Model Checking Aspect-Oriented Design Specification

Dianxiang Xu, Izzat Alsmadi, and Weifeng Xu
Department of Computer Science
North Dakota State University
Fargo, ND 58105, USA
E-mail: {dianxiang.xu, izzat.alsmadi, weifeng.xu}@ndsu.edu

Abstract

Aspects can be used in a harmful way that invalidates desired properties. Rigorous specification and analysis of aspect design is thus highly desirable. This paper presents an approach to model-checking state-based specification of aspect-oriented design. It is based on a rigorous formalism for capturing crosscutting concerns with respect to the design-level state models of classes. An aspect model not only encapsulates pointcuts and advice, but also supports inter-model declarations, aspect precedence, and references to the behaviors of other classes in advice models. For verification purposes, we convert the aspect-oriented state model of a system into woven models and further transform the woven models and the non-base class models into FSP processes. The generated FSP processes are checked by the LTSA model checker against the desired system properties. We have applied our approach to the modeling and verification of a non-trivial aspect-oriented cruise control system. A total of 21 properties that provide a comprehensive coverage of the system requirements are successfully formalized and verified.

1. Introduction

Aspect-Oriented Programming (AOP) [12] modularizes crosscutting concerns into aspects with the advice invoked at the specified points of program execution. It is expected to "improve reuse and ease of change…, and ultimately creating more value for producers and consumers alike" [18]. 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" [16]. Aspects can be used in a harmful way that invalidates desired properties [10][11] and even destroys the conceptual integrity of programs [16]. A piece of around advice may completely alter the behavior of the base classes no matter whether it is expected or unexpected. Therefore, aspects must be applied with care. To assure the quality of an aspect-oriented system, rigorous analysis of aspect design is highly desirable. Existing methods for aspect-oriented design modeling have focused on the formalisms for aspect specification. Since UML is a widely applied tool for modeling object-oriented design, exploring the meta-level notation of UML or extending the UML notation has been a dominant approach for specifying crosscutting concerns [17]. This approach, however, lacks the ability of rigorous verification due to the informal or semi-formal nature of UML.

This paper presents an approach to model-checking state-based specification of aspect-oriented design. It is based on rigorous notations (e.g. pointcuts, advice, aspects) for capturing crosscutting concerns with respect to the design-level state models of classes. 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 by an explicit weaving mechanism. Then we transform the woven models and the class models not affected by the aspects into FSP processes. Finally we apply the LTSA model checker [14] to verifying the generated FSP processes against the desired system properties. Our experiment has shown that the model-checking approach is highly effective in assuring the quality of aspect-oriented design.

The rest of this paper is organized as follows. Section 2 is a brief introduction to the LTSA model-checker. Section 3 describes aspect-oriented state models for design specification. Section 4 discusses verification of the aspect-oriented models. Section 5 presents the empirical study. Section 6 reviews the related work. Section 7 concludes the paper.

2. Background: LTSA and FSP

The model checker LTSA (Labeled Transition System Analyzer) [14] mechanically verifies whether or not a model satisfies the particular properties required of a system when it is implemented. A model is a simplified, abstract description of the behavior of a system. 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 (Finite State Processes), 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, a sequential composition, a conditional process, or is defined using action prefix ("->") and choice ("|"). Shared actions in concurrent processes indicate synchronization between the processes. Parallel composition ("||") can be used to form composite processes.

LTSA allows system properties to be defined as (safety and progress) property processes and/or Fluent Linear Temporal Logic (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). Properties can also be specified as state-oriented logical propositions in FLTL. As states in FSP are implicit, LTSA takes an approach that maps an action trace into a sequence of abstract states described by *fluents*. A fluent is defined as *fluent FL* = $<\{s_1,...s_m\}, \{e_1,...e_n\}$ *initially B*, where B is the initial value, $s_1,...s_m$ are the initiating actions, and $e_1,...e_n$ are the terminating actions. *FL* becomes true when any of the initiating actions occur and false when any of the terminating actions occur. In other words, a fluent holds at a time instant if and only if it holds initially or some initiating actions has occurred, and in both cases, no terminating action has yet occurred. An action fluent is a fluent such that the action itself is the initiating action and other actions are the terminating ones. An action fluent becomes true immediately when the action occurs and false when the next action occurs. Fluent expressions can be constructed by applying normal logical operators (conjunction, disjunction, negation, implication, and equivalence) to fluents. 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.

