

A Rigorous Approach to Uncovering Security Policy Violations in UML Designs

Lijun Yu, Robert France, Indrakshi Ray, Sudipto Ghosh
Colorado State University, USA
{lijun,france,iray,ghosh}@cs.colostate.edu

Abstract

There is a need for rigorous analysis techniques that developers can use to uncover security policy violations in their UML designs. There are a few UML analysis tools that can be used for this purpose, but they either rely on theorem-proving mechanisms that require sophisticated mathematical skill to use effectively, or they are based on model-checking techniques that require a “closed-world” view of the system (i.e., a system in which there are no inputs from external sources). In this paper we show how a lightweight, scenario-based UML design analysis approach we developed can be used to rigorously analyze a UML design to uncover security policy violations.

In the method, a UML design class model, in which security policies and operation specifications are expressed in the Object Constraint Language (OCL), is analyzed against a set of scenarios describing behaviors that adhere to and that violate security policies. The method includes a technique for generating scenarios. We illustrate how the method can be applied through an example involving role-based access control policies.

1. Introduction

Designers of software systems that are required to enforce or adhere to security policies must be able to analyze their designs to uncover policy violations. To support rigorous analysis, designs must be expressed in analyzable forms. Formal modeling languages such as Z [19] and Alloy [14] can be used to create analyzable designs, but these languages are not widely used in industry. The Unified Modeling Language (UML) [1] is an industrially popular standard object-oriented modeling language that includes a language, called the Object Constraint Language (OCL) [2], for expressing operation specifications and constraints on system states. Currently, there are very few tools for rigorously analyzing UML designs. Some tools rely on

sophisticated theorem-proving capabilities (e.g., see [24]), while others utilize model-checkers that require a closed-world view of the software, in which inputs from external sources (e.g., operation parameters) are not allowed (e.g., see [23]).

In this paper, we present a lightweight method for rigorously analyzing a UML design model. The method is lightweight, in that it does not guarantee that all possible violations will be found. It is rigorous, in that it will uncover all violations within the scope of a set of scenarios. Here, a scenario describes a sequence of state transitions, where a transition is the result of a completed operation execution. The method uses a scenario-based analysis technique that we developed to determine whether a scenario is allowed or disallowed in a UML design class model that includes operation specifications expressed in the OCL [7][8].

A verifier using the method first produces a set of scenarios describing legal and illegal behaviors: A legal scenario describes behavior that adheres to security policies, while an illegal scenario describes behavior that violates security policies.

After the scenarios are produced, the verifier then uses the scenario-based analysis technique to check whether the legal scenarios are allowed, and the illegal scenarios are not permitted in the design model. The design model under analysis is a UML class model in which security policies are modeled as invariants expressed in the OCL. The analysis is essentially a consistency check between the behaviors defined in the scenarios and the behaviors described in the UML class model. Identified discrepancies can be the result of defects in the class model or in the scenarios.

A key challenge is providing the verifier with automated assistance for producing scenarios. A naïve syntax-based generation approach can produce scenarios by considering all possible sequences of operation calls. The verifier will then be required to manually label each generated scenario as legal or illegal. For industrial-strength systems, this naïve approach will generate too many scenarios to make manual labeling of scenarios infeasible. In this paper

we present a scenario generation technique that improves upon the naïve approach by taking into consideration domain-specific knowledge about sequences of operation calls that reflect typical usages and sequences that should not be allowed. This knowledge is encoded in operation call sequence patterns that are used by the verifier to generate scenarios.

The method is illustrated using a design model of an application that uses Role-Based Access Control (RBAC) [6] to manage access to protected resources. UML has been used to specify access control policies. Epstein and Sandhu study the feasibility of using UML to support role engineering [4]. Ahn and Shin study how to express RBAC constraints using OCL [3]. In earlier work, we show how violations of RBAC policies such as separation of duty, prerequisite and cardinality constraints can be modeled using anti-patterns of object configurations [5]. None of the above approaches offer a method for systematically analyzing UML designs to uncover policy violations.

The work described in this paper extends our previous work by providing a systematic approach to producing scenarios used in the analysis. Furthermore, we illustrate how the approach can be used to systematically analyze a design to uncover policy violations.

The remainder of the paper is organized as follows. In Section 2 we provide background information on RBAC and we present a UML design class model that includes OCL invariants that express RBAC policies. In Section 3 we provide an overview of the scenario-based UML design analysis technique we developed in previous work. In Section 4 we describe the analysis method and illustrate its use. In Section 5 we discuss related work in verification and analysis of access control policies, and in Section 6 we conclude the paper.

2. RBAC

RBAC is an access control model used to protect sensitive information resources [6]. In core RBAC, permissions are granted to roles, and roles are assigned to users. The assigned roles that a user activates in a session determine the resources that the user can access in the session. In hierarchical RBAC, roles are organized into hierarchies of junior and senior roles. A senior role dominates its junior roles, that is, a senior role inherits all the permissions of its junior roles, and a junior role inherits all the assigned users of the senior role. Separation of duty (SOD) constraints are added in the constrained form of RBAC. There are two forms of SOD constraints: A static separation of duty (SSD)

constraint prohibits the assignment of conflict of interest roles to the same user, and a dynamic separation of duty (DSD) constraint prohibits the simultaneous activation of conflict of interest roles by the same user.

