Xu DX, El-Ariss O, Xu WF *et al.* Aspect-oriented modeling and verification with finite state machines. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 24(5): 949–961 Sept. 2009

Aspect-Oriented Modeling and Verification with Finite State Machines

Dian-Xiang Xu[1,4] (徐殿祥), *Senior Member*, *IEEE*, Omar El-Ariss[2], Wei-Feng Xu[3] (许巍峰), *Senior Member*, *IEEE* and Lin-Zhang Wang[4] (王林章), *Member*, *CCF*, *ACM*, *IEEE*

[1] *National Center for the Protection of the Financial Infrastructure, Dakota State University, Madison, SD 57042, U.S.A.*

[2] *Department of Computer Science, North Dakota State University, Fargo, ND 58105, U.S.A.*

[3] *Computer and Information Science Department, Gannon University, 109 University Square, Erie, PA 16541, U.S.A.*

[4] *State Key Laboratory of Novel Software Technology, Department of Computer Science and Technology, Nanjing University Nanjing 210093, China*

E-mail: dianxiang.xu@dsu.edu; omar.elariss@ndsu.edu; xu001@gannon.edu; lzwang@nju.edu.cn

Received December 1, 2008; revised June 23, 2009.

Abstract Aspect-oriented programming modularizes crosscutting concerns into aspects with the advice invoked at the specified points of program execution. Aspects can be used in a harmful way that invalidates desired properties and even destroys the conceptual integrity of programs. To assure the quality of an aspect-oriented system, rigorous analysis and design of aspects are highly desirable. In this paper, we present an approach to aspect-oriented modeling and verification with finite state machines. Our approach provides explicit notations (e.g., pointcut, advice and aspect) for capturing crosscutting concerns and incremental modification requirements with respect to class state models. For verification purposes, we compose the aspect models and class models in an aspect-oriented model through a weaving mechanism. Then we transform the woven models and the class models not affected by the aspects into FSP (Finite State Processes), which are to be checked by the LTSA (Labeled Transition System Analyzer) model checker against the desired system properties. We have applied our approach to the modeling and verification of three aspect-oriented systems. To further evaluate the effectiveness of verification, we created a large number of flawed aspect models and verified them against the system requirements. The results show that the verification has revealed all flawed models. This indicates that our approach is effective in quality assurance of aspect-oriented state models. As such, our approach can be used for model-checking state-based specification of aspect-oriented design and can uncover some system design problems before the system is implemented.

Keywords aspect-oriented modeling, finite state machines, modeling, verification, model checking

1 Introduction

As a new software development paradigm, Aspect-Oriented Programming (AOP)[1−2] modularizes crosscutting concerns into aspects with the advice invoked at the specified points of program execution[3]. It is expected to "improve comprehensibility, reuse and ease of change..., increasing adaptability and ultimately creating more value for producers and consumers alike"[4]. An aspect-oriented system consists of aspects and base classes (or components) that can be woven into an executable whole. The base classes in an aspect-oriented system can also be executed independently. From the system architecture perspective, aspects often crosscut multiple base classes. From the base class perspective, however, aspects are essentially incremental modifications to base classes with additional operations and constraints for separate concerns. They provide a paradigm of "programming by difference", which constructs new components by specifying how they differ from the existing components[2]. The incremental modifications of aspects to base classes can impose a significant impact on the object states of base classes. For example, an incremental modification aspect can alter the state transitions defined by the state models of their base classes[5].

While the ability to modularize crosscutting concerns appears to improve 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"[3]. Aspects can be used in a harmful way

Regular Paper

The research was supported in part by the ND EPSCoR IIP-SG via NSF of USA under Grant No. EPS-047679. The fourth author was supported in part by the National Natural Science Foundation of China under Grant No. 60603036, the National Basic Research 973 Program of China under Grant No. 2009CB320702, and the National High-Tech Research and Development 863 Program of China under Grant No. 2009AA01Z148.

that invalidates desired properties[6] and even destroys the conceptual integrity of programs[3]. Therefore, aspects must be applied with care. To assure the quality of an aspect-oriented system, rigorous analysis and design of aspects are highly desirable. Existing methods for aspect-oriented modeling have focused on the formalisms for aspect specification. Since UML is a widely applied tool for object-oriented modeling, exploring the meta-level notation of UML or extending the UML notation has been a dominant approach for specifying crosscutting concerns[7]. This approach, however, lacks the ability of rigorous verification due to the informal or semi-formal nature of UML.

In this paper, we present a rigorous approach to aspect-oriented modeling and verification with finite state machines①. Our approach exploits finite state machines to model objects (classes) and provides explicit notations (inter-model declaration, pointcut, advice, aspect) for capturing crosscutting concerns and incremental modification requirements with respect to class state models. In our approach, an aspect-oriented state model consists of class models, aspect models, and aspect precedence. For verification purposes, we first compose aspect models into class models through a weaving mechanism. Then we transform the woven models and the class models not affected by the aspects into FSP (Finite State Process). Finally we apply the LTSA (Labeled Transition System Analyzer) model checker[9] to verifying the generated FSP processes against the desired system properties. According to the system requirements, properties are represented by property processes and/or temporal assertions in linear temporal logic.

We have applied our approach to the modeling and verification of three event-based simulation systems: cruise control, telecommunication and banking. We have successfully built the aspect-oriented state models of these systems. To further evaluate the effectiveness of verification, we created 46 flawed aspect models in the cruise control and telecommunication systems and verified them against 95 system requirements. The results show that the verification has revealed all flawed models. Therefore, our approach is highly effective in assuring the quality of aspect-oriented models.

The rest of this paper is organized as follows. Section 2 describes aspect-oriented modeling with finite state machines. Section 3 discusses how to check aspect-oriented state models with LTSA. Section 4 presents the empirical study. Section 5 reviews the related work. Section 6 concludes the paper.

2 Aspect-Oriented Modeling with State Machines

In this section, we first introduce class and aspect models and then illustrate aspect-oriented modeling through the aspects in an aspect-oriented cruise control system.

2.1 Class Models

Definition 1 (State Model). *A state model M is a triple (S, E, T), where*:

1) *S is a finite set of states*;

2) *E is a finite set of events*;

3) *$T \subseteq S \times E \times \Phi \times S$ is a set of transitions, where Φ is a set of regular logic formula in some language. $(s_i, e, \phi, s_j) \in T$ means that action $e \in E$ transforms state $s_i \in S$ to state $s_j \in S$ under condition $\phi \in \Phi$. ϕ is called the guard condition of the transition.*

For a state model, we may specify an initial state $s_0 \in S$. Definition 1 does not include the initial state as part of a state model because state models will also be used to specify aspects (the state model for an aspect does not need an initial state). For convenience, we use α to denote the state before an object is created (as in [10]) and the *new* event to represent the constructor (we often omit α in state diagrams, though). Usually, a class model includes α in S and *new* in E. Object construction transition, $(\alpha, new[\phi], s_0) \in T$, creates an object with initial state s_0 under condition ϕ. As an aspect-oriented program has a number of state models, we denote the component $X \in \{S, E, T\}$ of state model M as $M.X$. In a state model M for class C, events and transitions are related to methods of class C. Specifically, we interpret each transition $(s_i, e, \phi, s_j) \in M.T$ as follows:

- s_i and s_j are abstract states of objects of class C.
- e is corresponding to a method, say $m(\tau_1\nu_1, \tau_2\nu_2, \ldots, \tau_k\nu_k)$, in the specification of class C, where τ_i $(1 \leqslant i \leqslant k)$ is the type of parameter ν_i. τ_i can be a fundamental data type or an object type (i.e., class).
- ϕ is a logical condition constructed by using constants, instance fields of class C, or explicit parameters ν_i $(1 \leqslant i \leqslant k)$ of method m. If τ_i is an object type and f is a public function (method with a return value) of τ_i, then function call $\nu_i.f$ is allowed to occur in logical formulas.
- (s_i, e, ϕ, s_j) is a call to method m under state s_i that satisfies guard condition ϕ and achieves state s_j.