3. Aspect-Oriented State Models

3.1. Class Models

A state model M consists of states S, events E, and transitions T. Transition $(s_i, e[\phi], s_j) \in T$ means that event $e \in E$ results in state $s_j \in S$ from state $s_i \in S$ under guard condition ϕ (ϕ is optional). For the state model of a given class, S, E, and T represent object states, public constructor/methods, functionality implemented by the constructor/methods, respectively. $s \in S$ can be a concrete object state or a state invariant. A guard condition is a logical formula constructed by using constants, instance variables, and functions (methods with return values).

For convenience, we use α to denote the state before an object is created (as in [2]) 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. Thus we can determine the initial state of a given state model from its object construction transition. To distinguish states and events of different classes, we use $C.e$, $C.s$ and $C\ (s_i, e[\phi], s_j)$ to denote the event e, state s, and transition $(s_i, e[\phi], s_j)$ in the state model of class C.

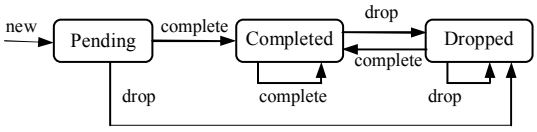

Figure 1. The state model for *Connection* class

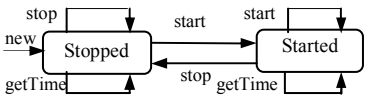

Figure 2. The state model of *Timer* class

As part of our running example, Figures 1 and 2 show the state models of classes *Connection* and *Timer* in the aspect-oriented *Telecom* simulation [1]. The states of the *Connection* class are *Pending*, *Completed*, and *Dropped*, and the events are *new*, *complete* and *drop*. Typically, a connection is established by the

complete event at the *Pending* state and then dropped by the *drop* event at the *Completed* state. The states in the *Timer* class model are *Stopped* and *Started*; the events are *new*, *start*, *stop*, and *getTime*.

3.2. Modeling Aspect-Oriented Design

As in AOP [12], 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 with respect to abstract class models (similar to the UML 2.0 protocol state machines [20], except for the post-conditions of transitions), as opposed to the abstraction level of programming constructs or control flow graphs. The preliminary modeling formalism was originally developed for the purposes of test generation from aspect-oriented state models [23][25]. A major problem with the model-based testing is that we have to inspect the aspect-oriented state models by hand when test execution reports a failure. If the models are proven correct, it can be determined that the failure has to do with the code. This paper exploits a generalized formalism for specification of aspect-oriented design so that verification of correctness can be automated. It thus improves the model-based testing process for aspect-oriented programs.

An aspect model consists of inter-model declarations (*ID*), state pointcuts (*SP*), transition pointcuts (*TP)*, and advice models (*AM*). An inter-model declaration introduces one or more new transition (state or event) to the base models. For an introduced transition $C(s_i, e[\phi], s_j)$, if s_i, s_j, and/or e are not yet in base model C, then they become a new state or event in C. A join point is a transition or state in a base model. A pointcut picks out a group of join points. Pointcuts are defined as follows:

(1) pointcut <cutname> <transition-variable>:
<base><transition> {,<base> <transition>}
(2) pointcut <cutname> (<state-variable>):
<base>.<state>{,<base>.<state>}

where (1) and (2) define transition and state pointcuts, respectively; <cutname> identifies a pointcut; <transition-variable> is a formal transition, $(s_i, e[\phi], s_j)$, where s_i, e, and s_j are variables; and <base>.<state> refers to a state 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.

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. An advice model that modifies a transition (e.g. the guard condition or resultant state) in a base model can simply have one transition. Figure 3 shows the model for a *Checking* aspect that applies to the *Connection* class in Figure 1. The first pointcut *completeAtDropped* picks out the transition join point *(Dropped, complete, Completed)* in the *Connection* model. The advice (with an empty model) means that at the *Dropped* state, the *complete* event is not applicable. The third pointcut *dropAtPending* picks out the transition *(Pending, drop, Dropped)*. The advice is that the resultant state of the *drop* event at the *Pending* state should be *Pending* (remain unchanged).

