Rigorous Analysis of UML Access Control Policy Models

Wuliang Sun, Robert France, and Indrakshi Ray
Department of Computer Science
Colorado State University
Fort Collins, USA
{*sunwl, france, iray*}*@cs.colostate.edu*

***Abstract*—The use of the Unified Modeling Language (UML) for specifying security policies is attractive because it is expressive and has a wide user base in the software industry. However, there are very few mature tools that support rigorous analysis of UML models. Alloy is a formal specification language that has been used to rigorously analyze security policies, but few practitioners have the background needed to develop good Alloy models. We propose a new approach to policy analysis in which designers use UML at the front-end to describe their security policies and the Alloy Analyzer is used at the back-end to analyze the modeled properties. The automated UML-to-Alloy and Alloy-to-UML transformations obviate the need for security designers to understand the intricacies of the Alloy specification language. The proposed approach supports the analysis of both functional and structural aspects of security policies.**

***Keywords*-Alloy; LRBAC; UML;**

I. Introduction

Security policies are an essential part of a security regime that shields systems from unauthorized accesses. Security policies for software developed using Model-Driven Engineering (MDE) approaches are likely to be modeled using the Unified Modeling Language (UML), the de-facto MDE language. Uncovering and correcting errors in policy models after the policies have been enforced in software can be expensive. Consequently, it is important that policy models be rigorously analyzed to uncover errors at the early stages of software development.

In this paper we restrict our attention to security policies that can be conveniently expressed using the standard UML class modeling notation and the OCL (Object Constraint Language) [16]. Our experience suggests that security policies that constrain how protected system elements are accessed can be conveniently expressed in class models augmented with OCL invariants and operation specifications. Structural aspects of these access control policies can be expressed in terms of classes and relationships among classes, while access control functionality can be be represented by operations associated with OCL pre- and post-conditions.

Tools such as USE [11] and OCLE [4] can be used to analyze structural aspects of access control policies, but they provide very poor support, if any, for analyzing access control functionality. There are UML research tools that support behavioral analysis (e.g., [7]), but these tools are either based on specialized forms of UML models that developers must become familiar with in order to use effectively, or require developers to use behavioral models such as statemachines to specify their properties, even when the properties can be more conveniently expressed as class models.

Alloy [6] is a formal specification language that has been used to specify security policies (e.g., see [14] [12]). It has very good tool support in the form of the Alloy Analyzer that translates an Alloy specification into a boolean formula that is evaluated by embedded SAT-solvers. However, in order to effectively use Alloy, the security designer must be familiar with the mathematical concepts underlying Alloy elements. Many practitioners in industry today do not have the mathematical background needed to use Alloy effectively.

In this paper, we present a rigorous approach to uncovering errors in access control policy class models. The approach allows a security designer to check whether it is possible for a system to move from a valid state with specified characteristics to a target invalid state. If analysis uncovers a sequence of operation calls that move the system from a valid state to an invalid state, then the designer uses the trace information provided by the analysis to uncover the error in the policy model. At the front-end, a security designer uses the UML class model notation and the OCL to model security policies and to specify the properties to verify. A property-to-verify specifies the valid and invalid states that will be checked by the Alloy Analyzer at the back-end. The approach uses UML-to-Alloy and Alloy-to-UML transformations to shield the security designer from the back-end use of the Alloy language and analyzer.

The transformations used in the approach build upon the UML2Alloy transformation tool [1] developed by Bezhad Bordbar's research group. The UML2Alloy tool currently supports only transformation of structural properties. The work proposed in this paper extends this work by providing support for transforming functional behavior specified in a UML class model to a Alloy model that specifies behavioral traces.

The approach also builds upon our previous work on the Scenario-based UML Design Analysis (SUDA) approach [18], [17]. A designer uses SUDA to check whether a specific functional scenario is supported by a design class model in which operations are specified using the OCL. In

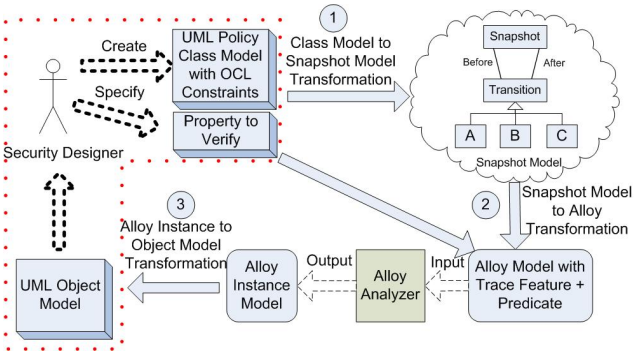


Figure 1: Approach Overview

SUDA, the property to be verified is expressed as a specific sequence of state transitions (a functional scenario). The approach described in this paper goes further in that the property to be verified is expressed in terms of valid and invalid states, and analysis attempts to uncover scenarios that start in a specified valid state and end in a specified invalid state. In summary, SUDA is used to answer the question "Is the given scenario supported by the UML class model?", while the approach described in this paper is used to answer the question "Is there a scenario supported by the UML class model that starts in a specified valid state and ends in a specified invalid state?".