In this section we present a RBAC policy model in two parts: in the first part we give a UML design class model that describes RBAC classes and operations, in the second part we describe RBAC constraints using OCL invariants.

2.1. RBAC design class model

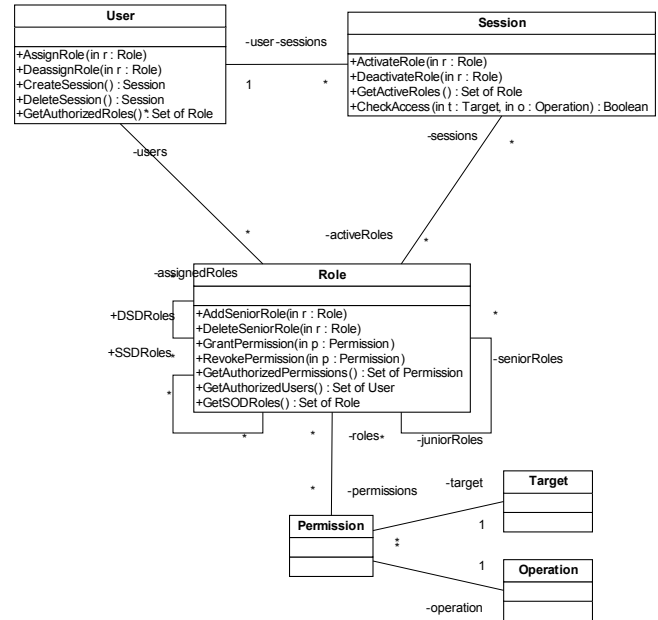


Figure 1. Hierarchical RBAC design class model

In the hierarchical RBAC design class model shown in Fig. 1, the *User*, *Role* and *Session* classes model users, roles and sessions entities in RBAC. The *Permission* class describes RBAC permissions in terms of operations that can be performed on targets. The *assignedRoles* association end determines the set of roles directly assigned to a user. The operation *GetAuthorizedRoles()* returns all roles directly and indirectly assigned to a user. The *activeRoles* association end determines the set of roles directly activated in a session, and the operation *GetActiveRoles()* returns all roles directly activated in a session. The association end *permissions* is the set of all permissions directly associated with a role, and the operation *GetAuthorizedPermissions()* returns all permissions directly and indirectly associated with a role. The *seniorRoles* and *juniorRoles* association ends define the role hierarchy relationships. The *SSDRoles*

association end defines the set of role pairs that are constrained by SSD. The *DSDRoles* association end defines the set of role pairs that are constrained by DSD.

Operations are specified using the OCL. For example, the operation *GetAuthorizedRoles()* in *User* is defined using a query operation *GetDominatedRoles()* as follows:

```
// Get set of authorized roles to the
user.
context User::GetAuthorizedRoles()
: Set(Role)
post: result =
self.assignedRoles.GetDominatedRoles()->
asSet()

// Get set of dominated roles to the
role.
context Role::GetDominatedRoles():
Set(Role)
body:
let oneStep:Set(Role)= Set{self} in
result = if oneStep.juniorRoles-
>isEmpty() then
    oneStep
else
    oneStep->union(oneStep.juniorRoles
    .GetDominatedRoles())->asSet()
endif
```

The operations that are involved in the analysis presented in Section 3 are given below.

```
context User::AssignRole(r:Role)
// Assign a role to the user.
pre:
not self.GetAuthorizedRoles()-
>includes(r)
post: self.GetAuthorizedRoles()-
>includes(r)

context Session::ActivateRole(r:Role)
// Activate a role in the session.
pre:
not self.GetActiveRoles()->includes(r)
post: self.GetActiveRoles()->includes(r)

context
Session::GetActivateRoles:Set(Role)
// Return activated roles in the session.
pre: true
post: result = self.activeRole

context Role::AddSeniorRole(r:Role)
// Add a senior role to current role.
pre: true
post: self.seniorRoles->includes(r) and
r.juniorRoles->includes(self)
```

```
context Role::CheckAccess(t:Target,
o:Operation):Boolean
// Query operation that checks
permissions // of all active roles to see
whether there
// is a match for the target and
operation.
pre true
post: result = self.GetActiveRoles().
GetAuthorizedPermissions()->exists (p |
p.target = t and p.operation = o)
```

2.2. RBAC constraints

2.1.1. Role activation constraint. A fundamental constraint in role activation is that a role can be activated by a user only if it has been assigned to the user. We express this constraint as an OCL invariant named *RBAC_Policy_1*:

RBAC_Policy_1: A user can only activate roles that are assigned to him.

```
context Session
inv RBAC_Policy_1:
self.user.authorizedRoles->
includesAll(self.activeRoles)
```

