Formal Software Architecture Design of Secure Distributed Systems

Huiqun Yu, Xudong He, Shu Gao, and Yi Deng
School of Computer Science
Florida International University
Miami, FL 33199, USA
{yhq|hex|sgao01|deng}@cs.fiu.edu

Abstract

This paper proposes a formal software architecture design method for distributed systems. The underlying formalism is the Software Architecture Model (SAM), a general software architecture model combining Petri nets and temporal logic. We present a two-tier structure for architectural modeling. The upper level models the workflow of a distributed system. Each place at the upper level is a super-place that corresponds to a lower level Petri net. An initial distributed architecture can be directly derived from the upper level model. Security of the architecture is checked using the dependence relation of the model. Security policies are enforced by systematically reconstructing the initial architecture. A Travel Planner is used as the example to demonstrate our approach to secure software architecture design of distributed systems.

Keywords: *Software architecture, security, distributed systems, specification, design*

1. Introduction

The fundamental objectives of software security are to protect the software systems and their data. Breach of software security could cause a loss of money or even disaster. With the widespread use of Internet, the risks from malicious attacks are increasing.

A number of techniques have been devised to protect software from these attacks [15]. These include operating system protection mechanisms and cryptographic techniques. However, these techniques, which are general in purpose, cannot directly enforce security that has no precise definition. A high degree of assurance of software security is usually achieved by independent verification of the security properties apart from good design practices and testing processes. To this end, many security models and policies were proposed [10, 14]. Various formal security verification methods were established in order to prove the correctness of security policies against the corresponding models [12, 13, 8]. Unfortunately, security modeling and verification have been largely independent of system requirements and system design. Significant benefits can be gained by integrating system design modeling with security policy enforcement [4].

This paper addresses the problem of secure software architecture design for distributed systems. We use SAM as the underlying formalism. SAM is a general software architecture model based on a dual formalism combining Petri nets and temporal logic [7]. It has been successfully applied to specification and analysis of several software systems at the architectural level, including a control and command system, a flexible manufacturing system, and an electronic commerce system [17, 19, 20]. This paper shows that specification and enforcement of software security are well suited to the SAM framework. We propose a two-tier structure for architectural modeling of distributed systems. The upper level models the workflow of a distributed system. Each place at the upper level is a super-place that corresponds to a lower level Petri net. An initial distributed architecture can be directly derived from the upper level model. The dependence relation of the architecture model is extracted in order to give source and sink of every workflow in the model and the dominating elements for flowing. Security of the architecture model is checked using the dependence relation. Security policies are enforced by systematically reconstructing the initial architecture.

The main contributions of this paper are as follows.

- *An architectural model for secure distributed systems*: Architectural level modeling of software systems provides better system understanding, multiple levels of reuse, clear dimensions for system evolution and ease for system management. The SAM-based approach provides a unified model for software architecture and software security.

- *A formal method for security design*: We provide a for-

mal method for checking the security of software architecture using the dependence relation of the model, and propose a security design method that automatically enforces security policies by reconstructing the distributed software architecture.

The rest of the paper is organized as follows: Section 2 gives a brief introduction to the SAM model and its theoretical basis. Section 3 presents a software architecture model for distributed systems. Section 4 presents security modeling and design method. Section 5 is the conclusion.

2. The Foundations of SAM

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 Petri nets, while their properties (or constraints) are specified by temporal logic formulas. The interfaces of components and connectors are *ports*. In the following, we will briefly introduce the formal basis of SAM: predicate transition nets (PrT nets) [6], and a linear-time temporal logic (LTL) [9].

