Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2013-IJ-003

Verifying Aspect-Oriented Models Against Crosscutting Properties.

Zhanqi Cui Linzhang Wang Xi Liu Lei Bu Jianhua Zhao Xuandong Li

Postprint Version. In International Journal of Software Engineering and Knowledge Engineering, Vol. 23, No. 05, World Scientific, 2013, pp. 655-676.

International Journal of Software Engineering
and Knowledge Engineering
Vol. 23, No. 5 (2013) 655–676

DOI: 10.1142/S0218194013400123

VERIFYING ASPECT-ORIENTED MODELS AGAINST CROSSCUTTING PROPERTIES

ZHANQI CUI[*,†,‡], LINZHANG WANG[*,§],
XI LIU[*], LEI BU[*], JIANHUA ZHAO[*]
and XUANDONG LI[*]

[*]*State Key Laboratory of Novel Software Technology*
Department of Computer Science and Technology
Nanjing University
Nanjing, 210046, China

[†]*Key Laboratory of Optical Fiber Sensing*
and Communications, Ministry of Education
School of Communication and Information Engineering
University of Electronic Science and Technology of China
Chengdu, 611731, China
[‡]*zqcui@seg.nju.edu.cn*
[§]*lzwang@nju.edu.cn*

Dealing with crosscutting concerns has been a critical problem in software development processes. To facilitate handling crosscutting concerns at design phases, we proposed an aspect-oriented modeling and integration approach with UML activity diagrams. The primary concerns are depicted with UML activity diagrams as primary models, whereas crosscutting concerns are described with aspectual extended activity diagrams as aspect models. Aspect models can be integrated into primary models automatically. The AOM approach can reduce the complexity of design models. However, potential faults that violate desired properties of the software system might still be introduced during the modeling or integration processes. The verification technique is well-known for its ability to assure the correctness of models and uncover design problems before implementation. We propose a framework to verify aspect-oriented UML activity diagrams based on Petri net verification techniques. For verification purpose, we transform the integrated activity diagrams into Petri nets and prove the consistency of the transformation. Then, crosscutting concerns in system requirements are refined to properties in the form of CTL formulas. Finally, the Petri nets are verified against the formalized properties to report whether the aspect-oriented design models satisfies the requirements. Furthermore, we implement a tool named Jasmine-AOV to support the verification process. Case studies are conducted to evaluate the effectiveness of the proposed approach.

Keywords: Aspect-oriented modeling; verification; model checking; activity diagram; Petri net.

1. Introduction

Dealing with crosscutting concerns has been a critical problem during software development life cycles. Aspect-oriented programming (AOP) [1] provides a viable programming level solution by modularizing crosscutting concerns into aspects. Aspect-oriented modeling (AOM) handles crosscutting concerns by providing a higher level of abstraction to alleviate software complexity in the design phase. In particular, earlier awareness of cross-cutting concerns in the model-centric design can guide the subsequent implementation and validation activities.

To facilitate handling crosscutting concerns at earlier software development phases, in our previous work, we proposed an AOM approach based on UML activity diagrams [2]. The approach shifts aspect-oriented techniques [1] from a code-centric to a model-centric, which is employed to handle the crosscutting concerns during design phases. Thus, it alleviates software complexity in a more abstract level. The primary functional concerns are modeled with activity diagrams, and crosscutting concerns are modeled with aspectual activity diagrams, respectively. Then the overall system design model, which is also an activity diagram, is integrated by weaving aspect models into primary models.

However, aspect-oriented modeling techniques cannot guarantee the correctness of produced design models. For instance, wrong weaving sequences may cause the integrated models to violate system crosscutting requirements. Design models are widely used as a basis of subsequent implementation [3, 4] and testing [5–7] processes. As a result, it is costly if defects in design models are discovered at later implementation and testing stages. Therefore, assuring the correctness of the aspect-oriented design models is vitally important. So far, the applicable approach is manual review, which is time consuming and depends on reviewers' expertise. However, existing automatic verification tools cannot deal with aspect-oriented activity diagrams directly.

In order to ensure that crosscutting concerns are correctly modeled, we propose a rigorous approach to automatically verify aspect-oriented models (activity diagrams) by using Petri net based verification techniques. Firstly, the integrated activity diagrams are translated into Petri nets. Then, crosscutting concerns in system requirements are refined to properties in the form of CTL formulas. Finally, the Petri nets are verified against the formalized properties.

The rest of this paper is organized as follows. Section 2 presents backgrounds of activity diagrams, Petri nets, and a running example. Section 3 discusses the verification of aspect-oriented activity diagrams. Section 4 presents two case studies and evaluations of our approach. Section 5 reviews the related work. Finally, Sec. 6 concludes the paper and discusses the future work.

2. Background

In this section, we briefly introduce UML activity diagrams, Petri nets, and a running example that will be employed to demonstrate our approach in the following sections.

2.1. *Activity diagrams and Petri nets*

The UML activity diagram is a powerful tool to describe control flow based program logic at different levels of abstraction. Designers commonly use activity diagrams to describe the sequence of behaviors between classes in a software system. Nodes and edges are two kinds of elements in activity diagrams. Nodes in activity diagrams are connected by edges. We formally define activity diagrams as follows.

Definition 1. (Activity Diagram). An activity diagram AD is a 3-tuple (N, E, F), where:

- $N = \{n_1, n_2, \ldots, n_i\}$ is a finite set of **nodes**, which contains action, initial/final, decision/merge and fork/join nodes, $n_I \in N$ is the initial activity state, $N_F \subseteq N$ is a set of final activity states, N_a is the subset of action nodes in N, that $N_a = \{n_i \mid n_i \in N \text{ and } n_i \text{ is an action node}\}$;
- $E = \{e_1, e_2, \ldots, e_j\}$ is a finite set of **edges**;
- $F \subset (N \times E) \cup (E \times N)$ is the **flow** relation between nodes and edges.

We call $ep = N_0 \xrightarrow{E_0} N_1 \cdots \xrightarrow{E_{k-1}} N_k \xrightarrow{E_k} \cdots \xrightarrow{E_{n-1}} N_n$ is a **path** in AD, where $N_k \subseteq N$ and $E_k \subseteq E$, $N_{k+1} = (N_k - \{n_k\}) \cup \{n_{k+1} \mid (n_k, e_k) \in F \wedge (e_k, n_{k+1}) \in F \wedge e_k \in E_k\}$. The **action node sequence** of ep is denoted as $AS(ep) = N_{a0} \rightarrow N_{a1} \rightarrow \cdots \rightarrow N_{ak} \cdots \rightarrow N_{an}$, which is the projection of ep on action node set N_a, where $N_{ak} = \{n_i \mid n_i \in N_a \wedge n_i \in N_k\}$. By removing all the empty set N_{ak} in AS (ep), we can get the **action trail** τ(ep) of $AS(ep)$.

Due to the nature of UML is semi-formal and UML diagrams are design-oriented models, translating activity diagrams into formal verification-oriented models is necessary before verification. In this approach, we translate activity diagrams into Petri nets, because in UML 2, the semantics of activity diagrams is explained in terms of Petri net notations [8], like tokens, flows, etc. The Petri net is a formal specification language that is widely used to model software behaviors. A Petri net consists of places, transitions, and arcs. Like UML activity diagrams, Petri nets offer a graphical notation for stepwise processes that include choice, iteration, and concurrent execution. On the other hand, Petri nets have a precise mathematical definition of their execution semantics, with a well-developed mathematical theory for process analysis. A Petri net is formally defined as follows.

Definition 2. (Petri net) A Petri net [9] is a 4-tuple $PN = \{P, T, A, M_0\}$, where

- P is a finite set of **places** and T is a finite set of **transitions**, and P and T are disjoint;
- A is a finite set of **arcs** connect between places and transitions, where $A \subseteq (P \times T \cup T \times P)$;
- M_0 is the initial **marking**.