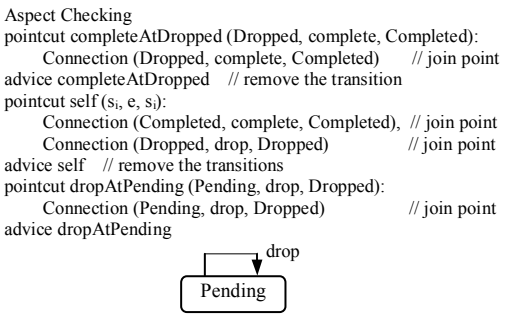

Figure 3. The *Checking* aspect model

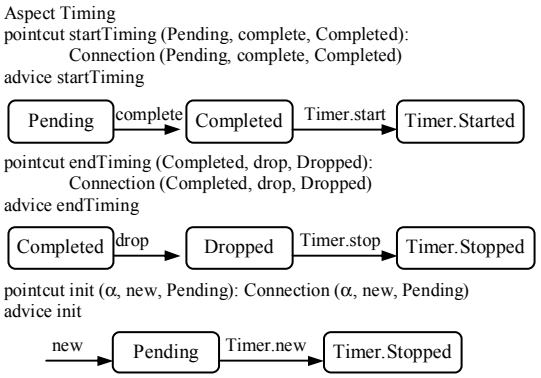

Figure 4. The *Timing* aspect model

Figure 4 shows the model of the *Timing* aspect in the *Telecom* simulation. The first pointcut picks out the transition *(Pending, complete, Completed)* in the *Connection* model. The advice is to start timing once this transition has happened. Similarly, the second pointcut picks out the transition *(Completed, drop, Dropped)*. The advice is to stop timing once the transition has happened. The third pointcut picks out the object creation transition of *Connection*, the advice is to create a *Timer* object and get ready for timing (*Timer* is called a non-base class in an advice model - it is used but not affected by aspects).

Note that both *Checking* and *Timing* take *Connection* as the base class. 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. In the *Telecom* example, we have *Checking* > *Timing*, i.e., *Checking* is applied before *Timing*. Multiple pointcuts in the same aspect can also share join points. The order in which their advice is applied to the shared transitions depends on their occurrences in the aspect model. As such, the aspect-oriented state model of a system consists of class models, aspect models, and a precedence relation on the aspect models.

4. Checking Aspect-Oriented Models

To verify an aspect-oriented state model, we first weave aspect models 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 resultant process into LTSA. LTSA then verifies whether or not the properties are violated. If violated, it reports a trace to property violation (i.e., counterexample). This helps improve the aspect-oriented state model or examine correctness of system properties. Figure 5 shows the general process for verifying the aspect-oriented state models. A prototype tool has been implemented in the MACT (Model-based Aspect Checking and Testing) toolkit to automate the transformation from aspect-oriented state models into FSP processes. In the following, we focus on the two core components of the verification process: weaving for checking and converting woven models and class models into FSP behavior processes.

4.1. 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 Section 3.1. Specifically, a generalized transition in a woven model is of the form $(s_i, e_1[\phi_1]->e_2[\phi_1]->...-> e_k[\phi_k], s_j)$ where ϕ_l (l=1,…k) is the guard for event e_l. It means the sequence of guarded events $e_1[\phi_1]->e_2[\phi_1]->...-> e_k[\phi_k]$ (called a *composite* event) results in state s_j from s_i. Typically, one of these events belongs to the base class whereas the rest are events of other classes involved. If there is only one event in the sequence, the transition reduces to a traditional one.

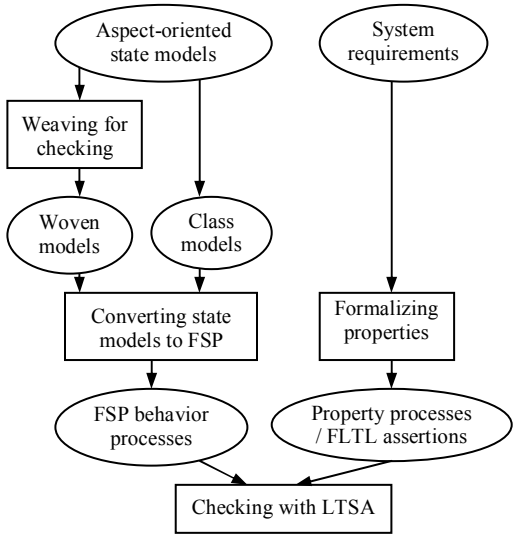

Figure 5. The model-checking process

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 non-base classes, combine the transitions that use states and events of the non-base classes into composite events (leaving out the states of the non-base 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.