① This article is a substantial extension of the conference paper[8]. The new materials include rigorous definitions of the aspect-oriented state models, new running example throughout the paper, in-depth discussions on aspect verification, and additional empirical studies.

2.2 Aspect Models

As in AOP[1−2], aspects in our approach are explored to modularize concerns that crosscut or are separate from primary concerns (i.e., classes). Our approach, however, aims to capture crosscutting features at the level of abstract finite state machines (as described in Subsection 2.1; they are similar to the UML 2.0 protocol state machines[11], except for the post-conditions of transitions), as opposed to the abstraction level of programming constructs or control flow graphs. In our approach, an aspect model consists of inter-model declarations (ID), state pointcuts (SP), transition pointcuts (TP), and advice models (AM).

Definition 2 (Inter-Model Declaration). *An inter-model declaration is defined as follows*:

$$declare\ \langle base\rangle\langle transition\rangle\ \{,\langle base\rangle\langle transition\rangle\}$$

where $\langle base\rangle\langle transition\rangle$ *refers to a transition in base model.*

An inter-model declaration introduces one or more new transition to the base model. For a declared transition $C(s_i, e[\phi], s_j)$, if s_i, s_j, and/or e are not yet in C, then they become a new state or event in C. The new transitions, states, and events can be used in subsequent pointcut definitions. A join point is a transition or state in a base model. A pointcut picks out a group of join points.

Definition 3 (Pointcut). *Pointcuts are defined as follows*:

1) *pointcut* $\langle cutname\rangle\langle transition\text{-}variable\rangle$: $\langle base\rangle\langle transition\rangle\ \{,\langle base\rangle\langle transition\rangle\}$,

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

where 1) *and* 2) *define transition and state pointcuts, respectively;* $\langle cutname\rangle$ *identifies a pointcut;* $\langle transition\text{-}variable\rangle$ *is a formal transition,* $(s_i, e[\phi], s_j)$*, where* s_i*,* e*, and* s_j *are variables;* $\langle base\rangle\langle transition\rangle$ *refers to an existing transition (join point) in the base model;* $\langle base\rangle.\langle state\rangle$ *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.*

Definition 4 (Advice Model). *An advice model is defined as*: *advice* $\langle transition\text{-}cut\rangle\ \langle state\text{-}model\rangle$.

The advice for a pointcut, specified by a state model, describes the control logic applied to each join point picked out by the pointcut. An advice model can be empty, which means removal of the transitions picked out by the pointcut from the base models.

Definition 5 (Aspect Model). *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, we can specify an explicit precedence relation between aspects. 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 6. (Aspect-Oriented State Model). *An aspect-oriented state model with* m *class models and* n *aspect models is a triple* $(\{C_i\}, \{A_j\}, R)$ *where* $\{C_i\}$ $(1 \leqslant i \leqslant m)$ *is a set of class models,* $\{A_j\}$ $(1 \leqslant j \leqslant n)$ *is a set of aspect models, and* R *is an aspect precedence relation over* $\{A_i\}$*, respectively.*

2.3 Aspect-Oriented Modeling

Aspect-oriented modeling involves identifying and specifying primary and crosscutting concerns (i.e., classes and aspects). As class modeling with state machines has been well-studied, this paper focuses on aspect modeling. The key to state-based modeling of aspects is to identify and specify the impacts of aspects on their base classes and the relations with other classes. In event-based systems, aspects can lead to a variety of impacts and relations, such as:

- removing transitions from the state models of base classes;
- changing the resultant states of transitions in the state models of base classes;
- modifying the guard conditions of transitions in the state models of base classes;
- adding new transitions among existing states in the state models of base classes;
- introducing new states and events and thus transitions between new and existing states;
- referencing the states/events of other classes (called *integrated classes*) for integration.

In addition, a particular aspect can be a complex combination of different impacts and relations. For illustration purposes, let us consider different types of aspects in the aspect-oriented reconstruction of a legacy cruise control system[9]. Fig.1 shows the architecture of the aspect-oriented cruise control system, where a small circle represents a crosscutting relationship between a base class and an aspect. *CarSimulatorGUI*, *CarSimulator* and *CarSpeed* constitute an executable subsystem (i.e., car simulator without cruise control facility). The three aspects are

CarSimulatorFix, *CruiseControlIntegrator*, and *SpeedControlIntegrator*. *CarSimulatorFix* is an incremental modification aspect, enforcing the precondition "engine is on" of events *accelerate* and *brake* in *CarSimulator*. If this precondition were not enforced, a safety problem would occur — the car would start accelerating immediately when, at the initial system state (engine is off), one first accelerates the car and then turns on the ignition. The *CruiseControlIntegrator* aspect composes *CarSimulator* with such cruise control components as *CruiseDisplay* and *Controller*, whereas the *SpeedControlIntegrator* aspect composes *SpeedControl* with *Controller*.

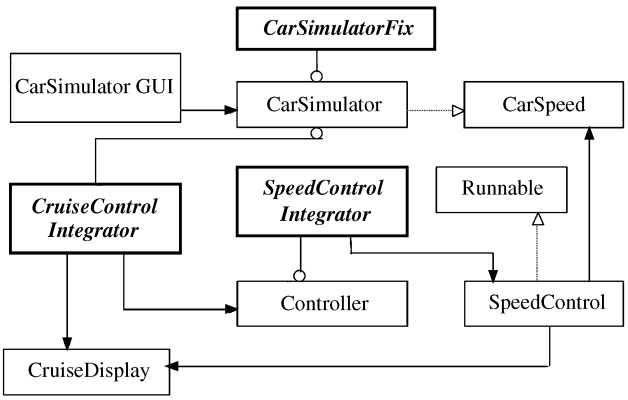

Fig.1. Aspect-oriented cruise control system.

Fig.2 shows the state model of *CarSimulator* class. The events are *engineOn*, *engineOff*, *accelerate*, and *brake*. Cruise control events (i.e., *on*, *off*, and *resume*) are not included because cruise control is considered as a separated concern through aspect-orientation (a car simulator may or may not have a cruise control facility). The six states capture all the relationships between three state variables: *ignition*, *throttle level* and *brake pedal*. *Ignition* depends on *engineOn* and *engineOff* events, whereas *throttle level* and *brake pedal* depend on *accelerate* and *brake* events. For clarity, a label with multiple events separated by "," indicates multiple transitions that share the start and resultant states. For example, (OFF_{10}, $\langle$*engineOff*, *accelerate*$\rangle$, OFF_{10}) refers to two transitions (OFF_{10}, *engineOff*, OFF_{10}) and (OFF_{10}, *accelerate*, OFF_{10}).

