Constructing Authorization Systems Using Assurance Management Framework

Hongxin Hu, *Student Member, IEEE*, and Gail-Joon Ahn, *Senior Member, IEEE*

***Abstract*—Model-driven approach has recently received much attention in developing secure software and systems. In addition, software developers have attempted to employ such an emerging approach in the early stage of software development life cycle. However, security concerns are rarely considered and practiced due to the lack of appropriate systematic mechanisms and tools. In this paper, we introduce a multilayered software development life cycle (SDLC), which is based on an assurance management framework (AMF), focusing on the development of authorization systems. AMF facilitates comprehensive realization of formal security model, security policy specification and verification, generation of security enforcement codes, and rigorous conformance testing. We also articulate our experience in analyzing role-based authorization requirements and realizing those requirements in constructing a role-based authorization system.**

***Index Terms*—Authorization, model-driven approach, role based, unified modeling language (UML).**

I. Introduction

SECURITY has become a core ingredient of nearly most modern software and information systems. Unfortunately, the integration of such critical security concerns throughout the overall software development process has been exercised in an ad hoc manner. Security countermeasures are often integrated into existing systems after significant security problems are discovered during the administration or usage phase. This afterthought approach causes several severe problems such as unsatisfied security requirements, integration difficulties, and mismatches between design models and actual implementations. In order to effectively address security aspects in the software development process, more convenient and mature mechanisms should be designed to help software developers analyze and capture security concerns from the early stage of software development life cycle.

Several issues should be taken into account for accommodating such requirements. First, there exists a gap between security models and building secure systems with those models. Security models are generally described in some forms of formalism. However, software developers are reluctant to fully adopt a formal security model for their development tasks. Consequently, it is very important to have mechanisms and corresponding tools to aid software developers or system administrators in understanding and articulating a specific-security model and associated policies in the software analysis and design stages. Second, it is crucial to verify and validate the security model and policies before actual implementation commences, such that flaws and conflicts in the system design can thus be identified as early as possible and efficiently resolved. Furthermore, the security model and policies that are specified with modeling languages should be translated to security enforcement codes for deriving appropriate security properties. Additionally, the consistency between the design model and its implementation, and the correctness of the translation should be evaluated.

To address these issues, we introduce an assurance management framework (AMF), which ensures that formal security models are fully realized in real systems through model representation, policy specification and validation, and generation of security enforcement codes. In addition, formal verification of security models and corresponding policies, and automatic test case generation are supported in this framework. Moreover, we believe that more practical engineering processes for secure software development should be addressed so that software developers can easily adopt our approach in their development practices. Therefore, we further propose a multilayered software development life cycle (SDLC) for building role-based authorization systems based on AMF framework. This multilayered SDLC includes four development phases: authorization requirement analysis, authorization model and policy verification, authorization system design and implementation, and conformance testing. In order to demonstrate the applicability of this engineering approach, we utilize a role-based-banking-authorization system as a case study.

The rest of this paper is organized as follows. Section II introduces our assurance management framework. We discuss our layered software development life cycle for constructing role-based-authorization systems in Section III. In Section IV, we elaborate the processes of building a role-based-banking-authorization system to demonstrate the feasibility of our approach. Section VI concludes this paper with future research directions.

II. Assurance Management Framework

In this section, we introduce our assurance management framework [19], which is depicted in Fig. 1. The framework is designed for facilitating the secure software development. In the modeling stage, formal specifications of security model and

Manuscript received June 8, 2009; revised November 27, 2009 and February 26, 2010; accepted March 30, 2010. Date of publication May 10, 2010; date of current version June 16, 2010. This paper was recommended by Associate Editor E. R. Weippl. This work was supported in part by the National Science Foundation under Grant NSF-IIS-0242393, Grant NSF-IIS-0900970, and Grant NSF-CNS-0831360 and in part by the Department of Energy Early Career Principal Investigator Award under Grant DE-FG02-03ER25565.

The authors are with Security Engineering for Future Computing Laboratory, the School of Computing, Informatics, and Decision Systems Engineering, Ira A. Fulton Schools of Engineering, Arizona State University, Tempe, AZ 85281-8809 USA (e-mail: hxhu@asu.edu; gahn@asu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TSMCC.2010.2047856

1094-6977/$26.00 © 2010 IEEE

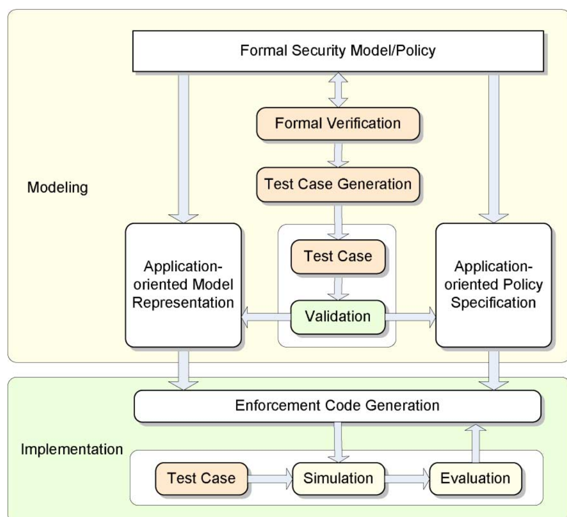

Fig. 1. Assurance management framework.

policies are verified. Then, application-oriented security model representation and corresponding security policy specification are derived from formal specification of security model and policy, which can also be utilized to produce test cases automatically. Furthermore, the generated test cases are used to check conformance to the formal specification. In the implementation stage, security enforcement codes are generated systematically from the application-oriented specifications. The correctness and conformance of the generated codes with respect to the formal specification are also evaluated by using the generated test suites in the simulation. We divide all tasks in the AMF framework into two phases as follows:

1) Automatic realization of security model and policy
   a) *Application-oriented security model representation:* The representation of a security model should enable software engineers to integrate security aspects into the applications without knowing details of the security model. In this regard, a well-designed and general-purpose visual representation should be considered as a means to represent the security model in an intuitive fashion.
   b) *Application-oriented security policy specification:* Security policies are an important means for laying out high level security rules for organizations to avoid unauthorized accesses. A considerable amount of work has been carried out in the area of security policy specification. A high-level policy specification approach should be considered in the practical system development process so that security policies can be easily integrated into the system design by system developers.
   c) *Automatic generation of security enforcement codes:* It is also a crucial aspect to make the transparent transition from system design to secure system development. The goal of code generation in AMF is to automatically generate executable modules from the application-oriented specification of security model and policies by a well-known software engineering mechanism, such as the model-driven development (MDD) [31]. The generated security modules would be eventually integrated into the real systems to achieve an acceptable degree of assurance in secure system development.
2) Automatic analysis and testing of security model and policy
   a) *Automatic analysis of formal security model and policy:* One of the promising advantages in mathematical- and logic-based techniques for security model and policies is that formal reasoning of the security properties can be performed. Since the formal security model and policies serve as a basis for secure system development in AMF, obviously the formal specifications of model and policies should be proved based on the expected security properties. Formal verification offers a rich toolbox containing a variety of techniques such as model checking, SAT solving, and theorem proving, for supporting automatic-system analysis.
   b) *Automatic test case generation from formal specification:* While formal verification can prove violation or satisfaction of properties, it is not sufficient enough to practically guarantee the assurance. The proof only shows that a given formal specification fulfills a set of properties. However, we should consider that the actual implementation is influenced by other facts, such as platform, transformation approach, compiler, and so on. Consequently, the implemented module should be rigorously tested.
   The most significant recent development in testing is the application of verification approach, which generates test cases from the formal specification. We attempt to apply this notion to our framework so that we can derive test cases from the formal security model and policy. As a result, the generated test cases are fed into a validator and a simulator to check whether the system design and development comply with the formal specification.

III. Multilayered Software Development Life Cycle for Authorization Systems

In order to demonstrate the usability of AMF framework, more practical engineering processes for developing secure systems should be articulated. We now present a multilayered software development life cycle shown in Fig. 2, which is especially designed for software developers to build a role-based-authorization system based on our AMF framework. The layered software development life cycle consists of four development phases as follows:

A. Phase 1: Requirement Identification

This phase analyzes security requirements to determine the prospective actors and business processes of an authorization system. In addition, the organizational authorization rules need to be captured in this phase for reflecting authorization

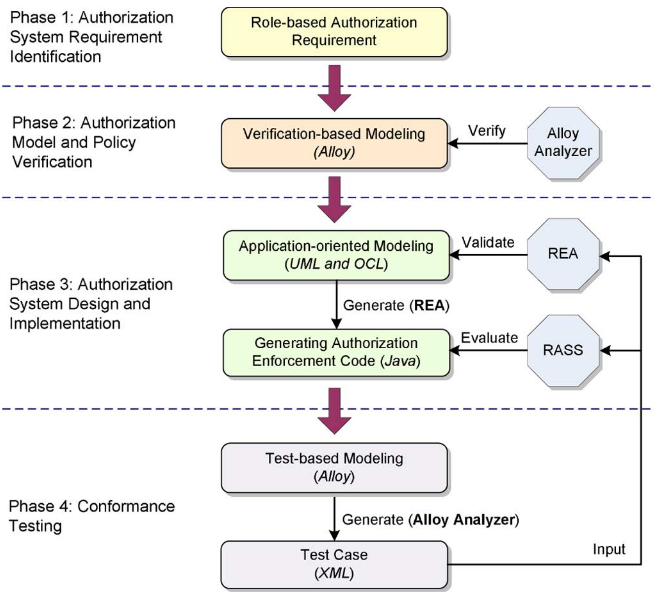

Fig. 2. Multilayered SDLC for constructing role-based-authorization systems with AMF framework and corresponding tools.

requirements in the subsequent development phases. Furthermore, corresponding entities, relations and constraints are identified and defined based on the authorization requirement analysis.

In this paper, we mainly focus on a role-based authorization. We adopt the NIST/ANSI RBAC (role-based access control) standard [3], [16] as the underlying security model since it includes most of RBAC features [29], and has been widely adopted in information-assurance community. Based on this RBAC standard, a domain-specific RBAC model is constructed from the identified RBAC elements and relations. Authorization constraints are an important means in RBAC for laying out high-level security rules including organizational and system-centric perspectives. An RBAC-constraint-specification language RCL2000 [6] is utilized to define authorization constraints formally based on the identified authorization requirements. The RCL2000-based constraint specifications are used for policy analysis, test case generation, and application-oriented policy generation for the subsequent phases in the layered software development life cycle.

B. Phase 2: Model and Policy Verification

In AMF, the formal security model and policy serve as a foundation of secure software development. The correctness of the design and implementation is based on the premise that the formal security model and policy are valid. This phase formally verifies the specifications of security model and policy against a given set of security properties. Such rigorous examination allows us to provide a higher assurance in the layered software development life cycle.

To support this task, RBAC model and RCL2000-based constraints are reconstructed and converted, respectively, to alloy [20]. Then, we use alloy analyzer to further examine properties of RBAC model and corresponding constraints.

C. Phase 3: System Design and Implementation

To build a secure system based on a particular security model, it is very important to have an application-oriented representation of the security model for software engineers. In addition, a high-level policy-specification approach should be considered in the practical system development process so that security policies can be easily understood and incorporated into the system design by software engineers. We use unified modeling language (UML), which is a general-purpose visual-modeling language, for representing the domain-specific RBAC model for authorization-system design. Then, we adopt object constraint language (OCL) [37] for transforming RCL2000-based constraints to OCL-based constraint specifications that are easily embedded in UML-based RBAC representation. In AMF, executable modules are generated automatically from the UML/OCL-based specification of authorization model and policy. The generated authorization modules would be eventually used to construct role-based-authorization systems.

RBAC authorization environment (RAE) [4] is utilized to support the major features in AMF, especially for authorization system design and implementation. RAE tool is composed of three functional components: specification, validation, and code-generation components. In addition, the domain-specific RBAC model and constraints can be validated to detect misconfigurations and conflicts during the authorization system design phase.

D. Phase 4: Conformance Testing

Even though formal analysis can prove whether or not a given formal specification satisfies all of critical properties, it is necessary to perform a conformance testing. AMF automatically derives test cases from the formal specification of access control model and policy. Counterexamples are also articulated as part of test cases. Then, the generated test cases are used to check conformance of the application-oriented system design against the formal specification. In addition, the correctness and conformance of the generated codes with respect to the formal specification are evaluated as well. We use alloy analyzer with a SAT solver to achieve this goal by generating test cases from alloy-based model and policy specification. The generated test cases are fed into RAE to check the validity of RBAC model and constraint specifications as well as utilized by an RBAC-authorization-simulation system (RASS) [4] to evaluate the generated RBAC codes under simulation.