2.1. Predicate Transition Nets

Definition 1 (A PrT net) *A* PrT net *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* $\overrightarrow{F}$ *and* $\overleftarrow{F}$ *to denote* $F \cap (P \times T)$ *and* $F \cap (T \times P)$*, 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.*

Remarks:

- 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 **S** = (S, *OP*) and a set *Eq* of **S**-equations. Signature **S** = (S, *OP*) includes a set of sorts S and a family *OP*= ($OP_{s1,...,sn,s}$) of sorted operations for s_1, ..., s_n, $s \in S$. For each$s \in S$, we use CON_s to denote $OP_{,s}$ (the 0-ary operation of sort s), i.e. the set of constant symbols of sort s. The **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 **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 $\{k_1x_1, ..., k_nx_n\}$.
- 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 $\alpha = \{x_1 \leftarrow c_1, \ldots, x_n \leftarrow c_n\}$, 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.
- 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 $\tau^{\bullet}$), respectively.

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 an infinite sequence of markings: $\sigma = M_0, M_1, M_2, \ldots$, for each pair of markings (M_i, M_{i+1}), there is an enabled transition τ_i such that $M_i\tau_iM_{i+1}$ [1].

Definition 2 (Computation) *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.

[1] We assume that there be an idle transition (or stuttering) for every PrT net, which is always enabled.

2.2. A Linear-Time Temporal Logic

The requirement specification language is LTL. 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 $\sigma : s_0, s_1, \ldots$, 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 [9].

Two important classes of properties are: *safety* and *response*.

- Safety properties are those that can be expressed by a formula $\Box\psi$, for some past formula ψ.
- Response properties are those that can be expressed by a formula $p \Rightarrow \Diamond q$, for some past formula p and q.

A model $\sigma : s_0, s_1, \ldots$ satisfies a temporal formula if $(\sigma, 0) \models p$. A temporal formula p that is *valid* in a program P if it is satisfied under all computations of P.

Several correctness criteria of SAM models are identified in [7]. One basic criterion is *element correctness* which requires the property specification S to hold in the corresponding behavior model B, i.e. $Comp(B) \models S$.

3. An Architecture Model of Distributed Systems

3.1. The Architecture Model

We present a two-tier structure of distributed systems using SAM. At the upper level, each composition corresponds to the whole or a part of the workflow in the system. At the lower level, each composition corresponds to a specific task. A *composition constraint* describes the property related to several components and connectors. Instead, a *component constraint* defines the property of a single component. A component constraint at the upper level will be inherited as a composition constraint at the refined lower level. Every constraint is specified by an LTL formula. Figure 1 shows a graphical view of the software architecture model, where the higher level component A_2 (or task) is decomposed into a lower level PrT net.

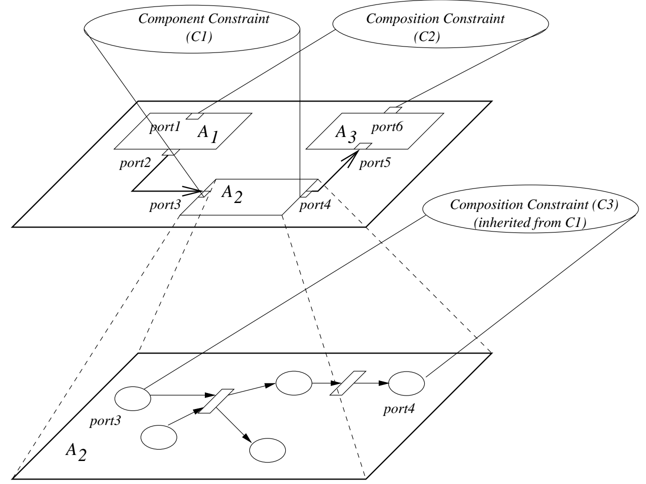

Figure 1. A two-tier model of distributed systems

A task at the upper level is a PrT net model, which describes lower level compositions with a unique input (place) and a unique output (place), as illustrated in Figure 2. We use I and O to denote input and output, respectively. In abstract view, a task is a super-place in the PrT net. There is a typing correspondence between the super-place with the task, i.e., for each super-place W, $\varphi(W) = \varphi(I) \times \varphi(O)$.