A marking M of PN is any subset of P. For any transition t, ${}^\bullet t = \{p_i \mid (p_i, t) \in A\}$ is the incoming places of t, $t^\bullet = \{p_i \mid (t, p_i) \in A\}$ is the outgoing places of t. A transition

t is enabled in a marking M if $^{\bullet}t \subseteq M$, otherwise, it is disabled. Let $enabled(M)$ be the set of transitions enabled in M.

We say $pnp = M_0 \xrightarrow{t_0} M_1 \cdots \xrightarrow{t_{k-1}} M_k \xrightarrow{t_k} \cdots$ is a path in PN, where $M_K \subseteq P$, $t_k \subseteq T$, $^{\bullet}t_k \subseteq M_k$, and $M_{k+1} = \{M_k - {^{\bullet}t_k}\} \cup t_k{^{\bullet}}$. Given a path $pnp = M_0 \xrightarrow{t_0} M_1 \cdots \xrightarrow{t_{k-1}} M_k \xrightarrow{t_k} \cdots$ in PN, the corresponding transition sequence of pnp is $TS(pnp) = enabled(M_0) \rightarrow enabled(M_1) \rightarrow \cdots \rightarrow enabled(M_k) \cdots$.

Places, transitions and arcs in Petri nets are drawn as circles, boxes and arrows, respectively. We do not consider weights of arcs in this paper for simplification.

2.2. *Aspectual extensions for activity diagrams*

The aspect-oriented modeling technique is adapted from our previous work in [2]. In this subsection, we briefly introduce aspectual extensions into the activity diagram to model crosscutting concerns.

For modeling crosscutting concerns, we follow such terms like "joinpoint", "pointcut", and "advice" from the terminology of AspectJ [10] into the UML activity diagram with similar meanings. Join points are elements in a primary model that are appropriate to insert advice models before or after. A pointcut model is used to filter join points in primary models to which the corresponding advices should be applied. An advice model is used to specify additional enhancements or mitigations for a crosscutting concern.

As described in Table 1, in order to model aspects, 7 stereotypes and 3 tagged values are added to the activity diagram. Extensive explanations and discussions about these extensions are beyond the scope of this paper and can be found in [2].

Table 1. Extensions for modeling aspects.

Extension	Type	Applies to	Description
≪Pointcut≫	Stereotype	Diagram	Indicate an activity diagram is a pointcut model.
advice	Tagged Value	≪Pointcut≫	Indicate the corresponding advice model of the pointcut model.
≪Joinpoint≫	Stereotype	Element	Denote the position of the join point element in a pointcut model.
≪Argument≫	Stereotype	Element	Indicate elements that serve as actual arguments for related formal parameters in the corresponding advice model.
parameter	Tagged Value	≪Argument≫	Denote the name of the element in the advice model which are related to the ≪Argument≫ element.
≪Advice≫	Stereotype	Diagram	Indicate an activity diagram is an advice model.
type	Tagged Value	≪Advice≫	Indicate the type of the advice model; "type" is either "Before" or "After".
≪Entry≫	Stereotype	Node	Denote where tokens flow in an advice model from primary models.
≪Exit≫	Stereotype	Node	Denote where tokens flow out an advice model to primary models.
≪Parameter≫	Stereotype	Element	Indicate elements that serve as formal parameters in an advice model.

In Table 1, the "Extension" column is the name of the extension, and the "Type" column indicates the type of the extension that can be either a stereotype or a tagged value. The "Applies to" column specifies the type of objects that the extension can be applied to. The "Description" column gives a brief introduction about the extension.

2.3. *CTL and LoLA*

Computation tree logic (CTL) [11] is a kind of branch time logic which can reason about many execution paths at one time. CTL provides two path quantifiers: universal (**A**) and existential (**E**) in combination with four temporal operators: **X**(next time), **F**(eventually), **G**(globally), and **U**(until). In a CTL formula, every temporal operator is preceded by a path quantifier. The syntax of CTL formula would be defined as:

Assume AP is the underlying set of atomic propositions, then

- p, $q \in AP$ are CTL formulas;
- $true \,|\, false \,|\, \neg p \,|\, p \wedge q \,|\, p \vee q$ are CTL formulas;
- ϕ, ψ are CTL formulas, then $\mathbf{AX}\,\phi \,|\, \mathbf{AF}\,\phi \,|\, \mathbf{AG}\,\phi \,|\, \phi\,\mathbf{AU}\,\psi \,|\, \mathbf{EX}\,\phi \,|\, \mathbf{EF}\,\phi \,|\, \mathbf{EG}\,\phi \,|\, \phi\,\mathbf{EU}\,\psi$ are CTL formulas.

LoLA (a Low Level Petri Net Analyzer) [12] has been implemented for the validation of reduction techniques for place/transition net state spaces. LoLA can verify various properties, such as reachability of a given state or state predicate, boundedness of the net or a place, deadlocks, dead transitions, reversibility, and CTL formulas. In LoLA, the path quantifiers **A**, **E** and temporal operators **X**, **F**, **G**, **U** are replaced by **ALLPATH**, **EXPATH**, **NEXTSTEP**, **EVENTUALLY**, **ALWAYS**, **UNTIL**, respectively. The verification task is inputted by a ".task" file in which a CTL formula is announced by the keywords of LoLA.

2.4. *Running example*

We adapt the order processing scenario from [8] as a running example to demonstrate our approach. The requirements of this scenario are described in Fig. 1. As Fig. 1 shows, there are 4 crosscutting concerns related to this scenario: authentication, validation, logging, and informing.

Figure 2 is the primary model of the order processing scenario, which consists of 3 main steps: fill order, ship order, and close order. Based on the requirements in Fig. 1 and our previous aspect-oriented modeling approach [2], the crosscutting concerns of the running example are modeled in Fig. 3.

In order to understand how crosscutting concerns will affect primary functionalities, aspect models are integrated with primary models to generate an overall system design model. Different weaving sequences would produce different integrated models. For example, we add an authorization aspect in the running example, which describes the logged-in user need to be checked whether she/he has the permission to

