

Secure Software Architectures Design by Aspect Orientation *

Huiqun Yu, Dongmei Liu

Department of Computer Science and Engineering
East China University of Science and Technology
Shanghai 200237, China
{yhq|dmliu}@ecust.edu.cn

Xudong He, Li Yang, Shu Gao

School of Computer Science
Florida International University
Miami, FL 33199, USA
{hex|lyang03|sgao01}@cs.fiu.edu

Abstract

Security design at architecture level is critical to achieve high assurance software systems. However, most security design techniques for software architectures were in ad hoc fashion and fell short in precise notations. This paper proposes a formal aspect-oriented approach to designing secure software architectures. The underlying formalism is the Software Architecture Model (SAM) that combines Petri nets and temporal logic. SAM supports a precise way to model the problem domain, its software architecture, and security aspects of the software architecture. An integrated architecture is obtained by weaving aspect models with the base architecture model. Mechanisms in SAM are amenable to analyzing correctness of the architecture design.

Keywords: *Software architecture, security, aspect orientation, SAM, formal method*

1 Introduction

Complex software systems are increasingly based on reused software modules, standard software components, and the assembly of existing programs into integrated program suites. This allows software manufacturers to market their products at a competitive price. This effort is facilitated by the advances of software architecture [25]: that is, its gross organization as a collection of interacting components. Software architecture plays a central role in managing the complexity of software design by better system structuring, multiple levels of reuse and clear dimensions for system evolution. One of the most important task is to design highly dependable security systems that protect the

softwares and resources. The task is increasingly complex and challenging with the rapid proliferation of the Internet. Several well-known security system architectures and models, including those in CORBA [3, 20], EJB [9] are cornerstones for designing scalable and flexible security systems. However, most security design techniques for software architectures were in ad hoc fashion and fell short in precise notations.

Separation of concerns is a general principle in software engineering introduced by Dijkstra [7] and Parnas [21] to control the complexity of evergrowing programs. Aspect-oriented programming (AOP) [16, 8] has been proposed to improve separation of concerns in software. AOP is based on the idea that computer systems are better programmed by separately specifying the various *concerns* of a system and their relationships, and then relying on the mechanisms in the underlying AOP environment to weave or compose them together into a coherent program. Concerns can range from high-level notions like security and quality of service to low-level notions like caching and buffering. Several successful AOP techniques have been proposed in literature, such as adaptive programming [17], AspectJ [15], Composition Filters [2], and multi-dimensional separation of concerns [18]. Most investigations so far have been focused on language constructs for aspect description and aspect weaving at code level. However, the significance of aspect-oriented design (AOD) at more abstract level has come to front recently [14, 24], because quality assurance in the early stage of software life cycle can result in better design, and shorter time to market.

The confluence of AOP and software architecture research was pointed out in [6], and security concerns such as authentication and access control, are suggested to be placed in architectural connectors. Role Models were used to define high level aspects in [10]. A role model is actually characteristics of certain UML models. Two kinds of language independent aspects, i.e. perspective aspects and aspect checks, were identified in [26], which help developers handle crosscutting dependencies among components at

*This work was partially supported by the NSF of China under grants No. 60473055 and No. 60373075, by the NSF of the USA under grants CCR-0098120 and HRD-0317692, and by the NASA of the USA under grant NAG 2-1440.

the design stage. But the above works fell short in precise notations for expressing different concerns and for manipulating these concerns in a systematic fashion, the latter being recognized in [23] as the essential ‘novelty’ of aspect-orientation. Without formal definitions, it is impossible to prove the correctness of the design.

This paper proposes a formal aspect-oriented approach to designing secure software architectures. A secure software architecture defines structure of the software system, the interaction and coordination among its components, which correctly enforces the security requirement. An aspect-oriented software architecture design consists of a model of essential functionality and a number of aspect models each modularized around a specific concern. We focus on aspects that reflect security concerns [22]. The integrated model of the system is obtained by “weaving” (composing) the model of essential functionality with the aspect models. To base AOD method on a solid foundation, we use SAM as the underlying formalism [27]. SAM is a dual formalism combining Petri nets and temporal logic, which has been successfully applied in high assurance software architecture design [12, 28, 5, 13].

The contribution of this paper includes:

1. proposing formal notations for aspect-oriented modeling at architectural level, and
2. establishing an aspect-oriented approach to designing secure software architectures.

The rest of the paper is organized as follows. Section 2 gives a brief introduction to SAM and its theoretical basis. Section 3 presents an AOD framework for designing secure software architectures. Section 4 is the conclusion.