2.1.2. Role hierarchy constraints. According to the definition of role hierarchy in the NIST RBAC standard [6], a senior role dominating its junior roles implies that the senior role inherits all the permissions of its junior roles, and a junior role inherits all the assigned users of the senior role. *RBAC_Policy_2* expresses this constraint:

RBAC_Policy_2: A senior role inherits all permissions from junior roles, and a junior role inherits all the users of its senior roles.

```
context Role
inv RBAC_Policy_2:
seniorRoles->forall(s |
s.authorizedPermissions->
intersection(self.authorizedPermissions)
= self.authorizedPermissions) and
self.seniorRoles->forall(s |
s.authorizedUsers->
intersection(self.authorizedUsers) =
s.authorizedUsers)
```

The role hierarchy is a partial order on roles and there should not be any cycles in the role hierarchy. We use an OCL query operation on roles called *Dominates* in the policy statement. The expression *r1.Dominates(r2)*, where *r1* and *r2* are roles, returns true if *r2* is a descendant of *r1* in a senior-junior role structure. The constraint is expressed by *RBAC_Policy_3*:

```

context Role::Dominates(r:Role):Boolean
pre true
post:
if (self.juniorRoles->includes(r)) then
result = true
else
result = self.juniorRoles->exists(j |
j.Dominates(r))
endif

```

RBAC_Policy_3: *There must be no cycles in senior-junior role relationships.*

```

context Role
inv RBAC_Policy_3:
not self.Dominates(self)

```

2.1.3. Separation of duty constraints.

RBAC_Policy_4 expresses the static separation of duty constraint, and RBAC_Policy_5 expresses the dynamic separation of duty constraint:

RBAC_Policy_4: *Conflict of interest roles cannot be assigned to the same user (SSD).*

```

context User
inv RBAC_Policy_4:
not self.GetAuthorizedRoles()->exists(r1,
r2 | r1.SSDRoles->includes(r2))

```

RBAC_Policy_5: *Conflict of interest roles can not be activated by the same user simultaneously (DSD).*

```

context User
inv RBAC_Policy_5:
not self.sessions.GetActiveRoles()->
exists(r1, r2 | r1.DSDRoles->
includes(r2))

```

3. Scenario-based UML design analysis technique

A UML design class model can be viewed as a characterization of valid system states, where a system state is a configuration of objects. A system state is called a *snapshot* in this paper. Tools such as USE [10] and OCLE [11] can be used to check whether a snapshot is valid with respect to a UML class model.

The scenario-based UML design analysis technique extends the applicability of these tools to behavior. The technique is used to determine whether behavior defined in a scenario is allowed by a UML design model [7][8]. A verifier charged with validating a UML design model produces a set of scenarios describing legal and illegal behaviors, and then uses the scenario-based analysis technique to check whether the behaviors described in the scenarios are allowed or not in the UML design class model.

The UML design class models against which the scenarios are evaluated consist of operations

specifications expressed in the OCL. A scenario describes a sequence of snapshot transitions, where each transition is triggered by an operation call and describes the net effect of the operation's execution. More precisely, a snapshot transition consists of (1) the name and parameter values of the operation that triggers the transition, (2) a before-snapshot describing the state of the system before the operation is executed, and (3) an after-snapshot describing the state of the system after the operation has executed.

In order to use tools such as USE and OCLE to support scenario-based analysis we developed an approach for generating a class model that characterizes valid snapshot transitions from a UML class model. The generated class model of behavior is called a *Snapshot Model*. Developers can use tools such as USE and OCLE to check that a scenario (sequence of snapshot transitions), satisfies the constraints expressed in a Snapshot Model.

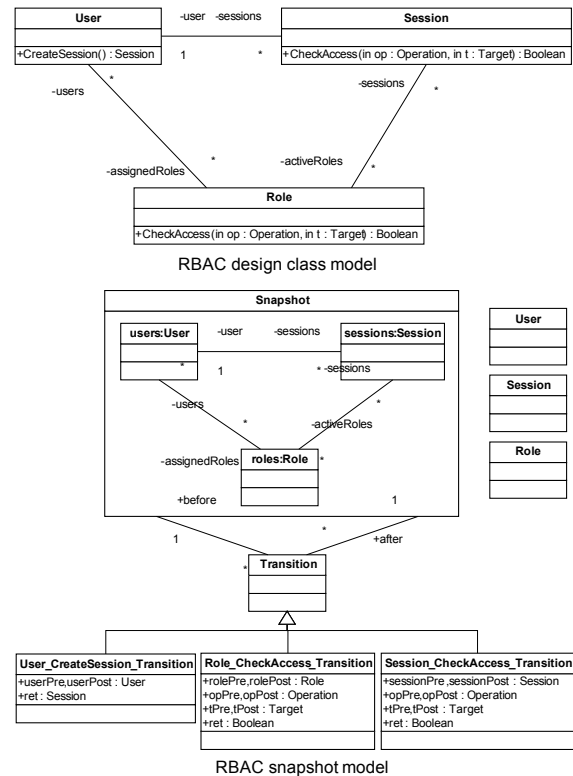


Figure 2. Partial RBAC Class Model and its Snapshot Model

The Snapshot Model is a key artifact in the approach. In previous work we described how a Snapshot Model can be mechanically generated from a UML design class model [8]. The Snapshot Model generated from part of the RBAC design model is shown in Figure 2.

