Enabling Verification and Conformance Testing for Access Control Model

Hongxin Hu and Gail-Joon Ahn
The University of North Carolina at Charlotte
{hxhu,gahn}@uncc.edu

ABSTRACT

Verification and testing are the important step for software assurance. However, such crucial and yet challenging tasks have not been widely adopted in building access control systems. In this paper we propose a methodology to support automatic analysis and conformance testing for access control systems, integrating those features to Assurance Management Framework (AMF). Our methodology attempts to verify formal specifications of a role-based access control model and corresponding policies with selected security properties. Also, we systematically articulate testing cases from formal specifications and validate conformance to the system design and implementation using those cases. In addition, we demonstrate feasibility and effectiveness of our methodology using SAT and Alloy toolset.

Categories and Subject Descriptors

D.2.1 [**Requirements/Specifications**]: Language, Methodologies, Tools; D.2.4 [**Software/Program Verification**]: Model Checking, Validation; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Design, Security, Languages, Verification

Keywords

Access Control, Model-based Verification, Model-based Testing, SAT Solver, Alloy

1. INTRODUCTION

Security has become a necessary part of nearly most modern software and information systems. Software developers utilize models extensively, particularly in the early software development lifecycle to improve software quality. Unfortunately, security concerns are rarely considered as part of this process due to the lack of appropriate mechanisms and tools to help software developers analyze and capture security concerns in software design and development phases.

We have previously proposed the Assurance Management Framework (AMF) [3], which ensures formal security models to be fully realized in real systems through security model representation, security policy specification and validation, generation of security enforcement codes, and evaluation of generated codes under simulation. This framework can minimize the gap between security models and development of secure systems. Furthermore, the validation and simulation steps in the framework can provide certain assurance for the secure system design and implementation. In AMF, the formal security model and policy are basis for secure software development. The correctness of the design and implementation of the security model and policy are based on the premise that the formal security model and policy are valid. As a result, the formal specifications of security model and policy must undergo rigorous verification. This essential issue should be addressed in the framework. In addition, during the stage of system design and implementation, we validate the security model and relevant polices by producing a set of security configurations as system states and by checking these states against the security policies. Additionally, the generated security enforcement codes from system design are evaluated by the scenario-based simulation. However, the processes of generating system states for the validation and simulation in the framework need to be systematically verified and supported. In this paper, we attempt to address these important issues.

In order to identify the errors and flaws in the formal specifications of security model and policy, formal verification techniques should be employed to provide a higher assurance examining the correctness of adopted security model and policy. Furthermore, creating system states and test cases manually is tedious, time-consuming, and often not sufficient enough for proving the presence of errors in system design and development phases. Thus, automatic and effective testing techniques should be applied for ensuring the conformance to system design with respect to the formal specification.

Formal verification and conformance testing are two well-established techniques for validating software systems. In formal verification, a formal specification of a system is proved correct with a set of higher-level properties that the system should satisfy. In conformance testing [19], an actual implementation of the system is compared with those of its formal specification by means of interactions between

SACMAT'08, June 11–13, 2008, Estes Park, Colorado, USA.

the implementation and test cases, which are derived from the formal specification.

In this paper, we present a methodology composing automatic verification and conformance testing for secure system development. The objective of the verification is to ensure that the formal specifications of security model and policy is verified against a given set of security properties before being applied to system design and implementation. And the conformance testing is used to validate compliance of system design and implementation with the formal specification of security model and policy. In this step, test cases are derived automatically from the formal specification, which needs to be verified before. It is quite clear that the verification and conformance testing techniques play complementary roles for ensuring secure software development based on our AMF framework. In the context of access control model, we divide the verification into two thrusts such as functional property verification and constraint verification. Also, we introduce the concept of an authorization state space to assist tasks for identifying unique characteristics of constraints in access control model specification during a course of the constraint analysis process. Corresponding processes for the formal verification and automatic test generation are articulated as well. In addition, we demonstrate how our methodology can be applied for RBAC using SAT and Alloy toolset.

The main contributions of this paper are two-fold. First, we introduce an enhanced assurance management framework that facilitates rigorous analysis and testing for security model and policy via formal verification and automatic test generation, in addition to existing features of AMF including comprehensive realization of security model and policy in real systems through security model representation, security policy specification and generation of security enforcement code. Second, we propose a methodology composing model-based verification and model-based testing strategies for access control to ensure secure software development.

The rest of this paper is organized as follows. We address the enhanced AMF framework in Section 2. Section 3 discusses our integration approach for model-based verification and model-based testing for access control. In Section 4, we demonstrate our approach for RBAC verification and conformance testing using SAT and Alloy. Several related works are discussed in Section 5. Section 6 concludes the paper and elaborates the future directions.

2. ENHANCED ASSURANCE MANAGEMENT FRAMEWORK (AMF)

In this section, we present our enhanced assurance management framework, which is depicted in Figure 1. The framework is designed for facilitating the secure software development. In the modeling stage, formal specifications of security model and policy are verified. Then, application-oriented security model representation and application-oriented 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 also are evaluated by using the generated test suites in the simulation. We divide all tasks in the AMF framework into two categories as follows:

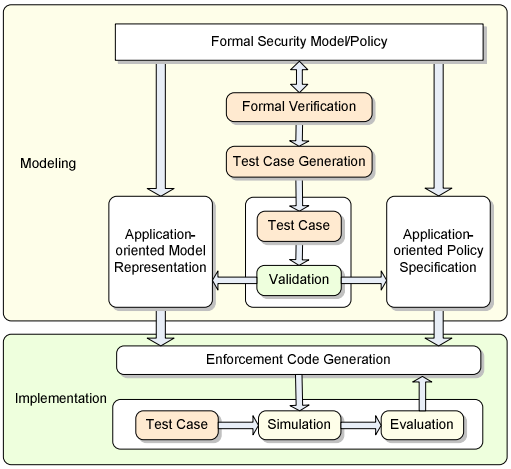

Figure 1: Assurance management framework.