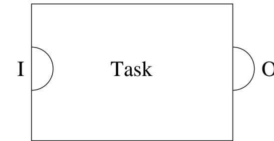

Figure 2. The structure of a task

At the lower level, each composition is a PrT net, which refines its upper level counterpart (or super-place). The refinement relation contains two aspects.

- *Interface Conformation*: The input and output of super-place are two places in the lower level PrT net.
- *Property Preservation*: The upper level component constraint is inherited as a lower level composition constraint.

Established techniques in SAM, such as reachability tree analysis, deductive proof, and structural induction, are applicable to formally analyzing the refinement relation of architecture models [19, 18, 7, 20].

An example (A Travel Planner) Consider a travel agency which processes reservations for flight seats, hotel rooms and rental cars [2]. The tasks involved in the travel agency 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 less 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 3 gives a PrT net model of the Travel Planner. In the underlying specification *SPEC* = (S, *OP*, *Eq*), where S includes elementary sorts such as Integer and Boolean, and also sorts Usr, Time, Route, 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.

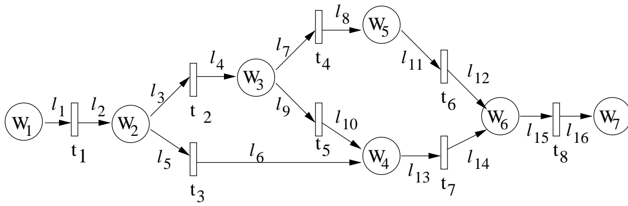

Figure 3. A PrT net model of the Travel Planner

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

- $P = \{W_1, W_2, W_3, W_4, W_5, W_6, W_7\}$.
- $T = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8\}$.
- $F = \{(w_1, t_1), (t_1, w_2), (w_2, t_2), (t_2, w_3), \ldots, (w_6, t_8), (t_8, w_7)\}$.
- $\varphi(W_1) = (Usr \times Time \times Route) \times (Usr \times Time \times Route)$,
  $\varphi(W_2) = \varphi(W_3) = (Usr \times Time \times Route) \times (Usr \times Time \times Route \times Price \times Ok)$,
  $\varphi(W_4) = \varphi(W_5) = (Usr \times Time \times Route) \times (Usr \times Time \times Flight)$,
  $\varphi(W_6) = (Usr \times Time \times Route) \times (Usr \times Time \times Hotel \times Price)$,
  $\varphi(W_7) = (Usr \times Time \times Route) \times (Usr \times Time \times Car \times Price)$.
- $Pre(t_1) = true$, $Post(t_1) = (M'(W_1) = \emptyset \wedge M'(W_2) = l_2)$,
  $Pre(t_2) = (price > 400 \vee ok = false)^2$, $Post(t_2) = (M'(W_2) = \emptyset \wedge M'(W_3) = l_4)$,
  $Pre(t_3) = (price' \leq 400 \wedge ok' = true)$, $Post(t_3) = (M'(W_2) = \emptyset \wedge M'(W_4) = l_6)$,
  $Pre(t_4) = (ok'' = true)$, $Post(t_4) = (M'(W_3) = \emptyset \wedge M'(W_5) = l_8)$,
  $Pre(t_5) = (ok''' = false)$, $Post(t_5) = (M'(W_3) = \emptyset \wedge M'(W_4) = l_{10})$,
  $Pre(t_6) = true$, $Post(t_6) = (M'(W_5) = \emptyset \wedge M'(W_6) = l_{12})$,
  $Pre(t_7) = true$, $Post(t_7) = (M'(W_4) = \emptyset \wedge M'(W_6) = l_{14})$,
  $Pre(t_8) = true$, $Post(t_8) = (M'(W_6) = \emptyset \wedge M'(W_7) = l_{16})$.
  Note that for each transition τ, $R(\tau) = Pre(\tau) \wedge Post(\tau)$.
- $L = \{(W_1, t_1) \mapsto l_1, (t_1, W_2) \mapsto l_2, \ldots, (W_6, t_8) \mapsto l_{15}, (t_8, W_7) \mapsto l_{16}\}$, 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.
- $M_0(W_1) = \{((usr_0, time_0, route_0), _)\}$,
  $M_0(W_2) = M_0(W_3) = M_0(W_4) = M_0(W_5) = M_0(W_6) = M_0(W_7) = \emptyset$.