IV. Constructing a Role-Based Authorization for a Financial Service System

In this section, we illustrate the applicability of our layered software development life cycle based on system requirements for a role-based-banking-authorization system derived from [12]. A domain-specific RBAC model is built based on the RBAC standard to reflect the domain-specific authorization requirements. We call our customized RBAC model for this financial service environment as *Bank-RBAC model*.

A. Authorization Requirement Analysis

1) System Requirement of the Banking Application: The banking application can be used by various bank officers to perform transactions on customer accounts. The bank officers, called "actors" in the system requirement analysis, include *teller*, *customer service representative*, *loan officer*, *accountant*, *accounting manager*, *internal auditor*, and *branch manager*. The business processes called "use cases" in the system requirement analysis are articulated including the following basic tasks: 1) create, delete, or modify customer deposit accounts; 2) create, delete, or modify customer loan accounts; 3) create general ledger report; and 4) modify or verify the ledger posting rules.

The participating actors and business tasks performed by each actor in the banking system are defined as follows:

1) *teller*: input and modify customer deposit accounts;
2) *customerServiceRep*: create and delete customer deposit accounts;
3) *loanOfficer*: create and modify status of loan accounts;
4) *accountant*: create general ledger reports.
5) *accountingManager*: in addition to the inherited privileges from *accountant*, modify ledger-posting rules;
6) *internalAuditor*: verify ledger posting rules; and
7) *branchManager*: perform all privileges inherited from other actors.

Since some actors may perform the functions of others, there exist dependencies between the actors. For example, we may have two actors who are accountants but one actor is senior to another actor with additional privileges in the system. Thus, we can model the dependency relations between *accountingManager* and *accountant*, and between *branchManager* and all other actors. It implies that *accountingManager* inherits all privileges of *accountant* and *branchManager* inherits capabilities from all actors.

In the banking system, several organizational authorization rules should be enforced to support common security principles such as separation of duty and least privilege. We address these authorization rules in the banking system as follows.

Rule 1: Some bank officers, such as *customerServiceRep* and *accountingManager*, cannot be performed by the same user.

Rule 2: Some users cannot act as the same bank officer.

Rule 3: Some bank officers, such as *customerServiceRep* and *loanOfficer*, cannot be activated by the users in the same transaction session.

Rule 4: A user can play a bank officer role x only if the user has been assigned to a different bank officer role y.

Rule 5: A banking business process can be performed by a bank officer only if the bank officer already has the ability to perform another banking business process.

Rule 6: The number of users assigned to a bank officer should be restricted. For example, only one user can be assigned to an *internalAuditor*.

2) Identifying RBAC Entities, Relations, and Constraints for the Banking Application: Identifying RBAC entities. Since an actor in use case analysis portrays business responsibilities in a given system, the actors defined earlier for the banking application can be treated directly as corresponding roles [11], [38] for the role-based-banking-authorization system. Therefore, we can identify following seven bank roles: *teller*, *customerServiceRep*, *loanOfficer*, *accountant*, *accountingManager*, *internalAuditor*, and *branchManager*. The bank use cases represent the bank business processes performed by each bank actor. Hence, we also derive the permissions from the same use cases accordingly. Table I lists the permissions identified for the banking authorization system as well as the corresponding bank operations and bank objects.

TABLE I
BANK PERMISSIONS AND CORRESPONDING BANK OPERATIONS AND BANK OBJECTS

Permission	Operation	Object
createDepositAccount	Create	Deposit account
deleteDepositAccount	Delete	Deposit account
inputDepositAccount	Input	Deposit account
modifyDepositAccount	Modify	Deposit account
createLoanAccount	Create	Status of loan account
modifyLoanAccount	Modify	Status of loan account
modifyLedgerReport	Modify	General ledger report
createLedgerPostingRule	Create	Ledger posting rule
verifyLedgerPostingRule	Verify	Ledger posting rule

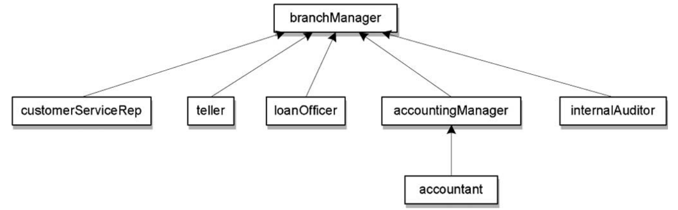

Fig. 3. Role hierarchy for banking-authorization system.

TABLE II
ASSIGNMENTS BETWEEN PERMISSIONS AND ROLES

BankRole	BankPermission
customerServiceRep	createDepositAccount,
teller	inputDepositAccount, modifyDepositAccount
loanOfficer	createLoanAccount, modifyLoanAccount
accountingManager	modifyLedgerReport
internalAuditor	createLedgerPostingRule
branchManager	verifyLedgerPostingRule

Identifying RBAC relations: There are dependency relations between actors in the banking use case. These dependency relations can be mapped to inheritance relations in role hierarchy. Fig. 3 shows the role-hierarchy structure in the banking application.

We identified the actors and the business processes that can be performed by each actor in the banking application so far. With this identification, we now capture the assignment relations between permissions and roles. Table II summarizes all permission-to-role assignments in the banking application.

Identifying RBAC constraints: Several organizational rules need to be captured in system requirements for the banking application. These organizational authorization rules are represented and enforced in RBAC systems by means of authorization constraints. Several RBAC constraints including static separation of duty, dynamic separation of duty, prerequisite condition, and cardinality rules are used to support these organizational authorization rules. Table III shows six RBAC constraints and corresponding organizational authorization rules identified for the banking application.