2 An Introduction to SAM

2.1 SAM Models

In SAM, a software architecture is defined by a hierarchical set of compositions in which each composition consists of a set of components, a set of connectors and a set of constraints to be satisfied by the interacting components. Basically, behaviors of components and connectors are modeled by predicate transition nets (PrT nets) [11], while their properties (or constraints) are specified by temporal logic formulas [19]. The interfaces of components and connectors are *ports*. Figure 1 shows a graphical view of a SAM architecture model, where the higher level component A3 is decomposed into a lower level sub-architecture.

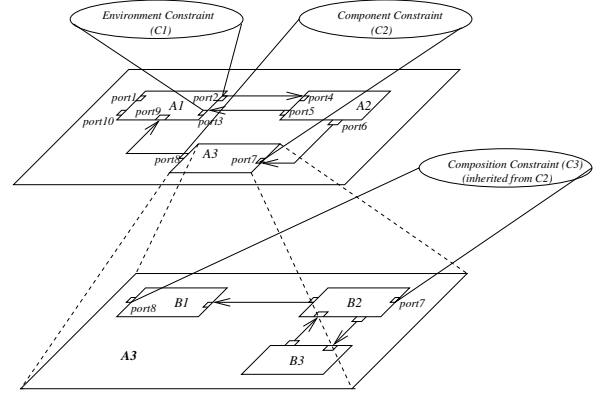


Figure 1. A SAM architecture model

Formally, a software architecture model in SAM consists of a set of compositions $C = \{C_1, C_2, \dots, C_k\}$ and a hierarchical mapping h relating compositions. Each composition $C_i = \{C_{m_i}, C_{n_i}, C_{S_i}\}$ consists of a set C_{m_i} of components, a set C_{n_i} of connectors, and a set C_{S_i} of composition constraints. An element $C_{i_j} = (S_{i_j}, B_{i_j})$, either a component or a connector, in a composition C_i has a property specification S_{i_j} (an LTL formula) and a behavior model B_{i_j} (a PrT net). Each composition constraint in C_{S_i} is also defined by an LTL formula. The interface of a behavior model B_{i_j} consists of a set of places (called *ports*) that is the interaction among relevant components and connectors. A component C_{i_j} can be refined into a lower-level composition C_ℓ , which is defined by $h(C_{i_j}) = C_\ell$.

2.2 Predicate Transition Nets

A predicate transition net (PrT net) [11] is a tuple $\Pi = (P, T, F, \varphi, R, L, M_0)$ where:

- P is a finite set of places.
- T is a finite set of transitions. We require that $P \cap T = \emptyset$.
- F is the set of arcs and is called the *flow relation*, i.e. $F \subseteq P \times T \cup T \times P$. We use \vec{F} and \overleftarrow{F} to denote $F \cap (P \times T)$ and $F \cap (T \times P)$, respectively. By convention, the preset and postset of a place p (or a transition τ) are denoted by $\bullet p$ and p^\bullet (or $\bullet \tau$ and τ^\bullet), respectively.
- φ is a mapping, which assigns each place in P to a sort.
- R is a mapping, which associates each transition τ in T with a first-order logic formula. The general form of $R(\tau)$ is $Pre(\tau) \wedge Post(\tau)$, where $Pre(\tau)$ specifies the precondition of τ , and $Post(\tau)$ describes the functionality of τ . We use $\psi(\tau)$ to denote the set of variables that appear in $Pre(\tau)$.

- L is a mapping, which labels each arc with a corresponding multi-set.
- M_0 is the initial marking.

There is an underlying algebraic specification for any PrT net, which defines data, operators and their properties. Formally, the underlying specification $SPEC = (S, OP, Eq)$ consists of a signature $\mathbf{S} = (S, OP)$ and a set Eq of \mathbf{S} -equations. Signature $\mathbf{S} = (S, OP)$ includes a set of sorts S and a family $OP = (OP_{s_1, \dots, s_n, s})$ of sorted operations for $s_1, \dots, s_n, s \in S$. For each $s \in S$, we use CON_s to denote $OP_{\cdot, s}$ (the 0-ary operation of sort s), i.e. the set of constant symbols of sort s . The \mathbf{S} -equations in Eq define the meanings and properties of operations in OP . We often simply use familiar operations and their properties without explicitly listing the relevant equations.

Tokens of a PrT net are ground terms of the signature \mathbf{S} , written $MCON_S$. The set of labels is denoted by $Label_S(X)$ (X is the set of sorted variables disjoint with OP). Each label can be a multiple set expression of the form $\{k_1 x_1, \dots, k_n x_n\}$.