In an early use of the approach, we analyzed a Location aware Role-Based Access Control model (LRBAC) developed by Indrakshi Ray's research group [10] [8] [9]. In LRBAC, access control decisions are based on the user's role, his location during the time of access, and the location of the object that he is trying to access. Although the LRBAC developers formally described the model using the Z specification language [9], we were able to uncover a significant error in the formalization using the approach.

The rest of the paper is organized as follows. Section II presents background material needed to understand the work described in this paper. Section III describes the UML/Alloy analysis approach and illustrates its use on the LRBAC demonstration case study. Section IV discusses related work, and Section V concludes the paper with a pointer to future directions.

II. Background

Figure 1 presents an overview of the approach. The dotted area includes the front-end activities and models. The front-end models are the only models that a security designer needs to directly manipulate. The security designer is responsible for 1) modeling access control policies using UML class model notation and the OCL, and 2) specifying the property-to-verify. The property-to-verify is expressed in terms of two $object\ configuration\ patterns$: One that characterizes the form of valid states (object configurations) to analyze, and the other characterizes target invalid states. Object configurations representing software states are called *snapshots* in this paper.

The back-end activities use three transformations (indicated in Fig. 1). Transformation 1 transforms the UML policy model to a class model, called a *snapshot model*, that specifies valid snapshot transitions, where a transition describes the effect of an operation on a state. The approach uses the UML-to-snapshot model transformation defined in the SUDA approach [19] here. Transformation 2 converts the snapshot model to an Alloy model. The property-to-verify is transformed to an Alloy predicate, referred to as the *verification predicate*, that is added to the Alloy model generated from the snapshot model. The resulting Alloy model is fed into the Alloy Analyzer and the verification predicate is evaluated. The Alloy Analyzer is used to determine if there exists an operation invocation sequence that starts from a specified valid snapshot and ends in a specified invalid snapshot. If the Analyzer finds a sequence then Transformation 3 is needed to convert the Alloy instance model of the sequence to a UML object model describing the sequence.

In the remainder of this section we describe (1) the class model-to-snapshot model transformation defined in the SUDA approach, (2) the Alloy trace mechanism, and (3) the LRBAC model used to illustrate our approach.

A. Snapshot Model: A Static Model of Behavior

Software behavior can be represented as a sequence of state transitions, where each transition is triggered by an operation invocation. Yu et al.[19] proposed a scenario-based static analysis approach, called SUDA, that allows a developer to check whether a particular sequence of state transitions is supported by a design class model in which operations are specified in OCL. In SUDA, a design class model with operation specifications is transformed to a static model of behavior, called a snapshot model. A snapshot represents a system object configuration at a particular time. A key concept in a snapshot model is the snapshot transition. A snapshot transition describes the behavior of an operation in terms of how system state changes after the invoked operation has completed its task. It consists of a before state, an after state, and the operation invocation that triggers the transition. An operation invocation is described by the operation name and the parameter values used in the invocation.

Figure 2 shows an example of a UML snapshot model generated from a UML design class model. The instances of class $Snapshot$, are snapshots, and the instances of class $Transition$ are transitions that each relates a *before* snapshot with an *after* snapshot. A snapshot consists of linked instances of classes in a design model (i.e., an object configuration).

Each operation in the original design class model (e.g., $Af(b : B)$ in the class A) is transformed into a specialization

of class $Transition$ (e.g., Af). The parameters of each operation (e.g., b in Af(b:B)) in the original design class model are transformed into references (shown as attributes) in the $Transition$ specialization. Moreover, if a parameter has a class type, it is transformed into two references. For example, the parameter b in the operation $Af(b : B)$ is transformed into $bPre : B$ and $bPost : B$, one of which specifies the parameter's state before the execution of the operation ($bPre$) and the other specifies the parameter's state after the execution of the operation ($bPost$). Also, two references (e.g., $aPre : A$ and $aPost : A$) pointing to before and after states of the object on which the operation is called are generated and placed in the specialized $Transition$ class representing the operation.

Operation specifications in the design class model are transformed into transition invariants that precisely specify the before and after snapshots that are associated with $Transition$ instances. For example, the operation specification for $Af(b : B)$:

Context *A::Af(b:B)*
Pre: *self.AB→excludes(b)*
Post: *self.AB = self.AB@pre→including(b)*

is transformed to the following transition invariant on the $Transition$ class Af:

Context *Af*
inv*:*
before.aset→includes(aPre) and
before.bset→includes(bPre) and
aPre.AB→excludes(bPre) and
after.aset→includes(aPost) and
after.bset→includes(bPost) and
aPost.AB = aPre.AB→including(bPost) and
after.aset→excluding(aPost)=before.aset→excluding(aPre)
after.bset→excluding(bPost)=before.bset→excluding(bPre)

More details on the transformation approach can be found in [17][18]. In SUDA, the generated snapshot model and a sequence of snapshot transitions are fed into an OCL analysis tool (e.g., USE and OCLE) to determine whether the behavior described by the snapshot sequence conforms to the behavior defined in the snapshot model.

B. Dynamic Analysis using Alloy

Alloy [6] is a textual modeling language based on first-order relational logic. An Alloy model consists of $signature$ declarations, $fields$, $facts$ and $predicates$. Each $field$ belongs to a $signature$ and represents a relation between two or more $signatures$. $Facts$ are statements that define constraints on the elements of the model. $Predicates$ are parameterized constraints that can be invoked from within $facts$ or other $predicates$.

