

Integrating Security Administration into Software Architectures Design *

Huiqun Yu, Xudong He, Yi Deng, Lian Mo
School of Computer Science
Florida International University
Miami, FL 33199, USA
{yhq|hex|deng|lmo01}@cs.fiu.edu

Abstract

Software architecture plays a central role in developing software systems that satisfy functionality and security requirements. However, little has been done to integrate system design with security enforcement, which would otherwise benefits both development process and system's quality of service (QoS). This paper proposes a formal method to integrate security administration into software architecture design. We use the Software Architecture Model (SAM), a general software architecture model combining Petri nets and temporal logic, as the underlying formalism. Several techniques for designing functionality of software architectures are presented. Security modeling and administration methods are proposed. As such, SAM serves as a common platform for modeling, design and analysis of secure software architectures.

Keywords: *Software architecture, security, formal method, design, analysis*

1. Introduction

Software security is a critical concern for modern information enterprises. Breach of software security could cause a loss of money or even disaster. Software architecture plays a central role in developing software systems that satisfy functionality and security requirements [12]. Two major elements of architectures are components and connectors. Important security concerns, such as authentication and access control, arise out of interactions between components. However, architecture descriptions are typically expressed informally and accompanied by box-and-line drawings indicating the global organization of computational entities and interaction among them [1]. While informal description

of software architecture may provide useful documentation, it is impossible to analyze an architecture for consistency or determine non-trivial properties. There is no way to check that a system implementation is faithful to its architectural design.

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 policy models were proposed [11]. Various formal security verification methods were established in order to prove the correctness of security policies against the corresponding models [10, 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 [3].

To address the above problems, we propose a formal approach to designing secure software architectures based on SAM [14]. SAM is a general software architecture model based on a dual formalism combining Petri nets and temporal logic. Security system architecture design in SAM includes two parts. One is the functionality part, which deals with the overall structure of the software architecture. The other is the security part, which handles security requirement modeling, specification, and enforcement. Several heuristics are proposed in order to guide the architectural design at both element level and composition level. Software security is enforced through well-defined rules. Analysis techniques are presented to ensure the correctness of architectural design. The main contribution of this paper is providing a formal method for integrating security administration into software architecture design on a common semantic domain.

The rest of the paper is organized as follows: Section 2 presents software architecture design techniques in SAM. Section 3 proposes security administration method. Section 4 is the conclusion.

* Supported in part by the NSF under grants HRD-0317692 and CCR-0226763, and by NASA under grant NAG 2-1440.

2. Software Architectures Design

2.1. The Structure of SAM Models

A SAM software architecture is defined by a hierarchical set of compositions, each of which 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. In this paper, we use predicate transition nets (PrT nets) [4], and a linear-time temporal logic (LTL) [9]. The interfaces of components and connectors are *ports (places)*. One interface requirement is that the input and output ports of the element must be maintained at a lower level.

2.2. Element Level Design

In SAM, each element (either a component or a connector) is specified by a tuple $\langle S, B \rangle$, where S is a property specification (written in LTL), and B is a behavior model (defined by a PrT net). To define an element constraint S , we can either directly formulate the given user requirements or carry out a cause and effect analysis by viewing input ports as cause and output ports as effects. Canonical forms [9] for a variety of properties such as safety, guarantee, obligation, response, persistence and reactivity are used as guidelines to define property specifications.

The general procedure to develop B includes the following steps.

1. Use all the input and output ports as places of B ;
2. Identify a list of events directly from the user requirements or through Use Case analysis [2];
3. Represent each event with a simple PrT net;
4. Merge all the PrT nets together through shared places to obtain B ;
5. Apply the transformation technique [6] to make B more structured and/or meaningful.

2.3. Composition Level Design

SAM supports both top-down and bottom-up system development approaches. The top-down approach is used to develop a software architecture specification by decomposing a system specification into specifications of components and connectors and by refining a higher-level component into a set of related sub-components and connectors at a low

level. The bottom-up approach is used to develop a software architecture specification by composing existing specifications of components and connectors and by abstracting a set of related components and connectors into a higher-level component. Often both the top-down approach and the bottom-up approach have to be used together to develop a software architecture specification.

In SAM, only a pair consisting of a related component and connector can be composed meaningfully. Suppose that $\langle S_1, B_1 \rangle$ and $\langle S_2, B_2 \rangle$ be a pair of a related component and connector, i.e. they share some ports. Their composition is obtained through: (1) composing B_1 and B_2 by merging identical ports, and (2) composing S_1 and S_2 by conjoining $S_1 \wedge S_2$.