A *marking* M of a PrT net is a mapping, $P \rightarrow MCON_S$, from the set of places to multi-sets of tokens. An *occurrence mode* of a PrT net is a substitution $\alpha = \{x_1 \leftarrow c_1, \dots, x_n \leftarrow c_n\}$, which instantiates typed label variables. We use $e : \alpha$ to denote the result of instantiating an expression e with α , in which e can be either a label expression or a constraint.

Given a marking M , a transition $\tau \in T$, and an occurrence mode α , τ is *enabled* if the following condition is satisfied: $\forall p : p \in P. (L(p, \tau) : \alpha \subseteq M(p)) \wedge Pre(\tau) : \alpha$

The *firing* of an enabled transition τ at the marking M and occurrence mode α returns the marking M' defined by $M'(p) = M(p) - L(p, \tau) + L(\tau, p)$, for all $p \in P$. We call M' the τ -successor of M , denoted by $M\tau M'$. A *run* σ of the PrT net Π at the initial mark M_0 is a sequence of markings: $\sigma = M_0, M_1, M_2, \dots$, for each pair of markings (M_i, M_{i+1}) , there is an enabled transition τ_i such that $M_i\tau_i M_{i+1}$.

Given a PrT net Π with an initial marking M_0 , we define the *computation* of Π (denoted by $Comp(\Pi)$) to be the set of all possible runs at the initial marking M_0 .

2.3 A Linear-Time Temporal Logic

The requirement specification language is a linear-time temporal logic (LTL) [19]. A *temporal formula* in LTL is constructed from state formulas to which we apply the boolean connectives and temporal operators. Common temporal operators include \Box (always) and \Diamond (eventually),

A *model* for a temporal formula p is an infinite sequence of states $\sigma : s_0, s_1, \dots$, where each state s_j provides an interpretation for the variables mentioned in p . The semantics

of a temporal formula p in a given model σ and position j is denoted by $(\sigma, j) \models p$. We define:

- For a state formula p , $(\sigma, j) \models p \Leftrightarrow s_j \models p$
- $(\sigma, j) \models \neg p \Leftrightarrow (\sigma, j) \not\models p$
- $(\sigma, j) \models p \vee q \Leftrightarrow (\sigma, j) \models p$ or $(\sigma, j) \models q$
- $(\sigma, j) \models \Box p \Leftrightarrow (\sigma, i) \models p$ for all $i \geq j$
- $(\sigma, j) \models \Diamond p \Leftrightarrow (\sigma, i) \models p$ for some $i \geq j$

A detailed definition of LTL can be found in [19].

A model $\sigma : s_0, s_1, \dots$ satisfies a temporal formula if $(\sigma, 0) \models p$. A temporal formula p that is *valid* over a PrT net Π specifies a property of Π , i.e., states a condition that is satisfied by all computations of Π .

2.4 An Example: The Travel Planner

Consider a travel agency which processes reservations for flight seats, hotel rooms and rental cars [1]. The tasks involved in the Travel Planner are :

- W_1 : Input travel information,
- W_2 : Reserve a ticket with Continental Airlines,
- W_3 : If W_2 fails or if the ticket costs more than \$400, reserve a ticket with Delta Airlines,
- W_4 : If the ticket at Continental costs no more than \$400, or if the reservation at Delta fails, purchase the ticket at Continental,
- W_5 : If Delta has a ticket, then purchase it at Delta,
- W_6 : Reserve a room at Sheraton, if there is flight reservation, and
- W_7 : Rent a car at Hertz.

Figure 2 gives a PrT net model of the Travel Planner. Tasks are modeled as places in the PrT net. In the underlying specification $SPEC = (S, OP, Eq)$, where S includes elementary sorts such as Integer and Boolean, and also sorts $Usr, Time, Route, Discount, Price, Flight, Hotel,$ and Car derived from Integer. Ok is an alias of Boolean. S also includes structured sorts such as set and tuple obtained from the Cartesian product of the elementary sorts; OP includes standard arithmetic and relational operations on Integer, logical connectives on Boolean, set operations, and selection operation on tuples; and Eq includes known properties of the above operators.