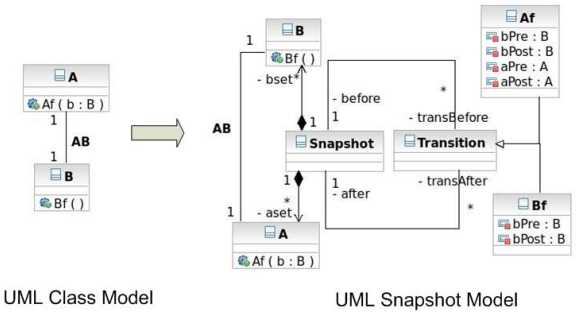


Figure 2: Example of a UML Snapshot Model Generated from a UML Class Model

Alloy provides a trace mechanism that associates the transitions triggered by operation invocations with states defined by $signatures$. Alloy defines an $ordering$ type that can cast a set of $states$ into a sequence of $states$. A $trace$ fact defines $states$ that are reachable through the invocation of sequence of operations.

C. Location-aware Role-Based Access Control Model

The Location-aware Role-Based Access Control (LRBAC) model (see Figure 3), proposed by Ray et. al. [10] [8] [9], is used in this paper to demonstrate the analysis technique in Section III-C. LRBAC is an access control model used to protect sensitive information resources based on Role-Based Access Control (RBAC) model [13]. LRBAC extends RBAC by incorporating the $location$ concept. In a RBAC model, $permissions$ are granted to $roles$, and $roles$ are assigned to $users$. The assigned $roles$ that a $user$ activates in a $session$ determine the resources ($objects$) that the $user$ can access ($operate$) in the $session$.

In a LRBAC model, a new class $Location$ is introduced, and associated with $User$, $Object$, $Role$, and $Permission$ entities. A $user$ can only be in one $location$ at any given time, while a $location$ can be associated with multiple $users$. $UserLoc$ is the association between $User$ and $Location$. Given a $user$, $user.UserLoc$ returns the $location$ of the $user$. Similarly, an $object$ is associated with one $location$ only, while a $location$ can have many $objects$. $Roles$ are associated with $locations$ by two relationships: $AssignLoc$ and $ActivateLoc$. Given a $role$, $role.AssignLoc$ returns the set of $locations$ in which that $role$ can be assigned, while $role.ActivateLoc$ returns the set of $locations$ in which that $role$ can be activated. A $role$ can be assigned to a $user$ only if $user.UserLoc$ is a member of $role.AssignLoc$. Similarly, a $user$ can activate a $role$ only if $user.UserLoc$ is a member of $role.ActivateLoc$. Also, $permissions$ are associated with $locations$ by two relationships: $PermRoleLoc$ and $PermObjLoc$. Given a $permission$, $permission.PermRoleLoc$ returns the set of allowable $locations$ for the $roles$ associated with the $permission$, and $permission.PermObjLoc$ returns the set

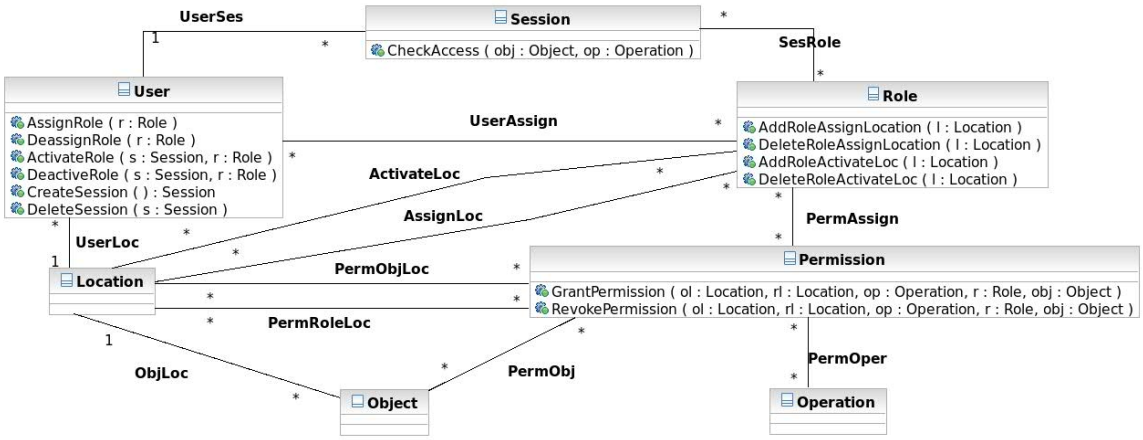


Figure 3: LRBAC Class Model

of allowable $locations$ for the $objects$ associated with the $permission$.

In class $Role$, both operation $AddRoleAssignLocation$ and $AddRoleActivateLoc$ take a $location$ parameter, and associate the $location$ with a $role$. The difference between these two operations is that the $location$ in the former operation refers to the one in which the role can be assigned, while the $location$ in the latter operation refers to the one in which the role can be activated. Both $DeleteRoleAssignLocation$ and $DeleteRoleActivateLoc$ take a $location$ parameter, and remove the $location$ from a set of $locations$ associated with a $role$.