3.2. An Initial Distributed Control Architecture

A distributed system control has many advantages over its centralized counterpart. These include fewer message exchanges between the central control module and the task execution, and less control burden by one single central controlling authority, which is desirable in autonomous environments [11]. We use the notion of *agent* to denote such kind of autonomous entities. Specifically in this context, an agent is a module that contains tasks (i.e. super-places in the upper level model) and their postsets.

An initial distributed control architecture is derived from the upper level PrT net model by

- taking each task and its postset as a distributed agent, and

[2] The 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'''.

- adding a *central control* module, which initiates the first agent in the workflow and collects the final result from the last agent.

The example (continued) Figure 4 below is the initial control architecture of the Travel Planner. In addition to the central control module, there are seven agents in the initial control architecture.

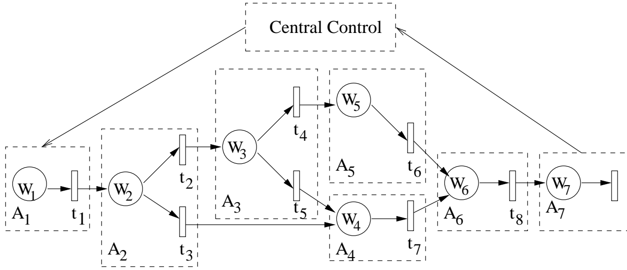

Figure 4. An initial distributed control architecture

However, one obvious problem for the distributed control is security. With a distributed control, the entire workflow is sent to the agent A_1 by the central control module. After the execution of W_1, A_1 forwards the remaining workflow to the following agents, in this case, A_2. After executing W_2, A_2 evaluates the following dependences and forwards the remaining workflow to the next appropriate agent, and so on. The agent A_2 has the knowledge that if it fails or if the ticket costs more than $400, the task needs to be sent to A_3 (on behalf of Delta airlines), which is a competing company. Due to this fact, A_2 can manipulate the price of the ticket and may reduce it to $399, which may result in a loss of business to A_3, or may prevent the customer from getting a better price than $399 that may potentially be offered by Delta. Security issues are investigated in the next section.

4. Security Modeling and Design

4.1. Security Models

A security policy is a set of properties that specify how an organization manages, protects, and distributes sensitive information. These properties can be access control properties, or information flow properties, or both. The Chinese Wall policy [3] was introduced as an attempt to balance commercial discretion with mandatory controls. The model is based on a hierarchical organization of data objects as follows:

- *Basic objects* are individual items of information (e.g. files), each concerning a single corporation;
- *Company datasets* define groups of objects that refer to the same corporation;
- *Conflict of interest classes (COI)* define company datasets that refer to competing corporations.

Given the object organization, the Chinese Wall policy prevents information flows that cause conflict of interest for individual consultants. A modified Chinese Wall security model is capable of addressing the problem of security in distributed systems.

Definition 3 (Sensitive dataset) *Given a distributed architecture model, an agent A in the model, and a task w in the agent, a variable v is* sensitive *to the task w if either it appears in $Pre(w^\bullet)$ and it is in the dataset of A, or it belongs to the intersection of variable set from $L(^\bullet w, w)$ and COI of A. The* sensitive dataset of the task w *(denoted by* sensitive(w)*) is the set of variables that are sensitive to w. We select one of the tasks in A as the* main task *of A. The* sensitive dataset of the agent A *(denoted by* sensitive(A)*) is assigned to the sensitive dataset of its main task.*

For example, the sensitive datasets of tasks in the initial architecture model (illustrated in Figure 4) are:
$sensitive(W_1) = sensitive(W_4) = sensitive(W_5) = sensitive(W_6) = sensitive(W_7) = \emptyset$,
$sensitive(W_2) = \{price, price'\}$, $sensitive(W_3) = \{price\}$.

