfy the 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 the properties of the expected object states (oracle values) that a correct implementation should produce when executing the test. Note that determining the actual parameters of all object construction and method invocations in an abstract test sequence is known as the undecidable path sensitization problem [6]. In MACT, these actual parameters need to be assigned manually in order to generate executable test code. An abstract test sequence reduces to a concrete test sequence if the object construction and method invocations have no parameters.

\(^1\)Cruise control has been widely used for empirical evaluation of traditional testing strategies. An aspect-oriented cruise control simulation [16] built from a legacy system [15] presents three typical aspects for precondition enforcement and separation of concerns.

Copyright © 2010 John Wiley & Sons, Ltd.  

DOI: 10.1002/stvr
2.2. Aspect models

In MACT, an aspect model consists of inter-model declarations (ID), state pointcuts (SP), transition pointcuts (TP), and advice models (AM). The following is a brief introduction to these concepts.

**Definition 1 (Inter-model declaration)**

An inter-model declaration is defined as follows:

\[
\text{declare} \langle \text{base} \rangle \langle \text{transition} \rangle \{,\, \langle \text{base} \rangle \langle \text{transition} \rangle \}
\]

where \( \langle \text{base} \rangle \langle \text{transition} \rangle \) refers to a transition in base model.

An inter-model declaration introduces one or more new transitions (state or event) to the base model. For an introduced transition \( B(s_i, e[p, q], s_j) \), if \( s_i, s_j, \) or \( e \) are not yet in the base model \( B \), then they become a new state or event in \( B \). The new transitions, states, and events can be used in subsequent pointcut definitions. Figure 2 shows a portion of the state model of \( \text{CruiseControllerIntegrator} \) aspect. The ‘\text{declare}’ clause introduces transitions and thus the \( \text{on} \) event into the base class \( \text{CarSimulator} \).

**Definition 2 (Pointcut)**

Pointcuts are defined as follows:

1. pointcut\((\text{cutname})\langle \text{transition} - \text{variable} \rangle : \langle \text{base} \rangle \langle \text{transition} \rangle \{,\, \langle \text{base} \rangle \langle \text{transition} \rangle \}
2. pointcut\((\text{cutname})\langle \text{state} - \text{variable} \rangle : \langle \text{base} \rangle .\langle \text{state} \rangle \{,\, \langle \text{base} \rangle .\langle \text{state} \rangle \}

where (1) and (2) define transition and state pointcuts, respectively; \( \langle \text{cutname} \rangle \) identifies a pointcut; \( \langle \text{transition} - \text{variable} \rangle \) is a formal transition, \((s_i, e[p, q], s_j)\), where \( s_i, e, p, q, \) and \( s_j \) are variables; \( \langle \text{base} \rangle \langle \text{transition} \rangle \) refers to an existing transition (join point) in the base model; \( \langle \text{base} \rangle .\langle \text{state} \rangle \) refers to an existing state (join point) in the base model. A transition or state variable serves as a unified reference to multiple transitions or states in one or more base models.

A pointcut picks out a group of join points; a join point is a transition or state in a base model. In Figure 2, for example, the state pointcut \( \text{IGNITION} \) picks out the state join point \( \text{CarSimulator} \).

---

An aspect model is a structure \langle ID, SP, TP, AM \rangle, where ID, SP, TP, and AM are a list of inter-model declarations, state pointcuts, transition pointcuts, and advice models, respectively.

Multiple pointcuts in the same aspect may share join points. The order in which their advice is applied to the shared transitions depends on their occurrences in the aspect model. Inter-aspect interference may also exist when multiple pointcuts in different aspect models share join points but provide conflicting advice. To deal with aspect interference, an explicit precedence relation between aspects can be specified. It is a partial-order relation on the given set of aspect models. As such, an aspect-oriented state model consists of class models, aspect models, and a precedence relation on the aspect models.

Definition 3 (Advice model)
An advice model is defined as: advice \langle \text{transition-cut} \rangle \langle \text{state-model} \rangle.

The advice for a pointcut, specified by a state model, describes the control logic applied to each join point picked out by the pointcut. For example, the advice of cruiseOn in Figure 2 means that, after a join point transition happens, the event on is issued to the CruiseController object if isMinSpeedReached() holds (i.e. the car speed has reached the required minimum for cruising). This event leads CruiseController to the Cruising state (for clarity, the prefix ‘CruiseController.’ to Cruising is omitted in Figure 2). Likewise, the advice of the pointcut init creates a CruiseController object with the initial state Inactive when the CarSimulator object is constructed; the advice of the pointcut ignitionOn leads CruiseController to the Active state through an invocation to its method engineOn.

In general, an advice model can express a broad range of impacts that an aspect may impose on its base classes [17]. Examples include removing transitions from base models, changing the resultant states of transitions, modifying the guard conditions of transitions, adding new transitions among existing states, introducing new states and events and thus transitions between new and existing states, and incorporating states and events of other classes for modeling object interactions.

Definition 4 (Aspect model)
An aspect model is a structure \langle \text{class models}, \text{aspect models}, \text{precedence relation} \rangle, where \text{class models} is a set of class models, \text{aspect models} is a set of aspect models, and \text{precedence relation} is an aspect precedence relation over \text{aspect models}.

Multiple pointcuts in the same aspect may share join points. The order in which their advice is applied to the shared transitions depends on their occurrences in the aspect model. Inter-aspect interference may also exist when multiple pointcuts in different aspect models share join points but provide conflicting advice. To deal with aspect interference, an explicit precedence relation between aspects can be specified. It is a partial-order relation on the given set of aspect models. As such, an aspect-oriented state model consists of class models, aspect models, and a precedence relation on the aspect models.

Definition 5 (Aspect-oriented state model)
An aspect-oriented state model with \text{m} class models and \text{n} aspect models is a triple \langle \{C_i\}, \{A_j\}, R \rangle, where \{C_i\}(1 \leq i \leq m) is a set of class models, \{A_j\}(1 \leq j \leq n) is a set of aspect models, and \text{R} is an aspect precedence relation over \{A_j\}, respectively.

Aspect-oriented state models focus on the composition of crosscutting concerns, which is different from the merging of hierarchical state models [18]. Hierarchical models represent different levels of abstractions, whereas aspects as crosscutting concerns are at the same level of abstraction as the base models. As AOP for object-oriented languages, aspects in aspect-oriented state models are much like meta-level operations on base models.

2.3. Checking aspect-oriented state models
Model checking is to verify whether or not a model satisfies the particular properties required of a system through exhaustive exploration of the state space. Model-checking in MACT is based on the model checker LTSA. In LTSA, a system model is represented by an FSP behavior process, whereas a property is formulated as a safety process or fluent linear temporal logic (FLTL) assertion. FSP processes are in essence labeled transitions systems (LTS), where transitions in a state machine are labeled with action names. An FSP process consists of one or more local processes separated by commas. A local process can be a primitive local process (END, STOP, ERROR, a reference to another local process), a sequential composition, a conditional process, or is defined using action prefix (‘\text{\textless}’ >’) and choice (‘\text{\textbar}’). For example, the following MAKER process manufactures an item

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr
and then signals that the item is ready for use by a USER process:

\[
\text{MAKER} = (\text{make} \rightarrow \text{ready} \rightarrow \text{MAKER}).
\]

\[
\text{USER} = (\text{ready} \rightarrow \text{use} \rightarrow \text{USER}).
\]

MAKER and USER share the action \textit{ready}; they must execute it at the same time. The composite process $\parallel \text{MAKER}\_\text{USER} = (\text{MAKER} \parallel \text{USER})$ describes the model of a simple manufacturing system that consists of MAKER and USER. ‘$\parallel$’ refers to composition. A safety property process $P$ asserts that any trace including actions in the alphabet of $P$ is accepted by $P$. FLTL assertions are formed by applying temporal operators to fluent expressions. They specify the desired properties that are true for every possible execution of a system. Fluent expressions can be constructed by applying normal logical operators (conjunction, disjunction, negation, implication, and equivalence) to fluents.