Ray et. al. [9] use the Z specification language for presenting a formal LRBAC model. In this paper, we use OCL to specify LRBAC operations and invariants. For example, the operation $DeleteRoleAssignLocation$ is specified using OCL as follows:

// Remove a location from a set of locations associated with a role
Context *Role::DeleteRoleAssignLocation(l:Location)*
// Precondition: location l has been associated with the role
Pre: *self.AssignLoc→includes(l)*
// Postcondition: location l has been removed from a set of locations associated with the role
Post: *self.AssignLoc=self.AssignLoc@pre→excluding(l)*

III. Approach

In this section, we describe the snapshot model-to-Alloy model transformation (Section III-A) and the Alloy instance-to-object model transformation (Section III-B). The class model-to-snapshot model transformation was outlined in the previous section. We also describe how the approach was used to analyze a LRBAC model (Section III-C).

A. Snapshot Model to Alloy Model Transformation

Figure 4 shows the Alloy model generated from the snapshot model in Fig. 2. The figure identifies the parts that are generated by the 4-step transformation algorithm presented in Algorithm 1.

In step 1, each class that is part of the $Snapshot$ class in the design class model is transformed to a signature in Alloy. For example, class A in Fig. 2 is transformed to a signature $sig\ A\{\}$ in Fig. 4. If a class has attributes, its attributes are transformed to fields of the signature corresponding to the class.

In step 2, the $Snapshot$ class is transformed to a $Snapshot$ signature containing fields that specify the object configuration within a snapshot. Two groups of fields in the $Snapshot$ signature are used to specify object configurations: fields defining a set of objects (e.g. aset:set A), and fields defining links between objects (e.g. AB: A one→one B). Linked objects in a snapshot must be in the domain defined by the $Snapshot$ signature. This constraint is expressed as a fact associated with the $Snapshot$ signature. For example, the fact $AB = AB :> bset\ \&\ aset <: AB$ in Fig. 4 specifies that linked objects either belong to $aset$ or $bset$. The Snapshot signature also includes a field (e.g., $OperID$) that is used to identify the operation that causes a transition to the snapshot when the snapshot is part of a sequence of transitions. There is an identifier type for each operation in the original class model (e.g., ID_Af is the identifier that corresponds to the operation Af).

In step 3, each $Transition$ specialization in the snapshot model is transformed to a predicate in Alloy. If a $Transition$ specialization has attributes, its attribute are transformed to parameters of the predicate. Two more parameters, $before$ and $after$ with the type $Snapshot$, are added to each predicate to represent the system states before and after the transition. OCL invariants associated with

each $Transition$ specialization in the snapshot model are transformed into the body of the predicate corresponding to the $Transition$ specialization using UML2Alloy [2][1][3]. Objects and links that are not changed during the transition are explicitly specified in the predicate. An equality that identifies the operation causing the transition (e.g. *after.OperID = ID_Af*) is also included in each predicate.

After the snapshot model is transformed to an Alloy model, a verification predicate is generated from the property-to-verify provided by the user. This predicate will check whether a snapshot with the specified invalid configuration pattern is reachable from a snapshot with the specified valid configuration pattern through operation executions. An example of a verification predicate generated from the property-to-verify shown in Fig. 6 can be found in Section III-C.

Algorithm 1 Snapshot Model to Alloy Model Transformation Algorithm

Input: UML Snapshot Model
Output: Alloy Model
Algorithm Steps:
Step 1. Transform each class that is part of $Snapshot$ class to a signature in Alloy.
Step 2. Transform the $Snapshot$ class to a $Snapshot$ signature containing fields that specify the object configurations within a snapshot.
Step 3. Transform each $Transition$ specialization to a predicate in Alloy.
Step 4. Declare $Snapshot$ signature as ordering type and define a $trace$ fact to associate transitions between two consecutive snapshots with operations.

B. Alloy Instance to UML Object Model Transformation

Algorithm 2 Alloy Instance to UML Object Model Transformation Algorithm

Input: Alloy Instance in XML, Original Class Model in XMI
Output: UML Object Model in XMI
Algorithm Steps:
Step 1. Extract fields and tuples associated with snapshot atoms from an Alloy XML model, and generate a table that includes information for all snapshots.
Step 2. Split the table based on snapshot atom label to produce a sequence of tables, where each table contains information about a single snapshot.
Step 3. Generate a UML object model in XMI describing a sequence of snapshot transitions using information from the sequence of tables produced in Step 2.