2.4. An Example

Consider a simple clinical information (SCI) system, which manages and maintains the personal health information. The patients' medical documents, including basic information, test results and treatment records, are classified into different access levels by the security administrator. The users (or roles), such as registration clerks, nurses, technicians, physicians etc., have different clearance levels to access those documents. The architecture of the SCI system includes four components as illustrated in Figure 1.

- the *Application System (AS)*, which provides users with documents access services,
- the *Access Interface System (AIS)*, which coordinates the interactions between other components,
- the *Policy Evaluator (PE)*, which performs evaluation decisions based on certain security policies that govern the access to the protected resources, and
- the *Database Management System (DBMS)*, which manages the medical documents.

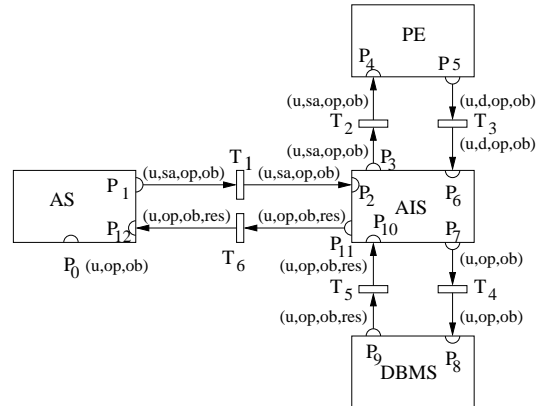


Figure 1. The SCI software architecture

Table 1. Variables in Figure 1

Variable	Description
u	User name
op	Requested operation
ob	Object name
sa	Static attributes of the user
d	Access control decision
res	Result feedback

The properties of components and connectors can be specified by LTL formulas.

- AS_S – A user request will be sent to the AIS service:
 $\forall(u, op, ob). \exists sa. (\Box(P_0(u, op, ob) \rightarrow \Diamond P_1(u, sa, op, ob)))$
- AIS_S –
 (1) AIS will invoke PE once a request is received:
 $\forall(u, sa, op, ob). (\Box(P_2(u, sa, op, ob) \rightarrow \Diamond P_3(u, sa, op, ob)))$
 (2) If AIS gets a positive decision, it will forward the user request to DBMS. Otherwise it will directly inform the user that the request was ‘denied’:
 $\forall(u, d, op, ob). (\Box((P_6(u, d, op, ob) \wedge d = 'Y'$
 $\rightarrow \Diamond P_7(u, op, ob))$
 $\wedge (P_6(u, d, op, ob) \wedge d = 'N'$
 $\rightarrow \Diamond P_{11}(u, op, ob, 'denied'))))$
- PE_S – When PE is invoked, it will return access control decision:
 $\forall(u, sa, op, ob). \exists d. (\Box(P_4(u, sa, op, ob) \rightarrow \Diamond P_5(u, d, op, ob)))$
- $DBMS_S$ – Once DBMS receives a request from AIS, DBMS will feedback a result:
 $\forall(u, op, ob). \exists res. (\Box(P_8(u, op, ob) \rightarrow \Diamond P_9(u, op, ob, res)))$

The following are connector property specifications, where every connector plays the role of a pipe.

- $$T_1: \forall(u, sa, op, ob). (\Box(P_1(u, sa, op, ob) \rightarrow \Diamond P_2(u, sa, op, ob)))$$
- $$T_2: \forall(u, sa, op, ob). (\Box(P_3(u, sa, op, ob) \rightarrow \Diamond P_4(u, sa, op, ob)))$$
- $$T_3: \forall(u, d, op, ob). (\Box(P_5(u, d, op, ob) \rightarrow \Diamond P_6(u, d, op, ob)))$$
- $$T_4: \forall(u, op, ob). (\Box(P_7(u, op, ob) \rightarrow \Diamond P_8(u, op, ob)))$$
- $$T_5: \forall(u, op, ob, res). (\Box(P_9(u, op, ob, res) \rightarrow \Diamond P_{10}(u, op, ob, res)))$$
- $$T_6: \forall(u, op, ob, res). (\Box(P_{11}(u, op, ob, res) \rightarrow \Diamond P_{12}(u, op, ob, res)))$$

The composition-level property specification (denoted by DES) is obtained by conjoining the property specifications of all components and connectors, i.e.

$$DES: AS_S \wedge AIS_S \wedge PE_S \wedge DBMS_S \\ \wedge T_1 \wedge T_2 \wedge T_3 \wedge T_4 \wedge T_5 \wedge T_6$$

One overall requirement REQ of the SCI system is that every user request must be processed, which can be specified by the following LTL formula:

$$\forall(u, op, ob). \exists res. (\Box(P_0(u, op, ob) \rightarrow \Diamond P_{12}(u, op, ob, res)))$$

One way to ensure the composition-level correctness is to show $DES \vdash REQ$. The following is a proof outline.