TABLE III
IDENTIFIED CONSTRAINTS, SUPPORTED AUTHORIZATION RULES AND CORRESPONDING COLLECTIONS FOR THE BANKING SYSTEM

Constraint		Rule	Collection
Statical SoD constraint	SSoD-CR	Rule 1	SCR
	SSoD-CU	Rule 2	SCR, SCU
Dynamical SoD constraint	Session-based DSoD	Rule 3	DCR
Prerequisite constraint	Prerequisite-Role	Rule 4	PR
	Prerequisite-Permission	Rule 5	PP
Cardinality constraint	Cardinality-Role	Rule 6	RUC

RBAC constraints can be formally specified with RCL2000. We illustrate two typical RBAC constraints specified in RCL2000 for the banking application as follows:

Constraint 1: (SSoD-CR): The number of conflicting roles, which are from the same conflicting role set, authorized to a user cannot exceed the cardinality of the conflicting role set.

$$|\,\mathtt{roles}^{*}(\mathtt{OE}(\mathtt{U})) \cap \mathtt{GS}(\mathtt{OE}(\mathtt{SCR}))\,| \leq \mathtt{GC}(\mathtt{OE}(\mathtt{SCR}))$$

Constraint 2: (User-based DSoD): The number of conflicting roles, which are from the same conflicting role set, activated directly (or indirectly via inheritance) by a user cannot exceed the cardinality of the conflicting role set.

$$|\,\mathtt{roles}^{*}(\mathtt{sessions}(\mathtt{OE}(\mathtt{U}))) \cap \mathtt{GS}(\mathtt{OE}(\mathtt{DCR}))\,| \leq \mathtt{GC}(\mathtt{OE}(\mathtt{DCR}))$$

As shown in Table III, six collections, SCR, SCU, DCR, PR, PP, and RUC, are required for defining constraints in the banking application. These collections specified in constraint specifications should be instantiated to enforce constraints properly at runtime in the banking application. For instance, in the banking application, the static conflicting role collection (SCR) for the *SSoD-CR* constraint is instantiated as follows, where the cardinality of the conflicting role sets in SCR is *one*. It implies the conflicting roles in the same conflicting role set cannot be assigned to a user simultaneously.

```
SCR = {({customerServiceRep,
accountingManager}, 1),
({customerServiceRep, internalAuditor},1),
({loanOfficer, accountingManager}, 1),
({loanOfficer, internalAuditor}, 1),
({accountingManager, internalAuditor},1),
({teller, accountant}, 1),
({teller, loanOfficer}, 1),
({teller, internalAuditor}, 1),
({accountant, loanOfficer}, 1),
({accountant, internalAuditor}, 1)}}
```

The *user-based DSoD* constraint defines the dynamic conflicting roles cannot be activated by a user at the same time. For example, the dynamic conflicting role set (DCR) for the *user-based DSoD* constraint with the cardinality can be defined as follows:

```
DCR = {({customerServiceRep,
        loanOfficer}, 1)}.
```

B. Authorization Model and Policy Verification

1) Model Representation in Alloy: In order to verify role-based-authorization model and policy, the authorization model should be verified. First, we represent the authorization model in alloy. The alloy representation of the bank-RBAC model with respect to NIST/ANSI RBAC is given as follows:

```
module Bank-RBAC
------------ Core Bank-RBAC-------------
sig BankUser {}
sig BankRole {}
sig BankOperation {}
sig BankObject {}
sig BankPermission {BankOperation,
                    BankObject}
sig BankSession {}
sig URA{
  ura: BankUser->BankRole}
sig PRA{
  Pra: BanKPermission->BankRole}
sig US{
  us: BankUser!->BankSession}
sig SR{
  sr: BankSession->BankRole }
sig PB{
  pb: BankOperation->BankObject}
----------Hierarchical Bank-RBAC--------
sig RRA{
  hierarchy: BankRole->BankRole}
-------Constrained Bank-RBAC-----------
sig SCR{
  conflict_role: set BankRole,
  cardinality: Int}
sig SCU{
  conflict_user: set BankUser,
  cardinality: Int}
sig DCR{
  conflict_role: set BankRole,
  cardinality: Int}
sig PR{
  prerequisite_role: set BankRole}
sig PP{
  prerequisite_permission:
  set BankPermission}
sig RUC{
  role: BankRole,
  cardinality: Int}
```

This alloy specification includes the core elements and relations in the bank-RBAC model. Role-hierarchy relation is also defined to support an hierarchical RBAC. Six signatures are also defined to support RBAC constraints in alloy specification.

2) Constraint Verification in Alloy: A critical task for specifying constraints is to determine whether a set of constraint expressions reflects the desired authorization requirements properly. Normally, constraints prohibit an action or state occurring in the system. Two issues should be considered carefully while analyzing a given set of constraints against the expected authorization requirements. First, constraints may be too weak, named *under-constraint* to grant undesired system states. A *safety* problem (i.e., the leakage of a right to an unauthorized user) can be

resulted from the weak constraints. Second, constraints may be too strong, named *over-constraint* to deny desired system states. Strong constraints can cause *availability* problems. For example, an entitled user cannot own the right to access a resource.

Using formal verification, both over- and under-constraints for an access control model specification are analyzed automatically with a set of given access control properties.

a) Identifying Under-Constraint: If unexpected authorization property is satisfied by the access-control-model specification, *under-constraint* is detected.

In the banking application, the following authorization property with a given role hierarchy is prohibited because of separation of duty principles.

1) *Two conflicting roles, such as teller and accountant, are authorized to the same user.*

This unexpected authorization property can be specified in alloy as follows:

```
pred Check_SSoD-CR[disj teller, accountant:
BankRole, u: BankUser, scr: SCR]{
  teller in scr.conflict_role &&
  accountant in scr.conflict_role &&
  scr.cardinality = 1 &&
  teller in u.(URA.ura).~*(RRA.hierarchy)
  && accountant in u.(URA.ura).
  ~*(RRA.hierarchy)
}
  run Check_SSoD-CR
```

In the banking application, suppose the policy designer only specifies a *simple SSoD-CR* constraint, which ignores the role-hierarchy relation in constraint expressions. An RCL2000 policy expression for this *simple SSoD-CR* constraint has been given in [6]. The policy is then transformed to alloy expression using an RAE tool. The following alloy fact specifies an alloy-based *simple SSoD-CR* constraint:

```
fact SSoD-CR {
  all u:BankUser | all scr:SCR |
    #(u.(URA.ura) &
    scr.conflict_role) <= scr.cardinality }
```

When running the predicate `Check_SSoD-CR` defined earlier in alloy analyzer, we can discover that conflicting roles, *teller* and *accountant* are *indirectly* assigned to a user. It indicates that the unexpected authorization property is held by the constraint specification. Thus, we can conclude this *simple SSoD-CR* constraint is too week with respect to the given authorization property.

b) Identifying over-constraint: If the verifier discovers that the expected authorization property is not satisfied by the access-control-model specification, this points out the defined constraints are too strong. Thus, the constraint definitions should be refined by reducing the restriction of constraints.