In the initial architecture model, each agent contains only one task, which is the corresponding main task by default. Therefore, the sensitive datasets of agents are:
$sensitive(A_1) = sensitive(A_4) = sensitive(A_5) = sensitive(A_6) = sensitive(A_7) = \emptyset$,
$sensitive(A_2) = \{price, price'\}$, $sensitive(A_3) = \{price\}$.

Definition 4 (A security policy) *A distributed control architecture $\Pi = (P, T, F, \varphi, R, L, M_0)$ is* secure*, if for every agent 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 agent and its tasks, neither task's decisive data value should be sensible to its own agent, nor agent's information should be leaked to another agent in the same COI.

For example, according to the above security criterion, the initial distributed control architecture in Figure 4 is not secure, because W_2 is a task in the agent A_2 and $t_2 \in W_2^\bullet$, but $\psi(t_2) \cap sensitive(A_2)$ is $\{price\}$, instead of $\emptyset$.

4.2. A Formal Security Design Method

We first define a new concept called the *dependence relation*. Then, we show how to use dependence relation to check security of the architecture model, and finally, we demonstrate that the security policy can be correctly enforced through reconstructing the distributed architecture.

Definition 5 (Dependence relation) *Given an upper level 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 4. We have:

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

Given an architecture model of a distributed system, the security policy in the workflow level can be enforced by the following steps:

1. Construct an initial control architecture from the workflow Π.
2. Compute the dependence relation $DR(\Pi)$.
3. For each $(w_i, w_j, V) \in DR(\Pi)$, we reconstruct the architecture in the ways that are illustrated in Figure 5.