- (1) Assume precedence $P_0(u, op, ob)$
- (2) \forall, \exists and \Box instantiation in AS_S :
 $P_0(u, op, ob) \Rightarrow \Diamond P_1(u, sa, op, ob)$
- (3) Apply modus ponens rule to (1) and (2):
 $\Diamond P_1(u, sa, op, ob)$
- (4) Instantiate \forall and \Box in T_1 to \Diamond in (3):
 $\Diamond P_1(u, sa, op, ob) \rightarrow \Diamond \Diamond P_2(u, sa, op, ob)$
- (5) Apply modus ponens rule to (3) and (4):
 $\Diamond \Diamond P_2(u, sa, op, ob)$
- (6) Apply \Diamond absorbing rule to (5):
 $\Diamond P_2(u, sa, op, ob)$
- (7) By repeating the above Steps (4) to (6) to all subsequent element property specifications $AIS_S, T_2, PE_S, T_3, AIS_S, T_4, DBMS_S, T_5, AIS_S, T_6$, we can derive the formula in Step (8).
- (8) $\Diamond P_{12}(u, op, ob, res)$
- (9) Eliminate precedence assumption in (1) by (8):
 $P_0(u, op, ob) \rightarrow \Diamond P_{12}(u, op, ob, res)$
- (10) \Box, \exists , and \forall generalization in (9):
 $\forall(u, op, ob). \exists res. (\Box(P_0(u, op, ob) \rightarrow \Diamond P_{12}(u, op, ob, res)))$

Thus we proved $DES \vdash REQ$.

For the element-level correctness analysis, we need to show that the property specification S holds in the corresponding behavior model B . To this end, several automatic verification techniques were developed [5, 15], which include symbolic model checking, theorem proving, and reachability tree analysis.

3. Security Administration Method

3.1. Security Policy Modeling

A security policy model is a mathematical restatement of the security policy that must be enforced by the computer system. In the following, we present a framework for administration of security policies based on SAM. The administration commands is a generalization of the take-grant model [13].

Places We assume that the data types include E (Entities), Sub (Subjects), Obj (Objects), W (Rights), where $E = Sub \cup Obj$. We assume that $W = \{t, g, r, w\}$, which contains four access rights: *take*, *grant*, *read*, and *write*, respectively. Each place p represents a subject in

Sub. The type (or called inscription) of the place p is $\varphi(p) = (E \xrightarrow{m} \wp(W))$, where \wp is power set operator, and \xrightarrow{m} is mapping operator. Access matrix can be derived from the markings of places. $(e \mapsto \alpha) \in M(p)$ means that the subject p has α right(s) on the entity e under the marking M .

Transitions The state changing commands include four rules: the *take-rule*, *grant-rule*, *create-rule*, and *revoke-rule*. Each rule corresponds to a transition in a PrT net. In the following, s, s_1, s_2 denote subjects, α, β denote subsets of access rights, and e denotes an entity. We assume that firing of the transition tr update the marking from M to M' .

- *The take-rule*

The command $take(s_1, s_2, \beta)$ makes the subject s_1 to grant its β right(s) to s_2 . Formally, the command corresponds to the transition tr in Figure 2, where the dashed parts are newly added by applying the command, while the solid parts are the old ones.

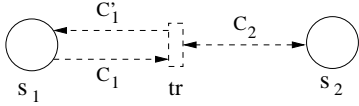


Figure 2. A PrT net for the take-rule

- $C_1 = M(s_1)$
- $C_2 = M(s_2)$
- $C'_1 = C_1 \dagger \{(e \mapsto \alpha \cap \beta) \mid (e \mapsto \alpha) \in C_2 \wedge \alpha \cap \beta \neq \emptyset\}$, where \dagger is the overriding operator.
- $R(tr) = Spec(M(s_1), M(s_2), \beta) \wedge (M'(s_1) = C'_1)$

Informally, $Spec(M(s_1), M(s_2), \beta)$ is a first-order formula of security policy specification¹ that stipulates the relationship on capabilities of s_1 and s_2 , as well as the set β . After firing tr , certain access rights in β of the subject s_2 are passed to the subject s_1 .

- *The grant-rule*

Using the command $grant(s_1, s_2, \beta)$, s_1 grants its β right(s) to s_2 . Its formal definition is similar to that of the take-rule, and omitted.

- *The create-rule*

The command $create(s, e, \beta)$ makes s to create an entity e and to claim β right(s) to e , whose formal definition is illustrated in Figure 3.