The *CarSimulatorFix* aspect, as shown in Fig.3, takes *CarSimulator* as the base class. Pointcut *atIgnitionOff* crosscuts two transitions (OFF_{00}, *accelerate*, OFF_{10}) and (OFF_{00}, *brake*, OFF_{01})). They mean that when *ignition* is off, *throttle level* is 0 and *brake pedal* is 0, *accelerate* and *brake* events update *throttle level* and *brake pedal* in the base model, respectively. The advice is that these events should not change the state under the given situation. The precondition (i.e., engine is *on*) of *accelerate* and *brake* is enforced by changing the resultant states of transitions from OFF_{10}/OFF_{01} to OFF_{00} (or by removing original transitions and add new transitions). In other words, when the engine is off, *accelerate* (or *brake*) will not change the engine state or the throttle level (or brake padel).

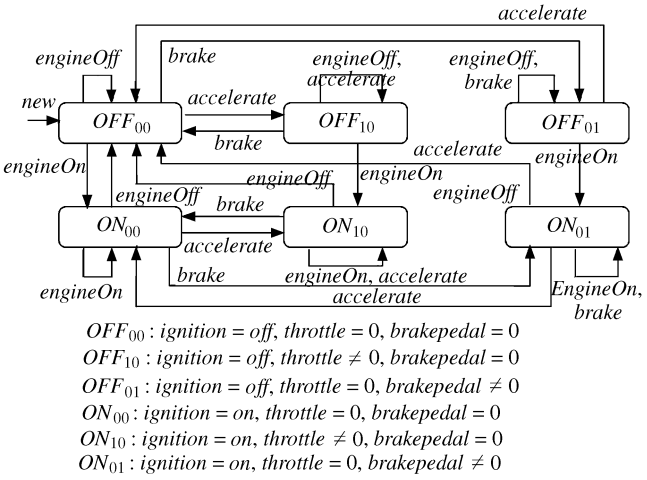

Fig.2. State model of *CarSimulator* class.

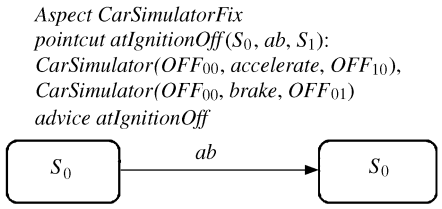

Fig.3. State model of *CarSimulatorFix* aspect.

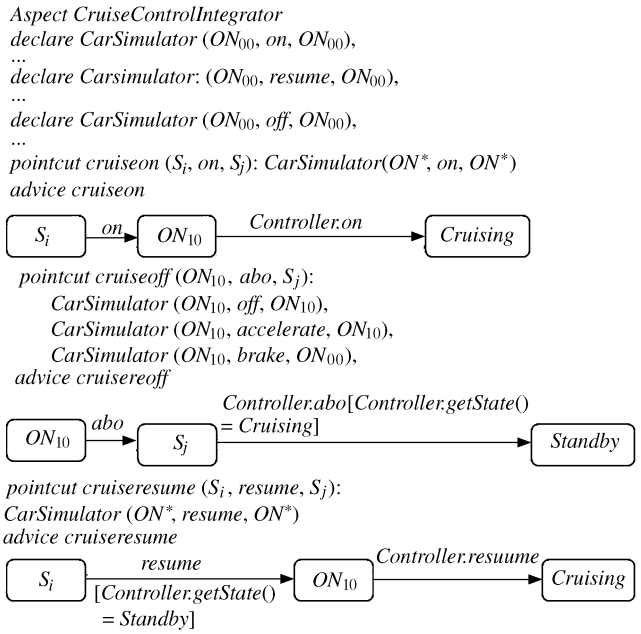

Fig.4. Portion of the *CruiseControlIntegrator* aspect model.

CruiseControlIntegrator, as partially shown in Fig.4, is a complex aspect that composes *CarSimulator* and *Controller* while introducing new cruise control events, *on*, *resume* and *off*, and thus new transitions for cruise control operations. The *declare* clauses define the new events and transitions with respect to the base model *CarSimulator*. Some of the new transitions (for events *on*, *off*, *resume* when engine is on) are further refined by the advice of pointcuts *cruiseon*, *cruiseresume* and *cruiseoff*. The functionality of integration is demonstrated by references to the states and events of *Controller*. *Controller* events are prefixed by "*Controller*" (e.g., *Controller.on* in the advice of *cruiseon*). An event without such a prefix (e.g., *on* in the advice of *cruiseon*) refers to an event for *CarSimulator*/*CruiseControlIntegrator*. For clarity, the prefix is omitted for all *Controller* states, such as *Cruising*, *Standby*, *Active*, and *Inactive*.

Consider the concern of turning on the cruise control by the *on* event. When the state of *CarSimulator* is ON_{00}, ON_{01}, or ON_{10}, the *on* event will lead *CarSimulator* to ON_{10} and *Controller* to *Cruising*. In other words, this concern crosscuts three transitions in the *CarSimulator* state model with the same advice. Thus, in Fig.4, these transitions are picked out by the *cruiseon* pointcut. The advice of this pointcut is that the resultant states of the transitions are changed to ON_{10} and the *Controller* state is set to *Cruising* through event *Controller.on*. Similarly, the concern of turning off the cruise control also crosscuts three transitions for the events *off*, *accelerate*, and *brake*. This is captured by the *cruiseoff* pointcut, whose advice is that the resultant states of these transitions remain unchanged but the *Controller* state is set to *Standby*.

The *CarSimulatorFix* and *CruiseControlIntegrator* aspects share the base class *CarSimulator*. To resolve the interference, *CarSimulatorFix* has higher precedence since the safety precondition needs to be handled first. Note that *Controller* is an integrated class with respect to *CruiseControlIntegrator*. It is also the base class of the *SpeedControlIntegrator* aspect, which integrates *SpeedControl*.

3 Checking Aspect Models

In our approach, model checking of aspects is based on the model checker LTSA. The input to LTSA includes behavior processes represented by FSP and system properties represented as property processes and/or FLTL (Fluent Linear Temporal Logic) assertions. LTSA verifies whether or not the properties are satisfied by the behavior processes through exhaustive exploration of the state space of the behavior processes. To verify aspect models, we first weave them into their base class models. This results in woven state models. Then we convert the woven models and the models of those classes not modified by the aspects into respective FSP behavior processes and verify if they have unreachable states. Meanwhile, we formalize the properties to be verified according to the system requirements. The properties are expressed as (safety and progress) property processes and/or FLTL assertions. Finally, we compose all behavior and property processes into a system-level process and feed the resulting 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). This helps improve the aspect-oriented state model or examine correctness of system properties.