The Snapshot Model consists of a structured

Snapshot class that characterizes snapshots (object configurations) and a *Transition* class that relates before and after snapshots for an operation. The three specializations of the *Transition* class characterize transitions triggered by the three operations shown in the partial RBAC design class diagram shown above the Snapshot Model. For example, the class *User_CreateSession_Transition* characterizes transitions triggered by invocations of the *CreateSession()* operation defined in the *Session* class. The attributes in the *Transition* subclasses represent scalar operation parameters and references to before and after states of the objects accessed by the operations. For example, the attribute *userPost* is a reference to the start state of the *User* object created in an invocation of *CreateSession()*. The created session object is stored in *ret* and there is no before state for the session object that is created during the execution of an operation.

Operation specifications are transformed to invariants on the Snapshot Model. This algorithm is described in our previously published paper [8].

A scenario is described by a UML sequence diagram annotated with descriptions of the effects of each operation invoked in the scenario. We use a UML model animator called UMLAnT to produce snapshot transitions from a sequence model.

4. A policy analysis method

In this section we describe how the scenario-based technique can be used as the basis for a rigorous policy analysis method. A verifier uses the method to produce a set of scenarios describing legal and illegal behaviors, that is, behaviors that conform to and that violate the policies, respectively. The scenarios are then evaluated against a Snapshot Model generated from an RBAC design model in which policies are expressed as invariants.

The method is comprised of two steps:

- (1) *Scenario Generation*: In this step, the verifier produces scenarios, that is, sequences of snapshot transitions. Each scenario must be labeled as illegal or legal before proceeding to the next step.
- (2) *Analysis*. In this step the verifier applies the scenario-based UML design analysis technique (as described in Section 3) to check whether the scenarios are accepted or rejected by the UML design model. If legal scenarios are rejected or illegal scenarios are accepted by the UML design model, it means that the design model, the scenarios, or both are problematic.

In the first step, the verifier produces sequence

diagrams that are animated to produce scenarios that consist of sequences of snapshot transitions. These sequence diagrams must refer to the class elements defined in the UML class model, but the behaviors of the operations invoked in the scenarios are defined independently by the verifier. The scenarios thus reflect the verifier's understanding of how the operations are to behave in a legal scenario or "misbehave" in an illegal scenario. The verifier describes behavior using a UML action language we developed called the Java-like Action Language (JAL) [9]. The sequence diagrams are animated to produce scenarios. A model animator that we developed, called UMLAnT [9], can be used to generate scenarios.

In the remainder of this section we describe the scenario generation technique, and we show how the policy analysis method can be used to analyze access control policy constraints in the RBAC design model. The example used in this section is intentionally small to better focus the description on the method.

4.1. Scenario generation

The scenario generation technique we developed is based upon a naïve scenario generation algorithm. The naïve algorithm generates too many scenarios, and thus we extend it by allowing the verifier to target specific families of scenarios by specifying patterns.

The naïve scenario generation algorithm does the following: (1) builds an *operation invocation tree* from a set of operations and parameter values, (2) traverses the operation invocation tree to produce all possible sequences of operation invocations, and (3) animates each sequence of operation invocations to produce a sequence of snapshot transitions. The verifier must then label each of the generated snapshot transition sequences as legal or illegal.

Each node in an operation invocation tree represents a particular invocation of a system operation on an object. The invocation is referred to as an *operation instance*. Each node contains an object identifier (the receiver of the operation call), an operation name and a value for each operation parameter. The root of the tree represents the system initialization point and it contains information about the start state. Child nodes represent operation invocations that can occur after the invocation represented by the parent node. A scenario is a path that starts at the root and ends at any node in the tree.

To reduce the number of scenarios produced by the above algorithm, the extended technique we developed allows a verifier to (1) limit the depth of the tree, (2) limit the number of objects of a class that can be in a start state, and (3) explicitly define a small domain for

each input parameter of the operation. For example, given an operation *User::AssignRole(r:Role)*, the verifier can restrict the *User* domain size to 2 users objects, and define a small domain for the *Role* parameter as follows: *Domain(Role) = {clerk, seniorClerk}*.

The extended generation technique allows a verifier to specify patterns of operation sequences that restrict (1) the operation calls that are used to build the operation invocation tree and (2) the order in which operations can be invoked. These patterns are called *operation invocation patterns*. An operation invocation pattern is a characterization of particular sequences of operation invocations that the verifier feels typifies good and problematic usages of the system. The patterns are manually created using the best available domain expertise and experience related to the sequences of operations that are likely to uncover policy violations. The patterns are described in terms of constraints on initial states and on the sequencing of operation calls. The use of these patterns allows the verifier to focus the analysis on particular sequences of invocation calls.