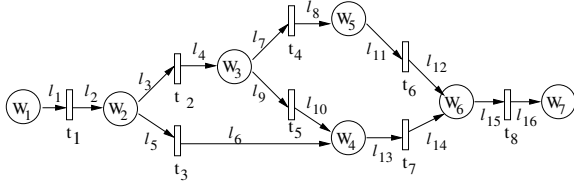


Figure 2. A PrT net model of the Travel Planner

Formally, the PrT net model is $\Pi = (P, T, F, \varphi, R, L, M_0)$, where:

- $P = \{W_1, W_2, W_3, W_4, W_5, W_6, W_7\}$.
- $T = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8\}$.
- $F = \{(w_1, t_1), (t_1, w_2), (w_2, t_2), (t_2, w_3), \dots, (w_6, t_8), (t_8, w_7)\}$.
- $\varphi(W_1) = \varphi((Usr \times Time \times Route \times Discount) \times (Usr \times Time \times Route \times Discount))$, where $\varphi(X)$ is the power set of X ,
 $\varphi(W_2) = \varphi((Usr \times Time \times Route \times Discount) \times (Usr \times Time \times Route \times Price \times Ok))$,
 $\varphi(W_3) = \varphi((Usr \times Time \times Route) \times (Usr \times Time \times Route \times Price \times Ok))$,
 $\varphi(W_4) = \varphi(W_5) = \varphi((Usr \times Time \times Route) \times (Usr \times Time \times Flight))$,
 $\varphi(W_6) = \varphi((Usr \times Time \times Route) \times (Usr \times Time \times Hotel \times Price))$,
 $\varphi(W_7) = \varphi((Usr \times Time \times Route) \times (Usr \times Time \times Car \times Price))$.

- $Pre(t_1) = true, Post(t_1) = (M'(W_1) = \emptyset \wedge M'(W_2) = l_2)$,
 $Pre(t_2) = (price > 400 \vee ok = false)^1, Post(t_2) = (M'(W_2) = \emptyset \wedge M'(W_3) = l_4)$,
 $Pre(t_3) = (price' \leq 400 \wedge ok' = true), Post(t_3) = (M'(W_2) = \emptyset \wedge M'(W_4) = l_6)$,
 $Pre(t_4) = (ok'' = true), Post(t_4) = (M'(W_3) = \emptyset \wedge M'(W_5) = l_8)$,
 $Pre(t_5) = (ok''' = false), Post(t_5) = (M'(W_3) = \emptyset \wedge M'(W_4) = l_{10})$,
 $Pre(t_6) = true, Post(t_6) = (M'(W_5) = \emptyset \wedge M'(W_6) = l_{12})$,
 $Pre(t_7) = true, Post(t_7) = (M'(W_4) = \emptyset \wedge M'(W_6) = l_{14})$,
 $Pre(t_8) = true, Post(t_8) = (M'(W_6) = \emptyset \wedge M'(W_7) = l_{16})$.
 Note that for each transition τ , $R(\tau) = Pre(\tau) \wedge Post(\tau)$.

¹The variables *price* and *ok* are actually projections of the corresponding labels, e.g. *price* is the 4th element in the 2nd element of l_3 . Similarly are *price'*, *ok'*, *ok''*, and *ok'''*.

- $L = \{(W_1, t_1) \mapsto l_1, (t_1, W_2) \mapsto l_2, \dots, (W_6, t_8) \mapsto l_{15}, (t_8, W_7) \mapsto l_{16}\}$, where each label is a type-respecting singleton set. For example, $l_3 = \{(-, (usr, time, route, price, ok))\}$, $l_4 = \{((usr, time, route), -)\}$, where $-$ is a wildcard.
- $M_0(W_1) = \{((usr_0, time_0, route_0, discount_0), -)\}$,
 $M_0(W_2) = M_0(W_3) = M_0(W_4) = M_0(W_5) = M_0(W_6) = M_0(W_7) = \emptyset$.

Note that places in Figure 2 are super-places that may be further refined to lower level PrT nets. For example, a possible refinement of the place W_2 is illustrated in Figure 3. In SAM model, hierarchical relationship among places (or transitions) are defined by a hierarchical mapping h . We require that $\varphi(W_2) = \varphi(Input) \times \varphi(Output)$.

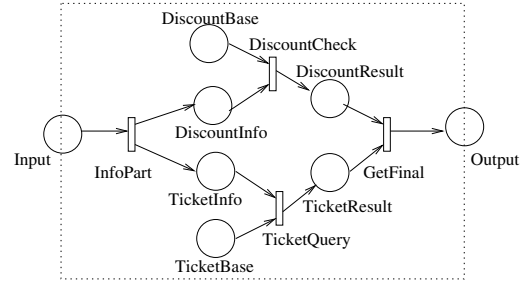


Figure 3. A refinement of task W_2

3 Secure Software Architectures Design

3.1 An AOD Framework

An AOD framework for design secure software architectures is illustrated in Figure 4. The framework addresses design issues in order to establish sound software architecture models for secure software architectures based on “separation of concerns” principle. It consists of four kinds of models:

- A *problem domain model* gives precise description of basic functionality and their relationships of the targeted software system.
- A *base architecture model* defines software architecture that provides basic functional modules and their connections. A functional module reflects an autonomous software entity, and usually includes a set of components and/or connectors.
- A *security aspect model* describes security requirements, identifies vulnerabilities and threats, and provides mechanisms that enforce security policies into the architecture.