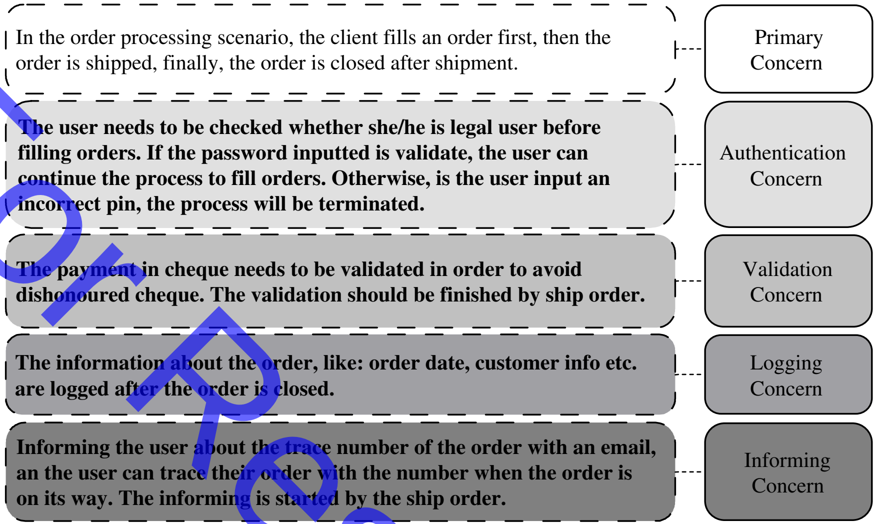
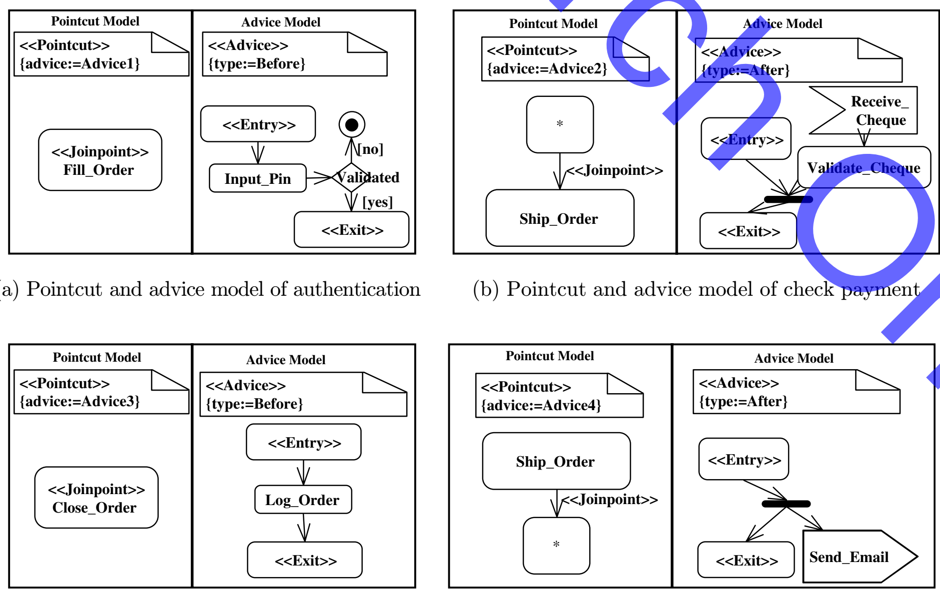


Fig. 1. The requirements of the order processing scenario.

Fig. 2. The primary model of the order processing scenario.



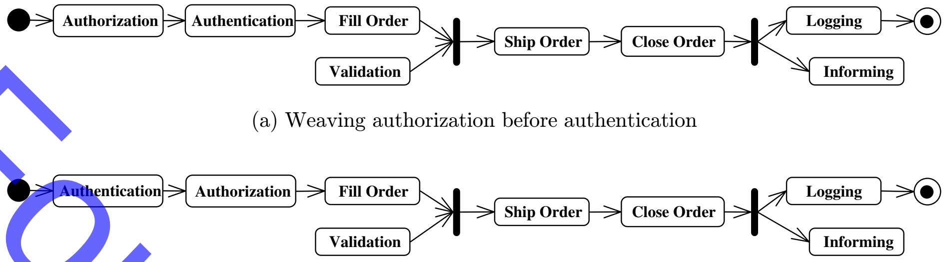
(a) Pointcut and advice model of authentication

(b) Pointcut and advice model of check payment

(c) Pointcut and advice model of logging

(d) Pointcut and advice model of informing

Fig. 3. Pointcut and advice models of the order processing scenario.



(a) Weaving authorization before authentication

(b) Weaving authentication before authorization

Fig. 4. Two different integrated models of the order processing scenario.

fill orders. If the authorization aspect is woven before authentication, then the result of integration is shown in Fig. 4(a). Otherwise, if the authentication aspect is woven before authorization, then the result of integration is shown in Fig. 4(b). As we know, the legal user has to be logged-in before being checked whether the corresponding permission is granted or not. As a result, the authentication aspect should be woven firstly, and Fig. 4(b) is the correct integration result we expected. Extensive explanations and algorithms about the integration approach can be also found in our previous paper [2].

3. Verifying Aspect-Oriented Models

In our previous work [2], aspect-oriented models, including primary models, aspect models, as well as integrated models, were all depicted with UML activity diagrams. Since the correctness of the integration process cannot be guaranteed, how to ensure the consistence between the integrated activity diagrams and crosscutting requirements becomes a critical research problem. In UML 2, the semantics of activity diagrams is explained in terms of Petri nets. There are also various automatic tools, i.e. LoLA [12], verifying Petri nets against specified properties. As a result, if we can translate activity diagrams into Petri nets, we could verify the activity diagram models by verifying corresponding Petri net models for specific properties. In this section, we first discuss transformation from activity diagrams to Petri nets, and then present the verification against crosscutting concerns.

3.1. *Transforming from activity diagrams to Petri nets*

We adapt the mapping semantics of control-flows in UML 2 activities in [13] to convert activity diagrams into Petri nets. Basically, action nodes and fork/join nodes are translated to net transitions, control nodes (initial, final, decision, and merge nodes) become net places, and edges are transformed to net arcs. Auxiliary transitions or places are added when the ends of an arc both are transitions or both are places. For simplification, were strict an activity diagram only consists of action nodes, control nodes, and control flows in this approach. For bridging Petri nets and UML activity diagrams smoothly, we define a Petri net corresponding to a UML activity diagram by extending Definition 2.

Definition 3. (PN4AD) A Petri net transformed from an activity diagram $AD = (N, E, F)$, is a tuple $PN4AD = (P, T, T_A, A, M_0)$, where (P, T, A, M_0) is a PN, and $T_A \subseteq T$ is a finite set of transitions which are transformed from action nodes N_A of AD.

Based on the mapping rules in [13], we construct an algorithm to transform activity diagrams to Petri nets. The algorithm is described in Algorithm 1. With the algorithm, the activity diagram of the running example in Fig. 4(b) is converted to the Petri net in Fig. 5. The transformation of more complex activity diagrams (containing data flows, exceptions, and expansions etc.) is straightforward based on transformation rules in [14].

Algorithm 1. Convert an activity diagram into a Petri net

```
INPUT: AD := an activity diagram
OUTPUT: PN(P, T, A, M_0) := a Petri net
for each node n in AD
        if n is an initial node, final node, decision node, or merge node
                Generate a corresponding place in PN.P
        else // action node, fork node, or join node
                Generate a corresponding transition in PN.T
for each edge e in AD
        N_1 := source node of e in AD
        N_2 := target node of e in AD
        M_1 := corresponding place or transition of N_1 in PN.P
        M_2 := corresponding place or transition of N_2 in PN.P
        if both N_1 and N_2 ∈ (initial nodes ∪ final node ∪ decision
          node ∪ merge node)
                Generate an auxiliary transition T_1 in PN.T
                Generate an arc start from M_1 to T_1 in PN.A
                Generate an arc start from T_1 to M_2 in PN.A
        else if both N_1 and N_2 ∈ (action node ∪ fork node ∪ join node)
                Generate an auxiliary place P_1 in PN.P
                Generate an arc start from M_1 to P_1 in PN.A
                Generate an arc start from P_1 to M_2 in PN.A
        else
                Generate an arc start from M_1 to M_2 in PN.A
for each place without an incoming arc
        Generate an initial token for that place in PN.M_0
return PN
```

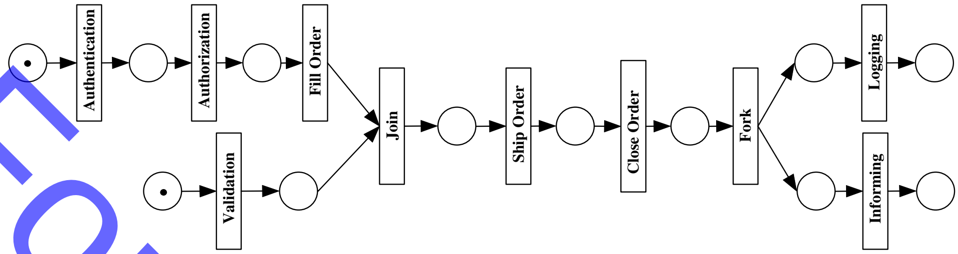


Fig. 5. The Petri net transformed from the order processing scenario.