For example, a verifier can create the following pattern of operation calls for analyzing role activation behavior:

Initial State Constraint

u in Domain(User) // There is at least one user
#Role>3 //At least 4 roles are in the start state

Call Pattern

u.CreateSession().return(s:Session){1,2}
u.AssignRole(){2,4}
u->s.ActivateRole(){1..4}

The first part of the pattern description constrains the initial state. In this case the initial state must consist of a *User* object, *u*, and at least four roles.

The second part is the pattern of operation calls. The expression *caller->callee.Op()* (e.g., see last line of the above Call Pattern) identifies the sender (*caller*) and receiver (*callee*) of an operation call message. If the caller is omitted then it is assumed that the message is coming from an external actor. The analysis we perform using the Snapshot Model does not require that the sender of an operation call be known; this information is currently used only to visualize the operation sequence as a sequence diagram that shows both senders and receivers of messages.

The pattern describes the following sequences of operation calls:

1. Start with 1 or 2 calls to the *CreateSession()* operation for a user, *u* using any parameters (as indicated by the "." in the parameter list of the operation), and each successfully returning a new session, *s*, (indicated by *return(s:Session)*),

2. followed by 2 to 4 operation calls to the *AssignRole()* for user *u*, and
3. ending with 1 to 4 calls made by the user *u* to activate roles in the sessions previously created by calls to *CreateSession()*.

In order to generate snapshot transitions, a verifier must provide descriptions of operation behavior to the snapshot generation algorithm. The verifier uses a UML action language called the Java-like Action Language (JAL) to describe what the verifier considers legal and illegal effects of operations [9]. JAL descriptions can be interpreted by UMLAnT [9], an open-source Eclipse plugin developed by our research group. UMLAnT supports executing JAL descriptions of behavior and generating snapshot transition sequences.

For example, a verifier can define the legal effects of the operation *User::AssignRole* as follows:

```
JAL_User_AssignRole
if (!this.userRoles._exists(role))
{
    this.userRoles._add(role);
}
```

The verifier creates the JAL descriptions using information provided in requirements use cases and misuse cases. In the case of access control policies, the misuse cases can be based on the access control anti-patterns identified in our previous work [5].

Scenario generation algorithm

Inputs. UML design class model, maximum number of operations Max , parameter domain definitions, operation JAL definitions, tree node r . Operation invocation patterns.

Outputs. Set of scenarios.

Algorithm steps

For each operation call do:

If operations from root to current tree node r and op match an operation invocation pattern:

1. Create one tree node n and add it as child of r .
2. Store information about the operation call (e.g., operation name, parameters, receiving object identifier) in tree node n .
3. Execute desired JAL description associated with the operation using the start state stored in r to get the next system state. Store the next system state in tree node n .
4. Print the sequence of operation calls from the tree root to tree node n as an output scenario.
5. If $Max > 1$
 - a) Call the scenario generation algorithm recursively with tree node n and $Max - 1$ as maximum number of operations.

Figure 3. Scenario generation algorithm

Snapshot transitions are generated by traversing the operation invocation tree and interpreting the associated JAL descriptions of behavior using UMLAnT. The verifier then needs to determine whether the generated scenarios describe legal or illegal behaviors.

The scenario generation algorithm is described in Fig. 3.

4.2. Analyze RBAC constraints

In this sub-section we show how some of the RBAC constraints given in Section 2 can be analyzed using the method.

4.2.1. Analyze role activation constraint. To analyze the role activation constraint ($RBAC_Policy_1$), we use the following operation invocation pattern:

Initial State Constraint

$Domain(User) = \{Bob\}$

$Domain(Role) = \{clerk, seniorClerk\}$

Call Pattern

$[no\ Bob.AssignRole(r)]\{0..2\}$

$Bob.CreateSession(.)return(s:Session) \quad Bob->s.ActivateRole(r)\{1..2\}.$

The expression $[no\ Bob.AssignRole(r)]$ is used to

match all operation calls except calls of the form $Bob.AssignRole(r)$.

The above pattern describes sequences of operations which end with 1 or 2 invocations of the $ActivateRole()$ operation, and start with 0 to 2 operation invocations that do not include operation calls that assign the activated roles to the user Bob .

The verifier describes the effect of the $ActivateRole$ operation using JAL – The JAL description simply activates the role. Scenarios generated from this pattern would allow roles to be activated even though they are not assigned to the user. For this reason, the verifier knows that the pattern would produce illegal scenarios.

An example of an illegal scenario generated from the above pattern is shown in Fig. 4. The scenario starts from an initial system state with one user instance Bob and one $Role$ instance $clerk$. The user creates one session and activates the $clerk$ role. The activation succeeds and $clerk$ is added to the $activeRoles$ association of the session.