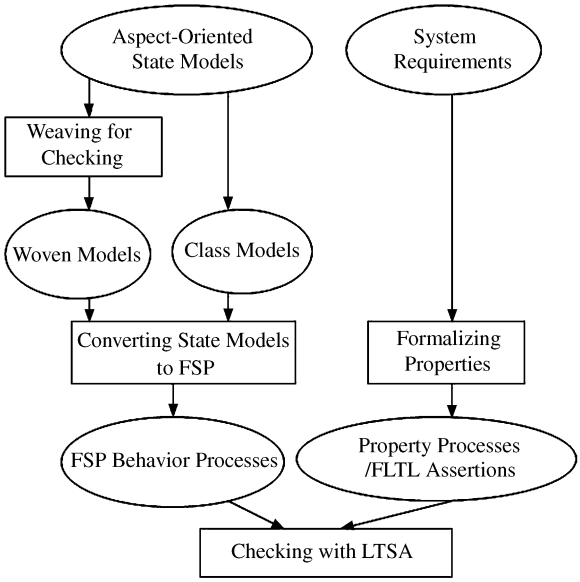

Fig.5. Model-checking process.

Fig.5 shows the general process for verifying aspect-oriented state models. In the following, we first give a brief introduction to LTSA and FSP, then we discuss the two core components of the verification process: weaving for checking and converting woven models and class models into FSP behavior processes. Finally, we will discuss property formalization issues.

3.1 Introduction to LTSA and FSP

LTSA mechanically verifies whether or not a model satisfies the particular properties required of a system when it is implemented. Through exhaustive exploration of the state space, LTSA checks for both desirable and undesirable properties for all possible sequences of events and actions. The modeling approach of LTSA is based on labeled transitions systems (LTS), where transitions in a state machine are labeled

with action names. Since representing state machines graphically severely limits the complexity of problems that can be addressed, LTSA introduces a textual (algebraic) notation, FSP, to describe system models. It can translate FSP descriptions to the equivalent graphical LTS description.

An FSP process consists of one or more local processes separated by commas. The description is terminated by a full stop. 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 ('→') and choice ('|'). For example, the following MAKER process manufactures an item 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 *ready*; they must execute it at the same time. Such a shared action implies synchronization between concurrent processes. Note that FSP is essentially based on actions and events. Although every FSP description has a corresponding graphical LTS description, the states of a process are implicit. Consider the MAKER process, the state after action *make* or *ready* is not identified, or cannot be referenced in the guard of a conditional local process. The composite process ‖ MAKER_USER = (MAKER ‖ USER) describes the model of a simple manufacturing system that consists of the two concurrent processes MAKER and USER. "‖" refers to parallel composition.

LTSA allows system properties to be defined as (safety and progress) property processes and/or FLTL assertions. A safety property process P asserts that any trace including actions in the alphabet of P is accepted by P. A progress property asserts that in an infinite execution of a target system, at least one of the actions listed in the property will be executed infinitely often (the progress properties are actually a subset of liveness properties). 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.

3.2 Weaving for Checking

In aspect models, inter-model declarations introduce new transitions, states, and events to base models. State and transition pointcuts are a naming mechanism for mapping state/event variables in advice models to the counterparts selected from base models by pointcut expressions. The selected transitions are then replaced with corresponding advice models or transitions. To represent woven state models, we slightly extend the state models described in Subsection 3.1. Specifically, a generalized transition in a woven model is of the form $(s_i, e_1[\phi_1] \rightarrow e_2[\phi_2] \rightarrow \cdots \rightarrow e_k[\phi_k], s_j)$ where ϕ_l $(l = 1, \cdots, k)$ is the guard for event e_l. It means the sequence of guarded events $e_1[\phi_1] \rightarrow e_2[\phi_2] \rightarrow \cdots \rightarrow e_k[\phi_k]$ (called a *composite* event) results in state s_j from s_i. Typically, one of these events belongs to the base class whereas others are events of the integrated classes. If there is only one event in the sequence, the transition reduces to a traditional one.

The basic entities of FSP processes are events/actions. Object states, particularly intermediate states, are implicit. Consider the advice of pointcut *cruiseon* in Fig.4. It states that turning on the cruise control is handled by two operations: *CarSimulator.on* (changing the *CarSimulator* state to ON_{10}) and *Controller.on* (setting the *Controller* state to *Cruising*). An FSP process for this advice would be composed of the event sequence *CarSimulator.on* → *Controller.on* without considering the intermediate state *Cruising*. Similarly, we can collapse the advice of *cruiseoff* pointcut in Fig.4 into the following two composite transitions: $(ON_{10}, abo[Controller.getState() = Cruising] \rightarrow Controller.abo, S_j)$ and $(ON_{10}, abo[Controller.getState() != Cruising], S_j)$. Note that *abo* as a variable can be *accelerate*, *brake*, or *off*.

Now we present the weaving algorithm that composes an aspect model with a base model for checking purposes. 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.

Algorithm 1. Weaving for Checking

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 have 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. In the cruise control system, the *CarSimulatorFix* aspect has higher precedence than *CruiseControlIntegrator*. We first weave the *CarSimulatorFix* aspect with its base class *CarSimulator*. The weaving result is shown in Fig.6. It is then used as the base model for weaving *CruiseControlIntegrator* aspect. The final woven model is shown in Fig.7, where for simplicity all guard conditions are omitted.

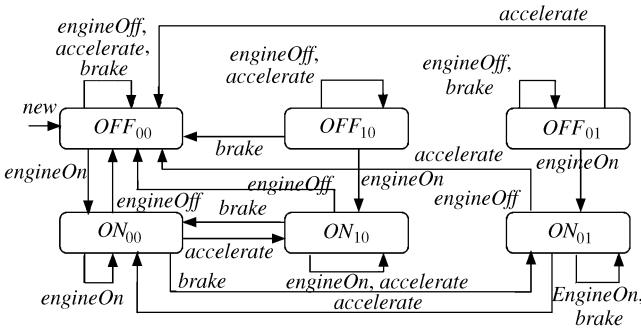

Fig.6. Woven model of *CarSimulator* class and *CarSimulatorFix* aspect.

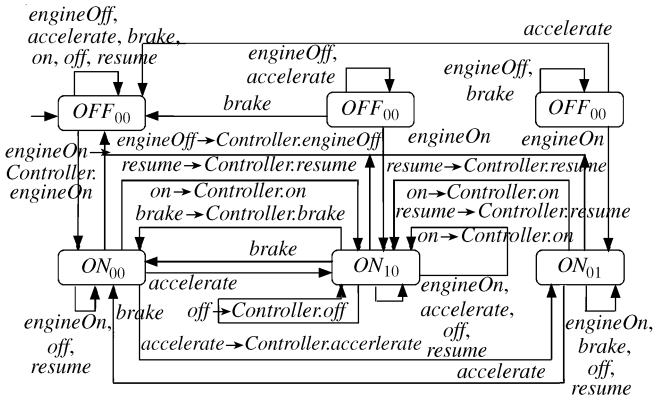

Fig.7. Woven model of *CarSimulator* class, *CarSimulatorFix* aspect and *CruiseControlIntegrator* aspect.