For a *PN4AD*, given a path $pnp = M_0 \xrightarrow{t_0} M_1 \cdots \xrightarrow{t_{k-1}} M_k \xrightarrow{t_k}$ in PN, and the corresponding transition sequence of pnp is $TS(pnp) = enabled(M_0) \to enabled(M_1) \to \cdots \to enabled(M_k)$. The **action transition sequence** of $TS(pnp)$ is $ATS(pnp) = A_0 \to A_1 \cdots \to A_k$, where $A_i = \{t_{i,j} \mid t_{i,j} \in enabled(M_i) \text{ and } t_{i,j} \in T_A\}$. By removing all the empty A_i in $ATS(pnp)$, we can get the **action trail** $\sigma(php)$ of $ATS(php)$.

In order to ensure the conformance of the transformation, we define a theorem as follows. The proof of the theorem is in Appendix A.

Theorem 1. *Given an Activity Diagram* $AD = (N, E, F)$, *and the corresponding* $PN4AD = (P, T, T_A, A, M_0)$, *for any path* ep *of* AD, *there must be a path* pnp *of* $PN4AD$, *and the action trail* $\tau(ep)$ *is equivalent with the action trail* $\sigma(pnp)$, *vice versa.*

Since the transformed Petri net shares the same action trail with the activity diagram, we can achieve the verification of the activity diagram by verifying the equivalent Petri net against same system properties defined on the sequence of action nodes.

3.2. *Verifying Petri nets*

Crosscutting concerns describe the running sequences between advices and primary behaviors in all paths of integrated models. These properties can be described in the form of CTL formulas [11] naturally. CTL formulas cannot be generated from aspect models by synthesizing conditions of join points specified by pointcut models and checking the corresponding advice models appears at right places. This is because that the context specified by a pointcut model would be changed after integration, and the join points matched by the pointcut model could no longer exist. In this approach, the properties to be checked are directly refined from crosscutting requirements.

3.2.1. *Properties specified from the requirement*

Based on the Petri net generated, we can easily analyze reachability, safety, liveness, and fairness properties [9]. In this approach, we only focus on checking properties that are closely related to crosscutting concerns. We categorize crosscutting concerns from two facets. Firstly, according to the execution sequence between actions in advice models and join points, a crosscutting concern can be either executing before or after join points. Secondly, the execution of a crosscutting concern is either sequential or parallel with the primary behaviors. Sequential crosscutting concerns are synchronous features that their running positions are restricted by the join points. Parallel crosscutting concerns are asynchronous features that are running concurrently with primary actions and they are finished or started by the join points.

(1) *Before-crosscutting concerns*

A before-crosscutting concern specifies some extra behaviors must be performed before matched join points. A before aspect would be either sequential or parallel with respect to the flows of primary models.

If it's a sequential aspect, the behaviors specified by the aspect model are executed before the join point node. The keyword of sequential before crosscutting concerns in requirements level is "before". In the integrated model, actions specified by the aspect model are executed between the join point node and the predecessor node of the join point in the primary model.

Otherwise, if it's a parallel aspect, the behaviors in the aspect model must be finished at the join point edge. The keyword of parallel before crosscutting concerns in requirements is "be finished by". In the integrated model, the actions of the crosscutting concern are running concurrently with the primary behaviors, and then synchronized at the join node which replaced the join point edge.

In corresponding Petri nets, assume jp is the transition transformed from one of the join point, ad is the transition transformed from the structured activity node that represents the advice model. The requirement of a before aspect can be represented in the form of the CTL formula as: $\mathbf{AG}\neg((ad \wedge \mathbf{EX}(\neg ad \wedge \neg jp)) \vee ((\neg ad \wedge \neg jp) \wedge \mathbf{EX} jp))$.

(2) *After-crosscutting concerns*

An after-crosscutting concern specifies some actions must be performed after matched join points. An after-crosscutting concern can also be either a sequential or a parallel aspect with respect to the flows of primary models.

If it's a sequential aspect, the behaviors specified by the aspect model are executed after the join point node. The keyword of sequential after crosscutting concerns in requirements level is "after". In the integrated model, actions specified by the aspect model are executed between the join point node and the successor node of the join point node in the primary model.

Otherwise, if it's a parallel aspect, the behaviors in the aspect model must be started by the join point edge. The keyword of parallel before crosscutting concerns in requirements is "be started by". In the integrated model, the actions of the crosscutting concern are enabled by the fork node, which replaced the join point edge, and then running concurrently with primary behaviors.

In corresponding Petri nets, assume jp is the net transition transformed from the join point, ad is the net transition transformed from the structured activity node that represents the advice model. The requirement of an after aspect can be represented in the form of the CTL formula as: $\mathbf{AG}\neg((jp \wedge \mathbf{EX}(\neg jp \wedge \neg ad)) \vee ((\neg jp \wedge \neg ad) \wedge \mathbf{EX} ad))$.

3.2.2. *Conflicts of multiple crosscutting concerns*

The CTL formula need to be adjusted if more than one crosscutting concerns (which are all "before" aspects or are all "after" aspects) match a same join point. Because the running sequence between one aspect and a join point can be affected by other aspects of the same before/after kind, which match the same join point. For instance, in the running example, the authentication and authorization concerns are conflicting because they both are before-crosscutting aspects and they have same join point, the "Fill_Order" action. The running sequence of the authentication aspect and the "Fill_Order" operation will be changed from "Authentication → Fill_Order" to "Authentication → Authorization → Fill_Order" after the weaving of the authorization aspect.

(1) *Conflicts between two before-crosscutting concerns*

For a before-crosscutting concern cc_1 with advice model ad_1 and join point jp_1, if any other before aspect, which matches the same join point jp_1 and weaves after cc_1, then some extra actions are performed after ad_1 and before jp_1. Assume it's a before-crosscutting concern cc_2 with advice ad_2 weaves after cc_1, then jp_1 should be replaced by ad_2 in the CTL formula of cc_1 as: $\mathbf{AG}\neg((ad_1 \wedge \mathbf{EX}(\neg ad_1 \wedge \neg ad_2)) \vee ((\neg ad_1 \wedge \neg ad_2) \wedge \mathbf{EX} ad_2))$.

(2) *Conflicts between two after-crosscutting concerns*

For a after-crosscutting concern cc_1 with advice model ad_1 and join point jp_1, if any other after aspect, which matches the same join point jp_1 and weaves after cc_1, then some extra actions are performed after jp_1 and before ad_1. Assume it's a after-crosscutting concern cc_2 with advice ad_2 weaves after cc_1, then jp_1 should be replaced by ad_2 in the CTL formula of cc_1 as: $\mathbf{AG}\neg((ad_2 \wedge \mathbf{EX}(\neg ad_2 \wedge \neg ad_1)) \vee ((\neg ad_2 \wedge \neg ad_1) \wedge \mathbf{EX} ad_1))$.