If the Alloy Analyzer generates an instance that satisfies the verification predicate, the Alloy instance-to-UML object model transformation takes the Alloy instance model in XML as an input and creates a UML object model in XMI. Figure 5 illustrates the three-step process used to generate UML object models. The algorithm outline for this transformation is described in Algorithm 2.

```
module Sample
open util/ordering[Snapshot] as SnapshotSequence

sig A{} sig B{}                                    Step 1

abstract sig ID{}                                  Step 2
One sig  ID_Af,  ID_Bf,  ID_Null extends ID{}
sig Snapshot{
OperID: one ID,
// Objects
aset: set A,
bset: set B,
// Links
AB: A one->one B,
} { // Linked objects must exist in the snapshot
AB = AB :> bset & aset <: AB
}

pred A_Af[disj before, after: Snapshot, aPre, aPost: A, bPre, bPost: B] {
after.OperID = ID_Af                               Step 3
// Precondition
aPre in before.aset
bPre in before.bset
bPre not in aPre.(before.AB)
// Postcondition
aPost in after.aset
bPost in after.bset
aPost.(after.AB) = aPre.(before.AB) + bPost
// Unchanged objects
after.aset - aPost = before.aset - aPre
after.bset - bPost = before.bset - bPre
// Unchanged links
after.AB - aPost <: (after.AB) = before.AB - aPre <: (before.AB)
}

pred B_Bf[disj before, after: Snapshot, bPre, bPost: B] {
after.OperID = ID_Bf
// Unchanged objects
after.aset = before.aset
after.bset = before.bset
// Unchanged links
after.AB = before.AB
}

fact traces{                                       Step 4
        all before: Snapshot - SnapshotSequence/last | let after =
                SnapshotSequence/next[before] | Some a: A | some b: B |
                        A_Af[before, after, a, a, b, b] || B_Bf[before, after, b, b]
}
```

Figure 4: An Alloy Model Transformed from the Snapshot Model in Figure 2

C. A Demonstration Case Study

We analyze functional features of a LRBAC model to demonstrate how this approach can be used to find errors in policy models. The LRBAC model that will be analyzed is shown in Fig. 3. The following are the steps performed in the case study.

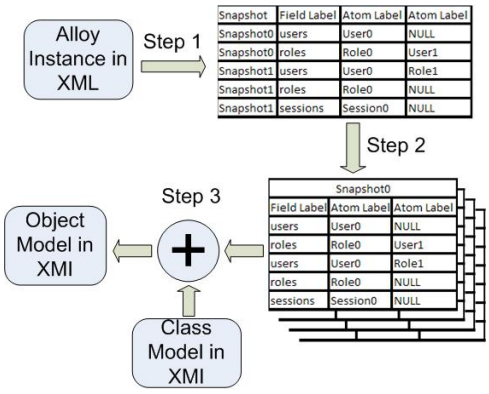


Figure 5: Alloy Instance to UML Object Model Transformation

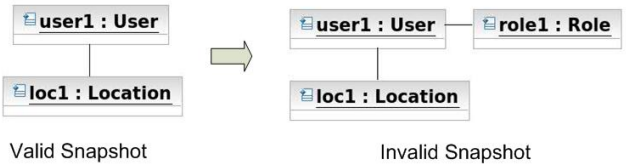


Figure 6: Valid and Invalid LRBAC Snapshot Patterns

1) Specifying the Property-to-Verify: In a LRBAC model, a role can be assigned to a user only if the location associated with the user is included in the locations associated with the role. Figure 6 shows a valid and an invalid LRBAC snapshot pattern that we used to check this constraint. The left part of Fig. 6 shows a user linked to a location. A valid snapshot with at least one user linked to a location is an instantiation of this pattern. The right part of Fig. 6 shows a user in a location linked to a role that does not include the user's location. A snapshot that has an instantiation of this object configuration pattern is considered invalid. The question that an analysis based on this property-to-verify answers can be expressed as follows: "Is there a sequence of operation invocations that takes the system from a valid state consisting of at least one user in a location to an invalid state in which the user is linked to a role that does not include the user's location?"

2) Generating the LRBAC Snapshot Model: The LRBAC model in Figure 3 is transformed to the snapshot model shown in Figure 7 using the algorithm outlined in the previous section. Each operation in the LRBAC model is transformed to a $Transition$ specialization, and each operation specification is transformed to an invariant for the $Transition$ specialization. For example, the operation specification for $DeleteRoleAssignLocation$ given in Section II-C is transformed to the following partially presented invariant:

Context *Role_DeleteRoleAssignLocation*
inv:
// Generated from precondition
before.roles→includes(rPre) and
before.locs→includes(lPre) and
rPre.AssignLoc→includes(lPre) and
// Generated from postcondition
after.roles→includes(rPost) and
after.locs→includes(lPost) and
rPost.AssignLoc=rPre.AssingLoc→excluding(lPost) and
// Unchange object configuration
after.roles→excluding(rPost)
=before.roles→excluding(rPre) and
...

3) Generating an LRBAC Alloy Model: The generated LRBAC snapshot model is transformed to an Alloy model using the transformation algorithm outlined in Section III-A. For example, the Alloy predicate $Role\_DeleteRoleAssignLocation$ partially shown below is generated from the transition class $Role\_DeleteRoleAssignLocation$ in the snapshot model:

pred *Role_DeleteRoleAssignLocation[disj before,after :Snapshot, rPre, rPost:Role, lPre, lPost:Location]* {
after.OperID = ID_DeleteRoleAssignLocation
// Precondition
rPre in before.roles
lPre in before.locs
lPre in rPre.(before.AssignLoc)
// Postcondition
rPost in after.roles
lPost in after.locs
rPost.(after.AssignLoc) = rPre.(before.AssignLoc) - lPost
// Unchanged objects
after.roles - rPost = before.roles - rPre
after.locs - lPost = before.locs - lPre
...
// Unchanged links
after.UserLoc = before.UserLoc
after.ObjLoc = before.ObjLoc
... }