1. Automatic realization of security model and policy
   - *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.
   - *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 specifying security policies. 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.
   - *Automatic generation of security enforcement codes.* It is also a crucial aspect to make the 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 policy by well-known software engineering mechanisms, such as the Model Driven Development (MDD). 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
   - *Automatic analysis of formal security model and policy.* One of the promising advantages in mathematical and logic based techniques for security

model and policy is that formal reasoning of the security properties can be achieved. Since the formal security model and policy serve as a basis for secure system development in AMF, obviously the formal specifications of model and policy should be proved correct against expected security properties. Formal verification offers a rich toolbox containing a variety of techniques such as model checking [6], SAT solving [15] and theorem proving [16], for supporting automatic system analysis.

- *Automatic test case generation from formal specification.* While formal verification can prove property violation or satisfaction, it is usually not sufficient in practice. The proof only shows that a given formal specification fulfills a property. However, the actual implementation is also influenced by other facts, such as platform, transformation approach, compiler, and so on. Consequently, software testing is necessary. The most significant recent development in testing is the application of verification technologies to drive the testing process to the generation of test cases from the formal specification. Thus, it is an attractive way to seek automated derivation of test cases from the formal security model and policy. As a result, the generated test cases can be fed into validator and simulator to check whether the system design and development comply with the formal specification.

3. INTEGRATION APPROACH

This section introduces our methodology composing formal analysis and conformance testing for building access control systems. As demonstrated in Figure 2, the formal access control model and policy are the core of the entire processes for serving the following tasks: (1) formal verification, (2) system design, and (3) test generation. Correspondingly, three high-level access control models–such as verification-based model, application-oriented model, and test-based model–are constructed based on the formal model. Our previous work [3] has addressed how a formal access control model and associated policies can be fully translated to application-oriented model representation and policy specification, which then generate enforcement codes. Here, we focus on two major enhancements: formal verification and conformance testing. In our approach, both model-based verification and model-based testing are supported by the same formal verification technology for the purposes of automatic verification and test case generation. A notable advantage of using model-based approach is to reduce the complexity of analysis, thus minimizing the state explosion problem.

In order to articulate our methodology clearly, we first define access control model specification as follows:

Definition 1 (Access Control Model Specification). *An access control model specification S is defined as S = (M, F, C), where:*

– M is an access control model representation, which defines sets of basic access control entities and relations.

– F is a set of access control function specifications, which specifies the features required by an access control system.

– C is a set of access control constraint specifications, which defines higher-level organizational policies.

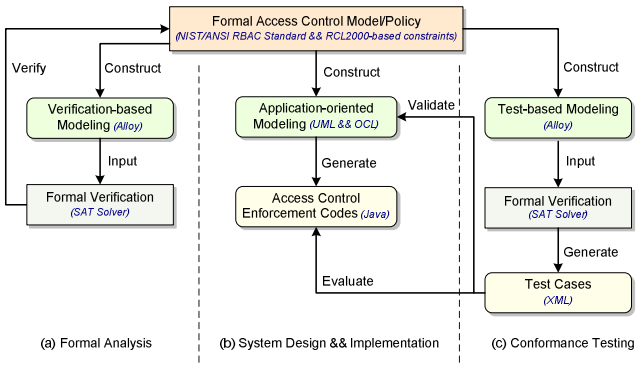

Figure 2: Integration approach.

Note that the access control model specification S can be decomposed to different sub-specifications for supporting our model-based verification and model-based testing approaches.

3.1 Model-based Verification for Access Control

We take into account the following verification problem for the access control model: given an access control model specification S and an access control model property P, does S satisfy P? We consider two kinds of property: access control functional property P_f and access control authorization property P_a.

Definition 2 (Access Control Functional Property). *An access control functional property P_f describes an expected operating result when performing an access control function in an access control system.*

Definition 3 (Access Control Authorization Property). *An access control authorization property P_a describes an authorization state that is achieved by an access control system.*

Therefore, the verification of an access control model is separated into two steps: access control function verification and access control constraint verification.

3.1.1 Function Verification

Definition 4 (Access Control Function Verification). *For an access control model specification $S_f = (M, f)$ and an access control functional property P_f, proving whether (M, f) satisfies P_f, denoted by $(M, f) \vDash P_f$, is called access control function verification.*

That is, if we can determine that (M, f) satisfies P_f, it means the access control functional property P_f is held on the access control model specification S_f. Hence, we can make sure the functional components in a formal model specification S_f are correct with respect to expected properties.

Figure 3 illustrates a reasoning process for formal verifi-

cation. The access control model specification S_f and the functional property P_f are encoded, and then fed into a formal verifier. The verifier in turn checks whether the functional property is violated or not. If a functional property violation is encountered, it means the access control model specification does not conform to the functional property, leading the refinement of model specification.