Consider the *Checking* aspect in Figure 3. It has no inter-model declarations and only the base class is involved. Nothing needs to be done in steps (2) and (3). Step 4 removes three transitions from the *Connection* state model and changes the resultant state of one transition. Weaving *Checking* with *Connection* will result in the woven model in Figure 6.

Figure 6. The woven model of *Checking* and *Connection*

A woven model can further be composed with other aspect models for the same base class. The order in which multiple aspects are applied is determined by the aspect precedence relation. As such, we can apply the *Timing* aspect to the woven model in Figure 6. Step (3) in the above algorithm compresses the advice of *startTiming*, *endTiming* and *init* into the following composite transitions, respectively:

(Pending, complete -> Timer.start, Completed)
(Completed, drop -> Timer.stop, Dropped)
(α, new - > Timer.new, Pending)

Then step (4) substitutes the join point transitions with the respective composite transitions. Thus, weaving the *Checking* and *Timing* aspects with *Connection* leads to the woven model in Figure 7. It depicts how timing is applied to the connection process.

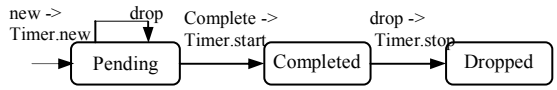

Figure 7. Woven model for Checking/Timing/Connection

4.2. From Woven Models to FSP Processes

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 non-base classes. Then we convert each woven model and class model (not modified by aspects) 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.

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* = ∅;
- 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 -> s';
  - S2.2.2 For each of other transitions, say *(s, ce, s')*, transform it to a clause | ce ->s', where "|" is the choice construct.
  - S2.2.3 For each event *e* in *E*, if there is one or more conditional transition $(s, e[\phi_1], s_1),...,(s, e[\phi_k], s_k)$ (suppose $\phi_1 \vee ... \vee \phi_k$ is not always true), generate a clause | e ->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[ϕ], s')*, such that the local process for *s'* is not generated yet, repeat Procedure 2 for *s'*.
- S2.5 Return the resultant process 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). For example, the generated FSP process for the woven model in Figure 7 is as follows:

```
CONNECTION = PENDING,
PENDING =
    (connection.complete -> timer.start -> COMPLETED
    | connection.drop -> PENDING),
COMPLETED =
  (connection.drop -> timer.stop -> DROPPED).
```

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 non-base class models. For the previous *Telecom* example, the system-level FSP process is:

```
|| TELECOM = (CONNECTION || TIMER).
```

Putting this together with the FSP processes for the woven model and the *Time* class model, we have obtained the complete FSP specification for the *Telecom* subsystem that consists of the *Connection* and *Timer* classes and *Checking* and *Timing* aspects.

5. Empirical Study

The running example in the previous sections has been verified against a number of properties. This section reports the application of our approach to a non-trivial aspect-oriented cruise control system. Its AspectJ implementation has 690 lines of code, including 143 lines of aspect code. As an aspect-oriented refactoring of a legacy Java applet [14], the system provides engine control (*engineOn*, *engineOff*, *accelerate*, *brake*) and cruise control (*on*, *off*, and *resume*) operations. Engine control events are processed by a *CarSimulator* object and cruise control events by a *Controller* object. Figure 8 shows the system architecture, where a small circle represents a relationship between a base class and an aspect. *CruiseControlIntegrator* composes *CarSimulator* with such cruise control components as *CruiseDisplay* and *Controller*, whereas aspect *SpeedControlIntegrator* composes *SpeedControl* with *Controller*. The *CarSimulatorFix* aspect solves a safety problem with the legacy system, which was found when we were testing the first executable aspect-oriented version. The failure is that the car starts accelerating immediately when, at the initial system state (engine is off), one first accelerates the car and then turns on the ignition.

According to the cruise control system requirements, we have formalized 21 properties, focusing on the required effects of the aspects. For example, the following two properties apply to the *CarSimulatorFix* and *CruiseControlIntegrator* aspects:

- The cruise controller cannot be active before the ignition has ever been on.
- The cruise controller should not be active after the controller or car engine is turned off.