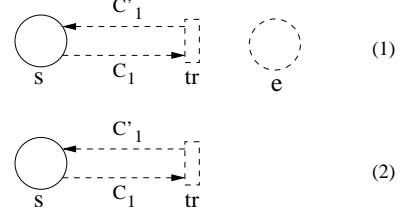


Figure 3. PrT nets for the create-rule

Case 1: if e is a subject, then a new place e and a new transition tr are added, and

- $C_1 = M(s)$
- $C'_1 = C_1 \dagger \{(e \mapsto \beta)\}$
- $R(tr) = Spec(M(s), M(e), \beta) \wedge (M'(s) = C'_1)$.

Case 2: if e is an object, then only a new transition tr is added, and

- $C_1 = M(s)$
- $C'_1 = C_1 \dagger \{(e \mapsto \beta)\}$
- $R(tr) = Spec(M(s), e, \beta) \wedge (M'(s) = C'_1)$.

- *The revoke-rule*

The command $revoke(s, e, \beta)$ removes β right(s) to the entity e from the subject s , whose formal definition is illustrated in Figure 4.

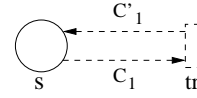


Figure 4. A PrT net for the revoke-rule

- $C_1 = M(s)$
- $C'_1 = C_1 \dagger \{e \mapsto (\alpha - \beta) \mid (e \mapsto \alpha) \in C_1\}$
- $R(tr) = Spec(M(s), M(e), \beta) \wedge (M'(s) = C'_1)$.

3.2. Security Administration

A security policy enforcement consists of a sequence of PrT nets, such that

1. the initial PrT net contains only one place that denotes the security administrator (the super user), and
2. at each step, one of the state changing rules is applied to the current PrT net to obtain a new PrT net.

The constraint for each transition in the PrT net guarantees the correctness of the security policy enforcement.

For the SCI system, let $Sub = \{s, u_1, \dots, u_m\}$ denote the set of the security administrator and users, and $Obj = \{d_1, \dots, d_n\}$ denote the set of the patients' medical documents. The security administrator construct a PrT model as follows.

¹Additional elements may be needed in order to specify a particular security policy. For example, to specify Multi-level security (MLS) policy, elements such as clearance levels, and the mapping from entities to clearance levels are necessary.

- The create-rule is used to create users and/or document. Hence, the security administrator take the access rights from all of the users.
- The revoke-rule is used to remove access rights from users.
- The take-rule and the grant-rule to exchange rights among users.

By applying these rules, a security policy model is obtained as illustrated in Figure 5. The security model plus its connections to ports P_4 and P_5 actually constitute a refinement of the *Policy Evaluator* in Figure 1.

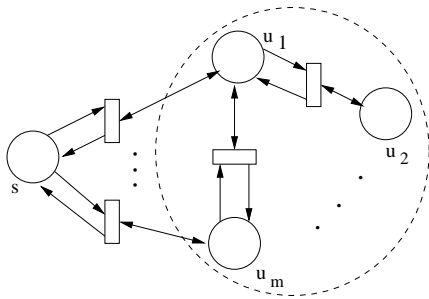


Figure 5. A security model of the SCI system

4. Concluding Remarks

This paper proposes a formal approach to designing secure software architectures. We use SAM, a general software architecture model combining Petri nets and temporal logic, as the underlying formalism. A systematic method for software architecture design and security administration proposed.

Our method has several advantages. Firstly, the method provides a rigorous way to modeling and designing secure software architectures. The method is based on a well established formalism SAM [5]. Not only SAM is capable of modeling complex software architectures, it also proves to be a powerful method for system analysis; Secondly, we integrate security administration into software architectures design, which is based on a common semantic model; Thirdly, the separation of security concerns from functionality decreases design complexity and helps to enhance system's QoS.

Interesting topics such as multi-policy enforcement, modularity in policy representation, composition, design and analysis tools are omitted in this paper. Another promising direction is aspect-oriented approach [7] to security system design, which addresses separation of concerns in software development by using specialized mechanisms to encapsulate concerns whose behavior crosscuts essential application functionality.

References

- [1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.
- [2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Longman, Inc., 1999.
- [3] 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.
- [4] 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.
- [5] X. He and Y. Deng. A framework for developing and analyzing software architecture specifications in SAM. *The Computer Journal*, 45(1):111–128, 2002.
- [6] X. He and J. Lee. A methodology for constructing predicate transition net specifications. *Software-Practice and Experience*, 21(8):845–875, 1991.
- [7] 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.
- [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] 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.
- [11] 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.
- [12] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., 1996.
- [13] L. Snyder. Formal models of capability-based protection systems. *IEEE Transactions on Computers*, C-30(3):172–181, 1981.
- [14] 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.
- [15] 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.