In the running example, the authentication and authorization aspects are conflicting, because they match the same join point: "Fill_Order". Suppose the authentication aspect was weaved before authorization aspect. Base on above conflicts solving rules, the crosscutting requirements of authorization aspect remain unchanged as:

$$\mathbf{AG}\neg\,((\mathit{Authorization} \wedge \mathbf{EX}(\neg \mathit{Authorization} \wedge \neg \mathit{Fill\_Order})) \vee ((\neg \mathit{Authorization} \wedge \neg \mathit{Fill\_Order}) \wedge \mathbf{EX}\mathit{Fill\_Order})) \tag{1}$$

and the crosscutting requirements of authentication aspect need to be changed from:

$$\begin{aligned}&\mathbf{AG}\neg\,((\mathit{Authentication} \wedge \mathbf{EX}(\neg \mathit{Authentication} \wedge \neg \mathit{Fill\_Order}))\vee\\ &\quad((\neg \mathit{Authentication} \wedge \neg \mathit{Fill\_Order}) \wedge \mathbf{EX}\mathit{Fill\_Order}))\end{aligned} \tag{2}$$

to:

$$\begin{aligned}&\mathbf{AG}\neg\,((\mathit{Authentication} \wedge \mathbf{EX}(\neg \mathit{Authentication} \wedge \neg \mathit{Authorization}))\vee\\ &\quad((\neg \mathit{Authentication} \wedge \neg \mathit{Authorization}) \wedge \mathbf{EX}\mathit{Authorization}))\end{aligned} \tag{3}$$

3.2.3. *Verification*

After the system crosscutting properties are refined as a set of CTL formulas. We can verify the Petri net against specified CTL formulas generated. If the verification is passed, it means the model satisfies the corresponding crosscutting requirements. Otherwise, the model violates the corresponding crosscutting requirements to some extent, which means further revision about the model is needed.

In the running example, the integrated model in Figs. 4(a) and 4(b) are both verified against the crosscutting requirements of authentication, authorization, validation, logging, and informing. First, the integrated models are transformed to Petri nets. Then the 5 crosscutting requirements are refined to 5 CTL formulas. Finally, Petri net analyzer LoLA is employed to verify the two Petri nets against the formalized crosscutting requirements, respectively.

The Petri net transformed from the model in Fig. 4(b) passes the verification process and output "result: true" for all the 5 CTL formulas. While the Petri net transformed from the model in Fig. 4(a) fails when verifying against the 2 CTL formulas generated from authentication and authorization requirements, and passes the verification against the other 3 CTL formulas. In the Petri net transformed from the model in Fig. 4(a), the fire of transition "Authentication" will enable transition "Fill_Order", which violates CTL formula (3). And the fire of transition "Authorization" will enable transition "Authentication", which violates CTL formula (1). This verification results show that the crosscutting requirements of authentication and authorization are not hold in the aspect-oriented model. After correcting the weaving preference fault and integrating the aspect model again, the new integrated model passes the verification process.

3.3. *Tool Implementation*

We implemented a tool named Jasmine-AOV[a] based on Topcased[b] and LoLA.[c] As Fig. 6 shows, this tool is composed of 4 main parts: Model Transformer, Crosscutting Concern Manager, CTL Generator, and Model Checker. The Model Transformer converts an activity diagram to a Petri net automatically. The inputs of Model

[a] Jasmine-AOV, http://seg.nju.edu.cn/∼zqcui/Jasmine-AOV
[b] Topcased, http://www.topcased.org/
[c] LoLA, http://www.informatik.uni-rostock.de/tpp/lola/

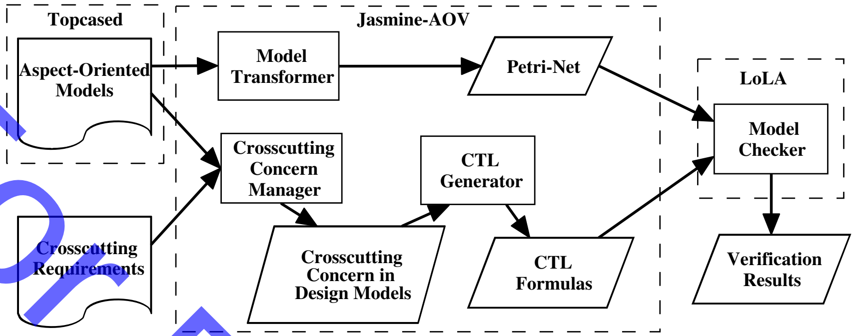


Fig. 6. The framework of Jasmine-AOV.

Transformer are UML diagrams designed by Topcased in the form of XML files and the outputs of the tool are Petri net files which are readable for LoLA to perform verification tasks. The Crosscutting Concern Manager is used to manage mapping relations between crosscutting concerns in requirements and elements in corresponding activity diagrams. It provides an assistant for mapping textual crosscutting requirements to design activity diagrams. The CTL Generator can automatically generate CTL formulas from crosscutting requirements that are mapped to design models. The CTL Generator also supports users to input CTL formulas manually. Model Checker is implemented by directly wrapped an existing checker: LoLA. It can verify the Petri net against crosscutting properties in the format of CTL formulas and report the result.

The screenshot of Jasmine-AOV is in Fig. 7. The "Crosscutting concerns" area manages the crosscutting requirements which are mapped to design models. The "New Crosscutting Concern"dialog provides a wizard for mapping textual crosscutting requirements to design activity diagrams. The "Petri net" area displays the Petri net transformed from the corresponding activity diagram. The "CTL Formulas" area lists the formulas refined from crosscutting concerns in the "Crosscutting concerns" area automatically or wrote by users manually. The "Verification Results" area outputs the results of verifying the Petri net in the "Petri net" area against the CTL formulas in the "CTL Formulas" area by LoLA.

Writing complex CTL formulas is not easy for a software engineer without proper training about formal methods. To tackle this problem, we implement the CTL Generator to assist generating CTL formulas automatically. As Fig. 7 shows, the user only need to select actions which is the advice, the join points, and the relationship between the advice and the join points, based on the textual description of the crosscutting concern. After this information is inputted, the CTL Generator generates a CTL formula for the crosscutting concern and adjusts CTL formulas if there is more than one aspect of the same before/after type apply on a same join point.

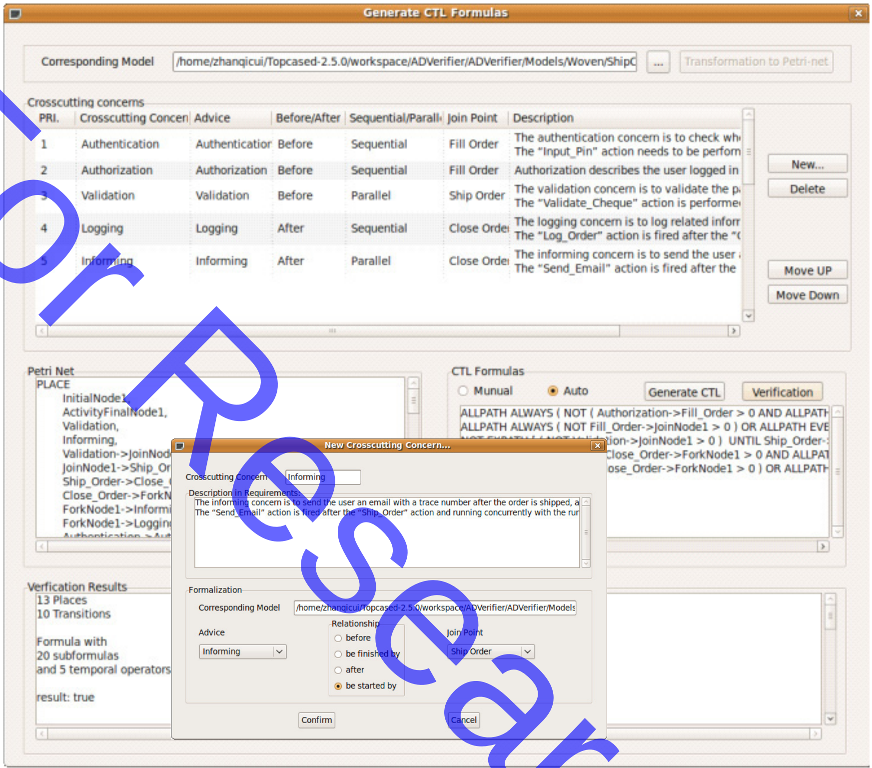


Fig. 7. The screenshot of Jasmine-AOV.