For a given aspect-oriented state model, we weave all aspects with their base classes and transform the model into a set of woven state models together with the models of those integrated classes. For verification purposes, the original base class models and aspect models are not used. For example, the aspect-oriented model of the cruise control system reduces to three models after weaving: *SpeedControl*, *Controller*, and *CarSimulator*, where *SpeedControl* identifies the model of class *SpeedControl*, *Controller* now refers to the woven model of *Controller* class and *SpeedControlIntegrator* aspect, and *CarSimulator* now represents the woven model of *CarSimulator* class, *CarSimulatorFix* aspect, and *CruiseControlIntegrator* aspect.

3.3 From State Models to FSPs

After aspect weaving, we convert each woven model and class model into an FSP process. To do so, we first generate the top-level FSP process named after the (base) class. This process starts with the initial state of the (base) class. For example, the woven model of the aspects *CarSimulatorFix* and *CruiseControlIntegrator* with their base class *CarSimulator* in Fig.7 is defined as:

$$CARSIMULATOR = OFF_{00},$$

where OFF_{00} is the initial state (local process). A local process is then created for each state s in the model. The body of this local process is determined by the transitions that start from the state s. Each basic transition (s, e, s') becomes a local process $e \rightarrow s'$ in the process body (composition events and guard conditions are discussed later). If there are more than one such transitions, they are composed by the choice ("|") construct. In Fig.7, the transitions starting with OFF_{00} include:

$(OFF_{00}, CarSimulator.engineOn \rightarrow Controller.engineOn,\ ON_{00})$,
$(OFF_{00},\ CarSimulator.engineOff,\ OFF_{00})$,
$(OFF_{00},\ CarSimulator.accelerate,\ OFF_{00})$,
$(OFF_{00},\ CarSimulator.brake,\ OFF_{00})$,
$(OFF_{00},\ CarSimulator.on,\ OFF_{00})$,
$(OFF_{00},\ CarSimulator.off,\ OFF_{00})$,
$(OFF_{00},\ CarSimulator.resume,\ OFF_{00})$.

The local process for OFF_{00} is thus as follows:

$OFF_{00} = (carSimulator.engineOn \rightarrow controller.engineOn \rightarrow ON_{00}$
$|carSimulator.engineOff \rightarrow OFF_{00}$
$|carSimulator.accelerate \rightarrow OFF_{00}$
$|carSimulator.brake \rightarrow OFF_{00}$
$|carSimulator.on \rightarrow OFF_{00}$
$|carSimulator.off \rightarrow OFF_{00}$
$|carSimulator.resume \rightarrow OFF_{00}),$

where the first letter of class names is lowercased. This is required of the action and event identifiers in FSP. The general algorithm for transforming a woven (or class) model into an FSP consists of two procedures: FSP process generation and recursive FSP local process generation. The algorithm is described below.

Algorithm 2. Conversion of a State Model into an FSP. Generating a complete FSP process for a given state model

Procedure 1. FSP Process Generation

Input: a state model

Output: an FSP process with all local processes

Steps:

S1.1 Let *TraversedStates* be all the states whose local processes are already generated. Initially *TraversedStates* $= \varnothing$;

S1.2 Find the initial state (denoted as *initState*) from the object construction transition of the model;

S1.3 The top-level process is *modelName* = *initState* (the object construction event is abstracted away), where *modelName* is the name of the (base) class;

S1.4 Generate the local process for *initState* using Procedure 2 below;

S1.5 Concatenate the top-level process in S1.3 with the subprocess in S1.4 and replace the last occurrence of ',' with ').', which means the end of a process;

S1.6 Report unreachable for any state in the state model but not in *TraversedStates*;

S1.7 Return the resulting process of S1.5.

Procedure 2. FSP Local Process Generation

Input: a state model and a state s in the model

Output: an FSP local process

Steps:

S2.1 The initial process text: $s = ($;

S2.2 Find all transitions in the model that start with state s. Suppose E is the set of events involved in the transitions;

S2.2.1 For the first transition, (s, ce, s'), transform it to a clause $ce \rightarrow s'$;

S2.2.2 For each of other transitions, say (s, ce, s'), transform it to a clause $|ce \rightarrow s'$, where "|" is the choice construct;

S2.2.3 For each event e in E, if there is one or more conditional transition $(s, e[\phi_1], s_1), \ldots, (s, e[\phi_k], s_k)$ (suppose $\phi_1 \vee \cdots \vee \phi_k$ is not always true), generate a clause $|e \rightarrow s$;

S2.2.4 Concatenate the initial process text, the clauses in the above steps, and "," (end of a local process);

S2.3 Add s into *TraversedStates*;

S2.4 For each transition, $(s, e[\phi], s')$, such that the local process for s' is not generated yet, repeat Procedure 2 for s'.

S2.5 Return the local process obtained in S2.2.4.

For clarity, Algorithm 2 does not deal with the naming convention. In fact, it has to follow the naming convention of LTSA. Specifically, we capitalize process (i.e., model) and local process (i.e., state) names and use a lower case for the first letter of each event name. To differentiate the events of different classes, we always prefix an event with its class name (starting with a lower case letter according to the LTSA naming convention, though).

Finally, we need to define the system-level process for an aspect-oriented state model. To do so, we compose the FSP processes for all woven state models and class models not affected by aspects. For the cruise control system model, aspects *CarSimulatorFix* and *CruiseControlIntegrator* are woven into *CarSimulator*; *SpeedControlIntegrator* is woven into *Controller*; *SpeedControl* remains unchanged. Thus the system-level process is:

$$\|\mathit{CruiseControl} = \mathit{CarSimulator}\| \mathit{Controller}\|\mathit{SpeedControl}.$$

where "||" means parallel composition. Putting this together with the FSP processes for the *CarSimulator* and *Controller* woven models and the *SpeedControl* class model, we obtain the complete FSP specification for the cruise control system.

3.4 Aspect Verification

After the woven models and class models are transformed into FSP processes, we formalize system properties according to the system requirements and verify the FSP processes against the properties. As mentioned earlier, LTSA allows properties to be expressed as property processes and/or FLTL assertions.

We can verify an aspect-oriented state model in an incremental manner, that is, check the class models first and then the aspect models together with their base models and related class models. The incremental verification can determine aspect problems if the classes are already proven correct. Katz[6] has discussed how spectative, regulative, invasive aspects affect the properties of their base programs. This aspect category is helpful for identifying the properties that aspects should preserve or invalidate. In the cruise control system, for example, the requirement "OFF_{00} should be reached eventually after engine is turned on" must be met no matter whether the aspects are applied.

To verify correctness of aspects, we are also interested in the particular properties that are changed or introduced by aspects. Depending on the application, an aspect can be expected to validate or invalidate a specific property of its base classes. For example, when an aspect is used to enforce the contract of the base classes, it implies that the base classes have not yet enforced the contract. We can define properties that should be violated (or satisfied) by the base classes alone but satisfied (or violated) after the aspect is composed. Consider the *CarSimulator* model in Fig.2. It violates the following safety property:

property *SAFETY*= (
{*carSimulator.engineOn, carSimulator.engineOff,*

carSimulator.brake, carSimulator.on,
carSimulator.off, carSimulator.resume} → *SAFETY*
| {*carSimulator.accelerate*} → *SAFETYCHECK*),

SAFETYCHECK= (
{*carSimulator.engineOff, carSimulator.brake,*
carSimulator.on, carSimulator.off,
carSimulator.resume} → *SAFETY*
|{*carSimulator.accelerate*} → *SAFETYCHECK*
|{*carSimulator.engineOn*} → *ERROR*).