- A *secure architecture model* is the software architecture model that the security policies have been correctly enforced.

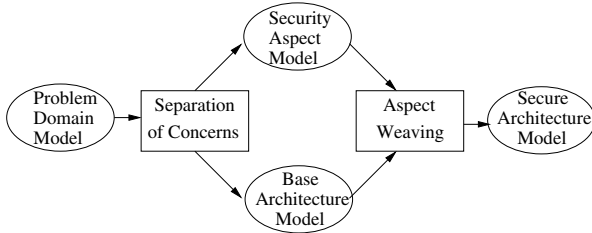


Figure 4. An AOD framework for secure software architectures

In the AOD framework, SAM is a unified language for component modeling and aspect modeling. There are two distinct levels of software architecture specification development in SAM, i.e. element level and composition level [12]. The element level specification deals with the specification of a single component or connector and the composition level specification concerns how to combine (horizontal) specifications at the same abstract level together and how to relate (vertical) specifications at different abstract levels.

Once the problem domain model is established, the remaining major issues involved in the framework are how to separate various concerns and how to weave aspect models into an integrated software architecture. *Separation of concerns* characterizes functionality and security aspects of the system, which consists of the following steps.

1. Identifying autonomous modules in the domain model and possibly adding some auxiliary modules to construct a base software architecture model².
2. Specifying security requirements on the base software architecture.
3. Identifying the vulnerabilities and threats, and constructing mechanisms that can be used to protect the system against such threats.

Aspect weaving generates a software architecture by weaving aspect models with the base architecture model. The steps include:

1. Pinpointing the location where the base architecture model and aspect models interact.
2. Defining the behavior of the system in order to enforce security policies on the base software architecture.

²Intuitively, a component can be cleanly encapsulated in a generalized procedure, which tends to be units of the system's functional decomposition, while an aspect cannot be cleanly encapsulated in a generalized procedure, but tends to be properties that affect the performance or semantics of the components in a systemic way [16].

3. Integrating aspect models with the base software architecture.

In the following, we will precisely define the aspect-oriented modeling elements in SAM, and apply them to weaving different concerns into the base architecture model. The Travel Planner is used as an illustrating example throughout the rest of this paper.

3.2 The Base Architecture Model

We introduce the notion of *block* to denote autonomous software entities (or modules). In form, a block is a part or a whole of a PrT net that models certain software module. We are interested in those contents and profiles of such kind module that have impact on the AOD method. As we know, SAM supports hierarchical software architecture modeling, which allows blocks in different hierarchies. In addition, the granularity and level of block grouping is flexible, which depend on the properties to be observed.

To characterize a block, two element categories of block are identified, i.e. *internal elements* and *external elements*.

- The *internal elements* are attributes that characterize of a block, such as its name, interesting values (constants or variables), type etc.
- The *external elements* present the interface of a block with its environment.

A *base architecture model* is a SAM model with block grouping, and each block represents an autonomous software entity.

The Example (Continued)

A distributed architecture for the Travel Planner is desirable because the system is inherently distributed. Moreover, a distributed architecture is more flexible and more reliable. To achieve maximal distributed implementation, a base architecture model of the Travel Planner can be derived from the PrT net of the problem domain model by

- taking each task and its postset as a distributed block, and
- adding a *central control* block, which initiates some block(s) in the domain model and collects the result.

Note that in addition to seven blocks (A_1 to A_7) that correspond to different tasks, there is a central control block in the base architecture model. As for the block A_2 , the internal elements include its name (A_2), its interesting variables ($price, ok, price', ok'$), its type information (representing Continental Airlines). The external elements include its communicating environment, i.e. the incoming

$$\begin{aligned}
sensitive(W_1) &= sensitive(W_4) = sensitive(W_5) \\
&= sensitive(W_6) = sensitive(W_7) = \emptyset, \\
sensitive(W_2) &= \{price, price'\}, sensitive(W_3) = \\
&\{price\}.
\end{aligned}$$

In the base architecture model, each block contains only one task, which is the corresponding main task by default. Therefore, the sensitive datasets of blocks are:

$$\begin{aligned}
sensitive(A_1) &= sensitive(A_4) = sensitive(A_5) \\
&= sensitive(A_6) = sensitive(A_7) = \emptyset, \\
sensitive(A_2) &= \{price, price'\}, sensitive(A_3) = \\
&\{price\}.
\end{aligned}$$