4. Evaluation and Case Suites

To evaluate the effectiveness of our approach, we have applied our approach to the design models adapted from the Ship Order example in [8] and the Telecom System.[d] The Ship Order example contains 5 crosscutting concerns and the Telecom System contains 6 crosscutting concerns. For both case studies, we transformed the integrated models to Petri nets, and mapped crosscutting requirements to the design models with the help of the tool. Then, corresponding CTL formulas of verification tasks are generated automatically. Finally, the Petri nets are checked against the CTL formulas generated.

The faults of aspect-oriented models, which can be caused by design defects or incorrect integration processes, are categorized as follows:

1. Aspect model faults

(a) **Incorrect weaving preference.** The priorities of aspect models are incorrectly assigned. This kind of faults will lead to match join pints faults or running sequence changed unexpected.

[d] AJDT Toolkit: http://www.eclipse.org/ajdt

Table 2. Model mutants of the 2 case studies.

Fault types		Ship order	Telecom system
Aspect model faults	Incorrect weaving preference	1	1
	Incorrect binding	5	3
Pointcut model faults	Overmatch join points	5	6
	Mismatch join points	5	6
	Incorrect position of join point	5	6
Advice model faults	In correct type of advice models	5	6
Number of model mutants in total		26	28
Mutants killed		22	28
Equivalent mutants		4	0

(b) **Incorrect binding between pointcut model and advice model.** The pointcut model is incorrectly mapped to an unrelated advice model. This kind of faults will result in improper advice models apply at some join points.

2. Pointcut model faults

(a) **Overmatch/Mismatch join points.** The pointcut model matches extra join points or miss some join points should be matched. The consequence of this kind of faults is that extra advices are performed at unexpected join points or desired advices are not going to be performed at join points.

(b) **Incorrect position of join points.** The element which serves as a join point in the pointcut model is incorrectly appointed. The phenomenon of this kind of faults is that advices are applied at incorrect points of the primary model.

3. Advice model faults

(a) **Incorrect type of advice models.** The type of the advice model is declared incorrectly. This kind of faults will cause the running sequence between the advice model and the primary model change unexpectedly.

To further evaluate the ability of our approach to detect the faults of aspect-oriented models, mutated models are created based on preceding category of aspect model faults. 26 and 28 model mutants are constructed for the 2 case studies, respectively. 22 out of 26 mutants for ship order case study and all mutants for telecom system case study are killed because they violate the crosscutting requirements from various ways and these violations are detected by the verification process. The 4 alive mutants of the ship order case study are manually checked, and turn out to be equivalent mutants. This result illustrates the ability of our approach to find the faults in aspect-oriented models and to improve the quality of design models. Table 2 classifies all these model mutants by their fault types and the verification results.

5. Related Work

There are many research projects on bringing aspect-oriented ideas to software requirement engineering from different perspectives. Whittle and Araujo [15] focus on scenario-based requirements and composing them with aspects to generate a set of state machines that represent the composed behaviors from both aspectual and non-aspectual scenarios. In contrast, our approach is carried out at the design level instead of requirement level. However, our approach can be enhanced with the aspect mining tool at requirements level, like EA-Miner [16], by inputting crosscutting concerns detected by these tools to our Jasmine-AOV tool for verification.

There is also a large body of research on aspect-oriented modeling. But most of them do not concern about the correctness of the integrated model and provides verification supports. In addition to supporting aspect-oriented modeling and integration, our approach also formally checks whether crosscutting concerns in requirements are correctly designed. Xu *et al.* proposed to model and compose aspects with finite state machines, and then transformed to FSP processes and cheeked by LTSA model checker against all system requirements [17]. Whereas our approach is carried out on activity diagrams and only focuses on checking crosscutting concerns instead of general system requirements. Furthermore, we categorize 4 kinds of crosscutting concerns and generate CTL formulas automatically from crosscutting concern specifications, which bridges the gaps between crosscutting requirements and aspect-oriented design models. We also provide a solution for the conflicts between crosscutting concerns.

Several model checking techniques have been presented for aspect-oriented programs. Denaro *et al.* first reported a preliminary experience on verifying deadlock freedom of a concurrent aspect [18]. They first derived PROMELA process templates from aspect-oriented units, and then analysis the aspect-oriented program with SPIN. Ubayashi and Tamai [19] proposed to apply model checking techniques to verify whether the result of weaving classes and aspects contained unexpected behaviors like deadlocks. These approaches can find realistic defects in the aspect-oriented programs. In contrast, the approach in this paper is carried out at the model level other than the program level. As a result, our approach can identify system faults at an earlier stage, and save costs to revise programs when detecting design faults at implementation or maintenance phase.

6. Conclusions and Future Work

This paper presents a framework to verify aspect-oriented UML activity diagrams by using Petri net based verification techniques. We add lightweight extensions to standard activity diagrams with stereotypes and tagged values to support the modeling of aspects. Then the aspect models are integrated with primary models. For verification purpose, we transform the integrated activity diagrams into Petri nets. Then, crosscutting properties of the system are refined as a set of CTL formulas. Last, the Petri nets are verified against the refined CTL formulas. The verification results report whether the Petri net satisfy the requirements or not. Thus, we can reason whether the integrated activity diagram meets the requirement since they are equivalent. In other words, we can claim that the aspect-oriented modeling is correct with respect to specified crosscutting requirements. Two case studies have been carried out to demonstrate the feasibility and effectiveness of our approach. Concerning the future work, we will focus on testing system implementations against aspect-oriented models have been verified.

Acknowledgments

We would like to thank Professor Karsten Wolf at University Bamberg, who is the author of LoLA, for his help in dealing with problems encountered when integrating LoLA into our tool Jasmine-AOV. This work is supported by the National Natural Science Foundation of China (Nos. 91118007, 61021062, 61170066), and by the National 863 High-Tech Program of China (No. 2012AA011205). The preliminary version of this paper is published in SEKE 2012 [20].

Appendix A. Proof of Theorem 1

Theorem 1. *Given an Activity Diagram* $AD = (N, E, F)$, *and the corresponding* $PN4AD = (P, T, T_A, A, M_0)$, *for any path* ep *of* AD, *there must be a path* pnp *of* $PN4AD$, *and the action trail* $\tau(ep)$ *is equivalent with the action trail* $\sigma(pnp)$, *vice versa.*

To prove this theorem, on one hand, we need to prove for any path in AD, $ep = N_0 \xrightarrow{E_0} N_1 \xrightarrow{E_1} \cdots \xrightarrow{E_{m-1}} N_m$ in AD, there is a legal path $pnp = M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} M_n$ in $PN4AD$ which shares the same action trail with ep.

Proof. (by induction on the number of flow edges in ep)

Basis: $m = 0, ep$ has only one node set N_0, which contains nodes without any incoming edges (initial and receive signal nodes), according to the transformation rules, for each of the node in N_0, we have initial marking $M_0 \subseteq PN4AD.P$, and since the place in M_0 doesn't have incoming arcs, then it holds a token. Therefore, $pnp = M_0$ is a legal path in $PN4AD$. Clearly, their action trails are the same.

Induction Hypothesis: Assume this argument holds for all the prefix paths $ep_k = N_0 \xrightarrow{E_0} N_1 \xrightarrow{E_1} \cdots \xrightarrow{E_{k-1}} N_k$ in ep with number of flow edges smaller than k, where $m > k \geq 0$.

Induction Step: Let $ep_{k+1} = N_0 \xrightarrow{E_0} N_1 \xrightarrow{E_1} \cdots N_k \xrightarrow{E_k} N_{k+1}$ be the prefix of ep with $k+1$ flow edges. As $N_k \xrightarrow{E_k} N_{k+1}$, and $N_{k+1} = \{N_k\{n_k\}\} \cup \{n_{k+1} \mid (n_k, e_k) \in F \wedge (e_k, n_{k+1}) \in F \wedge e_k \in E_k\}$, we will discuss different kinds of n_k accordingly.