4) Analyzing the Alloy Model: To analyze the Alloy model, the property-to-verify is first transformed to an Alloy predicate (the verification predicate):

pred valid2invalid{
// Specify that the first snapshot is valid
let first = SnapshotSequence/first | first.OperID = ID_Null
all u:first.users | u.(first.UserAssign) = none and u.(first.UserLoc) != none
all ses:first.sessions | ses.(first.SesRole) = none
all r:first.roles | r.(first.ActivateLoc) = none and r.(first.AssignLoc) = none
// Query whether there exists a path from a valid state to an //invalid state

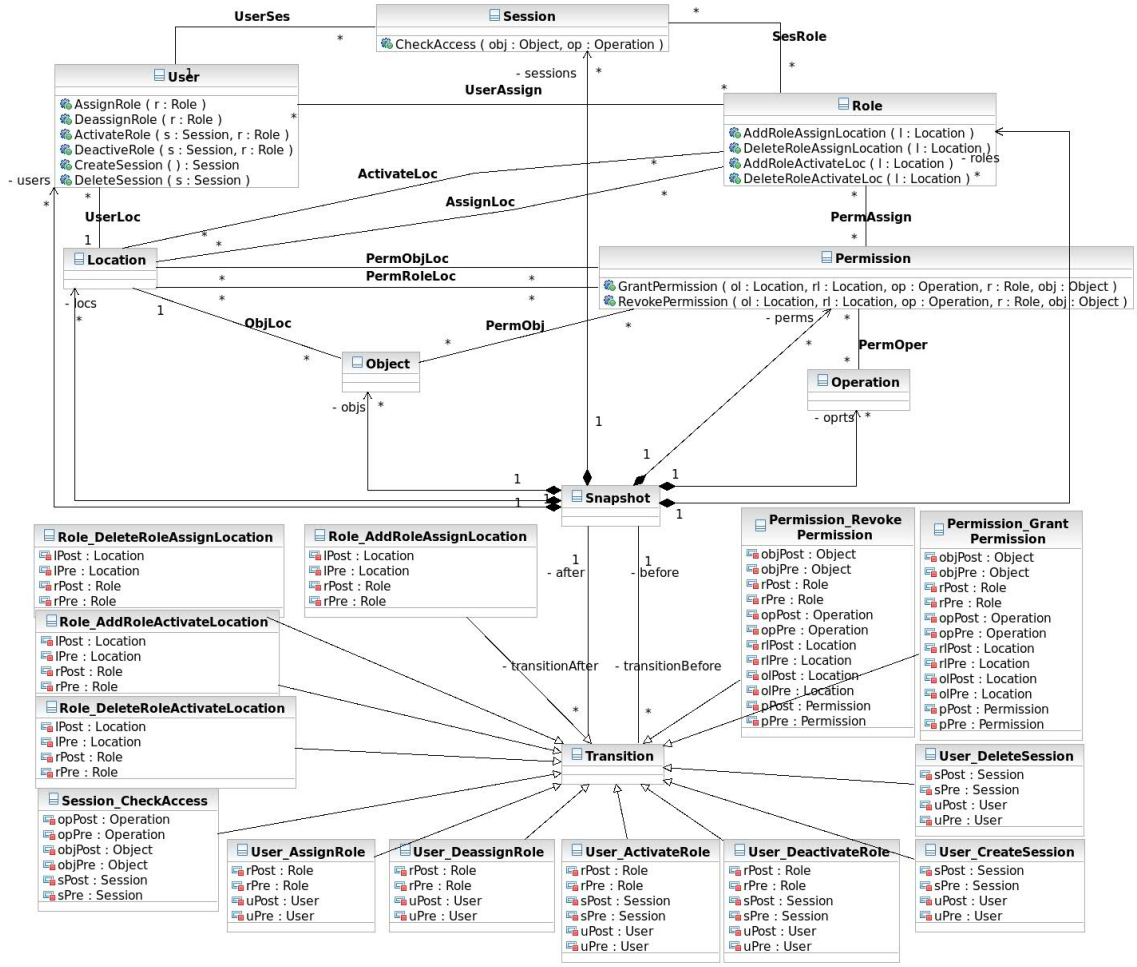


Figure 7: LRBAC Snapshot Model

some s: Snapshot - first | some r:s.roles | some u:s.users | some l:s.locs | l not in r.(s.AssignLoc) and r in u.(s.UserAssign) and l in u.(s.UserLoc)
}

The Alloy Analyzer uses the verification predicate to query whether there exists a reachable path from a valid snapshot to an invalid snapshot, and returns an instance that satisfies the predicate. In this case study, the Analyzer did find a path. Figure 8 shows a sequence of snapshots produced by the Analyzer for the verification predicate. The first snapshot in Fig. 8a shows that in the initial state user $User1$ is in location $Location3$, and has not been assigned any roles. Note that since this is the start state, $OperID$ has the value ID_Null. The second snapshot in Fig. 8b shows that location $Location3$ has been assigned to role $Role$. $OperID$ is $ID\_AddRoleAssignLocation$, indicating that operation $AddRoleAssignLocation$ caused the transition from the first snapshot to the second snapshot. The third snapshot in Fig. 8c shows that role $Role$ has been assigned to user $User1$. $OperID$ is $ID\_AssignRole$, indicating that operation $AssignRole$ caused the transition. The fourth snapshot in Fig. 8d is invalid since the link between role $Role$ and location $Location3$ has been removed. $OperID$ indicates that operation $DeleteRoleAssignLocation$ caused the transition from the third snapshot to the fourth snapshot. The Alloy instance describing this sequence of snapshots is transformed to a UML object diagram describing snapshot transitions using the algorithm. Space does not allow us to show the object model that was produced.