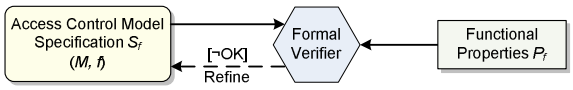

Figure 3: Function verification.

3.1.2 Constraint Verification

A critical task for specifying constraints is to determine whether a set of constraint expressions really reflects the desired authorization requirements properly. Normally, constraints prohibit [1] 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.

A concept of *authorization state space* is introduced to identify under- and over-constraints in an access control model specification. An authorization state space represents an entire space that an access control system probably covers. In other words, all possible system states of an access control system consist of an authorization state space. Regarding access control requirements, an authorization state space can be divided into two subspaces: (1) the *desired* authorization state subspace S_d, which contains authorization states that should be allowed to occur in an access control system according to the authorization requirements. (2) the *undesired* authorization state subspace S_u, which contains authorization states that should be prohibited to appear in an access control system against the authorization requirements. On one hand, we are able to specify the desired authorization state subspace with the expected authorization properties P_{a^+} and the undesired authorization state subspace with the unexpected authorization properties P_{a^-}, respectively. On the other hand, from the perspective of access control specification, an authorization state space can be divided into permitted authorization state subspace S_p and prohibited/constrained authorization state subspace S_c. The most ideal view of an authorization state space is that the desired authorization state subspace is contained by the permitted authorization state subspace, and the undesired authorization state subspace is included in the prohibited authorization state subspace. It means the specified constraints meet the authorization requirements accordingly.

[1]Constraints can also be used to enforce obligation. We will not cover this aspect in this paper. However, we believe the approach introduced in this paper is applicable to analyze obligation constraints as well.

Unfortunately, the ideal view is far from the reality. Two situations may exist in practice.

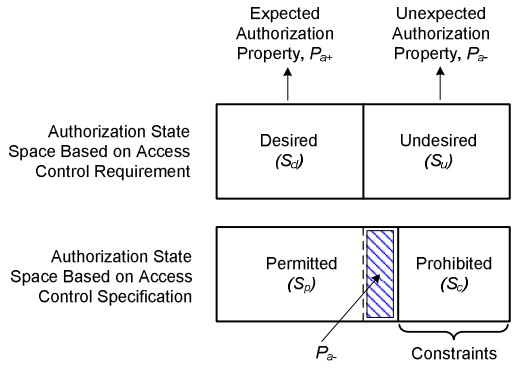

Figure 4: Identifying under-constraint.

Figure 4 depicts one case, which demonstrates that the permitted authorization state subspace S_p covers *partial* undesired authorization state subspace S_u due to the reason of *under-constraint*.

When the prohibited authorization state subspace S_c is a subset of the undesired authorization state subspace S_u, and there is an overlap between the permitted authorization state subspace S_p and the undesired authorization state subspace S_u, *under-constraint* occurs in the constraint specifications.

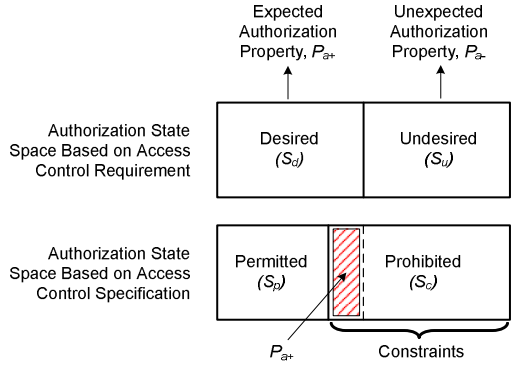

Figure 5: Identifying over-constraint.

Another case is shown in Figure 5. *Over-constraint* is presented in this case, where the permitted authorization state subspace S_p is covered by the desired authorization state subspace S_d, and the prohibited authorization state subspace S_c contains *partial* desired authorization state subspace S_d.

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 general definition for access control constraint verification is given as follows:

Definition 5 (Access Control Constraint Verification). *For an access control model specification $S_c = (M, F, c)$ and an access control authorization property P_a, proving whether (M, F, c) satisfies P_a, denoted by $(M, F, c) \vDash P_a$, is called access control constraint verification.*

In order to identify *under-constraint*, the unexpected authorization property P_{a^-} is used to replace P_a, and an expression that addresses the analysis for *under-constraint* can be defined as follows: $(M, F, c) \models P_{a^-} \Rightarrow C \downarrow$, where C↓ denotes *under-constraint*.

As demonstrated on the bottom part of the Figure 4, if an unexpected authorization property P_{a^-}, which represents the authorization subspace $S_p \cap S_u$, is satisfied by the access control model specification S_c, *under-constraint* is detected. Figure 6 depicts the process of constraint verification for checking *under-constraint* based on the above definition. If the verifier proves an unexpected authorization property P_{a^-} is held on the access control model specification S_c, we can conclude that the given constraint specifications are too weak, and should be strengthened to exclude undesired authorization properties.

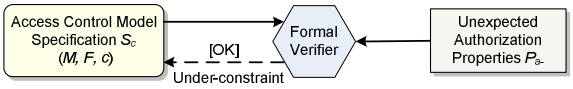

Figure 6: Constraint verification: under-constraint.