- Obviously, if n_k is a final node, then the argument holds.
- If n_k is a decision node or merge node, there is a corresponding place p_j in $PN4AD.P$, as there is a path $pnp_{j+1} = M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{j-1}} M_j \xrightarrow{t_j} M_{j+1}$ in $PN4AD$ equivalent with ep, p_j must holds a token and $p_j \in M_j$.
  - □ If n_{k+1} is a decision node or merge node or final node, there is a corresponding place p_{j+1} in $PN4AD$. According to the transformation rule, one transition t_j

and two arcs will be added into *PN4AD.T* and *PN4AD.A* to connect p_j and p_{j+1}, as p_j holds a token, then t_j is enabled. So we can fire transition t_j in *PN4AD* and get a new marking M_{j+1}, which is $M_j - \{p_j\} + \{p_{j+1}\}$, so we can get a path $pnp_{j+1} = M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{j-1}} M_j \xrightarrow{t_j} M_{j+1}$ in *PN4AD*. As ep_k and pnp_j shares the same action trail, and during the $k+1$th step in *ep* and $j+1$th step in *pnp*, there is no action node related behavior, the corresponding action trails both remain the same. Note, we do not consider the potential impact of the successor of pnp_{j+1} to enabled M_{k+1} here, as it is still unseen so far.

□ If n_{k+1} is an action node, fork node or join node, there is a corresponding transition t_{j+1} in *PN4AD.T*. n_{k+1} is fireable means all predecessor nodes of n_{k+1} are contained in N_k. So for the corresponding transition t_{j+1} in *PN4AD.T*, all predecessor places of t_{j+1} are contained in M_j. As a result, transition t_{j+1} after M_j, which is generated from n_{k+1}, is the next transition to be fired for pnp_j in *PN4AD*.

  * If n_{k+1} is a fork or join node, similar with above proof, nothing is related with action node. Thus, the action trails keep the same and still equivalent with each other.
  * If n_{k+1} is an action node, n_{k+1} will appear in the end of $\tau(ep_{k+1})$. On the other side, as t_{j+1} is added after pnp_j, M_j will be updated with t_{j+1}. Therefore, both the action trail of ep_k and pnp_j will be updated with n_{k+1}, and thus still keeps the same.

- If n_k is an action node or join node, there is a corresponding transition t_j in *PN4AD.T*. As pnp_j is a legal path in *PN4AD*, t_j is enabled.

□ If n_{k+1} is a decision node or merge node or final node, there is a corresponding place p_{j+1} in *PN4AD*. According to transformation rule, an arc will be add from t_j to p_{j+1}, so $pnp_{j+1} = M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{j-1}} M_j \xrightarrow{t_j} M_{j+1}$, $(M_{j+1} = M_j - {}^{\bullet}t_j + \{p_{j+1}\})$, is a legal path in *PN4AD*. As both n_k and t_j are dismissed from the current action trail, the action trail of ep_{k+1} and pnp_{j+1} keeps the same. Similarly, the potential impact of the transitions enabled by M_{j+1} is not considered here.

□ If n_{k+1} is an action node, fork node, or join node, there is a corresponding transition t_{j+1} in *PN4AD.T*. According to the rule, a place p_{j+1} and two arcs will be added to connect t_j and t_{j+1}. The fire of t_j makes p_{j+1} contains a token. As a result, t_{j+1} is enabled in *PN4AD*, and the path $pnp_{j+1} = M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{j-1}} M_j \xrightarrow{t_j} M_{j+1}$, $(M_{j+1} = M_j - {}^{\bullet}t_j + p_{j+1})$, is a legal path in *PN4AD*.

  * If n_{k+1} is a fork or join node, similar with above proof, only n_k is dismissed from both action trails. Thus, the action trails keep the same and still equivalent with each other.
  * If n_{k+1} is an action node, n_k will be dismissed while n_{k+1} will appear in the end of $\tau(ep_{k+1})$. On the other side, as t_{j+1} is added after pnp_j, M_j will be updated with t_{j+1}. Therefore, both the action trail of ep_k and pnp_j will be updated with n_k and n_{k+1}, and thus still keeps the same.

- If n_k is a fork node, there is a corresponding transition t_j in *PN4AD.T*. As pnp_j is a legal path in *PN4AD*, t_j is enabled as well. For any $\{n_{k+1,i} \mid (n_k, e_k) \in F \wedge (e_k, n_{k+1,i}) \in F$, where $e_k \in E_k\}$
  - □ If $n_{k+1,i}$ is a decision node or merge node or final node, there is a corresponding place $p_{j+1,i}$ in *PN4AD*. According to transformation rule, an arc will be added from t_j to $p_{j+1,i}$. After E_k is fired, t_j will be fired in *PN4AD* and $p_{j+1,i}$ will holds a token. So the path $pnp_{j+1} = M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{j-1}} M_j \xrightarrow{t_j} M_{j+1}$, $(p_{j+1,i} \in M_{j+1})$, is a legal path in *PN4AD*. Similar with above, as nothing related to action nodes is performed in the above step, and the potential impact of the so far invisible transitions after pnp_{j+1} is not considered, the action trail keeps the same with each other.
  - □ If $n_{k+1,i}$ is an action node, fork node, or join node, there is a corresponding transition $t_{j+1,i}$ in *PN4AD.T*. According to the rule, an place $p_{j+1,i}$ and two arcs will be added to connect t_j and $t_{j+1,i}$. After E_k is fired, t_j will be fired in *PN4AD*, $p_{j+1,i}$ will hold a token, and $t_{j+1,i}$ is enabled. So the path $pnp_{j+1} = M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{j-1}} M_j \xrightarrow{t_j} M_{j+1}$, $(p_{j+1,i} \in M_{j+1})$, is a legal path in *PN4AD*.
    - ∗ If $n_{k+1,i}$ is a fork or join node, the above step has nothing to do with action nodes. Thus, the action trails keep the same and still equivalent with each other.
    - ∗ If $n_{k+1,i}$ is an action node, $n_{k+1,i}$ will appear in the end of $\tau(ep_{k+1})$. On the other side, as $t_{j+1,i}$ is added after pnp_j, M_j will be updated with $t_{j+1,i}$. Therefore, both the action trail of ep_k and pnp_j will be updated with $n_{k+1,i}$, and thus still keeps the same.

Above all, given any path *ep* in *AD*, there is a corresponding path *pnp* in *PN4AD* shares the same action trail with *ep*.

The other direction of the theorem is that given any path in the generated *PN4AD*, $pnp = M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} M_n$ in *PN4AD*, there is a legal path $ep = N_0 \xrightarrow{E_0} N_1 \xrightarrow{E_1} \cdots \xrightarrow{E_{m-1}} N_m$ in *AD* which shares the same action trail with *pnp*. □

Proof. (by induction on the number of transitions in *pnp*)

Basis: $n = 0$, $pnp = M_0$, $M_0 \subseteq PN4AD.P$ is the initial marking in *PN4AD*. M_0 is the place set transformed from a node set N_0 of *AD*. According to the transformation rules, for each place in M_0 is generated form a node without any incoming edges in N_0 (initial and receive signal nodes). Clearly, the action trails are the same.

Induction Hypothesis: Assume this argument holds for all the prefix paths $pnp = M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{j-1}} M_j$ in *pnp* with number of transitions smaller than j, where $m > j \geq 0$.

Induction Step: Let $pnp_{j+1} = M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{j-1}} M_j \xrightarrow{t_j} M_{j+1}$ be the prefix of *pnp* with $j+1$ transitions. As $M_j \xrightarrow{t_j} M_{j+1}$, and $M_{j+1} = M_j - {}^\bullet t_j + t_j{}^\bullet$. We will discuss different kinds of t_j accordingly.