To check an aspect-oriented model, aspect models are first woven into their base class models (as shown in Figure 3). This results in woven models that capture the integrated behaviors of aspects and classes. Then the woven models and non-base class models (i.e. class models not modified by the aspects) are converted into respective FSP behavior processes. Meanwhile, the required properties are formalized as LTSA safety processes and/or FLTL assertions. Finally, all behavior and safety processes are combined into a composite process and feed the resultant process into LTSA. LTSA then verifies whether or not the properties are violated. If violated, LTSA reports a trace to property violation (i.e. counterexample). Each trace is a sequence of events (or state transitions) starting from the initial state.

The aspect weaver relies on the semantics of aspect models, which can be informally described as follows: inter-model declarations introduce new transitions, states, and events to base models. State and transition pointcuts are a naming mechanism for mapping states/events in advice models to the counterparts selected from base models by pointcuts. The selected transitions are then replaced with corresponding transitions in the advice models. If an aspect crosscuts multiple classes, it is woven into each of the base classes. Let ‘←’ be the assignment operator, $M.S$, $M.E$ and $M.T$ be the sets of states, events, and transitions of state model $M$, respectively.

**Definition 6**

Given base model $BM$ and aspect model $A = (ID, SP, TP, AM)$. The woven state model, $WM$, of composing aspect $A$ into base model $BM$ results from the following procedure:

1. Initially, $WM ← BM$.
2. For each inter-model declaration in $ID$ that is defined on $BM$, add each new transition into $WM.T$. If states (or events) used in the new transitions are not yet in $WM.S$ (or $WM.E$), add them into $WM.S$ (or $WM.E$).
(3) For each advice model in $AM$ that involves integrated classes, combine the transitions that use states and events of integrated classes into composite events (leaving out the states of integrated classes). Let $AM'$ denote the new set of advice models.

(4) For each transition pointcut in $TP$, replace each transition in $WM.T$ picked out by the pointcut with the corresponding advice model in $AM'$. If the advice model uses a state variable defined by some state pointcut in $SP$, then replace the state variable with the corresponding state in $WM.S$ according to the state pointcut.

A woven model can further be composed with other aspect models for the same base. The order in which multiple aspects are applied is determined by the aspect precedence relation. More details on the woven models and model checking can be found in the previous publication [16]. Correctness of the transformation from class and woven models to FSP is guaranteed because of the equivalent semantics of transitions in the state models and local processes in FSP.

3. TESTING ASPECT-ORIENTED PROGRAMS AGAINST STATE MODELS

Figure 4 shows the process for generating test code of aspect-oriented programs from their state models. Given an aspect-oriented state model, the woven models are created by the aspect weaver. Woven models are like class state models as described before. Then a transition tree is generated from a woven model through either a property-oriented or a structure-oriented test generator. Each path from the root to a leaf node in the transition tree indicates an abstract test sequence. If needed, the user can edit the transition tree, for instance, by providing actual parameters for the object construction and method invocations to form concrete test sequences. Finally, test code including test classes and wrapper aspects is generated from the transition tree. The test code and the aspect-oriented program under test form an executable whole. In this test generation process, all activities except for the editing of transition trees are automated (In addition, model mutation for model-checking with LTSA also requires some manual work. This will be discussed in Section 5.2).
Tests generated from the woven models can target aspect faults because the base classes can be first tested by the tests generated from the class models. If the base classes pass all of the state-based tests but the aspect-oriented program as a whole fails some of the tests, the failure would have to do with aspects [17]. It is worth pointing out that the existence of one-on-one relation between aspect modeling and implementation is not required. One aspect model can be corresponding to an implementation with multiple aspects. For example, the aspect-oriented cruise control implementation has two aspects, CarSimulatorFix and CruiseControlIntegrator, sharing the base class CarSimulator. The CarSimulator aspect enforces the precondition of engineOn to deal with a safety problem, whereas the CruiseControlIntegration aspect introduces new behaviors to all methods in CarSimulator. It is feasible to build one aspect model, say CruiseControlAspect, that covers the behaviors of both CarSimulatorFix and CruiseControlIntegrator. In this case, the two implementation aspects can be tested together with the base class CarSimulator by using the aspect-oriented state model \( \langle \{ \text{CarSimulator} \}, \{ \text{CruiseControl Aspect} \}, \{ \} \rangle \). Likewise, a complex implementation-level aspect may be specified by multiple aspect models. In this case, these aspect models need to be woven together before the tests are generated. In the cruise control experiment, the models for the CarSimulatorFix and CruiseControlIntegrator aspects were built separately. However, a new version of the cruise control program can be created by combining the code of the CarSimulatorFix and CruiseControlIntegrator aspects in the original version into one implementation aspect, say CruiseControlAspect. The implementation aspect CruiseControlAspect can be tested through the base class CarSimulator by using the aspect-oriented state model \( \langle \{ \text{CarSimulator} \}, \{ \text{CarSimulator Fix, CruiseControl Integrator} \}, \{ \} \rangle \).

A structure-oriented testing strategy generates tests directly from a woven or class model to meet the given coverage criterion (state coverage, transition coverage, or round-trip). A test suite is said to achieve the state (or transition) coverage if it covers each of the states (or transitions) at least once. A round-trip test suite consists of a set of test sequences such that the final object state of each sequence has occurred in some other sequence.

A property-oriented testing strategy creates tests from counterexamples of model checking. Note that model checking with LTSA has two inputs: behavior processes and system properties. Two strategies for testing through model checking have been developed: (a) checking with trap properties and (b) checking of model mutants. The former employs the correct model (processes) but incorrect FLTL properties. Counterexamples of trap properties are legal sequences of events that should be allowed to occur in the correct model. These counterexamples are used to test whether or not the implementation fulfills these behaviors. The latter uses incorrect models (processes) but correct properties (FLTL assertions and safety processes). Counterexamples from model mutants can be illegal sequences of events that should not occur in the correct model. These counterexamples are used to test whether or not the implementation allows such abnormal behaviors.

As shown in Figure 4, all property-oriented and structure-oriented strategies share the test generation infrastructure—transition trees. This facilitates test management (e.g. for providing actual parameters of object construction and method invocations and merging different test trees) and test code generation. The root of a transition tree represents the initial state; a child node contains the resultant state and the event of a transition from the state of the parent node. Each node is uniquely indexed by its location in the tree. As the root is not displayed, the root’s immediate children containing the initial states and constructors (called initial state nodes) would look like roots (called visible roots). Figure 5 shows a portion of a transition tree for the counterexamples.
of trap properties in the cruise control example. The node $1^\text{new} \rightarrow \text{OFF00}$ is the initial state node or visible root. $e \rightarrow S$ in each node means that $S$ is the resultant state of event $e$ of the transition from the parent state (the corresponding postcondition is displayed in a separate area). In a transition tree, each path from the initial state node to a leaf represents an abstract test sequence. It becomes a concrete test sequence when the required parameters of all object construction and method invocations are determined. In Figure 5, for example, the path, $1^\text{new} \rightarrow \text{OFF00}$, $1.1^\text{engineOn} \rightarrow \text{ON00}$, $1.1.3^\text{brake} \rightarrow \text{ON01}$, $1.1.3.1^\text{engineOff} \rightarrow \text{OFF00}$ (or simply $1.1.3.1^\text{new, engineOn, brake, engineOff}$), represents a test sequence. It is a concrete test sequence because $\text{new}$, $\text{engineOn}$, $\text{brake}$, and $\text{engineOff}$ require no parameters. Sections 4 and 5 will elaborate on transition tree generation for each of the test generation strategies.

Before generating test code from a transition tree, the user should provide the actual parameters, if required, of the object construction and method invocations and insert Java code into the sequences for test initialization and cleanup. In the cruise control experiment, for example, a number of tests need to reach the \textit{Cruising} state through the \textit{on} event. However, it requires a minimum car speed. To achieve this condition, the statement $\text{setCarSpeed}(\text{MinCruisingSpeed})$ can be inserted into the nodes. This statement will be executed before the $\text{on}$ event.