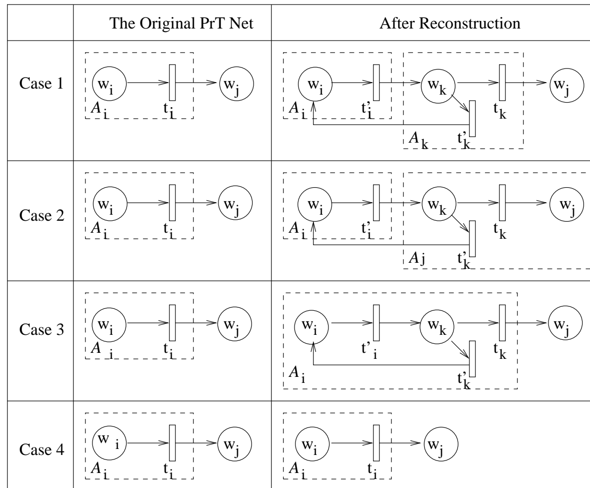

Figure 5. Four cases for reconstruction

- Case 1: If $V \cap sensitive(w_i) \neq \emptyset$, and $V \cap sensitive(w_j) \neq \emptyset$, then add a new task w_k and two transitions $t_k,\ t'_k$, which constitute a neutral agent A_k (i.e. both the dataset of A_k and the COI of A_k are $\emptyset$). Let $\varphi(w_k) = \varphi(w_i)$, $M_0(w_k) = \emptyset$, $Pre(t_k) = Pre(t_i)$, $Post(t_k) = Post(t_i)[w_i/w_k, w_k/w_j]$, $Pre(t'_k) = \neg Pre(t_i)$, $Post(t'_k) = Post(t_i)[w_i/w_k, w_j/w_i]$, $Pre(t'_i) = \emptyset$, $Post(t'_i) = Post(t_i)[w_j/w_k]$.
- Case 2: If $V \cap sensitive(w_i) \neq \emptyset$, and $V \cap sensitive(w_j) = \emptyset$, then add a new task w_k and two transitions $t_k,\ t'_k$, which are appended to the A_j. The main task of A_j is w_j. Let $\varphi(w_k) = \varphi(w_i)$, $M_0(w_k) = \emptyset$, $Pre(t_k) = Pre(t_i)$, $Post(t_k) = Post(t_i)[w_i/w_k, w_k/w_j]$, $Pre(t'_k) = \neg Pre(t_i)$, $Post(t'_k) = Post(t_i)[w_i/w_k, w_j/w_i]$, $Pre(t'_i) = \emptyset$, $Post(t'_i) = Post(t_i)[w_j/w_k]$.
- Case 3: If $V \cap sensitive(w_i) = \emptyset$, and $V \cap sensitive(w_j) \neq \emptyset$, then add a new task w_k and two transitions $t_k,\ t'_k$, which are appended to the agent A_i. The main task of A_i is w_i. Let $\varphi(w_k) = \varphi(w_i)$, $M_0(w_k) = \emptyset$, $Pre(t_k) = Pre(t_i)$, $Post(t_k) = Post(t_i)[w_i/w_k, w_k/w_j]$, $Pre(t'_k) = \neg Pre(t_i)$, $Post(t'_k) = Post(t_i)[w_i/w_k, w_j/w_i]$, $Pre(t'_i) = \emptyset$, $Post(t'_i) = Post(t_i)[w_j/w_k]$.
- Case 4: If $V \cap sensitive(w_i) = \emptyset$, and $V \cap sensitive(w_j) = \emptyset$, then keep the same structure.

According to the semantic definitions of $w_k,\ t_k$, and t'_k, we can easily see that the reconstructed architecture model has the same behaviors as the original model Π does, if we confine the observation over places in Π. What's more, the following theorem shows that the resulting architecture model is secure.

Theorem 6 (Security enforcement) *The above steps will result in a distributed software architecture, which enforces the security policy.*

Proof: For Case 1, 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 $\psi(t'_k) \cap sensitive(A_k) = \emptyset$; For Case 2, 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 $\psi(t_k) \cap sensitive(A_j) = \psi(t'_k) \cap sensitive(A_j) = V \cap sensitive(A_j) = \emptyset$; For Case 3, the proof is similar to Case 2; For Case 4, it is obvious that $\psi(t_i) \cap sensitive(A_i) = \emptyset$. By the definition of the security policy, we conclude the proof. $\square$

The example (continued) Using the above steps, we can obtain a distributed software architecture illustrated in

Figure 6, in which the security policy has been enforced. Note that, the agents A_2 and A_4 were reconstructed, and the agent A_8 was newly added.

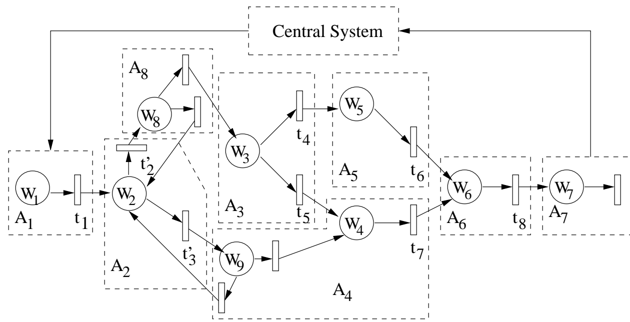

Figure 6. A secure distributed architecture

5. Conclusion

This paper proposed a formal approach to design of secure software architecture for distributed systems. The underlying formalism of SAM supports hierarchical modeling of distributed control. A two-tier architecture model is presented. The upper level models the workflow of distributed systems. Each of the place of the upper level is a super-place that corresponds to a lower level Petri net. An initial distributed architecture can be directly derived from the upper level model. Security of the architecture is checked using the dependence relation of the model. A security enforcement method is proposed, which automatically enforces security policies for the upper level workflow.