The expected authorization property P_{a^+} that addresses the authorization subspace $S_c \cap S_d$ is utilized to substitute P_a for identifying *over-constraint* as summarized in the following expression: $(M, F, c) \not\models P_{a^+} \Rightarrow C \uparrow$, where C↑ denotes *over-constraint*.

The bottom part of the Figure 5 depicts the *over-constraint* situation. Based on the above expression, we introduce a process for identifying *over-constraint* as shown in Figure 7. If the verifier checks the expected authorization property P_{a^+} is not satisfied by the access control model specification S_c, this points out the defined constraints are too strong. Thus, the constraint definitions should be refined by reducing the restriction of constraints.

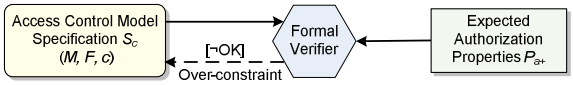

Figure 7: Constraint verification: over-constraint.

3.2 Model-based Testing for Access Control

Model-based testing is a software testing technology in which the models defined in software construction are used to drive the testing process. Numerous formal verification techniques have been used for model-based testing [20]. The idea of automated test generation from the formal verification is that counterexamples may be generated to illustrate a property violation by the formal verification, and counterexamples are interpreted as test cases. Our approach intends to use a formal specification of access control model and policy for automated derivation of test cases.

Two kinds of test case are generated for testing a constraint. One is called *negative test case*, denoted as T^-, which is considered as an undesired access control authorization state that should be denied by the constraint in the access control system. Another test case is named *positive test case*, denoted as T^+. This test case represents a desired access control authorization state and should be allowed to appear in the access control system.

The following expression specifies the generation of *negative test case* based on the satisfiability verification: $(M, F) \not\models c \Rightarrow T^-$. *Negative test case* T^- can be derived from a formal specification, in which an access control model specification $S_m = (M, F)$ does not satisfy the constraint specification c. A process is demonstrated in Figure 8. Since the constraint specification c is taken out from the access control model specification S_m, 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.

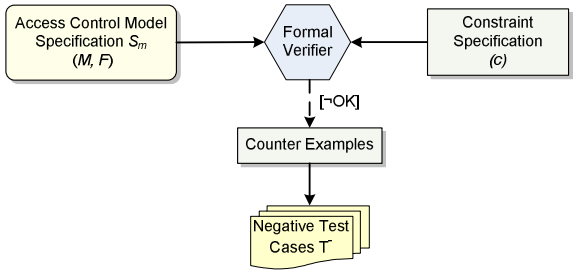

Figure 8: Generating negative test cases for constraints.

Positive test case T^+ is generated from a formal specification, as we draw the constraint specification c from the access control model specification $S_m = (M, F)$, and take the negated constraint specification $\neg c$ as the authorization property to verify the access control model specification S_m. Counterexamples are derived and utilized to build positive test cases. The following expression summarizes this characteristic: $(M, F) \not\models \neg c \Rightarrow T^+$. Corresponding process is shown in Figure 9.

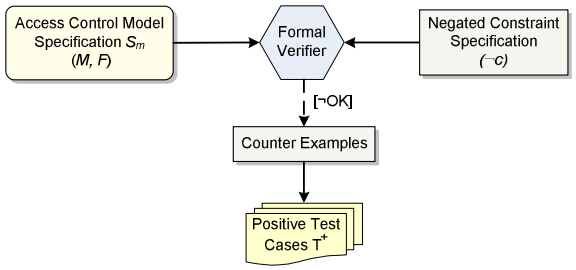

Figure 9: Generating positive test cases for constraints.

4. CASE STUDY: VERIFICATION AND CONFORMANCE TESTING FOR RBAC

In this section, we utilize SAT solving as an underlying formal verification technique to demonstrate automatic analysis and test generation for the formal specification of a RBAC model and associated constraints based on the approaches and definitions introduced in Section 3. We adopt the NIST/ANSI standard for RBAC [2] and a formal constraint specification language, Role-based Constraints Lan-

guage 2000 (RCL2000) [4]. Alloy is used as an intermediate language into which the RBAC model is constructed and the RCL2000-based constraints are translated. Then, using Alloy tool called Alloy Analyzer, which uses a SAT solver that supports enumeration, the RBAC model and corresponding constraints are analyzed, and test cases are generated from the RBAC model specification.

4.1 Alloy Overview

Alloy [9] is a structural modeling language based on first-order logic, and designed for the specification of object models through graphical and textual structure. An Alloy model is a structured specification composed with the following components: `Signature`, `Fact`, `Function`, `Predicate` and `Assertion`. The Alloy Analyzer [10] is an automated constraint solver for analyzing (verifying and validating) models written in Alloy. Alloy Analyzer provides two kinds of automatic analysis–*simulation* in which the consistency of a fact or predicate is demonstrated by generating a snapshot of the model; and *checking* in which a consequence of the specification is tested by attempting to generate a counterexample for an assertion. The former is useful for demonstrating the feasibility of a specification, where conflicting constraints could be detected, while the latter is for validating the correctness of a certain property in the system, where the assertion could be proved based on the facts defined in the model and within a finite scope of instances.