When generating test code, a test method is created for each test sequence from an initial state node to a leaf. Each event in a test sequence becomes an invocation to the corresponding constructor or method. The states and postconditions in the sequence are transformed into assertions following the constructor/method calls. For example, the method generated for the sequence $1.1.3.1^\text{new, engineOn, brake, engineOff}$ in Figure 5 is as follows:

```java
private static void modelTest11.1.3.1() {
    CarSimulator carsimulator = new CarSimulator();
    assertState (carSimulator, CarSimulator.OFF00);
    assertState(carSimulator.getCruiseController(), CruiseController.INACTIVE);
    carsimulator.engineOn();
    assertState(carSimulator, CarSimulator.ON00);
    assertState(carSimulator.getCruiseController(), CruiseController.ACTIVE);
    carsimulator.brake();
    assertState(carSimulator, CarSimulator.ON01);
    carsimulator.engineOff();
    assertState(carSimulator, CarSimulator.OFF00);
    assertState(carSimulator.getCruiseController(), CruiseController.INACTIVE);
}
```

$\text{assertState(Object, State)}$ is a method provided by the test execution infrastructure. It retrieves the actual state of $\text{Object}$ (e.g. $\text{carSimulator}$) and verifies whether or not it is equivalent to the model-level $\text{State}$ (e.g. $\text{CarSimulator.OFF00}$). Note that the named constant $\text{CarSimulator.OFF00}$ in the above test code is not defined in the $\text{CarSimulator}$ class. This issue is resolved by automated generation of a state-wrapper aspect that introduces the definitions of model-level states at runtime. Once all test methods for the entire tree are created, they are wrapped up into a test class and the main method is created by including an invocation to each test method. The test class, wrapper aspects, and the aspect-oriented program together form an executable whole.

The state wrapper aspect is generated according to the state specifications in the state model. For example, the following code snippet shows a portion of the wrapper aspect for $\text{CarSimulator}$. It is generated automatically from the state specifications described in Subsection 2.1.

```java
public aspect CarSimulatorUnderTest {
    declare parents: CarSimulator implements stateBasedClass;
    public static final String CarSimulator.OFF00="OFF00";
    public static final String CarSimulator.ON00="ON00";
    ...
```
public String CarSimulator.getModelState()
{
    if(!getIgnition() && getThrottle()==0 && getBrakepedal()==0)
    {
        return OFF00; 
    }
    else if(getIgnition() && getThrottle()==0 && getBrakepedal()==0)
    {
        return ON00; 
    }
    else ...
}

In general, the generated wrapper aspect includes three parts. First, it makes the base class through which aspects are tested to implement the interface StateBasedClass via the declare parents statement. A declare parents statement in AspectJ is used 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 test code. Second, the wrapper aspect defines the model-level states as named 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 test code, such as assertState(carsimulator, CarSimulator.OFF00). Third, the wrapper aspect defines the getModelState method for the class via an inter-type declaration. It maps each concrete object state to a model-level state. This makes it possible for the assertState method to compare concrete object states to the expected model-level states and report the pass/fail verdict of test execution. As such, wrapper aspects serve as a mechanism of runtime code instrumentation—they introduce new elements (named constants and methods) to classes but do not modify the source code. The generation of wrapper aspects works for all the state models in MACT because they are deterministic - states in a given state model are mutually exclusive (each object has only one state at a particular point of time) and one state will not be led to multiple different states by the same event and precondition. As a result, if-then-else statements can be used to determine the current model-level object state according to the state mappings.

4. STRUCTURE-ORIENTED ASPECT TEST GENERATION

This section describes how to generate the transition tree for a given structural coverage. Suppose \( \langle S, T, s_0 \rangle \) is a woven model or class model, where \( S \) is the set of states, \( T \) is the set of transition, and \( s_0 \) is the initial state. Each node in a transition tree consists of the current state, the event and precondition achieving the current state, the postcondition, the parent node, a list of child nodes. The state of the root node always represents the \( x \) state. The transition tree generation first creates the initial state node (as a child of the root) according to the object construction transition \((x, \text{new}[p_0, q_0], s_0)\). The initial state node is then expanded when the state model is traversed to meet the given coverage criterion.

Algorithm 1: Generate transition tree for the state (transition) coverage of a state model.
Input: state model \( \langle S, T, s_0 \rangle \), where object construction transition is \((x, \text{new}[p_0, q_0], s_0)\)
Output: transition tree (or simply the root node)
Declare: root, newNode, currentNode are nodes notCovered is the list of states(or transitions) that are currently not covered by the transition tree queue is a queue of nodes

1. begin
2. root.state \( \leftarrow x; \)
3. newNode.state \( \leftarrow s_0; \) newNode.event \( \leftarrow \text{new}; \) newNode.precondition \( \leftarrow p_0; \)
4. newNode.postcondition \( \leftarrow q_0; \)
5. newNode.parent \( \leftarrow \text{root}; \) root.children \( \leftarrow \emptyset; \)
6. queue \( \leftarrow \{\text{newNode}\}; \)
7. notCovered \( \leftarrow S \) (or \( T \));
8. while notCovered \( \neq \emptyset \) do
9. currentNode \( \leftarrow \) first node in queue;
for each transition \((currentNode.state, e\{p, q\}, s_j) \in T\), do
remove \(s\) (or the transition) from notCovered
newNode.state \(\leftarrow s_j\); newNode.event \(\leftarrow e\); newNode.precondition \(\leftarrow p\);
newNode.postcondition \(\leftarrow q\);
newNode.parent \(\leftarrow currentNode\); add newNode to currentNode.children;
add newNode to queue
end for
end while
return root
end

Algorithm 1 describes how the transition tree is generated to cover all the states (or transitions) of a state model using the breadth-first search. It first creates the initial state node and puts the new node into the queue for expansion (lines 3–5). To expand a node, it first retrieves each transition that starts with the state in the current node (line 9) and removes the resultant state (or transition) from the list of uncovered states (or transitions) (line 10). Then it creates a new node for this transition (lines 11–12), which is put into the queue for potential expansion (line 13). Whether it is expended depends on whether the states (or transitions) are already covered before it is selected from the queue (line 7).

Algorithm 2: Generate round-trip transition tree.
Input: state model \(\langle S, T, s_0 \rangle\), where object construction transition is \((x, new[p_0, q_0], s_0)\)
Output: transition tree (or simply the root node)
Declare: root, newNode, currentNode are nodes
queue is a queue of nodes

begin
root.state \(\leftarrow x\);
n newNode.state \(\leftarrow s_0\); newNode.event \(\leftarrow new\); newNode.precondition \(\leftarrow p_0\);
newNode.postcondition \(\leftarrow q_0\);
n newNode.parent \(\leftarrow root\); root.children \(\leftarrow \emptyset\);
queue \(\leftarrow \{newNode\}\);

while queue \(\neq \emptyset\) do
 currentsNode \(\leftarrow\) first node in queue;

for each event \(e\)

for each transition \((currentNode.state, e\{p_l, q_l\}, s_j) \in T\), do

newNode.state \(\leftarrow s_j\); newNode.event \(\leftarrow e\); newNode.precondition \(\leftarrow p_l\);
newNode.postcondition \(\leftarrow q_l\);
newNode.parent \(\leftarrow currentNode\); add newNode to currentNode.children;
if the current tree has no node whose state equals \(s_j\) and postcondition equals \(q_l\)
add newNode to queue
end if
end for