SAM is a general software architecture model based on a dual formalism combining Petri nets and temporal logic. SAM provides a unique hierarchical framework to tailor the dual formalisms of Petri nets and temporal logic for software architecture specification. In this paper, we chose a high level predicate transition nets , and a linear-time temporal logic as its theoretical basis. Other kind of Petri nets [16] and/or temporal logic [5] can also be used. SAM has been successfully applied to specification and analysis of several software systems at the architectural level. The work in this paper complements the SAM-based software architecture modeling techniques, and shows that specification and enforcement of software security are well suited to the SAM framework.

Several other works were proposed to deal with distributed system modeling and security enforcement at workflow level. In [1], a low level Perti net called TCPN was used to model and analyze workflows. But TCPN is not well suited to hierarchical structure modeling, and thus is not suitable for complex system modeling. Security enforcement was investigated in [2]. However, the proposed graph model of workflow is semi-formal and not self sufficient. In our method, once a formal architecture model is established, dependence relations of the workflows can be automatically derived, and security policy can be mechanically enforced by reconstructing the initial distributed control architecture. The main merits of our method over those works are: (1) The hierarchical architecture model is well suited to complex system modeling; (2) The SAM-based method has a solid theoretical basis that facilitates formal reasoning and design; (3) SAM provides a unified framework for both software architecture modeling and security enforcement.

It should be pointed out that our concern in this paper is only about theoretical aspect of secure software architecture modeling and design. We considered only a single security policy enforcement and used a simple example to illustrate its applicability. Interesting topics such as multi-policy enforcement, modularity in policy representation, composition, design and analysis tools, as well as implementation techniques, are omitted. We will explore these topics in the future.

Acknowledgements

This work is supported in part by the NSF under grants HDR-9707076 and CCR-0098120, and by NASA under grant NAG 2-1440.

References

[1] N. Adam, V. Atluri, and W. Huang. Modeling and analysis of workflows using Petri nets. *Journal of Intelligent Information Systems*, 10:131–158, 1998.

[2] V. Atluri, A. Chun, and P. Mazzoleni. A Chinese Wall security model for decentralized workflow systms. In *Proceedings of the 8th Conference on Computer and Communications Security*, pages 48–57, 2001.

[3] D. Brewer and M. Nash. The Chinese Wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 215–228, 1989.

[4] 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.

[5] E. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 16. Elsevier Science Publisher B.V., 1990.

[6] 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.

[7] X. He and Y. Deng. A framework for developing and analyzing software architecture specifications in SAM. *The Computer Journal*, 45(1):111–128, 2002.

[8] C. Ko and T. Redmond. Noninterference and intrusion detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 177–187, 2002.

[9] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[10] J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley & Sons, 1994.

[11] P. Muth, D. Wodtke, J. Weissenfels, A. Dttrich, and G. Weikum. From centralized workflow specification to distributed workflow execution. *Journal of Intelligent Information Systems*, 10:159–184, 1998.

[12] R. Peri. *Specification and Verification of Security Policies*. PhD thesis, The School of Engineering and Applied Science, University of Virginia, January 1996.

[13] R. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–165, 2000.

[14] P. Samarati and S. Vimercati. Access control: policies, models and mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design, LNCS 2171*, pages 137–196. Springer Verlag, 2001.

[15] J. Shim, A. Qureshi, and J. Siegel. *The International Handbook of Computer Security*. The Glenlake Publishing Company, Ltd., 2000.

[16] M. Trompedeller. *A Classification of Petri Nets*. http://www.daimi.dk /PetriNets/classification, 1995.

[17] J. Wang and Y. Deng. Incremental modeling and verification of flexible manufacturing systems. *Journal of Intelligent Manufacturing*, 10:458–520, 1999.

[18] J. Wang, Y. Deng, and G. Xu. Reachabilty analysis of real-time systems using time Petri nets. *IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics*, 30(5):725–736, 2000.

[19] 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.

[20] 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 Soecity Press, 2002.