4.2 RBAC Model Representation

The NIST/ANSI standard for RBAC gives a RBAC reference model, which defines sets of basic RBAC elements and relations, including a set of roles, a set of users, a set of permissions, relationships between users, roles, and permissions. We define a primary representation of the NIST/ANSI RBAC model in Alloy as follows:

```
module RBAC
sig User {}
sig Role {}
sig Operation {}
sig Object {}
sig Permission {Operation, Object}
sig Session {}
sig URA {
 ura:  User->Role}
sig PRA {
 pra:  Permission->Role}
sig US {
 us:  User!->Session}
sig SR {
 sr:  Session->Role}
sig PB {
 pb:  Operation->Object}
```

The above defines the core element sets and relations in a RBAC model. A role hierarchy relation supporting hierarchical RBAC is defined as follows:

```
sig RRA {
 hierarchy:  Role->Role}
```

In order to specify static separation of duty (SSoD) relations and dynamic separation of duty (DSoD) relations in the context of conflicting roles, which are addressed in the NIST/ANSI RBAC model, we give the following Alloy definitions [2]:

```
sig SCR {
 conflict_role:  set Role,
 cardinality:  Int}
sig DCR {
 conflict_role:  set Role,
 cardinality:  Int}
```

4.3 RBAC Constraint Specification

Policy designers can employ RCL2000 to specify complex authorization policies to meet high-level security requirements along with the NIST/ANSI RBAC standard. In order to reason about RCL2000 policy specifications using Alloy tool, we need to translate RCL2000 policy expressions to Alloy statements. RCL2000 supports six RBAC system functions `user`, `roles`, `sessions`, `permissions`, `operations` and `object`. These function expressions are represented in Alloy. For example, `roles(u)`, which returns all the roles assigned to the user `u`, is converted to `u.(URA.ura)`. In RCL2000, $\texttt{roles}^*$ and $\texttt{permissions}^*$ are defined as a variant of `roles` and `permissions` to support role hierarchy. For example, $\texttt{roles}^*(\texttt{u})$ returns a set of roles for which a given user is authorized. Such functions are able to converted to Alloy using "*", which denotes a reflexive transitive closure operator, and "$\sim$", which denotes transpose operator. Each term in RCL2000 is converted to corresponding Alloy operator. The detailed translation algorithm is described in Figure 10.

Next, we illustrate two typical RBAC constraints specified in RCL2000, and give an equivalent Alloy expressions generated by our translation algorithm.

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

RCL2000 Expression:

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

Translated Alloy Expression:

```
all u:User | all scr:SCR |
  #((u.(URA.ura).~*(RRA.hierarchy)) &
   scr.conflict_role) <= scr.cardinality
```

Table 1 explains the mapping from the RCL2000 expression to the Alloy expression for this constraint. All components in the RCL2000-based constraint expression can be mapped to corresponding Alloy components precisely.

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 exceeds the cardinality number of the conflicting role set.

RCL2000 Expression:

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

Translated Alloy Expression:

```
all u:User | all dcr:DCR |
  #(u.(US.us).(SR.sr).~*(RRA.hierarchy) &
   dcr.conflict_role) <= dcr.cardinality
```

[2]The separation of duty relations in the NIST/ANSI RBAC model can be extended to support conflicting permissions and conflicting users, using several definitions such as {SCP, DCP} and {SCU, DCU}, respectively.

Table 1: Mapping RCL2000 expression to Alloy expression for SSoD-CR constraint

RCL2000	Alloy	Meaning
OE(SCR)	`all scr: SCR | scr`	a collection which is a pairs of a conflicting role set and a cardinality for the conflicting role set
OE(U)	`all u:User | u`	a single user
roles*(OE(U))	`u.(URA.ura).~*(RRA.hierarchy)`	return all roles that are authorized to a single user considering role hierarchy
GS(OE(SCR)	`scr.conflict_role`	return a conflicting role set
GC(OE(SCR)	`scr.SetCardinality`	return the cardinality of a conflicting role set
$\cap$	$\&$	return the intersection of two sets
$\mid set \mid$	$\#set$	return the cardinality number of a set

Input: *RCL2000* expression; **Output**: *Alloy* expression

Let **Simple-OE** term be either **OE(*set*)** , or **OE(function(*element*))**, where ***set*** is an element of **{U, R, OP, OBJ, P, S, SCR, DCR, SCU, DCU, SCP, DCP, scr, dcr, scu, dcu, scp, dcp}** and ***function*** is an element of **{user, roles, roles*, sessions, permissions, permissions*, operations, object}**