negCond \(\leftarrow \neg(p_1 \lor p_2 \ldots \lor p_n)\)  
\(p_i\) is the precondition of each transition \((currentNode.state, e\{p_i, q_i\}, s_j)\)
if negCond \(\neq false\)
n newNode.state \(\leftarrow s_j\); newNode.event \(\leftarrow e\);
n newNode.precondition \(\leftarrow\) negCond; newNode.postcondition \(\leftarrow null\);
newNode.parent \(\leftarrow currentNode\); add newNode to currentNode.children;
end if
end for
end while
return root
end
The round-trip strategy is to generate test sequences from a state model such that the resultant object state of each sequence has occurred at least once in some other sequence. Algorithm 2 describes the generation of the round-trip tree using the breadth-first search. It first creates the initial state node and puts the new node into the queue for expansion (lines 3–5). To expand a node (line 7), it first retrieves each transition for each event that starts with the state in the current node (lines 8–9). Then it creates a new node for this transition (line 10–11). This node is added to the queue for expansion if the current tree does not have any node with the same state and postcondition (lines 12–14). For all these transitions (preconditions are $p_1, p_2, \ldots, p_n$), if $\neg(p_1 \lor p_2 \lor \ldots \lor p_n)$ is not false, then a dirty test node is created (lines 16–21). This algorithm extends the traditional [6] and the adapted [17] round-trip strategies by including postconditions as part of the termination condition of tree expansion (line 12). As postconditions can represent inter-object state invariants and aspects often involve interaction between multiple objects, the extension is critical for detection of aspect faults. Consider the state model of the individual class CarSimulator shown in Figure 1. When a CarSimulator object is at the state ON10, an invocation to the method brake changes the object state to ON00. In the aspect-oriented program, however, brake also changes the state of a Controller object to Standby when its current state is Cruising. This is because the aspect CruiseControlIntegrator intercepts the call to the brake method of the CarSimulator object and then invokes the brake method of the CruiseController object (refer to CruiseControlIntegrator’s state model is shown Figure 2). Such an effect required of the CruiseControlIntegrator aspect is captured by the postcondition CruiseController.Standby of the transition in the woven model of CarSimulator and CruiseControlIntegrator. When a faulty implementation of CruiseControlIntegrator does not achieve this postcondition, the fault cannot be revealed by merely checking whether or not the state ON00 is reached. The CarSimulator object being at the ON00 state does not imply that the CruiseController object is at the state Standby. For example, ON00 can be achieved by accelerator at ON01.

Given coverage criteria A and B, A is said to subsume B if a test suite that achieves coverage A always achieves coverage B. Suppose all states in each state model are reachable from the initial state. The round-trip coverage subsumes the transition coverage because the round-trip test suite must cover each transition. The transition coverage subsumes the state coverage because each state must appear in some transition. Suppose $m$ and $n$ are the number of states and transitions in a state model, 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 $O(m^2 \times n)$ [6].

5. PROPERTY-ORIENTED ASPECT TEST GENERATION

This section presents the two property-oriented strategies for test generation through model checking. To formalize properties for testing through model-checking, the system requirements that the state models must satisfy are enumerated and informally described. Each requirement is then represented as an LFTL formula. Consider the following safety requirement in the cruise control simulation: when the cruise controller is in the Cruising state, it will be able to eventually stop cruising through events brake, accelerator, or off. This can be formalized as the following FLTL formula in LTSA: $\text{assert } [[\text{CRUISING} \Rightarrow <\!(\text{brake} \lor \text{accelerator} \lor \text{off})\!]]$, where ‘[’’, ‘’<\!’’, and ‘’||’’ means ‘always’, ‘eventually’, and ‘or’, respectively.

5.1. Test generation using trap properties

Test generation from counterexamples of model checking trap properties makes use of the correct aspect-oriented model but negated FLTL properties. A counterexample of a trap property is a sequence of events (state transitions) that violates the trap property in the correct model. It starts from the initial state and is a legal sequence that contributes to (or does not violate) the correct property. This sequence can thus be used to test whether or not the implementation fulfills the functionality. It is thus a positive test.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr
Algorithm 3: Generate transition tree by checking a state model against trap properties.
Input: state model \( \langle S, T, s_0 \rangle \), where object construction transition is \( (x, \text{new}[p_0, q_0], s_0) \)
\( P = \{p_1, \ldots, p_n\} \) is a set of properties (FLTL assertions)
Output: transition tree (or simply the root node)
Declare: counters is the set of counterexamples
\( \text{ltsa}(M, p) \) is the set of counterexamples generated by checking property \( p \) against model \( M \) with \( \text{ltsa} \) root, newNode, currentNode are nodes

1. Begin
2. root.state \( \leftarrow \) \( x \);
3. newNode.state \( \leftarrow s_0 \); newNode.event \( \leftarrow \) new; newNode.precondition \( \leftarrow p_0 \);
   newNode.postcondition \( \leftarrow q_0 \);
4. newNode.parent \( \leftarrow \) root; root.children \( \leftarrow \emptyset \);
5. counters \( \leftarrow \emptyset \)
6. for each \( p_i \) in \( P \)
7.   counters \( \leftarrow \) counters \( \cup \) \( \text{ltsa}(M, \neg p) \);
8. end for
9. for each counterexample \( \langle s_0, e_1[p_1, q_1], s_1 \rangle, \ldots, \langle s_k-1, e_k[p_k, q_k], s_k \rangle \) in counters
10. currentNode \( \leftarrow \) the initial state node;
11. for \( i = 1 \) to \( k \)
12.   if there is a child of currentNode whose state equals \( s_i \), event equals \( e_i \),
       precondition equals \( p_i \)
13.     currentNode \( \leftarrow \) the found node
14.   else
15.     newNode.state \( \leftarrow s_i \); newNode.event \( \leftarrow e_i \);
16.     newNode.precondition \( \leftarrow p_i \); newNode.postcondition \( \leftarrow q_i \);
17.     newNode.parent \( \leftarrow \) currentNode; add newNode to the children of currentNode;
18.     currentNode \( \leftarrow \) newNode
19. end if
20. end for
21. end for
22. return root
23. end

Algorithm 3 describes how to generate the transition tree from the counterexamples of checking trap properties against a state model. Given a woven or class model and a set of system properties, it first creates the initial state node (lines 3–4). For each system property \( p \), it first invokes LTSA to check its trap property \( \neg p \) against the given model and collects all the counterexamples (lines 5–8). Each counterexample is converted into a sequence of nodes and then combined into the tree (lines 9–21). Consider the following counterexample of a trap property in the cruise control example:

\( (\text{OFF00}, \text{CarSimulator.engineOn, ON00}), (\text{ON00}, \text{CarSimulator.brake, ON01}), (\text{ON01}, \text{CarSimulator.engineOff, OFF00}) \)

It is converted into the path 1.1.3.1 in Figure 5. Consequently, test code can be generated from the transition tree.

5.2. Test generation using model mutants

A model mutant is an incorrect variation of a model. Test generation from counterexamples of checking model mutants makes use of model mutants and system properties. The philosophy of this testing strategy is as follows: a counterexample generated from a model mutant is a sequence of state transitions in the mutant that violates some required system property. It indicates a particular way that the aspect-oriented system may be designed or modeled incorrectly. The implementation likely makes the same error as indicated by the counterexample.
Note that some events in a counterexample can be illegal in the correct design because, for example, a model mutant may have incorrect or extra transitions. Such counterexamples are used to test whether or not the implementation would actually allow such normal or abnormal sequences.