The following is part of the LTSA output when the generated FSP process for the *CarSimulator* model in Fig.2 is checked against the above property:

Trace to property violation in *SAFETY*:
carSimulator.accelerate
carSimulator.engineOn.

The trace to property violation reflects the safety problem discussed in Subsection 2.3. The corresponding sequence of states and events in the class model is ⟨OFF_{00}, *accelerate*, OFF_{10}, *engineOn*, ON_{10}⟩. The above safety property, however, is satisfied after the *CarSimulatorFix* aspect is woven into *CarSimulater* if the aspect disables the event *CarSimulator.accelerate* at the initial state (i.e., if the advice of *atIgnitionOff* has an empty advice model, which means removal of the selected transitions in Fig.3). Note that the advice model in Fig.3 allows the *CarSimulator.accelerate* event to be received, yet without changing the state. Thus, ⟨*carSimulator.accelerate, carSimulator.engineOn*⟩ is a safe event sequence. In this case, the above property specification does not apply to the aspect model in Fig.3. In general, aspect verification may require modifying the concrete property specification of its base classes even for a similar property if the property relies on the ordering of events or states.

Properties with respect to an aspect may not make sense to the individual base classes. For instance, the *CruiseControlIntegrator* aspect in the cruise control system involves two classes. To verify its correctness, the cruise control properties are defined over the two classes, other than each individual class. They are in essence inter-class invariants enforced by the aspect. Obviously, they are meaningless when only an individual class is checked. This is similar for the *SpeedControlIntegrator* aspect. In addition to safety/progress property processes, FLTL assertions can be defined to express various system properties including safety and liveness. LTSA automatically verifies the given properties and checks to see if deadlock exists.

Moreover, our approach automatically inspects if there are unreachable states in an aspect-oriented state model. Consider the states OFF_{10} and OFF_{01}. They are reachable from the initial state OFF_{00} in the base model *CarSimulator* (Fig.2), but unreachable in the woven models (Figs. 6 and 7). It is the *CarSimulatorFix* aspect that makes them unreachable so as to avoid the unsafe situation. As a matter of fact, reachability analysis is conducted when a state model is converted into an FSP process (refer to step 1.6 of Algorithm 2). A generated FSP process contains no unreachable states. If a generated FSP process has missed an expected state, it indicates that the original aspect-oriented state model is incorrect. For the expected unreachable states, it is safe to remove them and the transitions associated with them. The removal does not lose any information for further verification. For example, Fig.6 can be reduced to Fig.8. The generated FSP processes for the two models are the same. Similarly, the model in Fig.7 is reduced when it is transformed into an FSP process.

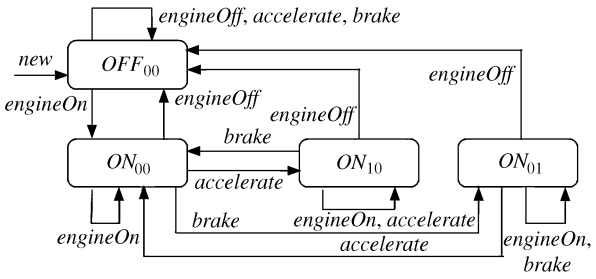

Fig.8. Reduced model of Fig.6.

4 Applications and Evaluations

We have fully implemented our approach in the Java-based tool MACT②. With the tool support, we have been able to apply our approach to three non-trivial event-based systems — the aspect-oriented cruise control system[8], telecom (a sample program in the AJDT toolkit③), and banking④.

To evaluate the effectiveness of verification, we have applied our approach as a heavyweight formal method to the cruise control and telecom systems — we have modeled all aspects and related classes and formalized and verified a comprehensive set of requirements for each subsystem (each subsystem consists of aspects of

②MACT (Model-based Aspect Checking and Testing) accepts textual specifications of aspect-oriented state models. We have also implemented a separate tool for the graphical notation of aspect modeling with state machines based on the open source UML tool ArgoUML (http://argouml.tigris.org). It can export the graphical representations of aspect and class state models to MACT.

③AspectJ Development Toolkit: http://www.eclipse.org/ajdt/. AspectJ is the representative AOP language built upon Java.

④http://www. manning.com/laddad/.

interest and related classes for modeling and verification). Such heavy-weight applications of formal methods have been a significant challenge. On one hand, it was difficult to define a comprehensive set of system requirements for each aspect and related classes. On the other hand, it was not easy to formalize every requirement (in fact property formalization can be more difficult than modeling). In practice, our approach can be used as a light-weight method, focusing on the modeling and verification of critical system components and requirements. This is the case for the banking system, which consists of 282 classes and 12 aspects (27 K lines of code). The application of our approach to the banking system has focused on the modeling and checking of the aspects for concurrency control of read/write access to the database and related classes.

Table 1. Subjects of the Empirical Study

Subjects	Cruise control	Telecom
Number of classes	9	10
Number of aspects	3	3
Number of system properties formalized	63	32
Number of aspect model mutants created	33	13

Table 1 lists the metrics of the cruise control and telecom systems. The underlying logic of the cruise control system is much more complex than telecom according to their aspect-oriented models and number of formalized requirements. For example, the former has twice as many formalized properties.

For each subsystem consisting of an aspect and related classes, we first built an aspect-oriented state model, formalized the properties according to the system requirements, and checked the aspect-oriented model against the properties. In this way, we obtained the correct model for each of the subsystems. This indicates that our approach has successfully verified the correct models. However, it is also interesting to know whether or not our approach can detect flaws in aspect-oriented state models. To further evaluate our approach, we created mutants (variations) of the correct aspect-oriented state models according to the fault category of aspect design and checked each model mutant to see if any property was violated.

The fault category of aspect models implies the various ways an aspect model can go wrong (as our focus is on aspects, we do not consider the potential faults of class model). It consists of the following fault types:

FT1: Incorrect pointcut with a missing join point. The consequence is that the desired advice is not applied to the join point.

FT2: Incorrect pointcut with an extra join point. The consequence is that extra advice is applied to the join point.

FT3: Incorrect advice where a transition has a wrong starting (ending) state or event/action.

FT4: Incorrect advice with a missing transition (state).

FT5: Incorrect advice with an extra transition (state).

FT6: Incorrect advice with a missing guard for a guarded transition or with an extra guard condition for an unguarded transition.

FT7: Incorrect aspect or advice precedence.

The above fault category is similar to the fault model of AOP[5,12]. They cover incorrect pointcuts, incorrect advice and incorrect aspect precedence, yet at different development phases (design vs. programming). The above fault category is specific to the aspect-oriented state models.

A model mutant of a correct aspect-oriented state model is a variation of the model. We create a model mutant by seeding one potential fault of the above category into an aspect model. It indicates a particular way that the aspect may be modeled incorrectly. We have created 33 and 13 mutants for the two applications, respectively. All of them were determined to be flawed because they either led to a deadlock or violated some property. This demonstrates that our approach is indeed effective for assuring the quality of aspect models.

5 Related Work

There is a growing body of work on aspect-oriented modeling with UML[13−21]. It exploits 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. A recent survey can be found in [7].