Dependence Relation

Given a PrT net $\Pi = (P, T, F, \varphi, R, L, M_0)$. A *dependence relation* (denoted by $DR(\Pi)$) for a workflow is a set of tuples:

$$DR(\Pi) = \{(w_i, w_j, V) \mid (w_i, w_j) \in \overrightarrow{F} \times \overleftarrow{F} \wedge V = \psi(w_i^\bullet \cap^\bullet w_j)\}$$

Intuitively, $DR(\Pi)$ gives source and sink of every workflow in the model and the dominating elements for flowing. For example, let Π represent the PrT model in Figure 5. We have:

$$\begin{aligned}
DR(\Pi) &= \{(W_1, W_2, \emptyset), (W_2, W_3, \{price, ok\}), \\
&(W_2, W_4, \{price', ok'\}), (W_3, W_5, \{ok''\}), \\
&(W_3, W_4, \{ok'''\}), (W_5, W_6, \emptyset), \\
&(W_4, W_6, \emptyset), (W_3, W_5, \emptyset)\}
\end{aligned}$$

A Security Policy

A distributed software architecture $\Pi = (P, T, F, \varphi, R, L, M_0)$ is *secure*, if for every block A in Π , and every task w in A , the following formula holds.

$$\Box(\forall t \in T. (t \in w^\bullet \rightarrow (\psi(t) \cap sensitive(A) = \emptyset)))$$

Intuitively, the security policy requires that for every block and its tasks, neither task's decisive data value should be sensible to its own block, nor block's information should be leaked to another block in the same *COI*. For example, according to the above security criterion, the initial distributed control architecture in Figure 5 is not secure, because W_2 is a task in the block A_2 and $t_2 \in W_2^\bullet$, but $\psi(t_2) \cap sensitive(A_2)$ is $\{price\}$, instead of \emptyset .

Join Points and Pointcuts

Given a base architecture model Π . Suppose $(w_i, w_j, V) \in DR(\Pi)$, and $\tau \in w_i^\bullet \cap^\bullet w_j$. The transition τ is a *join point*, if either $V \cap sensitive(w_i) \neq \emptyset$, or $V \cap sensitive(w_j) \neq \emptyset$, or both. Consequently, we divide the above join points in an architecture model into three categories (or types), each of which is a pointcut.

- *Pointcut of Type 1* contains all the join points τ , such that $V \cap sensitive(w_i) \neq \emptyset$, and $V \cap sensitive(w_j) = \emptyset$.
- *Pointcut of Type 2* contains all the join points τ , such

that $V \cap sensitive(w_i) = \emptyset$, and $V \cap sensitive(w_j) \neq \emptyset$.

- *Pointcut of Type 3* contains all the join points τ , such that $V \cap sensitive(w_i) \neq \emptyset$, and $V \cap sensitive(w_j) \neq \emptyset$.

Advices

Advices associate fragments of PrT nets with pointcuts, which specify the system behaviors at every join points in particular pointcuts. Table 1 illustrates the advices for different types of pointcuts.

Table 1. Pointcut structures and their advices

	Pointcuts and Types	Advices
Type 1 Pointcut		
Type 2 Pointcut		
Type 3 Pointcut		

- *Advice for pointcuts of Type 1:* Add a new task w_k and two transitions t_k, t'_k , which are appended to the A_j . Set the main task of A_j to be w_j . Let $\varphi(w_k) = \varphi(w_i)$, $M_0(w_k) = \emptyset$, $Pre(t_k) = Pre(t_i)$, $Post(t_k) = Post(t_i)[w_i/w_k, w_k/w_j]$, $Pre(t'_k) = \neg Pre(t_i)$, $Post(t'_k) = Post(t_i)[w_i/w_k, w_j/w_i]$, $Pre(t'_i) = \emptyset$, $Post(t'_i) = Post(t_i)[w_j/w_k]$.
- *Advice for pointcuts of Type 2:* Add a new task w_k and two transitions t_k, t'_k , which are appended to the block A_i . Set the main task of A_i to be w_i . Let $\varphi(w_k) = \varphi(w_i)$, $M_0(w_k) = \emptyset$, $Pre(t_k) = Pre(t_i)$, $Post(t_k) = Post(t_i)[w_i/w_k, w_k/w_j]$, $Pre(t'_k) = \neg Pre(t_i)$, $Post(t'_k) = Post(t_i)[w_i/w_k, w_j/w_i]$, $Pre(t'_i) = \emptyset$, $Post(t'_i) = Post(t_i)[w_j/w_k]$.
- *Advice for pointcuts of Type 3:* Add a new task w_k and two transitions t_k, t'_k , which constitute a neutral block A_k (i.e. both the dataset of A_k and the *COI* of A_k are \emptyset). Let $\varphi(w_k) = \varphi(w_i)$, $M_0(w_k) = \emptyset$, $Pre(t_k) = Pre(t_i)$, $Post(t_k) = Post(t_i)[w_i/w_k, w_k/w_j]$, $Pre(t'_k) = \neg Pre(t_i)$, $Post(t'_k) = Post(t_i)[w_i/w_k, w_j/w_i]$, $Pre(t'_i) = \emptyset$, $Post(t'_i) = Post(t_i)[w_j/w_k]$.