Algorithm 4: Generate transition tree by checking system properties against model mutants.
Input: state model $M = \langle S, T, s_0 \rangle$, where object construction transition is $\langle x, new[p_0, q_0], s_0 \rangle$

$P = \{ p_1, \ldots, p_n \}$ is a set of properties (FLTL assertions)

$\{ M_1, \ldots, M_m \}$ is a set of model mutants

Output: transition tree (or simply the root node)

Declare: counters is the set of counterexamples

$ltsa(M, p)$ is the set of counterexamples generated by checking property $p$ against model $M$ with ltsa root, newNode, currentNode are nodes

1. Begin
2. root.state $\leftarrow x$;
3. newNode.state $\leftarrow s_0$; newNode.event $\leftarrow$ new; newNode.precondition $\leftarrow p_0$;
   newNode.postcondition $\leftarrow q_0$;
4. newNode.parent $\leftarrow$ root; root.children $\leftarrow \emptyset$;
5. counters $\leftarrow \emptyset$;
6. for each $M_j$
7.   for each $p_j$ in $P$
8.     counters $\leftarrow$ counters $\cup$ ltsa($M_j, p_j$);
9.   end for
10. end for
11. for each counterexample \( \langle s_0, e_1[p_1, q_1], s_1 \rangle, \ldots, \langle s_{k-1}, e_k[p_k, q_k], s_k \rangle \) in counters
12. currentNode $\leftarrow$ the initial state node;
13. for $i = 1$ to $k$
14.   if $\langle s_{i-1}, e_i[p_i, q_i], s_i \rangle$ is not a transition in the correct model
15.      newNode.state $\leftarrow$ currentNode.state; newNode.event $\leftarrow e_i$;
16.      newNode.precondition $\leftarrow p_i$; newNode.postcondition $\leftarrow$ null;
17.      newNode.parent $\leftarrow$ currentNode; add newNode to the children of currentNode;
18.     break (exit the for loop);
19.   else
20.      if there is a child of currentNode whose state is $s_i$, event is $e_i$, precondition is $p_i$
21.         currentNode $\leftarrow$ the found node
22.      else
23.         newNode.state $\leftarrow s_i$; newNode.event $\leftarrow e_i$;
24.         newNode.precondition $\leftarrow p_i$; newNode.postcondition $\leftarrow q_i$;
25.         newNode.parent $\leftarrow$ currentNode; add newNode to the children of currentNode;
26.         currentNode $\leftarrow$ newNode
27.   end if
28. end for
29. end for
30. return root
31. end

Algorithm 4 describes how to generate the transition tree from the counterexamples of checking system properties against model mutants. It first creates the initial state node (lines 3–4). For each model mutant and each system property, it invokes LTSA to check the property against the model mutant and collects all the counterexamples (lines 5–10). Each counterexample is converted into a sequence of nodes and combined into the tree (lines 11–29). If a transition in a counterexample is not a legal transition in the correct model, a dirty test node is generated (lines 14–18), otherwise the conversion (lines 20–27) is the same as that in Algorithm 3 (line 12–19).
A major issue for test generation from model mutants is the systematic creation of model mutants. In this paper, model mutants were created in terms of the following seven types of aspect design faults (DF):

(DF1) Incorrect pointcut with a missing join point. The desired advice is not applied to the join point;
(DF2) Incorrect pointcut with an extra join point. The advice is unexpectedly applied to the extra join point;
(DF3) Incorrect advice (or inter-model declaration) where a transition has a wrong event or starting (ending) state;
(DF4) Incorrect advice with a missing transition;
(DF5) Incorrect advice with an extra transition;
(DF6) Incorrect advice with a missing or extra precondition for a transition;
(DF7) Incorrect aspect precedence or advice order.

These fault types provide a comprehensive coverage of state-based aspect design problems. They cover incorrect pointcuts, inter-model declarations, advice, and aspect precedence. Note that model mutants have to be syntactically correct and pass the compiler of the aspect-oriented state models. For a given aspect-oriented model, MACT automatically generates its model mutants with respect to the fault types DF1, DF4, DF6 (with missing precondition), and DF7. This is done by modifying the given model with the mutation operators of these fault types. For instance, TJPR (transition join point remover) is a DF1 mutation operator for creating a mutant with a missing transition join point by removing one transition join point from the given model. For other fault types, model mutants are created by walking through the correct model carefully and seeding one reasonable fault at each feasible spot. The future work plans to develop mutation operators for these fault types.

The fault model above is comparable to the one proposed by Ferrari et al. [19]. While they are defined for aspect-oriented models and programs, respectively, both have covered aspect faults with respect to pointcuts, inter-type declarations, and advice. The former has also considered faults with respect to aspect precedence, whereas the latter has considered faults with respect to base classes. In addition, the latter can be used to enhance future experiments due to the more refined lists of aspect faults.

6. TOOL IMPLEMENTATION

MACT has been implemented in Java. The current implementation has about 23 K lines of non-comment code. MACT accepts textual specifications of class models, aspect models, and project descriptions of aspect-oriented systems. Figure 6 is a screenshot of the MACT interface for project management, which presents part of a generated textual woven model in the cruise control project (the keyword Followedby means postcondition). The open source UML tool ArgoUML (http://argouml.tigris.org) has been extended for the graphical notation of aspect modeling with state machines. This extended ArgoUML can convert the graphical representations of aspect and class state models into the input of MACT.

To automate the generation of counterexamples for all trap properties and model mutants, substantial effort has been devoted to understanding and modifying the LTSA source code and testing the modifications. The main modifications to LTSA include: (a) generating and collecting multiple counterexamples per property, if present. The modified LTSA excludes duplicate or subsumed counterexamples and allows the user to configure the maximum counterexamples per property; (b) batch-checking all FLTL properties. The original LTSA allows the user to check one FLTL property at a time. The modified version can check all FLTL properties without human intervention; (c) creating and checking trap properties so as to produce counterexamples. The modified version creates the negated version of each FLTL property and verifies the trap property against the behavior processes.

MACT also provides various features for the management of projects and test data. Figure 7 shows the interface for transition tree management, where the popup menu lists the operations on
the transition tree. The transition tree (on the left) together with the user-provided test data and code (on the right) can be exported into a data file (it can be loaded later). The user can add, copy, and remove a node from a generated transition tree. This paper will not elaborate on these functions.

7. EMPIRICAL STUDY

This section presents the experiments, analyzes the results, and reports applications to larger programs.
TESTING AOP WITH FINITE STATE MACHINES

<table>
<thead>
<tr>
<th>Subjects</th>
<th>Cruise control</th>
<th>Telecom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code (LOC)</td>
<td>854</td>
<td>731</td>
</tr>
<tr>
<td>No. of classes</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>LOC</td>
<td>731</td>
<td>590</td>
</tr>
<tr>
<td>No. of implementation aspects</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>No. of aspect models</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>LOC</td>
<td>123</td>
<td>141</td>
</tr>
<tr>
<td>Formalized system properties</td>
<td>63</td>
<td>32</td>
</tr>
<tr>
<td>Number of model mutants</td>
<td>27</td>
<td>13</td>
</tr>
<tr>
<td>Number of program mutants</td>
<td>56</td>
<td>35</td>
</tr>
</tbody>
</table>

7.1. Experiments

The purpose of the empirical evaluations is to gain insights into the structure-oriented and property-oriented test generation strategies with respect to their cost-effectiveness. The main tasks include building the state models of classes, building aspect models, formalizing and checking a comprehensive set of system properties, creating model mutants, creating program mutants, generating executable tests for each testing strategy, and testing each of the program mutants with the generated tests. Table I lists the metrics of the two subjects (cruise control and telecom\(^1\)) for the comparative study. Aspects and related classes form units of modeling, checking, and testing. Each subject has three such units. The underlying logic of the cruise control system is much more complex than telecom according to their aspect-oriented state models, number of formalized requirements, number of model mutants, number of program mutants, etc. The model mutants were created according to the fault model of aspect design in Section 5.2. Here, the model mutants exclude those that produce no counterexamples. The computers used to conduct the experiments are desktop PCs (Intel Pentium 4, 2.6 GHz, 1 GB RAM) running Windows XP.

In principle, the aspect-oriented state models of a system can be built according to the system requirements before the system is implemented. In the experiments, however, the subject programs already existed before the aspect-oriented state models were created. Each existing program is viewed as the correct implementation of the application and there was little documentation on system requirements or design. An aspect model was built for each implementation aspect through reverse engineering. To do so, the base classes of each implementation aspect were first identified and modeled with state machines. The model of each aspect was created by the following process. First, the crosscutting behavior and requirements of the aspect were described informally. Second, the new states, events, and transitions that the aspect would introduce to each base class were determined. These states, events, and transitions were represented as inter-model declarations. Third, the join points (states and transitions) in the base class models affected by the aspect were identified. These joint points were grouped and then represented as pointcuts. Fourth, the behavior of the advice for each pointcut was identified and modeled as a finite state machine. This state machine became the advice model for the pointcut. Finally, the precedence relation was determined if the aspect shared join points with other aspects.