1. AO elimination
Replace all occurrences of *AO(expr)* with *(expr-{OE(expr)})*;
2. OE elimination
While There exists **Simple-OE** term in RCL2000 expression
case (i) **Simple-OE** term is *OE(set)*
choose *set* term;
call set_reduction procedure;
case (ii) **Simple-OE** term is *OE(function(element))*
choose *function(element)* term;
call reg_function_reduction procedure;
End
Procedure set_reduction
case (1) *set* is *U*
put *all u:User* | to right of existing quantifier(s);
replace all occurrences of *OE(U)* with *u*;
case (2) *set* is *R*
put *all r:Role* | to right of existing quantifier(s);
replace all occurrences of *OE(R)* with *r*;
……[a]
End
Procedure reg_funtion_reduction
case (1) *function(element)* term is *user(r)*
put *all u1:User= r.~(URA.ura)* | to right of existing quantifier(s);
replace all occurrences of *OE(user(r))* with *u1*;
case (2) *function(element)* term is *roles(u)*
put *all r1:Role= u.(URA.ura)* | to right of existing quantifier(s);
replace all occurrences of *OE(roles(u))* with *r1*;
……
End
3. System Functions elimination
While There exists *function(element)* term in RCL2000 expression
choose *function(element)* term;
call sys_function_reduction procedure;
End
Procedure sys_function_reduction
case (1) *function(element)* term is *user(r)*
replace all occurrences of *user(r)* with *r.~(URA.ura)*;
case (2) *function(element)* term is *roles(u)*
replace all occurrences of *roles(u)* with *u.(URA.ura)*;
……
End
4. Operators elimination
replace all occurrences of *set1* $\in$ *set2* with *set2 in set1*;
replace all occurrences of *set1* $\cap$ *set2* with *set1 & set2*;
replace all occurrences of *set1* $\cup$ *set2* with *set1* + *set2*;
replace all occurrences of $\mid set \mid$ with *# set*;
replace all occurrences of $\Rightarrow$ with =>;
replace all occurrences of $\wedge$ with &&;
replace all occurrences of $\neq$ with !=;
replace all occurrences of $\leq$ with <=;
replace all occurrences of $\geq$ with >=;
replace all occurrences of $\emptyset$ with *none*;

[a] Due to the page limit, we omitted the detailed procedures for OE elimination and system function elimination.

Figure 10: Translation algorithm.

4.4 RBAC Function Verification

The functional specification in the NIST/ANSI standard for RBAC defines various functions that role-based systems should provide. These functionalities are described in the standard using a set-based specification language, *Z*. Prior to applying these functional definitions for role-based system development, the correctness of these definitions should be checked rigorously. Formal verification is necessary for this objective.

In this subsection, we employ `DeleteRole` function as an example to demonstrate how the formal verification can assist in finding mistakes in the functional specifications. In hierarchical RBAC, the following functional properties should be achieved by the `DeleteRole` function.

1. The existing role is removed from the *Role* date set.
2. Any use-to-role assignment relation established by the role is removed.
3. Any permission-to-role assignment relation established by the role is removed.
4. Any role hierarchy relationship established by the role is removed.

The following is the functional definition for `DeleteRole` supporting hierarchical RBAC in the NIST/ANSI RBAC standard.

$DeleteRole(role{:}NAME) \lhd$
$role \in ROLES$
$UA' = UA \setminus \{u{:}Users \bullet u \mapsto role\}$
$assigned\_users' = assigned\_user \setminus \{role \mapsto assigned\_user(role)\}$
$PA' = PA \setminus \{op{:}OPS, obj{:}OBJS \bullet (op,obj) \mapsto role\}$
$assigned\_permissions' = assigned\_permissions \setminus \{role \mapsto assigned\_permissions(role)\}$
$ROLES' = ROLE \setminus \{role\} \rhd$

An Alloy function is constructed based on the above definition as follows:

```
fun DeleteRole(r:Role){
  r in Role =>
   all p:Permission |
   all u:User| (u->r) in URA.ura =>
    URA.ura = (URA.ura - (u->r))) &&
   (all p:Permission | (p->r) in PRA.pra =>
    PRA.pra = (PRA.pra - (p->r))) &&
   (Role = Role - r) }
```

We can also define an Alloy assertion to describe the RBAC functional properties P_f discussed earlier. Corresponding functional properties for `DeleteRole` operation with the notion of hierarchical RBAC are defined as follows:

```
assert Check_DeleteRole {
  all r:Role| all r':Role | all u:User |
  all p:Permission |
   DeleteRole(r) &&
   //The role is removed form the role set
   r !in Role &&
   //Corresponding UA relations are removed
   (u->r) !in URA.ura &&
   //Corresponding PA relations are removed
   (p->r) !in PRA.pra &&
```

```
    //Corresponding inheritance relations are removed
    (r->r') !in RRA.hierarchy &&
    (r'->r) !in RRA.hierarchy }
check Check_DeleteRole
```

By running Alloy Analyzer, we can validate this assertion against the RBAC model specification, which contains the `DeleteRole` function specification. The Alloy Analyzer will detect counterexamples, which identify violations of the assertion with respect to the function specification. After careful inspection, we found that the functional definition of `DeleteRole` for hierarchical RBAC in the NIST/ANSI RBAC standard misses a step for removing inheritance relations established by the role that is being deleted.

In [12], another formal definition of `DeleteRole` function for hierarchical RBAC is given. Using the same approach, we identified that the steps for removing UA relations and PA relations are missed in their specification.

4.5 RBAC Constraint Verification

In this subsection, we demonstrate how to identify under- and over-constraints with Alloy using the aforementioned approach in Section 3.

4.5.1 Identifying Under-constraint

Regarding separation of duty principles, the following authorization property considering the role hierarchy is unexpected:

- *Two conflicting roles are authorized to the same user.*

We specify this unexpected authorization property P_{a^-} in Alloy as follows:

```
  pred Check_SSoD[ disj r1,r2:Role, u:User, scr:SCR]
{
    //r1 and r2 are mutually exclusive
    r1 in scr.conflict_role &&
    r2 in scr.conflict_role &&
    scr.cardinality = 1 &&
    //r1 and r2 are authorized to the same user, u
    r1 in u.(URA.ura).~*(RRA.hierarchy) &&
    r2 in u.(URA.ura).~*(RRA.hierarchy) }
  run Check_SSoD
```

Suppose the policy designer only defines a simple *SSoD constraint*, which ignores the role hierarchy relation. We can translate the RCL2000 expression for the simple *SSoD constraint* to the Alloy expression, and put it into an Alloy fact as an Alloy constraint as follows:

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

When running the predicate `Check_SSoD` defined above, instances–in which conflicting roles are *indirectly* assigned to a user–are found by Alloy Analyzer. It means the unexpected authorization property is held by the constraint specification. In addition, we can conclude the constraint is too week with respect to the authorization property.