The analysis results suggests that the operation specification for $DeleteRoleAssignLocation$ may need to be strengthened since it allowed a transition from a valid snapshot to an invalid one. To improve the model, we strengthened the precondition of the operation by adding a clause that allows a role to be removed from a set of roles associated with a location only if the role is not assigned to any users. When the LRBAC model with this

modified operation specification is analyzed at the back-end by the Alloy Analyzer, no instance satisfying the verification predicate is found within a reasonably bounded scope.

IV. Related Work

USE [5] and OCLE [4] can be used to analyze structural properties of policy models, but provide very poor support for analyzing functional properties. The ModelRun tool [15] allows interactive verification of OCL properties and can load the UML model from the files created by other tools such as Rose 2000. However, like USE and OCLE it does not support analysis of operation specifications.

Schaad et. al. [14] used Alloy to formally analyze role-based access control policies. Samuel et. al. [12] proposed a framework for specification and verification of generalized spatio-temporal role-based access control model using Alloy. However, Alloy requires a user to have a good background in relational logic.

The analysis approach presented in this paper builds upon the work of Kyriakos [1] and Bordbar [3] on UML2Alloy to provide support for analyzing sequences of operation executions.

V. Conclusion and Future work

In this paper, we described an approach to formally analyzing functional aspects of security policy models expressed in UML. The approach involves transforming the UML models to Alloy models with traces to support analysis of sequences of operation executions. The results reported by the Alloy Analyzer are then transformed into object diagrams in UML that demonstrate how the property being verified has been violated. We applied the approach to an LRBAC specification to demonstrate how security developers can use the approach to detect errors in policy models.

A limitation of the approach is that it is restricted to analyzing functional behavior. We are currently exploring how we can extend the approach at both the front-end and back-end to support lightweight analysis of behaviors that enforce the policies. Specifically we are investigating how safety and liveness access control properties can be analyzed using model-checking techniques at the back-end in a usable manner.

References

[1] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. *Model Driven Engineering Languages and Systems*, pages 436–450, 2007.

[2] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On challenges of model transformation from UML to Alloy. *Software and Systems Modeling*, 9(1):69–86, 2010.

[3] B. Bordbar and K. Anastasakis. UML2Alloy: A tool for lightweight modelling of Discrete Event Systems. In *IADIS International Conference in Applied Computing*, volume 1, pages 209–216. Citeseer, 2005.

[4] D. Chiorean et al. Object constraint language environment (OCLE), version 2.02, 2004.

[5] M. Gogolla, F. Buttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1-3):27–34, 2007.

[6] D. Jackson. Software Abstractions: Logic, Language, and Analysis. *The MIT Press*, 2006.

[7] J. Jurjens. UMLsec: Extending UML for secure systems development. *The Unified Modeling Language*, pages 1–9, 2002.

[8] I. Ray and M. Kumar. Towards a location-based mandatory access control model. *Computers & Security*, 25(1):36–44, 2006.

[9] I. Ray, M. Kumar, and L. Yu. LRBAC: A location-aware role-based access control model. *Information Systems Security*, pages 147–161, 2006.

[10] I. Ray and L. Yu. Short paper: Towards a location-aware role-based access control model. In *Security and Privacy for Emerging Areas in Communications Networks, 2005. SecureComm 2005. First International Conference on*, pages 234–236. IEEE, 2006.

[11] M. Richters. The USE tool: A UML-based specification environment, 2001. *Internet: http://www. db. informatik. uni-bremen. de/projects/USE*, 20:133–147.

[12] A. Samuel, A. Ghafoor, and E. Bertino. A framework for specification and verification of generalized spatio-temporal role based access control model. Technical report, Citeseer, 2007.

[13] RS Sandhu, EJ Coyne, HL Feinstein, and CE Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[14] A. Schaad and J.D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 13–22. ACM, 2002.

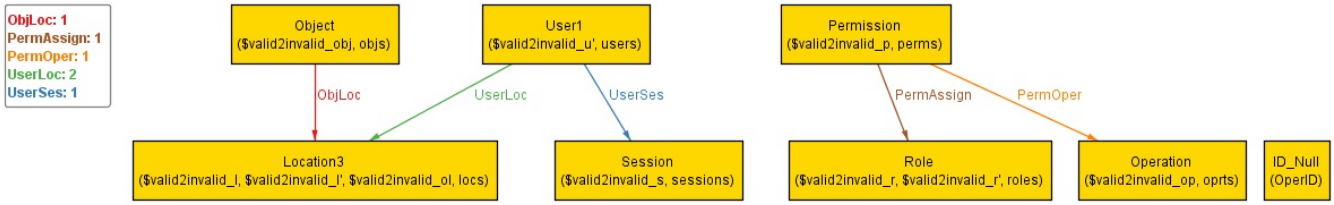
[15] A. Toval, V. Requena, and J.L. Fernandez. Emerging OCL tools. *Software and Systems Modeling*, 2(4):248–261, 2003.