For each subject program, the conformance between the class/aspect models and the existing correct program was verified as follows: first, when class/aspect models were built and system properties were formalized, they were checked by LTSA such that the models would not violate any property. This assured that the models were correct with respect to the system properties. Second, test cases were generated from class and aspect models with each of the testing strategies. Third, the test cases were used to test the corresponding class/aspect code in the correct implementation. If no test failed, the models were considered conforming to the correct implementation and all

\(^1\)http://www.eclipse.org/ajdt/.

tests generated from the models were considered valid for the correct implementation; otherwise the models and property formulas were modified until all tests generated from them had passed. The same valid tests for the correct implementation were then applied to each of the program mutants.

The mutants of aspect programs were created according to the following programming faults (PF) of aspects:

(PF1) Missing pointcut or incorrect pointcut missing join points. An incorrect pointcut may miss a join point for various reasons, such as misspelling method/constructor signature and using non-‘*’ rather than ‘*’.

(PF2) Pointcut picking out extra join points. An incorrect pointcut may pick out extra join points, for example, due to a wrong ‘*’ or an extra OR (||) component.

(PF3) Pointcut with incorrect focus of control flow, such as missing !within or !cflow.

(PF4) Incorrect advice type, such as mistaking an after (or before) type as a before (or after) type.

(PF5) Incorrect advice that fails to enforce precondition, for example, due to missing or wrong statements (typically if) for precondition validation.

(PF6) Incorrect advice that fails to establish postcondition or intra-class state invariant, for example, due to missing or incorrect state update of base class object.

(PF7) Incorrect advice that fails to preserve inter-class state invariant, for example, due to missing or incorrect state update for base class object or missing an update message to an integrated object.

(PF8) Incorrect effects of inter-type declarations.

(PF9) Incorrect precedence of interfering aspects.

The above list provides a comprehensive coverage of the likely faults in aspect code: incorrect pointcut, advice, inter-type declaration, and aspect precedence. In the experiments, the program mutants were created by walking through each of the aspects and modifying the code to inject a fault when feasible. They covered all fault types. Although the work on automated mutation of aspect-oriented programs has emerged recently [19–21], no mature tools were found when the experiments were started.

The following is a mutant of aspect SpeedControlIntegrator, obtained by commenting one line of code in the description of pointcut cruiseBAO: execution(void accelerator()) ||. The mutant does not pick out the join point: execution(void accelerator()).

```
public aspect SpeedControlIntegrator {
  ...
  pointcut cruiseBAO(CruiseController controller): target(controller) &&
  (execution(void brake()) ||
  // execution(void accelerator()) ||    //missing join point
  execution(void off()));

  before(CruiseController controller) : cruiseBAO(controller) {
    if (controller.getState() == CruiseController.CRUISING) {
      getSpeedControl(controller).disableControl();
    }
  }
  ...
}
```