Taking into account the following authorization properties for dynamic separation of duty principle.

1) *A user cannot activate two conflicting roles in the same session, but can activate them in the different session.*

We can specify this expected authorization property in alloy as follows:

```
assert Check_DSoD {
  all u: BankUser |
  all disj customerServiceRep,
    loanOfficer: BankRole |
 all disj s1,s2:BankSession |
 all dcr: DCR |
  customerServiceRep in dcr.conflict_role
  && loanOfficer in dcr.conflict_role &&
  dcr.cardinality = 1 &&
  (u->s1) in US.us &&
  (u->s2) in US.us &&
  (customerServiceRep->s1) in~SR.sr &&
  (loanOfficer->s1) !in~SR.sr &&
  (loanOfficer->s2) in~SR.sr }
 check Check_DSoD
```

With a *User-based DSoD* constraint[1] defined earlier, the following alloy fact specifies an alloy-based DSoD constraint:

```
fact DSoD {
  all u:BankUser | all dcr:DCR |
    #(u.(US.us).(SR.sr) & dcr.conflict_role)
      <= dcr.cardinality }.
```

Running "`check Check_DSoD`" in alloy analyzer, counterexamples are found, which indicate the expected authorization property expressed in assertion `Check_DSoD` is denied by the constraint specification. That is, the constraint is too strong, and should be weakened to contain the expected authorization properties.

C. Authorization System Design and Implementation

1) Model Representation in UML and OCL: We first represent domain-specific RBAC model using UML class diagram. *Role*, *User*, *Permission*, *Session*, *Object*, and *Operation* in the NIST/ANSI RBAC standard are parameterized by *BankRole*, *BankUser*, *BankPermission*, *BankSession*, *BankObject*, and *BankOperation* for the bank domain, respectively. A complete representation of the Bank-RBAC model including core bank-RBAC, hierarchical bank-RBAC, and constrained bank-RBAC in UML class diagram can be derived from the UML representation of RBAC standard [4]. Six classes `SCR`, `SCU`, `DCR`, `PR`, `PP`, and `RUC` are defined for constraint specifications in the banking application.

The functional specification of the RBAC standard defines various functions, such as administrative function, review function, and system function. All functions defined in bank-RBAC model are specified in OCL. For brevity, we elaborate several typical OCL-based functional definitions for bank-RBAC model.

a) Functional definition of core bank-RBAC: Administrative commands: These commands are for the creation and maintenance of RBAC element sets and relations by administrators.

[1]In order to reduce the complexity, we omit the role hierarchy in this constraint.

The functions for adding and deleting an element such as `AddRole` and `DeleteRole` can be addressed in UML class diagrams as well. A command specification for `AssignUser` is defined as follows:

```
context BankRole::AssignUser(u:BankUser)
pre : self.user->excludes(u)
post : self.user->includes(u).
```

Review functions: These functions are for administrators to query RBAC element sets and relations. Query operations do not change system states. Instead, each query returns a value or a set of attributes of corresponding RBAC element set or relation. In OCL, they are defined as a `body` expression. The following OCL definition supports a review function `UserPermissions`:

```
context BankUser::UserPermissions():
      Set(BankPermission)
body : self.role.permission->asSet()
```

Supporting system functions: The functions are applied to create and maintain RBAC dynamic properties with regard to users' sessions and access control decisions. `CheckAccess` checks whether an operation on an object is allowed to be performed in a particular session. OCL representation for this function is defined as follows:

```
context BankSession::CheckAccess(op:
      BankOperation, ob:BankObject):Boolean
pre : true
post : self.SessionRoles()->exists(r|
        r.permission->exits(p|
        p.operation->includes(op)
        and p.object->includes(ob))).
```

b) Functional definition of hierarchical bank-RBAC: There are four functions such as `AddSenior`, `DeleteSenior`, `AddJunior`, and `DeleteJunior` for administrators to maintain inheritance relationships among roles. We define two review functions `AllSeniors` and `AllJuniors` for role hierarchy. The following definition is for `AllSeniors`:

```
context BankRole::AllSeniors():Set(BankRole)
body : self.senior->union(self.senior
      ->collect(r|r.AllSeniors()))->asSet()
```

c) Functional definition of constrained bank-RBAC: The definitions related to constraint expressions are incorporated with corresponding components in UML-based model representation. We introduce two system functions `CheckStaticConstraints` and `CheckDynamic Constraints` for bank-RBAC model to enforce constraint expressions and to check conflicts. For example, when a user is assigned to a role, we need also enforce relevant constraints. The following `AssignUser` function includes assignment operations as well as constraint enforcement.

```
context BankRole::AssignUser(u:BankUser)
pre : self.user->excludes(u)
post : self.user->includes(u) and
     if (not self.CheckStaticConstraints())
       or (not u.CheckStaticConstraints())
     then
       self.user->excludes(u)
     endif
```

Using RAE tool, RCL2000-based RBAC constraints are translated automatically to corresponding OCL constraint expressions in the system-design phase. For instance, the OCL expression for *SSoD-CR* constraint for the banking application is translated as follows.

```
context BankUser
inv: let
  scr:SCR = SCR.allInstances()->any(true)
in
  self.AuthorizedRoles()->
  intersection(scr.RoleSet)->
  size()<=scr.SetCardinality
```

2) Model and constraint validation: As indicated in the modeling stage, the authorization tool RAE can help policy designers validate RBAC model and policies. The results of validation are utilized to find if the current model and constraints are adequate, detecting constraint conflicts. Our validation approach checks a set of system states against authorization policies. In UML-based representation, a system state is an UML object diagram, which can be changed by creating and deleting objects as well as inserting and removing links between objects. There are six components, `SCR`, `SCU`, `DCR`, `PR`, `PP`, and `RUC` that are used to specify conflicting sets. Constraint expressions employ these components to specify corresponding constraints. Each component is instantiated as an object during constraint validation.

An analyzer component in an RAE tool is responsible for model and constraint validation. The main task of the analyzer component is to parameterize and interpret RBAC constraint expressions and evaluate these constraints against the current system state. When conflicting element sets are changed, or constraint expressions are established or modified, the analyzer checks if the constraints are violated by the current system state. Also, if the system state is changed, the analyzer also evaluates all authorization constraints against the changed system state. If any of the authorization constraint is violated during this process, it indicates that the authorization constraint is false, or the system state is undesirable. If the system state violates the RBAC constraints, it generates a report that assists users to find the root causes for the conflict and to make decisions for resolving the conflict.