4.5.2 Identifying Over-constraint

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

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

We specify this expected authorization property P_{a^+} in Alloy as follows:

```
assert Check_DSoD {
  all u:User | all disj r1,r2:Role |
  all disj s1,s2:Session | all dcr:  DCR |
   //r1 and r2 are dynamic conflicting roles
   r1 in dcr.conflict_role &&
   r2 in dcr.conflict_role &&
   dcr.cardinality = 1 &&
   //u creates s1, s2
   (u->s1) in US.us &&
   (u->s2) in US.us &&
   //r1 and r2 cannot be activated in the
   //same session, but can be activated
   //in the different session
   (r1->s1) in ~SR.sr &&
   (r2->s1) !in ~SR.sr &&
   (r2->s2) in ~SR.sr }
check Check_DSoD
```

Assume the policy designer defines an *User-based DSoD* constraint [3] as we demonstrated before. We define an Alloy fact, which contains this constraint specification.

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

Running "`check Check_DSoD`" in Alloy Analyzer, counterexamples are found. It indicates 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. If we replace the *User-based DSoD* constraint with the *Session-based DSoD* constraint, the expected authorization property defined in assertion `Check_DSoD` is held.

4.6 Test Case Generation

As mentioned earlier, *negative test cases* T^- are derived from a formal access control model specification, in which the constraint specification is drawn out and serves as an authorization property for the formal verification, while *positive test cases* T^+ are generated from a formal specification, if we take the constraint specification out of the access control model specification, and consider the negated constraint specification as an authorization property.

We take the simple *SSoD constraint* as an example to demonstrate the process of automated test generation. The following assertion is defined to drive the negative test cases for the constraint specification (c).

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

Checking this assertion against the RBAC model specifi-

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

cation, in which *SSoD constraint* has been taken out, counterexamples are generated. These counterexamples are used to construct negative test cases as undesired system states to test the conformance of the *SSoD constraint* in both access control system design and implementation.

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

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

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. Through running this assertion, counterexamples are also generated. Then, positive test cases serving as desired system states are constructed from these counterexamples.

In order to generate more meaningful test cases for real application domains, Alloy signatures need to reflect all RBAC configuration components of the targeted application domain for producing specialized instances of the defined Alloy module. Then, running the constraint assertion with the scope enables Alloy to generate test cases. Suppose we have a banking system with a user `Bob` and two conflicting roles, `customerServiceRep` and `loanOfficer`. We first need to define the appropriate assignment of user, role and conflicting role set as follows. This Alloy definition is then provided to Alloy Analyzer so that it can run `SSoD` assertion defined earlier with the scope of *one* user and *two* roles. Finally, Alloy Analyzer can generate a negative test case for our conformance testing, such that the user `Bob` is assigned to two conflicting roles, `customerServiceRep` and `loanOfficer`.

```
one sig Bob extends User{}
one sig customerServiceRep,loanOfficer extends Role{}
fact SCR_rules {
  customerServiceRep in SCR.conflict_role &&
  loanOfficer in SCR.conflict_role }
```

4.7 Tool Support

In this section, we give a brief introduction to our toolset, which constitutes a toolchain with the Alloy Analyzer to facilitate the application of our methodology for automatic analysis, realization and conformance testing of RBAC model and constraints.

We developed RAE based on ArgoUML [1]. RAE tool is composed of three major functional components: specification component, validation component, and code generation component. Specification component in RAE is responsible for specifying RBAC model and constraints. In this component, UML class diagrams are utilized to represent RBAC model; UML object diagrams are used to represent snapshots of RBAC model at particular points; and an editing environment for constraints is provided to easily specify authorization constraints using RCL2000 and OCL. Validation component in RAE is in charge of violation checking so as to validate RBAC model and constraints through constructing

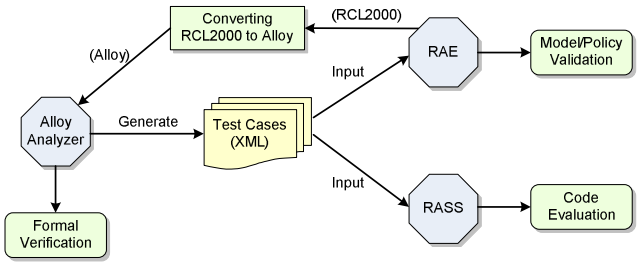

Figure 11: Toolchain supporting our approach.

a set of system states and check such states against authorization constraints. Code generation component in RAE is used to generate java codes automatically for RBAC model and constraints. The generated enforcement codes by RAE are utilized by developers to integrate into a real application system requiring RBAC features. In order to verify the conformance and correctness of the generated RBAC enforcement codes, we used a testbed, RASS, as a simulation environment. In RASS, RBAC function and constraint implementations are verified by running extensive test cases. We designed a web-based user interface and a storage layer to incorporate the generated RBAC enforcement modules. The web-based user interface provides the function of interaction between the users and system functions, which are provided by generated codes, and the storage layer stores the RBAC configuration.