This mutant can be killed by the following test:

```
Controller controller = new Controller();
controller.engineOn();
...
controller.accelerator()
...
controller.on();
...
controller.accelerator();
assertState(controller, Controller.STANDBY);
assertState(controller.getSpeedControl(), SpeedControl.D0);
...

The correct aspect should lead the speedControl object (i.e. controller.getSpeedControl()) to the state D0. This is verified by the assertion assertState(controller.getSpeedControl(), SpeedControl.D0);. However, the aspect mutant will result in the state SpeedControl.E1 (the state before the invocation controller.accelerator());. When controller.accelerator() is executed, the advice is not applied because the incorrect pointcut in the aspect mutant does not pick out the join point controller.accelerator(). Thus getSpeedControl(controller).disableControl();in the advice is not called to update the state.

7.2. Results

Tables II and III show the accumulative results of all aspect testing units in the two subjects, respectively. AS, AT, and RT denote the structure-oriented test generation strategies for the all-states, all-transitions, and round-trip coverage, respectively; TP refers to the strategy of test generation from counterexamples using trap properties. MM denotes the strategy of test generation from counterexamples using model mutants. For each aspect testing unit in a subject, the five test generation strategies were applied to the same state model. MM and TP also use the same FLTL properties. The test generation time (in milliseconds) means the time for generating tests from counterexamples and/or woven models. The counterexample generation time (in seconds) refers to the time for producing and collecting the counterexamples by model checking all model mutants or all trap properties. It does not apply to the structured-oriented strategies. The LOC of generated test code, generated test methods, and the test generation time do not include those for testing individual classes or class clusters, which is also an important part of the experiments.

<table>
<thead>
<tr>
<th>Strategies</th>
<th>Structure-oriented</th>
<th>Property-oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC of generated test code</td>
<td>AS</td>
<td>AT</td>
</tr>
<tr>
<td>No. of generated test methods</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>Test generation time (ms)</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>Counterexample generation time (s)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>No. of killed program mutants</td>
<td>17</td>
<td>25</td>
</tr>
<tr>
<td>% of killed program mutants</td>
<td>30</td>
<td>45</td>
</tr>
</tbody>
</table>

Table II. Experimental results for cruise control.

<table>
<thead>
<tr>
<th>Strategies</th>
<th>Structure-oriented</th>
<th>Property-oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC of generated test code</td>
<td>AS</td>
<td>AT</td>
</tr>
<tr>
<td>No. of generated test methods</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Test generation time (ms)</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Counterexample generation time (s)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>No. of killed program mutants</td>
<td>21</td>
<td>27</td>
</tr>
<tr>
<td>% of killed program mutants</td>
<td>60</td>
<td>77</td>
</tr>
</tbody>
</table>

Table III. Experimental results for telecom.
The findings from the experiments are summarized below:

(I) **Comparing the structure-oriented strategies**

The three strategies were all fast in test generation. AS and AT can be very poor at fault detection (only 30 and 45% for the cruise control) and RT is the most effective one. Although RT generated the most test code, the overhead of test generation and test execution was not significant because test generation and execution was fully automated in the experiments. As RT subsumes AS and AT, RT will be used as the representative of the structure-oriented strategies when compared to the property-oriented strategies.

(II) **Comparing the property-oriented strategies**

Both TP and MM can be powerful for fault detection—TP killed 83% of the telecom mutants and MM killed 79% of the cruise control mutants. They also complemented each other—one killed some program mutants that were not killed by the other, and vice versa. TP outperformed MM in telecom, whereas MM was superior to TP in cruise control. However, MM was more expensive than TP. In both subjects, it took 10 times as much time to generate the counterexamples from the model mutants than from the trap properties. MM checked each model mutant against all FLTL properties, whereas TP only checked the correct model against the trap properties. Checking one model mutant against all FLTL properties (not including the properties in the form of safety processes) is roughly comparable to checking the correct model against all trap properties. MM also incurred costs for creating some model mutants by hand.

(III) **Comparing the property-oriented strategies with the structure-oriented strategies**

RT, TP, and MM complemented each other—RT killed some program mutants that were not killed by TP or MM (in telecom) and vice versa (in cruise control). Both TP and MM outperformed RT in the cruise control experiment. In addition to the overhead for generating counterexamples through model checking, TP and MM may require more manual work than RT. Table IV shows the major cost factors of such manual work for the test generation strategies. All strategies require formal modeling of the system under test. MM needs additional effort for model mutation although some model mutants are generated automatically. TP and MM also formally check the models for correctness before the models are used for test generation. While the checking process is automated, the systematic formalization of required properties is a daunting task.

On the other hand, model checking for correctness offers an important benefit for model-based testing: when the execution of a test generated by TP or MM fails, it indicates that the implementation under test is faulty. However, this is not necessarily true for those tests generated by a structure-oriented strategy unless the models are already proven correct. Strictly speaking, a structure-oriented strategy should also verify the models for correctness to avoid generating incorrect or meaningless tests. This can be done either formally or informally (thus a question mark is used in Table IV). If RT employs model checking for formal verification, then TP and MM are not significantly more expensive than RT.

(IV) **Combining the property-oriented and structure-oriented strategies**

The combination of property- and structure-oriented strategies can improve the fault detection capability. Table V shows that the combined strategies have killed 82 and 86% of the mutants for cruise control and telecom, respectively. The combination is more effective than

<table>
<thead>
<tr>
<th>Table IV. Major cost factors of test generation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing strategies</td>
</tr>
<tr>
<td>Cost factors</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>Formal modeling</td>
</tr>
<tr>
<td>Model mutation</td>
</tr>
<tr>
<td>Formal verification</td>
</tr>
</tbody>
</table>

Table V. Fault detection capabilities of the combined strategies.

<table>
<thead>
<tr>
<th></th>
<th>Cruise control</th>
<th>Telecom</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of killed program mutants by combined strategies</td>
<td>46</td>
<td>30</td>
</tr>
<tr>
<td>% of killed program mutants by combined strategies</td>
<td>82</td>
<td>86</td>
</tr>
<tr>
<td>No. of killed program mutants by all structure-oriented strategies</td>
<td>38</td>
<td>29</td>
</tr>
<tr>
<td>Improvement over structure-oriented strategies</td>
<td>21%</td>
<td>3.5%</td>
</tr>
<tr>
<td>No. of killed program mutants by all property-oriented strategies</td>
<td>44</td>
<td>28</td>
</tr>
<tr>
<td>Improvement over property-oriented strategies</td>
<td>4.5%</td>
<td>7%</td>
</tr>
</tbody>
</table>

Table VI. Larger applications.

<table>
<thead>
<tr>
<th>Application</th>
<th>LOC</th>
<th>Classes</th>
<th>Aspects</th>
<th>Aspects tested</th>
<th>Strategies applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banking*</td>
<td>27 K</td>
<td>282</td>
<td>12</td>
<td>2</td>
<td>RT, TP, MM</td>
</tr>
<tr>
<td>EJBCOMPONENTS†</td>
<td>8.2 K</td>
<td>52</td>
<td>14</td>
<td>2</td>
<td>RT</td>
</tr>
</tbody>
</table>

*http://www.manning.com/laddad/.

The combination of structure-oriented and property-oriented strategies has not killed all mutants, although mutants killed by each category have covered all types of aspect faults in the fault model. For the mutants that were never killed, there are three main reasons: first, the behaviors of the mutated code are not captured by state models (i.e., they are not corresponding to the states or state transitions in the state models). Second, the behaviors of the mutated code are relevant to the state models, however, the model-level states used as oracles cannot catch incorrect values of implementation-level state variables. Consider the ON01 state defined by getIgnition() && getThrottle()==0 && getBrakepedal()!=0. getBrakepedal()!=0 evaluates true regardless of whether getBrakepedal() is 2 or 3. Third, the behaviors of the mutated code are related to state transitions but the postconditions of the state transitions are inaccurate. For example, getBrakepedal()!=0 is inaccurate about the expected value of state variable brakepedal. It cannot tell the difference between getBrakepedal()==2 and getBrakepedal()==3 when brakepedal is updated incorrectly by an aspect.

7.3. Larger applications

In the preceding subsection, the model-based approach has been applied to the two subjects as a heavyweight formal method for modeling, verification, and test generation. It is a heavyweight method in that the state models of all aspects and related classes are built and many system requirements are formalized as temporal logic formulas. This is desirable for a meaningful comparative study on the cost-effectiveness of the test generation strategies. Of course, this is too expensive. In practice, the approach can be applied as a lightweight formal method and selectively use the test generation strategies. Table VI shows two larger programs to which MACT has been applied. For the banking system, the work focused on the modeling, checking, and testing of the aspects for concurrency control of read/write access to the database and related classes. RT, TP, and MM are combined to generate aspect tests, but only a small set of requirements are formalized and only the automatically generated model mutants for test generation are included. For EJBCOMPONENTS, the models of aspects and related classes are verified via informal inspection, rather than model checking. The structure-oriented strategies are also applied to the test generation for this system.

There are two lessons learned from these applications. First, the interactions between aspects and base classes in larger applications tend to be more complex. For example, an aspect (e.g. for tracing)
may crosscut many base classes. Building complete models of such aspects would require modeling of a large number of classes. To reduce the modeling and testing effort, aspects can be modeled by considering the interactions with some, not all, of the base classes. Second, the approach can be used to test those aspects that do not change the states, events, and state transitions of their base classes, but only affect the postconditions of transitions. For example, a logging aspect typically outputs data to somewhere (e.g., data file or console window). The expected logging effect can be defined as a transition postcondition (e.g., file update) if the data are written to a data file. However, it can be difficult to automatically verify the postcondition oracles if the logging data are directed to the console window. In addition, it can be difficult to define accurate oracles (expected logging data) because they are often omitted in the design documentation.

7.4. Threats to validity

This paper assumes that the classes and aspects in an aspect-oriented program can be modeled with finite state machines. This is similar to the existing work on testing object-oriented programs with state models, which is also based on the assumption that classes can be modeled with finite state machines [6, 7, 9, 11]. Typically, the state model of a class captures the state transitions resulted from invocations to the methods (or message sequences). For a non-modal class (e.g., DateTime) which has no constraints on message sequences [6], it may not make much sense to build the state model. In this case, testing with the state model is very limited. In the experiments, a state model was built for each aspect in the subject programs although, as discussed in Section 3, the existence of one-on-one relation between aspect modeling and implementation is not required. In principle, it is possible to directly define the woven model of an aspect and its base class without specifying the aspect model and then use this model for test generation purposes. However, an aspect model can make more clear the impacts of the aspect on the base class, particularly when it involves multiple methods and/or multiple base classes.