The following example illustrates how the banking authorization model and constraints can be validated during the system design phase. Suppose a policy designer accidentally omits the following condition that he has defined in advance: *customerServiceRep* and *accountingManager* are mutually exclusive. Later, the policy designer decides to define that *accountingManager* is a prerequisite role of *customerServiceRep* due to the change of an organizational policy. When a user is assigned to *customerServiceRep*, the *Prerequisite-Role* constraint forces the user to be assigned to *accountingManager* as well. Clearly, we can observe that the *SSoD-CR* constraint is violated since *customerServiceRep* and *accountingManager* are conflicting roles and

they cannot be assigned to the same user. Hence, the policy designer should remove either prerequisite relation or mutual exclusion relation between *customerServiceRep* and *accountingManager*.

3) Generating Authorization Enforcement Code: The code generation in MDD approach enables the developers to build a real application by creating a platform-independent model and then transforming it to platform-dependent codes. Our code generation is to generate security enforcement codes with some degree of assurance based on model specifications represented by UML and OCL. As we addressed in the previous sections, all model components and constraints are evaluated so the enforcement codes generated from our model representation should fully reflect features and functionalities of the security model. Although we select the Java language as the target language in this paper, we believe the mechanisms can be also used for other languages. The process of mapping model specification to enforcement codes is performed by the adoption of the tools such as Octopus [2] and Dresden OCL toolkit [1]. Instead of discussing the details of the translating process, we elaborate some issues and solutions related to this translating process.

In our specification of bank-RBAC model, bank-RBAC model elements and relations are defined in the UML class diagram. The functionalities and constraints are specified with OCL expressions. To implement UML model elements, the classes, attributes, operations, and associations need to be translated into corresponding Java classes or operations. Then, each class in the model is mapped to one Java class; an operation for the class is created by one operation in Java class; and an attribute and its association with the class in the model generate a private class member and a `get` and `set` operations in the Java class. In addition, the basic types of OCL are mapped to corresponding Java types. For example, `Real` in OCL is mapped to `float` in Java. OCL collection type is implemented as a library using `Set` or `List` of Java language. It is slightly complicated when implementing this library, because OCL collections have a large amount of predefined operations. These operations should be defined as standard operations using Java. Based on the implemented standard OCL library, we can generate Java codes from OCL expressions directly.

In our implementation, two special system functions, `CheckStaticConstraints` and `CheckDynamic Constraints`, for constrained-RBAC are created automatically to collect and enforce static and dynamic constraint expressions respectively. Although we can use a universal function, such as a `CheckConstraints` function, to check all constraints for one component, for the purpose of making checking procedures more efficient, we provide two system functions for constraint checking: session-related constraint expressions are performed by `CheckDynamicConstraints`, and other constraints are enforced by `CheckStaticConstraints`.

D. Conformance Testing

In our conformance-testing approach, two kinds of test cases are generated for testing a constraint. One is called *negative test case*, which is considered as an undesired access control authorization state that should be denied by the constraint in the banking system. Another test case is named *positive test case*. This test case represents a desired access control authorization state and should be allowed to appear in the banking system.

Negative test case can be derived from a formal specification, in which an access-control-model specification does not satisfy the constraint specification. Since the constraint specification is taken out from the access control model specification, the authorization property expressed by constraint specification is not exactly held on the access-control-model specification. The verifier may generate counterexamples, which can be used to construct negative test cases.

Positive test case is generated from a formal specification, as we draw the constraint specification from the access-control-model specification, and take the negated-constraint specification as the authorization property to verify the access control model specification. Counterexamples are derived and utilized to build positive test cases.

We take the *SSoD-CR constraint* as an example to demonstrate the process of automated test case generation for the banking application. For instance, suppose the banking application has *one* user `Bob Dylan` and *two* conflicting roles, `customerServiceRep` and `loanOfficer`. We first need to add the following assignments into bank-RBAC model specification in alloy.

```
one sig Bob Dylan extends BankUser{}
one sig customerServiceRep,loanOfficer
  extends BankRole{}
fact SCR_rules {
  customerServiceRep in SCR.conflict_role
  && loanOfficer in SCR.conflict_role }
```

The following assertion is defined to derive the negative test cases for the constraint specification.

```
assert SSoD-CR {
  all u:BankUser | all scr:SCR |
    #(u.(URA.ura) & scr.conflict_role)
      <= scr.cardinality }
check SSoD-CR.
```

In order to derive positive test cases for the simple *SSoD constraint*, the *negated*-constraint specification is used as an authorization property. We define an assertion for this objective as follows:

```
assert Neg_SSoD-CR {
  all u:BankUser | all scr:SCR |
    #(u.(URA.ura) & scr.conflict_role)
      > scr.cardinality }
check Neg_SSoD-CR.
```

Note that the above assertion states the number of roles—which are from a conflicting role set—assigned to a user must exceed the cardinality number of the conflicting role set. Supposing the cardinality number is *one*, it means a user must own two or more conflicting roles.

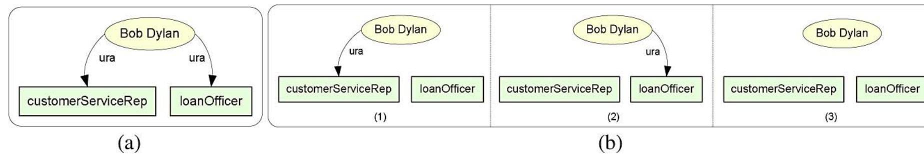

Fig. 4. Example of generated test case for *SSoD-CR* constraint. (a) Negative test case. (b) Positive test case.

Performing an `SSoD-CR` assertion, which contains *SSoD-CR constraint* specification, with the scope of *one* user and *two* roles. Alloy analyzer can generate a negative test case [see Fig. 4(a)] for the *SSoD-CR constraint*, such that the user `Bob Dylan` is assigned to two conflicting roles `customerServiceRep` and `loanOfficer`. On the other hand, through running a `Neg_SSoD-CR` assertion including *negated SSoD-CR constraint* specification, three positive test cases serving as desired system states are constructed as shown in Fig. 4(b). The generated test cases are utilized to check whether *SSoD-CR constraint* expression in system design and implementation phases complies with the formal constraint specification.