Since finite state models have long been in use for rigorous specification of object-oriented software[10], state-based aspect modeling is of particular interest. Elrad *et al.* have proposed an approach to aspect-oriented modeling with Statecharts[13,21]. Base state models and aspect state models are represented by different regions of Statecharts. An aspect first intercepts the events sent to the base state models and then broadcast the events to the base state models. Composition of base models and aspect models relies on a specific naming convention as the weaving mechanism is implicit. In comparison, our work uses a rigorous formalism for capturing crosscutting elements (inter-model declaration, join point, pointcut, and advice) with respect to the state models of classes. Aspects and classes are composed through an explicit weaving mechanism. Xu and Nygard[22] have developed aspect-oriented Petri

nets for threat-driven modeling and verification of secure software. The intended functions, threat scenarios, and security features of a system are all formalized by Petri nets. Verification is conducted with respect to the correctness and absence of threat scenarios, as opposed to desired system properties.

Several methods for model-checking aspect-oriented programs have been proposed. Ubayashi and Tamai[23] use model-checking to verify whether the woven code of an aspect-oriented program contains unexpected behavior. They also propose a model-checking framework that allows crosscutting properties to be defined as an aspect and thus separated from the program body. Denaro and Monga[24] report a preliminary experience with model-checking a concurrency control aspect. They manually build the aspect model in PROMELA (the input language of the SPIN model checker) and verify the deadlock problem of the synchronization policy. Since the transformation is done by hand, the conformance between the PROMELA program and aspect code remains an open issue. Nelson *et al.*[25] use both model checkers and model-builders to verify woven programs. The above work does not involve aspect-oriented modeling.

Krishnamurthi *et al.*[26] adapt 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. It assumes that the programs and advice are given as state machines, which represent the control-flow graphs of program fragments. In a series of papers, Katz and his group have addressed various issues of model-checking aspect-oriented code. In [27–28], model checking tasks are automatically generated for the woven code of aspect-oriented programs. This approach takes the advantage of the Bandera system[29] that generates input to model checking tools directly from Java code, and hence the woven code of AspectJ programs. In [30], they treat crosscutting scenarios as aspects and use model checking to prove the conformance between the scenario-based specification of aspects and the systems with aspects woven into them. In [31], they propose an approach to generic modular verification of code-level aspects. They check an aspect state machine against the desired properties whenever it is woven over a base state machine that satisfies the assumptions of the aspect. A single state machine is constructed using the tableau of the LTL description of the assumptions, a description of the joinpoints, and the state machine of the aspect code.

Our work is different from the above methods for model-checking aspect-oriented programs. The crosscutting notions (pointcuts, advice, and aspects) of the aspect-oriented state models in our approach are specified with respect to the design-level state models, as opposed to the programming constructs or control flow graphs of aspect-oriented programs. In the cruise control system model, for instance, the abstract state OFF_{00} (*ignition=OFF* && *throttle*=0 && *brakepadel*=0) involves three instance variables in the implementation. Aspect models are allowed to introduce new states, events, and transitions. Nevertheless, the approaches to modular verification of aspects[26,31] can be adopted to enhance our work.

Among the efforts to define formal semantics of aspects, some have been accompanied by proposals on employing the semantics for verification. For instance, Andrews[32] uses process algebras to offer a foundation for AOP. This work places emphasis on the correctness proofs of program weaving. It uses program equivalence to establish the correctness of a particular weaver. Xu *et al.*[33] reduce aspect verification to prior work on reasoning about implicit invocation systems. They suggest using model- rather than proof-theoretic techniques. It is not clear whether verification works in a way that is meaningful to aspects and what the formal properties about implicit invocation verification mean in the context of aspects. The above approaches to the aspect semantics are essentially orthogonal to our work.

6 Conclusions

We have presented the approach to aspect-oriented modeling and verification with finite state machines. The applications of our approach have demonstrated that aspect-orientation can provide an effective mechanism for dealing with crosscutting concerns and incremental modification requirements. Aspect-oriented verification through model checking can uncover system design problems before the system is implemented. This will reduce overall development costs due to earlier detection of problems.

Aspect-oriented modeling and verification can also facilitate detecting programming faults through model-based testing. For example, the model-based testing method[5] generates test cases from an aspect-oriented state model for exercising the resultant aspect-oriented program. When correctness of the model is assured by the model-checking method, each failure of test execution implies that the code is faulty (as long as the test oracle including test result evaluation is reliable). Therefore, combination of verification and model-based testing can assure high-quality system implementation.

Acknowledgment This work was done while the first author was with North Dakota State University.

Mr. Ganesh K. Vellaswamy contributed to the javacc parser for the aspect-oriented state models. Mr. Izzat Alsmadi defined an initial set of properties for the cruise control system. All these properties have been overridden by the empirical studies reported in this article. We thank Professors Jeff Magee and Jeff Kramer for providing the source code of LTSA, which greatly facilitated the integration of MACT with LTSA.

References

[1] Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold W G. An overview of AspectJ. In *Proc. ECOOP 2001*, Budapest, Hungary, June 18–22, 2001, pp.327–353.

[2] Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C V, Loingtier J M, Irwin J. Aspect-oriented programming. In *Proc. ECOOP 1997*, Jyväskylä, Finland, June 9–13, 1997, pp.220–242.

[3] Rinard M, Salcianu A, Bugrara S. A classification system and analysis for aspect-oriented programs. In *Proc. FSE 2004*, New Delhi, India, Feb. 5–7, 2004, pp.147–158.

[4] Sullivan K, Griswold W G, Song Y, Cai Y, Shonle M, Tewari N, Rajan H. Information hiding interfaces for aspect-oriented design. In *Proc. ESEC/FSE 2005*, Lisbon, Portugal, Sept. 5–9, 2005, pp.166–175.

[5] Xu D, Xu W. State-based incremental testing of aspect-oriented programs. In *Proc. the Fifth International Conf. Aspect-Oriented Software Development* (*AOSD 2006*), Bonn, Germany, March 20–24, 2006, pp.180–189.

[6] Katz S. Aspect Categories and Classes of Temporal Properties. Transactions on Aspect-Oriented Software Development I, Rashid A, Aksit M (eds.), *LNCS* 3880, 2006, pp.106–134.

[7] Schauerhuber A, Schwinger W, Retschitzegger W, Wimmer M. A survey on aspect-oriented modeling approaches. Technical Report, Vienna University of Technology, January 2006, http://www.wit.at/people/schauerhuber/publications/aom-Survey/AOMSurvey2006-01-30.pdf.

[8] Xu D, Alsmadi I, Xu W. Model checking aspect-oriented design specification. In *Proc. the 31st Annual International Computer Software and Applications Conference* (*COMPSAC 2007*), Beijing, China, July 23–27, 2007, pp.491–500.

[9] Magee J, Kramer J. Concurrency: State Models and Java Programs. Second Edition, John Wiley & Sons Ltd, 2006.

[10] Binder R V. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley, 2000.

[11] UML 2.0 superstructure and infrastructure specification. http://www.omg.org/technology/documents/formal/uml.htm.