In this paper, the quantitative evaluation of the testing strategies was based on two small AspectJ programs—both have less than 1000 lines of code. For these small programs, we were able to conduct heavyweight analysis, such as building the state models of all aspects and related base classes, formalizing the properties, and creating the model and program mutants. The weakness, however, is that they cannot reflect how the testing strategies will scale up for large-scale projects. Most aspects in the existing large open-source AspectJ programs are limited to such functionality as logging and tracing. The benefit of modeling and testing such aspects with finite state machines is not obvious because they primarily affect the output to a file or the screen, rather than state transitions. With the advances in aspect-oriented software development, large-scale projects are expected to be used to further evaluate the cost-effectiveness of the testing strategies.

Other threats to validity include completeness of formalized properties, systematic creation of model mutants, and systematic creation of program mutants. The fault detection capabilities and costs of property-oriented test generation strategies depend on the formalized system properties. While the formalization of properties for the two case studies was highly comprehensive, additional requirements may exist. Testing through model mutation depends on the mutation analysis of aspect models. Evaluation of the fault detection capabilities of the testing strategies also relies on the mutation analysis of the Aspect programs. Manual creation of the model mutants and program mutants seldom guarantees systematic mutation analysis. In addition, the fault models of aspect modeling and programming may not capture all the faults in real-world aspect-oriented software development.

8. RELATED WORK

The related work primarily includes testing aspect-oriented programs and modeling and checking of aspect-oriented systems. Although there is a large body of literature on state-based testing [6–11] and testing via model checking [13, 14], this paper will not review this literature because it is not concerned with the testing of aspect-oriented programs.
proposed an approach to combining specification-based testing with model checking, where tests are generated to meet the coverage of the specification model and its complement. It is not concerned about aspect testing or testing with model mutation.

8.1. Testing aspect-oriented programs

The existing work on testing aspect-oriented programs has focused on the fault models of aspect-oriented programs, implementation-based unit or integration testing of aspects, regression test selection, infrastructure for test execution, and structure-oriented test generation from design models (e.g. state models and UML diagrams). Unit testing deals with pieces of advice within an aspect, whereas integration testing targets the interaction between aspects and classes. Empirical study on the fault detection capability of aspect testing methods remains to be seen, though [23].

Alexander et al. proposed a fault model, including six types of aspect faults: incorrect pointcuts, incorrect aspect precedence, failure to establish postconditions, failure to preserve state invariants, incorrect focus of control flow, and incorrect changes in control dependencies [5]. Ferrari et al. [19] developed a more refined model of aspect faults with respect to pointcuts, inter-type declarations, advice, and base classes. Based on this model, they defined a mutation operator for mutation testing of aspect-oriented programs. Chakravarthi [20] developed an experimental tool for mutation analysis of AspectJ programs. This tool can create faulty AspectJ projects by changing a pointcut or advice type. Delamarre et al. [21] developed a tool for the mutation analysis of AspectJ pointcut descriptors. Different from this work, this paper proposes a framework for aspect test generation from state models and uses mutation analysis to evaluate the effectiveness of the testing strategies.

Zhao proposed a data flow-based approach to unit testing of aspect-oriented programs [24]. For each aspect or class, the approach performs testing at the intra-module, inter-module, and intra-aspect/intra-class levels. Zhao and Rinard also explored system dependence graphs to capture the additional structures in aspect-oriented features such as join points, advice, and aspects [25]. Control flow graphs were constructed at system and module levels, and then test suites were derived from control flow graphs. To reduce testing cost, Zhou et al. introduced an algorithm based on control flow analysis for selecting relevant test cases [26]. Lemos et al. [27] investigated control and data flow structural testing criteria for aspect-oriented programs. Wedyan and Ghosh developed a tool for measuring joinpoint coverage from the advice and class perspectives [28]. The advice perspective of joinpoint coverage measures the execution of the advice at each joinpoint it is woven into, whereas the class perspective of joinpoint coverage measures the execution of all the advices in each joinpoint in the class. Xie et al. proposed a framework for generating test inputs for AspectJ programs, where a wrapper class was created for each base class under test [29]. Harman et al. recently developed an alternative approach to automated test data generation for aspect-oriented programs [30]. Xu and Rountev developed an approach to regression test selection for aspects based on the control flow graphs and interaction graphs of AspectJ programs [31]. In summary, the above works have focused on code-based testing issues, such as test input generation, mutation analysis, code coverage, and regression test selection. In comparison, this paper focuses on testing whether or not an aspect-oriented program meets the system requirements through a model-based approach.

Xu et al. presented an approach to testing aspect-oriented programs with UML design models [32]. In this approach, each aspect-oriented model consists of class diagrams, aspect diagrams, and sequence diagrams. The sequence diagrams of the advice on the method are woven into the method’s sequence diagram. An AOF (Aspect-Object Flow) tree is generated from the woven sequence diagram and class/aspect diagrams. In the AOF tree, each path from the root to a leaf is an abstract test sequence. A concrete test is obtained by creating objects that satisfy the collective constraints in the sequence. The test generation process heavily relies on manual work. Xu and Xu adapted the round-trip testing strategy to state-based test generation for aspect-oriented programs [17]. As mentioned in Section 4, it has been improved in this paper. In addition, this paper provides four more testing strategies and presents empirical and comparative evaluations of the testing strategies. Badri et al. developed a framework, AJUnit, for
state-based unit testing of aspect-oriented programs [33]. This framework generates tests from statecharts according to transition coverage, advice coverage, and aspect coverage. This paper is different from AJUnit in several ways. First, this paper is based on aspect-oriented state models. Second, it generates aspect test code from various coverage criteria of state models and system properties. Third, it has conducted empirical evaluation of these testing strategies through mutation analysis.

8.2. Aspect modeling and checking

There is a growing body of work on aspect-oriented modeling with UML [34–36]. It either uses the meta-level notation of UML or extends the UML notation for specifying crosscutting concerns. Most of the work, however, is not concerned with aspect verification due to the informal or semi-formal nature of UML [37].

As finite state models have long been in use for rigorous specification of object-oriented software, state-based aspect modeling is of particular interest. Elrad et al. proposed an approach to aspect-oriented modeling with Statecharts [36]. Base models and aspect models are represented by different regions of Statecharts. An aspect first intercepts the events sent to the base models and then broadcasts the events to the base models. Composition of base models and aspect models relies on a naming convention. In comparison, this paper uses a rigorous formalism for capturing crosscutting elements (join points, pointcuts, and advice, etc.) with respect to the state models of classes.

Several methods for model checking aspect-oriented programs have been proposed. Ubayashi and Tamai [38] used model checking to verify whether the woven code of an aspect-oriented program contains unexpected behavior. Denaro and Monga [39] have reported a preliminary experience with model checking a concurrency control aspect. They manually built the aspect model in PROMELA (the input language of the SPIN model checker) and verified the deadlock problem of the synchronization policy. Nelson et al. [40] used both model checkers and model-builders to verify woven programs. The above work [18, 39, 41] does not involve aspect-oriented design modeling or test generation.

Krishnamurthi et al. [42] adapted model checking for verifying properties against advice modularly. Given a set of properties and a set of pointcut designators, this approach automatically generates sufficient conditions on the program’s pointcuts to enable verification of advice in isolation. In a series of papers [3, 4, 9, 30, 41, 43–45], Katz and his group have addressed various issues of model checking aspect-oriented code. Model checking tasks were automatically generated for the woven code of aspect-oriented programs [41]. They treated crosscutting scenarios as aspects and used model checking to prove the conformance between the scenario-based specification of aspects and the systems with aspects woven into them [43, 45]. They also proposed an approach to generic modular verification of code-level aspects [44].

This work is different from the above methods for model checking aspect-oriented programs. The crosscutting notions (inter-model declarations, pointcuts, advice, and aspects) of the aspect-oriented state models in this paper are specified with respect to the design-level state models, as opposed to the programming constructs or control flow graphs of aspect-oriented programs [16]. More importantly, this paper applies model checking to test generation from counterexamples. It not only verifies the aspect-oriented models but also generates tests from the models for exercising the resultant implementation.

9. CONCLUSIONS

The MACT framework for testing aspect-oriented programs with their state models has been presented. The aspect-oriented state model captures not only the behaviors of base classes, but also the impacts of the aspects on the state transitions of their base classes. From the woven models of aspects and their base classes, test cases are generated according to the structural coverage criteria of state models (e.g. state coverage, transition coverage, and round trip) or the counterexamples...
of checking the system properties against the state models (model checking with trap properties and with model mutation). The MACT framework makes it convenient to evaluate the various testing strategies. The empirical studies were based on the heavyweight analysis of two AspectJ Programs, including building the state models of all aspects and related base classes, formalizing a comprehensive set of system requirements, performing the mutation analysis of all aspect models and aspect code. The results have demonstrated that the testing strategies can detect many types of aspect faults with respect to the programming constructs of aspects—pointcuts, advice, inter-type declarations, and aspect precedence. The results have also demonstrated that the structure-oriented strategies and the property-oriented strategies can complement each other—one group of strategies killed some mutants that were not killed by the other group and vice versa. This indicates that integrating various testing strategies can be a more effective approach to testing aspect-oriented programs with state models.

State-based testing of procedural and object-oriented programs has gained attention in the past decades [6–9, 11, 13, 14, 46]. Similarly, there have been two groups of testing strategies—test generation according to coverage criteria of state models and test generation through model checking. The existing work, however, has not compared the cost-effectiveness of different testing strategies. Thus, MACT provides a framework for the comparative evaluation of the existing testing strategies for object-oriented programs because the aspect-oriented state models are a super set of class state models. On one hand, the testing strategies in the MACT framework are readily applicable to the testing of object-oriented programs. On the other hand, it is easy to incorporate new testing strategies into the MACT framework. The existing mutation analysis tools for object-oriented programs (e.g. MuJava for Java programs [47]) can be used for the evaluation of fault-detection capabilities.

This paper has used the coverage criteria of state models for test generation purposes. The future work will evaluate how much of the aspect code is covered by the test cases generated from these criteria. Meanwhile, the effectiveness of state-based testing and structural testing will be compared. The future work will also consider coverage criteria for preconditions of transitions in state models. Examples include branch coverage, multi-condition coverage, and full-predicate [11]. Among the five strategies, only round-trip includes branch coverage. The future work will investigate how the coverage criteria of transition preconditions can be combined meaningfully with the current test generation strategies. The combination is expected to retain the level of automation.

ACKNOWLEDGEMENTS

The authors thank Professors Jeff Magee and Jeff Kramer for providing the source code of LTSA.

REFERENCES