V. Related Work

There are several work on UML-based modeling of security model. Epstein and Sandhu [15] and Shin and Ahn [32] demonstrated the usage of UML for the representation of RBAC model. Ahn and Shin [7] showed how RBAC constraints can be expressed in UML using OCL. Jurjens [22] proposed an extension to UML that defines several new stereotypes toward formal security verification of elements. Basin *et al.* [10] and Lodderstedt *et al.* [24] defined a metamodel to generate security definition languages, an instance of which is SecureUML, a platform-independent language for RBAC. Doan *et al.* [14] attempted to accommodate mandatory access control issues in UML-based software design. Ray *et al.* [28] specified reusable RBAC policies using UML diagram templates and showed how RBAC policies can be easily integrated with the application. Alghathbar and Wijesekera [8] defined an approach AuthUML that includes a process and a modeling language to express RBAC policies via use cases. All of these approaches accommodated security requirements without considering the validation of security model and policy, and the translation to a concrete implementation. Our approach uses the standard UML to represent security model, supports the model and policy validation, and translates security model to enforcement codes.

In addition, there are several related work on the specification of access control policies such as formal logic-based languages [9], [21] high-level languages [13], [27], and visualization of access control policies [23], [34]. In our approach, we introduced more comprehensive mechanism, which can accommodate three approaches to specify authorization policies.

One important aspect of policy analysis is to formally check general properties of access control policies, such as inconsistency and incompleteness [17], [18]. Schaad and Moffett [30] specified the access control policies under the RBAC96 and ARBAC97 models and a set of separation of duty constraints in alloy. They attempted to check the constraint violations caused by administrative operations. Toahchoodee *et al.* [35] demonstrated how the spatiotemporal aspects in RBAC model could be verified with alloy. Our approach also uses alloy to analyze the formal specifications of an RBAC model and constraints, which are then used for access-control-system development. In addition, the verified specifications are used to automatically derive the test cases for conformance testing. In [33], Sohr *et al.* demonstrated how the USE tool, a validation tool for OCL constraints, can be utilized to validate authorization constraints against RBAC configurations. However, the USE tool mainly focuses on the analysis of OCL constraints and has some limitations for specifying models and policies.

Very few studies addressed how access control mechanisms could be tested. Recently, mutation analysis was applied to security policy testing. Xie *et al.* [25] proposed a fault model for XACML policies. The mutation operators were introduced to implement the fault model. Masood *et al.* [26] used formal techniques to conceive a fault model and adopt mutation for RBAC models. Traon *et al.* [36] also used mutation analysis and defined security policy mutation operators in order to improve the security tests. Compared with those approaches, our approach adopts formal verification technologies to facilitate *automated* generation of test cases from the formal specification of security model and policy. In addition, our work demonstrates how these test cases can be used to check the compliance of security system design and implementation with the formal specification.

VI. Concluding Remarks

We have proposed a multilayered SDLC for authorization systems based on our AMF, which is designed for analysis and realization of security models and policies. In addition, we have demonstrated how our approach could fulfill the requirements from both software engineers and security experts for the analysis, design, implementation and testing of security properties in constructing real authorization systems.

In our multilayered SDLC, our toolset—RAE and RASS—constitutes a set of modules including a formal analysis tool such as alloy analyzer to facilitate the features of our methodology. As part of future work, we would examine how such a formal analysis can be integrated seamlessly with our toolset. In addition, we plan to investigate the relationship between the size of represented model and the time required for verification and test case generation in our framework. In addition, we would attempt to extend AMF for dealing with composite policies and more complicated system properties such as temporal and context attributes. In addition, we would study how our approach can be applied to some emerging application domains [5].

REFERENCES

[1] Dresden OCL toolkit. (2008). [Online]. Available: http://dresden-ocl.sourceforge.net

[2] The Octopus Project. (2007). [Online]. Available: http://www.klasse.nl/octopus 2010.

[3] American National Standards Institute, Inc. *Role Based Access Control*, ANSI-INCITS 359–2004, 2004.

[4] G.-J. Ahn and H. Hu, "Towards realizing a formal RBAC model in real systems," in *Proc. 12th ACM Symp. Access Control Models Technol*. New York: ACM, 2007, pp. 215–224.

[5] G.-J. Ahn, H. Hu, and J. Jin, "Security-enhanced OSGi service environments," *IEEE Trans. Syst. Man Cybern. C: Appl. Rev.*, vol. 39, no. 5, pp. 562–571, Sep. 2009.

[6] G.-J. Ahn and R. S. Sandhu, "Role-based authorization constraints specification," *ACM Trans. Inf. Syst. Secur. (TISSEC)*, vol. 3, no. 4, pp. 207–226, Nov. 2000.

[7] G.-J. Ahn and M. E. Shin, "Role-based authorization constraints specification using object constraint language," in *Proc. 10th IEEE Int. Workshops Enabling Technol.: Infrastructure Collaborative Enterprises*, 2001, pp. 157–162.

[8] K. Alghathbar and D. Wijesekera, "authUML: A three-phased framework to analyze access control specifications in use cases," in *Proc. 2003 ACM Workshop Formal Methods Security Eng*. New York: ACM, 2003, pp. 77–86.

[9] J. Bacon, K. Moody, and W. Yao, "A model of OASIS role-based access control and its support for active security," *ACM Trans. Inf. Syst. Secur. (TISSEC)*, vol. 5, no. 4, pp. 492–540, 2002.

[10] D. Basin, J. Doser, and T. Lodderstedt, "Model driven security: From UML models to access control infrastructures," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 15, no. 1, pp. 39–91, 2006.

[11] E. Bertino, L. Khan, R. Sandhu, and B. Thuraisingham, "Secure knowledge management: Confidentiality, trust, and privacy," *IEEE Trans. Syst., Man, Cybern. A: Syst. Humans*, vol. 36, no. 3, pp. 429–438, May 2006.

[12] R. Chandramouli, "Application of XML tools for enterprise-wide RBAC implementation tasks," in *Proc. 5th ACM Workshop Role-Based Access Control*, Berlin, Germany, Jul. 2000, pp. 11–18.

[13] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," in *Proc. Int. Workshop Policies Distrib. Syst. Netw.*, Bristol, U.K., 2001, pp. 18–38.

[14] T. Doan, S. Demurjian, T. Ting, and A. Ketterl, "MAC and UML for secure software design," in *Proc. 2004 ACM Workshop Formal Methods Security Eng*. New York: ACM, 2004, pp. 75–85.