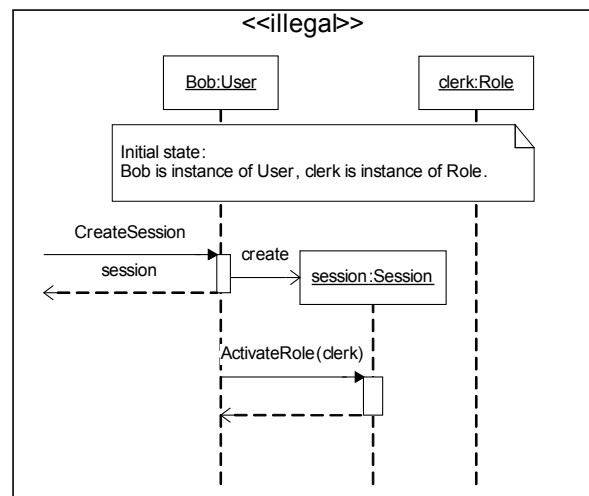


Figure 4. Role activation analysis scenario

The RBAC design model should reject the illegal behavior described by the scenario. Analysis with USE revealed that the RBAC design model is consistent with the scenario. The defect in the design class model is that the operation $Session::ActivateRole$ activates any role that is not activated. The pre-condition should check whether the role is assigned or not.

4.2.2. Analyze separation of duty constraints. We use the following operation invocation pattern to check enforcement of the SOD constraints:

Initial State Constraint

$Domain(User) = Bob$

$cashier$ in $Domain(Role)$

```

accountant in Domain(Role)
cashier in accountant.SSDRoles // the roles conflict
Call Pattern
[
[.]*
Bob.AssignRole(cashier)
Bob.AssignRole(accountant)
]{1}
[
Bob.CreateSession(.)return(s:Session)
s.ActivateRole(r){2..4} where(r = accountant and r =
cashier)
]{0..1}

```

The expression `[.]` matches any operation call and `"*"` represents the multiplicity "0 or more". The where clause stipulates that at least one of the `Activate()` calls must activate the `accountant` role, and at least one of the `Activate()` calls must activate the `cashier` role.

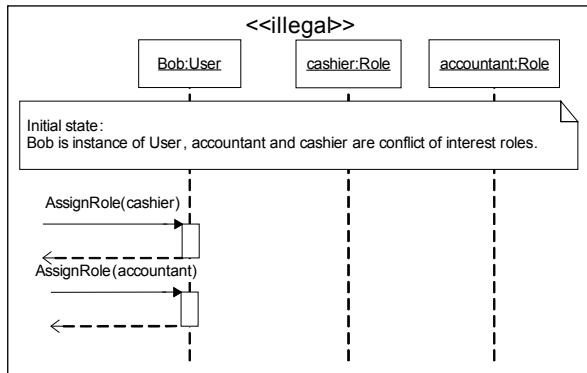


Figure 5. Static separation of duty analysis scenario

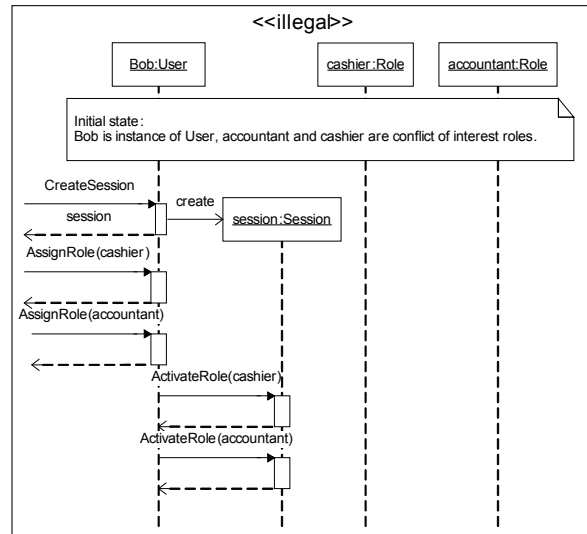


Figure 6. Dynamic separation of duty analysis scenario

The illegal scenario shown in Fig. 5 is generated from the pattern. It starts in a state consisting of two conflict of interest roles, `cashier` and `accountant`, and a user `Bob`. `Bob` is assigned `cashier` role first and then assigned the `accountant` role. The scenario violates the static separation of duty constraint defined as `RBAC_Policy_4` and thus it should be rejected by the RBAC design. In the design model, the `User::AssignRole` operation specified in Section 2 only checks whether the role is assigned to the user or not before it assigns the role, so that the illegal scenario is consistent with the RBAC design. To enforce the static separation of duty constraint in an RBAC design, the operation should also check whether the role to be assigned is in conflict of interest with roles that have been assigned to the user.

The generated illegal scenario shown in Fig. 6 was used to analyze the dynamic separation of duty constraint. In the scenario the user `Bob` is assigned two conflict of interest roles `cashier` and `accountant`, and `Bob` activates both roles in one session. Again, the `Session::ActivateRole()` operation does not check that the role to be activated is in a conflict of interest with a role in a session created by the user.