[12] Alexander R T, Bieman J M, Andrews A A. Towards the systematic testing of aspect-oriented programs. Technical Report, Colorado State University, 2004, http://www.cs.colostate.edu/∼rta/publications/CS-04-105.pdf.

[13] Aldawud O, Bader F, Elrad T. Weaving with statecharts. In *The Second International Workshop on Aspect-Oriented Modeling with UML*, Dresden, Germany, Sept. 30–Oct. 4, 2002.

[14] Chavez C, Lucena C. A meta-model for aspect-oriented modeling. In *The 2nd Int. Workshop on Aspect-Oriented Modeling with UML*, Dresden, Germany, Sept. 30–Oct. 4, 2002.

[15] Clarke S, Walker R J. Generic Aspect-Oriented Design with Theme/UML. Aspect-Oriented Software Development. Filman R E, Elrad T, Clarke S, Aksit M (eds.), Boston: Addison-Wesley, 2005, pp.425–458.

[16] Han Y, Kniesel G, Cremers A B. A meta model and modeling notation for AspectJ. In *The 5th Workshop on Aspect-Oriented Modeling with UML*, in conjunction with *UML 2004*, Lisbon, Portugal, October 11–15, 2004.

[17] Pawlak R, Duchien L, Florin G, Legond-Aubry F, Seinturier L, Martelli L. A UML notation for aspect-oriented software design. In *The 1st Int. Workshop on Aspect-Oriented Modeling with UML*, in conjunction with *AOSD 2002*, Enschede, The Netherlands, April 22–26, 2002.

[18] Stein D, Hanenberg S, Unland R. An UML-based aspect-oriented design notation. In *Proc. the 1st Int. Conf. Aspect-Oriented Software Development* (*AOSD 2002*), Enschede, The Netherlands, April 22–26, 2002, pp.106–112.

[19] Stein D, Hanenberg S, Unland R. On representing join points in the UML. In *The Second International Workshop on Aspect-Oriented Modeling with UML*, Dresden, Germany, Sept. 30–Oct. 4, 2002.

[20] Zavaleta E B, Fuster G G, Morillo J L. An approach to aspect modeling with UML 2.0. In *The 5th Workshop on Aspect-Oriented Modeling with UML*, in conjunction with *UML 2004*, Lisbon, Portugal, October 11–15, 2004.

[21] Elrad T, Aldawud O, Bader A. Expressing Aspects Using UML Behavior and Structural Diagrams. Aspect-Oriented Software Development, Filman *et al.* (eds.), Boston: Addison-Wesley, 2005, pp.459–478.

[22] Xu D, Nygard K. Threat-driven modeling and verification of secure software using aspect-oriented Petri nets. *IEEE Trans. Software Engineering*. April 2006, 32(4): 265–278.

[23] Ubayashi N, Tamai T. Aspect-oriented programming with model checking. In *Proc. the First International Conf. Aspect-Oriented Software Development* (*AOSD 2002*), Enschede, The Netherlands, April 22–26, 2002, pp.148–154.

[24] Denaro G, Monga M. An experience on verification of aspect properties. In *International Workshop on Principles of Software Evolution* (*IWPSE 2001*), Vienna, Austria, Sept. 10–11, 2001, pp.186–189.

[25] Nelson T, Cowan D D, Alencar P S C. Supporting formal verification of crosscutting concerns. In *Proc. Reflection*, Kyoto, Japan, Sept. 25–28, 2001, pp.153–169.

[26] Krishnamurthi S, Fisler K, Greenberg M. Verifying aspect advice modularly. In *Proc. the 12th ACM SIGSOFT Symposium on Foundations of Software Engineering* (*FSE 2004*), Newport Beach, USA, Oct. 31–Nov. 6, 2004, pp.137–146.

[27] Katz S, Sihman M. Aspect-validation using model-checking. In *Proc. the International Symposium on Verification*, in honor of Zohar Manna, *LNCS* 2772, Springer-Verlag, 2003, pp.389–411, Also early version in FOAL 2003.

[28] Sihman M, Katz S. Model checking applications of aspects and superimpositions. In *Proc. Foundations of Aspect-Oriented Languages*, Bonn, Germany, March 20–24, 2003, pp.51–60.

[29] Hatcliff J, Dwyer M. Using the Bandera tool set to model-check properties of concurrent Java software. In *Proc. the 12th Int. Conf. Concurrency Theory* (*CONCUR 2001*), Aalborg, Denmark, *LNCS 2154*, Larsen K G, Nielsen M (eds.), Springer-Verlag, Aug. 20–25, 2001, pp.39–58.

[30] Katz E, Katz S. Verifying scenario-based aspect specifications. In *Proc. the International Symposium of Formal Methods Europe*, Newcastle, UK, July 18–22, 2005, pp.432–447.

[31] Goldman M, Katz S. Modular generic verification of LTL properties for aspects. In *Proc. Foundations of Aspect Languages Workshop* (*FOAL 2006*), Bonn, Germany, March 20–24, 2006.

[32] Andrews J H. Process-algebraic foundations of aspect-oriented programming. In *Proc. Reflection*, Kyoto, Japan Sept. 25–28, 2001, pp.187–209.

[33] Xu J, Rajan H, Sullivan K. Aspect reasoning by reduction to implicit invocation. In *Proc. Foundations of Aspect-Oriented Languages*, Lancaster, UK, March 23, 2004.

Dian-Xiang Xu received the B.S., M.S., and Ph.D. degrees in computer science from Nanjing University, China in 1989, 1992, and 1995, respectively. He is currently an associate professor with the National Center for the Protection of the Financial Infrastructure at Dakota State University in South Dakota, USA. He was assistant professor of computer science at North Dakota State University from July 2003 to May 2009, research assistant professor and engineer of computer science at Texas A&M University from August 2000 to July 2003, and research associate at Florida International University from May 1999 to August 2000. Prior to that, he was associate professor and associate department chair of the Department of Computer Science and Technology, Nanjing University. His research interests are in the areas of software security, software testing, aspect-oriented software development, and applied formal methods. He is a senior member of the IEEE.

Omar El-Ariss received his B.S. and M.S. degrees in computer science from the Lebanese American University, Beirut, Lebanon in 2001 and 2005 respectively. He is currently pursuing his Ph.D. degree at the Computer Science Department in North Dakota State University. His research interests are in the areas of software safety, software security, software modeling and verification, and software testing.

Wei-Feng Xu received the B.S. and M.S. degrees in computer science from Southeast Missouri State University and Towson University in 2000 and 2002, respectively. He received his Ph.D. degree in software engineering from North Dakota State University in 2007. Currently, Dr. Xu is the director of Keystone Software Development Institute and an assistant professor of software engineering at Gannon University in Erie, PA. He also serves as an IT consultant for General Electric's Locomotive Remote Diagnostics Service Center in developing an innovative remote monitoring system. His research interests include software testing and aspect-oriented software development. He is a senior member of the IEEE.

Lin-Zhang Wang received his B.Sc. degree in computer science from the Shenyang University of Technology in 1995, and his Ph.D. degree in computer science from Nanjing University in 2005. He is currently an associate professor in the Department of Computer Science and Technology at Nanjing University, China. His research interest is in the area of software engineering, particularly model-driven software testing and verification. He is a member of IEEE, ACM, and CCF.