[15] P. Epstein and R. Sandhu, "Towards a UML based approach to role engineering," in *Proc. 4th ACM Workshop Role-Based Access Control,* New York: ACM, 1999, pp. 135–143.

[16] D. F. Ferraiolo, R. S. Sandhu, S. I. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed NIST standard for role-based access control," *ACM Trans. Inf. Syst. Secur. (TISSEC)*, vol. 4, no. 3, pp. 224–274, 2001.

[17] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies," in *Proc. 27th Int. Conf. Softw. Eng.*, ACM, New York, 2005, pp. 196–205.

[18] J. Y. Halpern and V. Weissman, "Using first-order logic to reason about policies," in *Proc. 16th IEEE Comput. Security Found. Workshop (CSFW)*, IEEE Computer Society, 2003, pp. 187–201.

[19] H. Hu and G. Ahn, "Enabling verification and conformance testing for access control model," in *Proc. 13th ACM Symp. Access Control Models Technol.*, New York: ACM, 2008, pp. 195–204.

[20] D. Jackson, "Alloy: A lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, 2002.

[21] S. Jajodia, P. Samarati, and V. S. Subrahmanian, "A logical language for expressing authorizations," in *Proc. IEEE Symp. Security Privacy*, Oakland, CA, May 1997, pp. 31–42.

[22] J. Jürjens, "UMLsec: Extending UML for secure systems development," in *Proc. 5th Int. Conf. United Model. Lang.*, 2002, pp. 412–425.

[23] M. Koch, L. V. Mancini, and F. Parisi-Presicce, "A graph-based formalism for RBAC," *ACM Trans. Inf. Syst. Secur. (TISSEC)*, vol. 5, no. 3, pp. 332–365, 2002.

[24] T. Lodderstedt, D. Basin, and J. Doser, "SecureUML: A UML-based modeling language for model-driven security," in *Proc. 5th Int. Conf. Model Eng., Concepts, Tools*, Dresden, Germany, September 30–October 4, 2002, *Lecture Notes Comput. Sci.*, vol. 2460, pp. 426–441.

[25] E. Martin and T. Xie, "A fault model and mutation testing of access control policies," in *Proc. WWW 2007: Proc. 16th Int. Conf. World Wide Web,* New York: ACM, 2007, pp. 667–676.

[26] A. Masood, A. Ghafoor, and A. Mathur, "Scalable and effective test generation for access control systems that employ RBAC policies that employ RBAC policies," Purdue Univ., Washington, DC, Tech. Rep. SERC-TR-285, 2005.

[27] T. Moses. (2005). eXtensible Access Control Markup Language (XACML), version 2.0, Oasis Standard [Online]. Available: Internet http://docs. oasis-open.org/xacml/2.0/access control-xacml-2.0-core-spec-os.pdf.

[28] I. Ray, N. Li, R. France, and D.-K. Kim, "Using UML to visualize role-based access control constraints," in *Proc. 9th ACM Symp. Access Control Models Technol. (SACMAT)*, 2004, pp. 115–124.

[29] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, "Role-based access control models," *IEEE Comput.*, vol. 29, no. 2, pp. 38–47, Feb. 1996.

[30] A. Schaad and J. D. Moffett, "A lightweight approach to specification and analysis of role-based access control extensions," in *Proc. SACMAT: Proc. 7th ACM Symp. Access Control Models Technol.*, New York: ACM, 2002, pp. 13–22.

[31] B. Selic, "The pragmatics of model-driven development," *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, Sep./Oct. 2003.

[32] M. E. Shin and G.-J. Ahn, "UML-based representation of role-based access control," in *Proc. 9th IEEE Int. Workshops Enabling Technol.: Infrastructure Collaborative Enterprises*, 2000, pp. 195–200.

[33] K. Sohr, G.-J. Ahn, and L. Migge, "Articulating and enforcing authorisation policies with UML and OCL," in *Proc. 2005 Workshop Softw. Eng. Secure Syst.—Building Trustworthy Appl.*, 2005, pp. 1–7.

[34] J. Tidswell and T. Jaeger, "An access control model for simplifying constraint expression," in *Proc. 7th ACM Conf. Comput. Commun. Security*, Athens, Greece, Nov. 2000, pp. 154–163.

[35] M. Toahchoodee, I. Ray, K. Anastasakis, G. Georg, and B. Bordbar, "Ensuring spatio-temporal access control for real-world applications," in *Proc. 14th ACM Symp. Access Control Models Technol.*, New York: ACM, 2009, pp. 13–22.

[36] Y. Traon, T. Mouelhi, and B. Baudry, "Testing security policies: Going beyond functional testing," in *Proc. 18th IEEE Int. Symp. Softw. Reliability*, 2007, pp. 93–102.

[37] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*. Reading, MA: Addison-Wesley, 2003.

[38] H. Zhu and M. Zhou, "Roles in information systems: A survey," *IEEE Trans. Syst. Man Cybern. C: Appl. Rev.*, vol. 38, no. 3, pp. 377–396, May 2008.

Hongxin Hu (S'09) is currently working toward the Ph.D. degree from the School of Computing, Informatics, and Decision Systems Engineering, Ira A. Fulton School of Engineering, Arizona State University, Tempe.

He is also a member of the Security Engineering for Future Computing Laboratory, Arizona State University. His current research interests include access control models and mechanisms, network and distributed system security, secure software engineering, and security in social network and cloud computing.

Gail-Joon Ahn (SM'07) received the Ph.D. degree in information technology from George Mason University, Fairfax, VA, in 2000.

He was an Associate Professor at the College of Computing and Informatics, and the Founding Director of the Center for Digital Identity and Cyber Defense Research and Laboratory of Information Integration, Security, and Privacy, University of North Carolina, Charlotte. He is currently an Associate Professor in the School of Computing, Informatics, and Decision Systems Engineering, Ira A. Fulton School of Engineering and the Director of Security Engineering for Future Computing Laboratory, Arizona State University, Tempe. His research interests include information and systems security, vulnerability and risk management, access control, and security architecture for distributed systems, which has been supported by the U.S. National Science Foundation, National Security Agency, U.S. Department of Defense, U.S. Department of Energy, Bank of America, Hewlett Packard, Microsoft, and Robert Wood Johnson Foundation.

Dr. Ahn is a recipient of the U.S. Department of Energy CAREER Award and the Educator of the Year Award from the Federal Information Systems Security Educators Association.