5. Related work

Sohr et. al. [12] propose both formal and practical approaches to analyze RBAC policies. The formal approach verifies RBAC policy specifications in first-order LTL (Linear Temporal Logic) via a theorem prover Isabelle; The practical approach analyzes RBAC policies specified in UML/OCL via USE tool [10] which automatically generates snapshots to validate policy invariants. The formal analysis approach with theorem prover guarantees reliability but it needs human intervention. The practical analysis approach based on automatic snapshot generation may help detect conflicting or missing constraints but it is not guaranteed because USE cannot generate all the snapshots. In addition, the approach focuses the analysis on OCL invariants while operation constraints in the policy design model are not analyzed.

Model checking techniques are used to check properties of states in state spaces. Zhang et al. [17] present a model checking algorithm and tool to assess access control specifications in propositional logic. The assessment checks whether the access control policy gives legitimate users enough permissions as well as whether it prevents intrusions. Schaad et al. [18] analyze separation of duty properties in ERP systems where delegation and revocation of workflow tasks and rights is frequent. Model checking techniques suffer

from state-space explosion problems. Efforts to reduce the time and space complexity include simplifying the model to an abstract one, partial order reduction and symbolic model checking which represents states and state transitions with Boolean formulas, however so far performance is still major issue of model checking. Alloy [14], a light-weight formal modeling notation and constraint analyzer, models structural aspects of systems in Z-like specifications and analyzes claims in first-order logic by simulating an example configuration or finding counter-examples. Zao et. al. [13] utilize Alloy to verify algebraic characteristics of RBAC schema. However, their work analyzes RBAC schema only instead of the whole policy design model, and Alloy analyzer is not efficient for analyzing large models as it has to reduce the search space of objects by limiting the number of objects in the logic expressions being analyzed. Neither Alloy nor model checking technique can be adapted to automatically generate scenarios.

UML model animation and simulation techniques execute UML design models and generate semantic object models by interpreting OCL in the UML design model or defining and applying graph transformation rules [20][21][22]. We create similar semantic snapshot models, but we do not execute the UML design model, instead we generate these snapshot models from scenarios created from the perspective of verifiers. Compared with existing UML model animation and simulation work, our work is more rigorous and we also automatically generate legal and illegal scenarios for analysis.

Song et. al. [15][16] verify enforcement of access control policies in the composition of access control aspect model and application model by manually discharging proof obligations. The limitation of their work is that the proof is not generated automatically.

Ray et. al [5] analyzes access control constraints in the application model by modeling violations of access control constraints as UML object diagrams and statically finding matches of such violations in the application model. The approach lacks tool support and the UML object diagrams often do not cover all the violations of access control constraints so that not all the policy violations in the application model can be detected.

6. Conclusions

In this paper we describe a lightweight, rigorous method for analyzing security policies. The method analyzes a UML design model by producing a set of scenarios and checking the consistency between the UML design model and these scenarios.

We propose an algorithm to automatically generate scenarios for analysis. The verifier uses domain knowledge and experience to reduce the number of scenarios that are generated by developing operation invocation patterns that focus the analysis on families of scenarios. The generation is semi-automatic because the verifier sometimes has to examine the scenarios and label them as legal or illegal.

Our future work includes building an Eclipse-based tool environment that integrates the UMLAnT, USE/OCLE tools and other mechanisms needed to support the method (e.g., the mechanisms for generating Snapshot Models from Design Class models and for generating scenarios). We also plan to develop techniques for systematically producing operation invocation patterns from requirements behavioral models. These tools will ease the task of the verifier charged with producing scenarios to evaluate a design. We will continue to investigate the extent that domain knowledge and experience can be captured and used to support a more systematic approach to generating legal and illegal scenarios.

Acknowledgements

This work was supported in part by AFOSR under contract number FA9550-07-1-0042.

References

- [1] Object Management Group, *Unified Modeling Language: Superstructure*, version 2.0 Final Adopted Standard.
- [2] Object Management Group, *Object Constraint Language Specification*, Version 2.0.
- [3] Ahn, G. and Shin, M. E. 2001. "Role-Based Authorization Constraints Specification Using Object Constraint Language". In *Proceedings of the 10th IEEE international Workshops on Enabling Technologies: Infrastructure For Collaborative Enterprises* (June 20 - 22, 2001). WETICE. IEEE Computer Society, Washington, DC, 157-162.
- [4] Epstein, P. and Sandhu, R. 1999. "Towards a UML based approach to role engineering". In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control* (Fairfax, Virginia, United States, October 28 - 29, 1999). RBAC '99. ACM, New York, NY, 135-143. DOI=<http://doi.acm.org/10.1145/319171.319184>
- [5] Ray, I., Li, N., France, R., and Kim, D. 2004. "Using UML to visualize role-based access control constraints". In *Proceedings of the Ninth ACM Symposium on Access Control Models and Technologies*, Yorktown Heights, New York, USA, June 02 - 04, 2004.
- [6] D.F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R.