3.4 Aspect Weaving

Given a base architecture model of a software system, a secure software architecture can be obtained by weaving aspect models with the base model. The weaving process includes the following steps.

Step 1: *Locating join points*

- (1) Analyzing security vulnerability of and threats to the software system based on security requirement.
- (2) Specifying join point conditions for connectors in the base architecture model, which reflects security vulnerability that connectors in the base model may be subject to.
- (3) Checking each connector in the base model to see whether it meets the join point condition or not. If so, the connector is a join point.

Step 2: *Constructing advices*

- (1) Identifying join points that have the same vulnerability and grouping them together as a pointcut.
- (2) Designing a mechanism (or an advice) for each pointcut such that the the vulnerability can be removed.

Step 3: *Weaving aspect*

- (1) Arranging a systematic way to searching for join points.
- (2) For each join point, modifying the base architecture model according to the corresponding advice.

The Example (Continued)

For the base architecture model Π of the Travel Planner, we first compute $DR(\Pi)$. For each $(w_i, w_j, V) \in DR(\Pi)$, and $\tau \in w_i^* \cap w_j$, check whether: (1) $V \cap sensitive(w_i) \neq \emptyset$, or (2) $V \cap sensitive(w_j) \neq \emptyset$ is true. By doing this, we find that in the base architecture model t_2 satisfies both (1) and (2), and t_3 satisfies (1) only. But the other connectors satisfy neither of them. Therefore, t_2 is join point that belongs to a pointcut of Type 3, and t_3 is a join point that belongs to a pointcut of Type 1. Following their corresponding advices, we can obtain a distributed software architecture model Π' as illustrated in Figure 7. Note that, the blocks A_2 and A_4 were reconstructed, and the block A_8 was newly added.

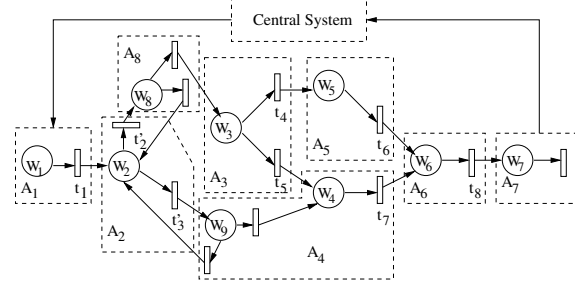


Figure 7. A secure distributed architecture model

3.5 Correctness of Aspect Weaving

One critical question is how to guarantee correctness of aspect weaving. The software architecture model in the AOD framework is essentially a SAM model extended with type information. Several correctness criteria of a SAM model are identified in [12]. One basic criterion is *element correctness*, i.e. the property specification S_{i_j} holds in the corresponding behavior model B_{i_j} , i.e. $Comp(B_{i_j}) \models S_{i_j}$.

To ensure the correctness of a SAM model is to show that all the constraints are satisfied by the corresponding behavior models. For example, we can show that our design of the Travel Planner is correct by the following two reasons. Firstly, according to the semantic definitions of w_k , t_k , and t'_k , it is easy to see that the reconstructed architecture model Π' has the same behaviors as the original model Π does, if we confine the observation over places in Π . Secondly, the integrated architecture model Π' in Figure 7 is secure in the sense that it correctly enforces the security policy. The advice for each pointcut takes the role of a refinement rule. Every refinement rule is correct, because: For Case 1, after reconstruction, we have $\psi(t'_i) = \emptyset$, and $\psi(t_k) = \psi(t'_k) = V$. Hence, $\psi(t'_i) \cap sensitive(A_i) = \emptyset$, and $\psi(t_k) \cap sensitive(A_j) = \psi(t'_k) \cap sensitive(A_j) = V \cap sensitive(A_j) = \emptyset$; Case 2 is similar to Case 1; For Case 3, after reconstruction, we have $\psi(t'_i) = \emptyset$, and $sensitive(A_k) = \emptyset$. Hence, $\psi(t'_i) \cap sensitive(A_i) = \emptyset$, $\psi(t_k) \cap sensitive(A_k) = \emptyset$, and $\psi(t'_k) \cap sensitive(A_k) = \emptyset$. This concludes the proof.