[16] J. Warmer and A. Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

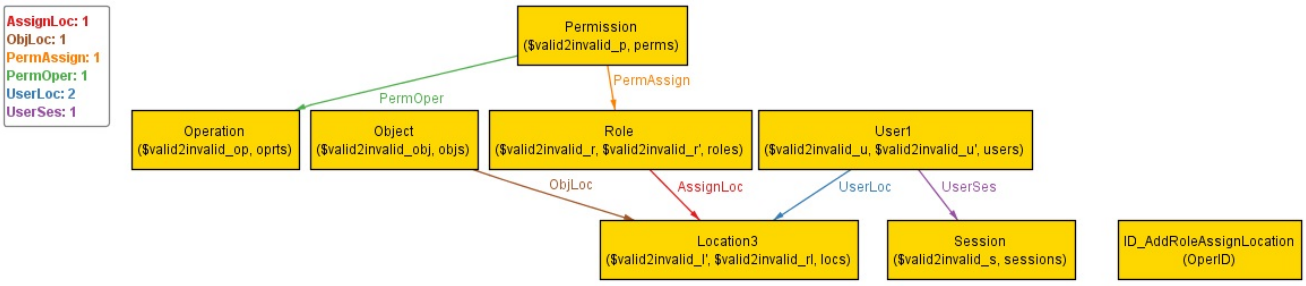
[17] L. Yu, R. France, and I. Ray. Scenario-Based Static Analysis of UML Class Models. *Model Driven Engineering Languages and Systems*, pages 234–248.

[18] L. Yu, R. France, I. Ray, and S. Ghosh. A Rigorous Approach to Uncovering Security Policy Violations in UML Designs. In *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*, pages 126–135. IEEE, 2009.

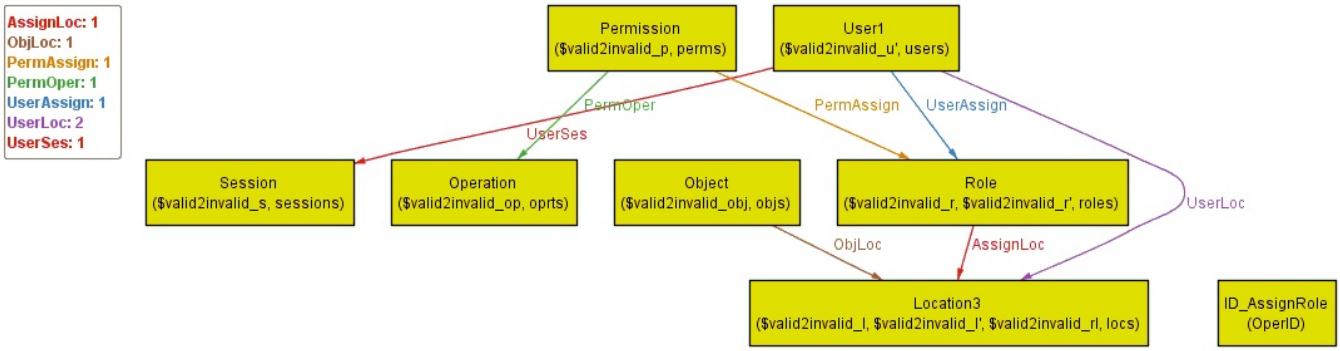
[19] L. Yu, RB France, I. Ray, and K. Lano. A light-weight static approach to analyzing UML behavioral properties. In *12th IEEE International Conference on Engineering Complex Computer Systems, 2007*, pages 56–63, 2007.



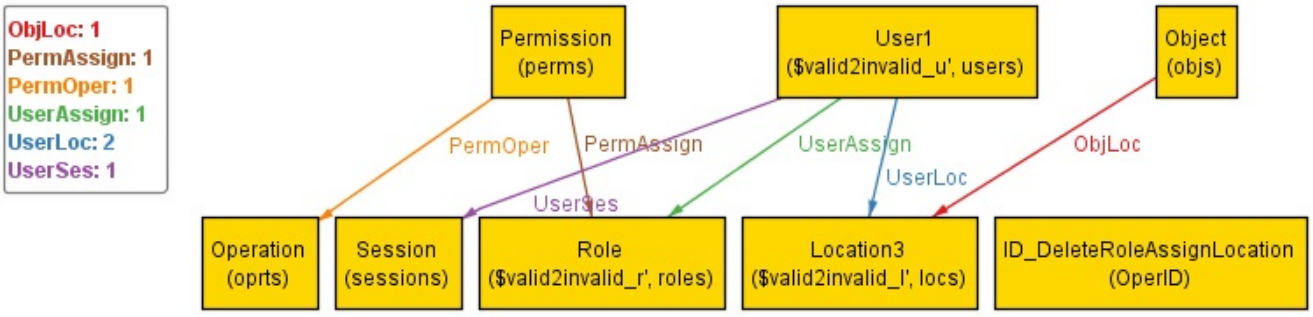
(a) First Snapshot: Valid



(b) Second Snapshot: Valid



(c) Third Snapshot: Valid



(d) Fourth Snapshot: Invalid

Figure 8: From Valid Snapshot to Invalid Snapshot