- If t_j is an auxiliary transition. According to the transformation rules, ${}^{\bullet}t_j = \{p_j\}$, $t_j{}^{\bullet} = \{p_{j+1}\}$, p_j, p_{j+1} are two places transformed from n_k and n_{k+1}, respectively. And in AD, n_k and n_{k+1} are two initial, final, decision, or merge nodes, the auxiliary transition t_j and two arcs are transformed form e_k, which start from n_k to n_{k+1}, to connect p_j and p_{j+1}. Therefore, $ep_{k+1} = N_0 \xrightarrow{E_0} N_1 \xrightarrow{E_1} \cdots N_k \xrightarrow{E_k} N_{k+1}$, $(N_{k+1} = \{N_k - \{n_k\}\} \cup \{n_{k+1} \,|\, (n_k, e_k) \in F \wedge (e_k, n_{k+1}) \in F \wedge e_k \in E_k\})$, is a legal path in AD. As there is no action node related behavior, the action trails of pnp_{j+1} and ep_{k+1} keeps the same. The potential impact of the successor n_{k+1} is not considered here, as it is still unseen so far.
- If t_j is not an auxiliary transition, then t_j is a transition transformed from n_{k+1} in AD, which is an action, fork or join node. For any $p_{j,i} \in {}^{\bullet}t_j$
  - □ If $p_{j,i}$ is an auxiliary place. Assume $\{p_{j,i}\} \subseteq t_{j-1}{}^{\bullet}$, according to the transformation rule, t_{j-1}, t_j are two transitions transformed from n_k and n_{k+1}, respectively. And in AD, n_k is also an action, fork, or join node, the auxiliary place and two arcs are transformed form e_k, which start from n_k to n_{k+1}, to connect t_{j-1} and t_j. Therefore, $ep_{k+1} = N_0 \xrightarrow{E_0} N_1 \xrightarrow{E_1} \cdots N_k \xrightarrow{E_k} N_{k+1}$, $(N_{k+1} = \{N_k - \{n_k\}\} \cup \{n_{k+1} \,|\, (n_k, e_k) \in F \wedge (e_k, n_{k+1}) \in F \wedge e_k \in E_k\})$, is a legal path in AD.
    - ∗ If n_{k+1} is a fork or join node, similar with above proof, nothing is related with action node. Thus, the action trails keep the same and still equivalent with each other.
    - ∗ If n_{k+1} is an action node, n_{k+1} will appear in the end of $\tau(ep_{k+1})$. On the other side, as t_j is added after pnp_{j+1}, M_j will be updated with t_j. Therefore, both the action trail of ep_k and pnp_j will be updated with t_j, which is transformed form n_{k+1}, and thus still keeps the same.
  - □ If $p_{j,i}$ is not an auxiliary place, then $p_{j,i}$ is a place transformed from n_k. In AD, n_k is an initial, final, decision, or merge node, an arc is transformed form e_k, which start from n_k to n_{k+1}, to connect $p_{j,i}$ and t_j. Therefore, $ep_{k+1} = N_0 \xrightarrow{E_0} N_1 \xrightarrow{E_1} \cdots N_k \xrightarrow{E_k} N_{k+1}(N_{k+1} = \{N_k - \{n_k\}\} \cup \{n_{k+1} \,|\, (n_k, e_k) \in F \wedge (e_k, n_{k+1}) \in F \wedge e_k \in E_k\})$, is a legal path in AD.
    - ∗ If n_{k+1} is a fork or join node, similar with above proof, nothing is related with action node. Thus, the action trails keep the same and still equivalent with each other.
    - ∗ If n_{k+1} is an action node, n_{k+1} will appear in the end of $\tau(ep_{k+1})$. On the other side, as t_j is added after pnp_{j+1}, M_j will be updated with t_j. Therefore, both the action trail of ep_k and pnp_j will be updated with t_j, which is transformed form n_{k+1}, and thus still keeps the same.

Above all, given any path pnp in $PN4AD$, there is a corresponding path ep in AD shares the same action trail with pnp.

Therefore, for any path *ep* of *AD*, there must be a path *pnp* of *PN4AD*, and the action trail $\tau(ep)$ is equivalent with the action trail $\sigma(pnp)$, vice versa. □

References

1. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier and J. Irwin, Aspect-oriented programming, in *Proceedings of the Annual European Conference on Object-Oriented Programming*, 1997, pp. 220–242.
2. Z. Cui, L. Wang, X. Li and D. Xu. Modeling and integrating aspects with UML activity diagrams, in *Proceedings of the ACM Symposium on Applied Computing*, 2009, pp. 430–437.
3. X. Li, Z. Liu, J. He and Q. Long, Generating a prototype from a UML models of system requirements, in *Distributed Computing and Internet Technology*, LNCS, Vol. 3347, 2005, pp. 135–154.
4. A. Fischer, Mapping UML designs to java, in *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2000, pp. 178–187.
5. L. Wang, J. Yuan, X. Yu, J. Hu, X. Li and G. Zheng, Generating test cases from UML activity diagram based on gray-box method, in *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, 2004, pp. 284–291.
6. C. Nebut, F. Fleurey, Y. L. Traon and J. Jezequel, Automatic test generation: A use case driven approach, *IEEE Transactions on Software Engineering* **32**(3) (2006) 140–155.
7. M. Chen, X. Qiu, W. Xu, L. Wang, J. Zhao and X. Li, UML activity diagram based automatic test case generation for java programs, *The Computer Journal* **52**(5) (2009) 545–556.
8. OMG, UML Superstructure v2.1, http://www.omg.org/technology/documents/formal/uml.htm.
9. T. Murata, Petri nets: Properties, analysis and applications, in *Proceedings of the IEEE*, Vol. 77, No. 4 Apr 1989, pp. 541–580.
10. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, An overview of AspectJ, in *Proceedings of the Annual European Conference on Object-Oriented Programming*, 2001, pp. 327–353.
11. E. M. Clarke and E. A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporallogic, in *Proceedings of Logic of Programs*, 1981, pp. 52–71.
12. K. Schmidt, LoLA: A low level analyser, in *Proceedings of the Application and Theory of Petri Nets*, 2000, pp. 465–474.
13. H. Störrle, Semantics of control-flow in UML 2.0 activities, in *Proceedings of the IEEE Symposium on Visual Languages — Human Centric Computing*, 2004, pp. 235–242.
14. H. Störrle, Structured nodes in UML 2.0 activities, *Nordic Journal of Computing* **11**(3) (2004) 279–302.
15. J. Whittle and J. Araujo, Scenario modelling with aspects, in *IEEE Software* **151**(4) (2004) 157–172.
16. A. Sampaio, A. Rashid, R. Chitchyan and P. Rayson, EA-Miner: Towards automation in aspect-oriented requirements engineering, *Transactions on Aspect-Oriented Software Development III*, LNCS, Vol. 4620, pp. 4–39.
17. D. Xu, O. E. Ariss, W. Xu and L. Wang, Aspect-oriented modeling and verification with finite state machines, *Journal of Computer Science and Technology* **24**(5) (2009) 949–961.
18. G. Denaro and M. Monga, An experience on verification of aspect properties, in *Proceedings of the 4th International Workshop on Principles of Software Evolution*, 2001, pp. 186–189.

19. N. Ubayashi and T. Tamai, Aspect-oriented programming with model checking, in *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, 2002, pp. 148–154.
20. Z. Cui, L. Wang, X. Liu, L. Bu, J. Zhao and X. Li, Verifying aspect-oriented activity diagrams against crosscutting properties with Petri net analyzer, in *Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering*, 2012, pp. 369–374.