They are inter-object state invariants between *CarSimulator* and *Controller* and thus affected by the *CarSimulatorFix* and *CruiseControlIntegrator* aspects. Similarly, the following two requirements apply to the *SpeedControlIntegrator* aspect:

- The cruising state cannot be entered before the speed control is enabled.
- The standby state cannot be entered before the speed control is disabled.

They are inter-object state invariants between *Controller* and *SpeedControl* and affected by the *SpeedControlIntegrator* aspect.

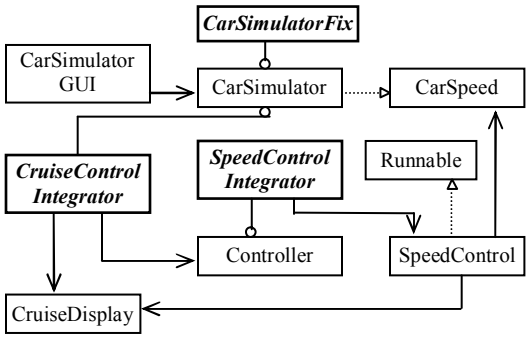

Figure 8. The aspect-oriented cruise control system

We have successfully verified all of the formalized properties against our aspect-oriented design model. No property violation was found. To further evaluate whether or not the model-checking approach can detect design defects, we created 33 variations (mutants) of the correct aspect-oriented model of the cruise control system according to the potential detects of aspect design (e.g. missing join points). 12 of them led to a deadlock and 21 violated one or more properties (e.g. variation 3-7 violated two properties #6 and #14). All mutants are determined to be flawed design models. This indicates that the model-checking approach is indeed effective in aspect verification.

6. Related Work

There is a growing body of work on aspect-oriented modeling with UML. This work exploits the meta-level

notation of UML or extends the UML notation for specifying crosscutting concerns. It is not concerned with the verification of aspect models due to the informal or semi-formal nature of UML [24]. A recent survey can be found in [17].

Since finite state models have long been in use for rigorous specification of object-oriented software [2], sate-based aspect modeling is of particular interest. Elrad et al. have proposed an approach to aspect-oriented modeling with Statecharts [4]. 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 (join points, pointcuts, and advice) with respect to state models. 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. 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 [19] use model-checking to verify whether the woven code of an aspect-oriented program contains unexpected behavior. They propose a framework that allows crosscutting properties to be defined as an aspect and thus separated from the program body. Denaro and Monga [3] report a preliminary experience with model-checking a concurrency control aspect. They manually build the aspect model in PROMELA (the SPIN input language) 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. [15] use both model checkers and model-builders to verify woven programs. The above work [3][15][19] does not involve aspect-oriented modeling.

Krishnamurthi et al. [13] 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 [9], model checking tasks are automatically generated for the woven code of aspect-oriented programs. In [8], 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 [7], 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 join points, 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. Aspect models are allowed to introduce new states, events, and transitions. Generally speaking, it is more difficult to handle the state space explosion problem at the code level than at the design level. Duo the complexity of code, "model checking programs (of real applications) often cannot completely analyze the program's state space since it runs out of memory" [21]. For assuring the quality of aspect code, we provide a combination of model-checking for correct design specification and model-based test generation for conformance testing of aspect code. Nevertheless, the approaches to modular verification of aspects [7][13] can be adopted to enhance our work.

7. Conclusions

We have presented a rigorous approach to automated verification of aspect-oriented design specification. This method can lead to two important benefits: (1) uncovering aspect design problems before code is written. This will reduce development costs due to the earlier detection of problems; and (2) determining programming faults through model-based testing. The model-based testing method [23] generates test cases from an aspect-oriented state model for exercising the resultant aspect-oriented program. A failure of test execution only indicates that the code does not conform to the model. 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, the combination of the model-checking and model-based testing methods can assure the quality of aspect-oriented programs.

The model-checking method also offers a potential for generating test cases from an aspect-oriented state model. The basic idea is to transform property violation traces (i.e., counterexamples) into test cases. Our future work will investigate how to define properties for test generation from counterexamples and integrate the generated test cases with the existing model-based testing method.

8. Acknowledgement

This work was supported in part by the ND EPSCoR IIP-SG via NSF Grant EPS-047679.

9. References

[1] AJDT (AspectJ Development Tools). http://www.eclipse. org/ajdt/

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

[3] Denaro, G. and Monga, M. "An experience on verification of Aspect Properties". In *International Workshop on Principles of Software Evolution (IWPSE)*, Vienna, Austria, Sept. 2001.