4 Conclusion

This paper has proposed a formal aspect-oriented approach to designing secure software architectures. The design process consists of problem domain modeling, separation of concerns, and system integration. SAM is used as the underlying formalism. The AOD method has several benefits. Firstly, the AOD method provides a rigorous way to identify notions in aspect-orientation and to reason about the correctness of aspect weaving. Secondly, the join point model in the AOD framework has powerful expressibility

due to hierarchical modeling ability of SAM and flexible granularity of blocks. Thirdly, the method supports reusable and reliable design of secure software architectures. Security aspects express the essential issues of security requirements and security enforcement mechanisms, and are reusable across different systems.

This paper presents some preliminary results in applying aspect orientation principle to secure software architectures design. Several related topics are not covered in this paper, such as security aspect partition, dependency between the aspect models, scalability of the aspect-oriented method, as well as tool support. We are interested in investigating these issues in the future.

References

- [1] V. Atluri, A. Chun, and P. Mazzoleni. A Chinese Wall security model for decentralized workflow systems. In *Proceedings of the 8th Conference on Computer and Communications Security*, pages 48–57, 2001.
- [2] L. Bergmans and M. Aksits. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, 2001.
- [3] B. Blakley. *CORBA Security: An Introduction to Safe Computing with Objects*. Addison-Wesley, 1999.
- [4] D. Brewer and M. Nash. The Chinese Wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 215–228, 1989.
- [5] Y. Deng, J. Wang, J. Tsai, and K. Beznosov. An approach for modeling and analysis of security system architectures. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1095–1115, 2003.
- [6] P. Devanbu and S. Stubblebine. Software engineering for security: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering, ICSE'00 Special Volume*, pages 227–239, 2000.
- [7] E. Dijkstra. *A Discipline of Programming*. Series in Computer Science. Prentice-Hall, 1976.
- [8] T. Elrad, R. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.
- [9] L. G. et al. Going beyond the sandbox: an overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of USENIX Symposium on Internet Technologies and Systems*, pages 103–112, 1997.
- [10] G. George, R. France, and I. Ray. Design high integrity systems using aspects. In *Proceedings of the 5th IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems*, pages 37–57, 2002.
- [11] X. He. A formal definition of hierarchical predicate transition nets. In *Proceedings of the 17th International Conference on Application and Theory of Petri Nets, LNCS 1091*, pages 212–229. Springer-Verlag, 1996.
- [12] X. He and Y. Deng. A framework for developing and analyzing software architecture specifications in SAM. *The Computer Journal*, 45(1):111–128, 2002.
- [13] X. He, H. Yu, T. Shi, J. Ding, and Y. Deng. Formally specifying and analyzing software architectural specifications using SAM. *Journal of Systems and Software*, 71(1-2):11–29, 2004.
- [14] M. Katara and S. Katz. Architectural views of aspects. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (ASOD 2003)*, pages 1–10. ACM, 2003.
- [15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 2072*, pages 327–353. Springer-Verlag, 2001.
- [16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1241*, pages 220–242. Springer-Verlag, 1997.
- [17] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44(10):39–41, 2001.
- [18] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 147–155. ACM, 1987.
- [19] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.
- [20] OMG. *CORBA Services: Common Object Services Specification, Security Service Specification*. Object Management Group, 1996.
- [21] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
- [22] C. Pfleeger and S. Pfleeger. *Security in Computing (3rd Edition)*. Prentice Hall PTR, 2003.
- [23] A. Rashid and L. Blair. Editorial: aspect-oriented programming and separation of crosscutting concerns. *The Computer Journal*, 46(5):527–528, 2003.
- [24] A. Rashid, A. Moreira, and J. Araújo. Modularisation and composition of aspectual requirements. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (ASOD 2003)*, pages 11–20. ACM, 2003.
- [25] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., 1996.
- [26] J. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis. VEST: an aspect-based composition tool for real-time systems. In *Proc. of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 58–69, 2003.
- [27] J. Wang, X. He, and Y. Deng. Introducing software architecture specification and analysis in SAM through an example. *Information and Software Technology*, 41:451–467, 1999.
- [28] H. Yu, X. He, Y. Deng, and L. Mo. A formal method for analyzing software architecture models in SAM. In *Proceedings of COMPSAC 2002*, pages 645–652. IEEE Computer Society Press, 2002.