A toolchain depicted in Figure 11 consists of three tools: Alloy Analyzer, RAE and RASS. We have enhanced the constraint editor in RAE to support the transformation from RCL2000 to Alloy. Thus, the policy designers are able to specify RBAC constraints with RCL2000, and then convert RCL2000-based constraints to Alloy expressions in RAE. The generated Alloy specifications for constraints can be forwarded to Alloy Analyzer. The formal specifications of a RBAC model are also constructed to Alloy, then analyzed by Alloy Analyzer as well. In addition, Alloy Analyzer allows to generate all nonisomorphic instances from an Alloy specification. These instances are then used as test cases, which are fed into RAE to construct system states. Importantly, such cases are checked against constraints to validate the RBAC model and constraint specifications in the stage of system design as well as utilized by RASS to evaluate the generated RBAC codes under simulation.

5. RELATED WORK

One important aspect of policy analysis is to formally check general properties of access control policies, such as inconsistency and incompleteness [5, 7, 8]. Schaad and Moffett [17] specified the access control policies under the RBAC96 model, the policy governing access to the access control policy under the ARBAC97 model, and a set of separation of duty constraints in Alloy. They attempted to check the constraint violations that may arise by administrative operations. Our approach also uses Alloy to analyze the formal specifications of a 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. Jaeger et al. [11] presented the concept of an access control space

and showed how it could be utilized to manage access control policies. In our work, a similar concept of an authorization state space is defined to help analyze constraints. Under- and over-constraints for the constraint specifications are identified based on the authorization state space analysis, which can conduct the formal constraint verification.

Very few works study how to test access control mechanisms. Recently, mutation analysis was applied to security policy testing. Xie et al. [13] proposed a fault model for XACML policies. The mutation operators were introduce to implement the fault model. Masood et al. [14] used formal techniques to conceive a fault model and adapt mutation to RBAC models. Traon et al. [18] also used mutation analysis and defined security policy mutation operators in order to improve the security tests. Comparing with these works, 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.

6. CONCLUSION AND FUTURE WORK

In this paper, we have proposed an approach integrating formal verification and conformance testing for access control model in AMF. We presented our model-based verification and model-based testing approaches, in which the formal specification of access control model and policy is verified with respect to selected security properties before being applied to secure system design and implementation. Also, we adopted the formal specification of NIST/ANSI standard RBAC model and demonstrated how test cases could be derived from formal specification, that are used to validate the secure system design and implementation conformance to formal specification by means of SAT and Alloy toolset. As part of future works, the verification and testing for a composition of policies will be studied in depth. Regarding more complicated secure system, we plan to investigate the relation between model size and the time required for verification and test case generation.

7. ACKNOWLEDGMENTS

This work was partially supported by the grants from National Science Foundation (NSF-IIS-0242393 and NSF-DUE-0416042), Department of Energy Early Career Principal Investigator Award (DE-FG02-03ER25565).

8. REFERENCES

[1] The ArgoUML Project. http://argouml.tigris.org.

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

[3] G.-J. Ahn and H. Hu. Towards realizing a formal RBAC model in real systems. In *SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 215–224, New York, NY, USA, 2007. ACM.

[4] G.-J. Ahn and R. S. Sandhu. Role-based authorization constraints specification. *ACM Trans. Inf. Syst. Secur. (TISSEC)*, 3(4):207–226, November 2000.

[5] A. K. Bandara, E. C. Lupu, and A. Russo. Using event calculus to formalise policy specification and analysis. In *POLICY '03: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, page 26, Washington, DC, USA, 2003. IEEE Computer Society.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[7] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 196–205, New York, NY, USA, 2005. ACM.

[8] J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. In *16th IEEE Computer Security Foundations Workshop (CSFW'03)*, pages 187–201. IEEE Computer Society, 2003.

[9] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[10] D. Jackson, I. Schechter, and H. Shlyahter. Alcoa: the alloy constraint analyzer. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 730–733, New York, NY, USA, 2000. ACM.

[11] T. Jaeger, X. Zhang, and A. Edwards. Policy management using access control spaces. *ACM Trans. Inf. Syst. Secur.*, 6(3):327–364, 2003.

[12] N. Li, J.-W. Byun, and E. Bertino. A critique of the ANSI standard on role based access control. *Technical Report TR 2005-29, Purdue University*, 2005.

[13] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 667–676, New York, NY, USA, 2007. ACM.

[14] A. Masood, A. Ghafoor, and A. Mathur. Scalable and effective test generation for access control systems that employ RBAC policies that employ RBAC policies. *SERC-TR-285, Purdue University*, 2005.

[15] D. G. Mitchell. *A SAT Solver Primer*. EATCS Bulletin (The Logic in Computer Science Column), Volume 85, February 2005, pages 112-133.

[16] A. Robinson and A.Voronkov. *Handbook of Automated Reasoning*. MIT Press, 2001.

[17] A. Schaad and J. D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 13–22, New York, NY, USA, 2002. ACM.

[18] Y. L. Traon, T. Mouelhi, and B. Baudry. Testing security policies: Going beyond functional testing. In *ISSRE '07. The 18th IEEE International Symposium on Software Reliability*.

[19] J. Tretmans. A formal approach to conformance testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI*, pages 257–276, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.

[20] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.