[4] Elrad, T., Aldawud, O., and Bader, A. "Expressing Aspects Using UML Behavior and Structural Diagrams". In Filman et al. [5], pp. 459-478.

[5] Filman, R.E., Elrad, T., Clarke, S., and Aksit, M. editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[6] Glusman, M. and Katz, S. "Model Checking Conformance with Scenario-based Specifications", In *Proc. of the 15th International Conference Computer Aided Verification (CAV'03)*, LNCS 2725, pp. 328-340, Boulder, CO, Springer, 2003.

[7] Goldman, M. and Katz, S. "Modular Generic Verification of LTL Properties for Aspects. In *Proc. of Foundations of Aspect Languages Workshop (FOAL06)*, 2006.

[8] Katz, E. and Katz, S. "Verifying Scenario-based Aspect Specifications". In *Proc. of the International Symposium of Formal Methods Europe*, pp. 432-447, Newcastle, UK, July 2005.

[9] Katz, S. and Sihman, M. "Aspect-Validation Using Model-Checking". In *Proc. of the International Symposium on Verification*, in honor of Zohar Manna, LNCS 2772, pp. 389-411, Springer.

[10] Katz, S. "Aspect Categories and Classes of Temporal Properties". *Transactions on Aspect-Oriented Software Development*, Rashid, A. and Aksit, M. (Eds.), LNCS 3880, 1:106-134, 2006.

[11] Katz, S. "Diagnosis of Harmful Aspects Using Regression Verification". In C. Clifton, R. Lammel, and G.T. Leavens, editors, *FOAL: Foundations of Aspect-Oriented Languages*, pp. 1-6, March 2004.

[12] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J. M. and Irwin, J. "Aspect-Oriented Programming". In *Proc. of ECOOP'97*, pp. 220-242, 1997.

[13] Krishnamurthi, S., Fisler, K., and Greenberg, M. "Verifying Aspect Advice Modularly". In *Proc. of FSE'04*, Newport Beach, CA, Nov. 2004.

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

[15] Nelson, T., Cowan, D.D. and Alencar, P. S. C. "Supporting Formal Verification of Crosscutting Concerns". In *Reflection*, pp. 153–169, 2001.

[16] Rinard, M., Salcianu, A., and Bugrara, S. "A Classification System and Analysis for Aspect-Oriented Programs". In *Proc. of FSE'04*, Nov. 2004.

[17] Schauerhuber, A., Schwinger, W., Retschitzegger, W., and Wimmer, M. "A Survey on Aspect-Oriented Modeling Approaches", *Technical Report*, Vienna University of Technology. January 2006.

[18] Sullivan, K., Griswold, W.G., Song, Y., Cai, Y., Shonle, M., Tewari, N, and Rajan, H. "Information Hiding Interfaces for Aspect-Oriented Design", In *Proc. of ESEC/FSE'05*, pp. 166-175, 2005.

[19] Ubayashi, N. and Tamai, T. "Aspect-Oriented Programming with Model Checking". In *Proc. of AOSD'02*, pp. 148-154, The Netherlands, 2002.

[20] *UML 2.0 Specification*. http://www.omg.org/ technology/documents/ formal/uml.htm

[21] Visser, W., Pasareanu, C.S., and Khurshid, S. "Test Input Generation with Java PathFinder". In *Proc. of ISSTA'04*, 2004.

[22] Xu, D. and Nygard, K. "Threat-Driven Modeling and Verification of Secure Software Using Aspect-Oriented Petri Nets". *IEEE Trans. on Software Engineering.* Vol. 32, No. 4, pp. 265-278, April 2006.

[23] Xu, D. and Xu, W. "State-Based Incremental Testing of Aspect-Oriented Programs". In *Proc. of AOSD'06*, pp. 180-189, Bonn, Germany, March 2006.

[24] Xu, D., Xu, W. and Wong, W. E. "Testing Aspect-Oriented Programs with UML Design Models", *International Journal of Software Engineering and Knowledge Engineering*, To appear.

[25] Xu, W. and Xu, D. "State-Based Testing of Integration Aspects". In *Proc. of Second Workshop on Testing of Aspect-Oriented Programs (WTAOP'06)*. In conjunction with ISSTA'06, July 2006, USA.