- Chandramouli. "Proposed NIST Standard for Role-Based Access Control". *ACM Transactions on Information and Systems Security*, 4(3), Aug. 2001.
- [7] Lijun Yu, Robert France, Indrakshi Ray, Kevin Lano, "A Light-weight Static Approach to Analyzing UML Behavioral Properties", *Proceedings of the International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, Auckland, New Zealand, July 2007.
- [8] Lijun Yu, Robert France, Indrakshi Ray, "Scenario-based Static Analysis of UML Class Models", *Proceedings of ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MODELS 2008)*, Toulouse, France, Sep. 28-Oct.3, 2008.
- [9] T. Dinh-Trong, N. Kawane, S. Ghosh, R. B. France, and A. A. Andrews. "A Tool-Supported Approach to Testing UML Design Models", *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, IEEE Computer Society Press, pp. 519-528, Shanghai, China, June 16-20, 2005.
- [10] Gogolla, M., Büttner, F., and Richters, M. 2007. "USE: A UML-based specification environment for validating UML and OCL". *Sci. Comput. Program.* 69, 1-3, Dec. 2007.
- [11] [D. Chiorean, M. Pasca, A. Cărcu, C. Botiza, S. Moldovan](#), "Ensuring UML Models Consistency Using the OCL Environment", *Electronic Notes in Theoretical Computer Science*, Volume 102, Nov. 2004, pages 99-110.
- [12] Sohr, K., Drouineaud, M., Ahn, G., and Gogolla, M. 2008. "Analyzing and Managing Role-Based Access Control Policies". *IEEE Trans. on Knowl. and Data Eng.* 20, 7 (Jul. 2008), 924-939. DOI=<http://dx.doi.org/10.1109/TKDE.2008.28>
- [13] Zao, J., Wee, H., Chu, J., Jackson, D.: RBAC Schema Verification Using Lightweight Formal Model and Constraint Analysis. Submitted to *SACMAT 2003*, <http://alloy.mit.edu/contributions/RBAC.pdf>
- [14] D. Jackson, "Alloy: a lightweight object modeling notation", *ACM Transactions on Software Engineering and Methodology*, Volume 11, Issue 2, April 2002, pages 256-290.
- [15] Eunjee Song, Raghu Reddy, Robert France, Indrakshi Ray, Geri Georg, Roger Alexander, "Verifiable Composition of Access Control Features and Applications", *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies (SACMAT 2005)*, Scandic Hasselbacken, Stockholm, June 1-3, 2005.
- [16] Eunjee Song, Robert France, Indrakshi Ray, and Hanil Kim, "Checking Policy Enforcement in an Access Control Aspect Model", *Proceedings of the International Conference on Convergence Technology and Information Convergence (CTIC) '07*, Anaheim, California, November 2007.
- [17] Nan Zhang, Mark D. Ryan and Dimitar Guelev, "[Evaluating Access Control Policies Through Model Checking](#)". *Eighth Information Security Conference (ISC'05)*. Lecture Notes in Computer Science volume 3650, pages 446-460, Springer-Verlag, 2005.
- [18] Schaad, A., Lotz, V., and Sohr, K. 2006. "A model-checking approach to analysing organisational controls in a loan origination process". In *Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies (Lake Tahoe, California, USA, June 07 - 09, 2006)*. SACMAT '06. ACM, New York, NY, 139-149. DOI=<http://doi.acm.org/10.1145/1133058.1133079>
- [19] Jim Davies and Jim Woodcock (1996). [Using Z: Specification, Refinement and Proof](#). Prentice Hall International Series in Computer Science. ISBN 0-13-948472-8. <http://www.usingz.com/text/online/>
- [20] Ermel, C., Holscher, K., Kuske, S., and Ziemann, P. 2005. "Animated Simulation of Integrated UML Behavioral Models Based on Graph Transformation". In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (September 20 - 24, 2005)*. VLHCC. IEEE Computer Society, Washington, DC, 125-133. DOI= <http://dx.doi.org/10.1109/VLHCC.2005.18>
- [21] Patricio Letelier Torres, Pedro Sánchez, "Validation of UML Classes through Animation", In *Proceedings of the International Workshop on Conceptual Modeling Quality, IWCMQ'02*, pp. 61-73.
- [22] Oliver, I. 1999. "Validation of Object Oriented Models Using Animation". In *Proceedings of the Workshop on Object-Oriented Technology (June 14 - 18, 1999)*. A. M. Moreira and S. Demeyer, Eds. Lecture Notes In Computer Science, vol. 1743. Springer-Verlag, London, 375-376.
- [23] Lilius, J.; Paltor, I.P., "vUML: a tool for verifying UML models", *Automated Software Engineering, 1999. 14th IEEE International Conference on*, Oct 1999 Page(s):255 – 258.
- [24] Jürjens, J. 2002. "UMLsec: Extending UML for Secure Systems Development". In *Proceedings of the 5th international Conference on the Unified Modeling Language (September 30 - October 04, 2002)*.