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 UML-to-Alloy and Alloy-to-UML transformations obviate the need for security designers to understand 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. 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.

While there are many languages for expressing software security policies (e.g., XACML [12], OWL/RDF [6], CIM-SPL [11], and PONDER [5]), software developers using Model-Driven Engineering (MDE) approaches are more likely to use the de-facto standard modeling language, the Unified Modeling Language (UML). In this paper we present a UML-based approach to analyzing access control policies. In the approach, structural aspects of access control policies are expressed in terms of classes and relationships among classes, while functional aspects (i.e., access control operations) are described by operation specifications.

Tools such as USE [16] 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, but these tools are either based on specialized forms of UML models that developers must become familiar with in order to use effectively (e.g., see [9]), or require developers to use behavioral models such as statemachines to specify their properties (e.g., see [10]), even when the properties can be more conveniently expressed in class models.

Alloy [8] is a formal specification language that has been used to specify security policies (e.g., see [17], [19]). 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. In the approach described in this paper, Alloy is used at the back-end to analyze the access control properties specified in UML class models. The approach allows a security designer to check whether a system can move from a valid to an invalid state as result of a sequence of access control operation calls. If analysis uncovers such a sequence of operation calls then the designer uses the trace information output by the analysis to help find the source of the errors in the access control UML class 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 at the University of Birmingham. The work proposed in this paper extends this prior work by providing support for transforming functional behavior specified in a UML class model to an Alloy model that specifies behavioral traces.

The approach also builds upon our previous work on the Scenario-based UML Design Analysis (SUDA) approach [22], [23]. 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 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?”,
Figure 1: Approach Overview

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) [13][14][15]. 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 [14], 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 manipulate directly. 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 source states, 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 UML-to-snapshot model transformation defined in the SUDA approach [24] is used for this purpose. 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.[24] 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 is an object configuration that represents a state of the system at a particular time. A snapshot transition describes the behavior of an operation in terms of its before and after effect on the system state. It consists of a before snapshot, an after snapshot, 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 simple 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., it is 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 execution of the operation ($bPre$) and the other specifies the parameter’s state after 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 specify the before and after snapshots that are associated with Transition instances. For example, the operation specification for \( Af(b : B) \):

```plaintext
// Add an object b to a set of bs associated with an // A object Context A::Af(b:B) // Precondition: object b is not linked to the object Pre: self.AB→excludes(b) // Postcondition: object b is linked to the object Post: self.AB = self.AB@pre→including(b)
```

is transformed to the following transition invariant on the Transition class \( Af \):

```plaintext
Context Af
inv:
// Generated from precondition before.aset→includes(aPre) and before.bset→includes(bPre) and aPre.AB→excludes(bPre) and // Generated from postcondition after.aset→includes(aPost) and after.bset→includes(bPost) and aPost.AB = aPre.AB→including(bPost) and // Unchanged parts of object configuration 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 [22][23]. In SUDA, the generated snapshot model and a sequence of snapshot transitions are fed into an OCL analysis tool (e.g., USE [16], OCLE [4]) 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 [8] 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.

Alloy provides a trace mechanism that associates the transitions triggered by operation invocations with states defined by signatures. The trace mechanism uses an ordering type that casts 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 (LR-BAC) model (see Figure 3), proposed by Ray et al. [15] [13] [14], is used in this paper to demonstrate the analysis technique in Section III-C. LR-BAC is an access control model used to protect sensitive information resources based on Role-Based Access Control (RBAC) model [18]. LR-BAC 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 LR-BAC 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 of allowable locations for the objects associated with the permission.

In class Role, operation AddRoleAssignLocation takes a location parameter, and associates the location with a role. Operation DeleteRoleAssignLocation takes a location parameter, and removes the location from a set of locations associated with a role.

Ray et. al. [14] 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:

```oclass
// 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 : setA`), 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, `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 attributes 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 into 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. The 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**

<table>
<thead>
<tr>
<th>Input: UML Snapshot Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Alloy Model</td>
</tr>
</tbody>
</table>

**Algorithm Steps:**

1. Transform each class that is part of `Snapshot` class to a signature in Alloy.
2. Transform the `Snapshot` class to a `Snapshot` signature containing fields that specify the object configurations within a snapshot.
3. Transform each `Transition` specialization to a predicate in Alloy.
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

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 for this transformation is outlined in Algorithm 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.

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 the valid and invalid LRBAC snapshot patterns 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 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 partially 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:

\[
\text{Context Role}_\text{DeleteRoleAssignLocation} \\
\text{inv:} \\
// Generated from precondition 
\text{before.roles} \rightarrow \text{includes(rPre)} \text{ and} \\
\text{before.locs} \rightarrow \text{includes(lPre)} \text{ and} \\
\text{rPre.AssignLoc} \rightarrow \text{includes(lPre)} \text{ and} \\
// Generated from postcondition 
\text{after.roles} \rightarrow \text{includes(rPost)} \text{ and} \\
\text{after.locs} \rightarrow \text{includes(lPost)} \text{ and} \\
\text{rPost.AssignLoc} = \text{rPre.AssignLoc} - \text{lPost} \\
// Unchanged parts of object configuration 
\text{after.roles} - \text{rPost} = \text{before.roles} - \text{rPre} \\
\text{after.locs} - \text{lPost} = \text{before.locs} - \text{lPre} \\
... \\
// Unchanged links 
\text{after.UserLoc} = \text{before.UserLoc} \\
\text{after.ObjLoc} = \text{before.ObjLoc} \\
... \\
\]

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 \text{Role}_\text{DeleteRoleAssignLocation} partially shown below is generated from the transition class \text{Role}_\text{DeleteRoleAssignLocation} in the snapshot model:

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

4) Analyzing the Alloy Model: To analyze the Alloy model, the property-to-verify is first transformed to an
The Alloy Analyzer uses the verification predicate to query whether there exists a path from a valid snapshot to an invalid snapshot, and returns an instance that satisfies the predicate if one is found. 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 outlined in Algorithm 2. Space does not allow us to show the object model that was produced.

The analysis results suggest 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

OCLE [4] can be used to analyze structural properties of policy models, but it provides very poor support for analyzing functional properties. USE [7] [20] [16] generates an object configuration (snapshot) conforming to the class model and evaluates the snapshot against OCL constraints. However, it can only check one snapshot transition at a time, unlike the approach described in this paper that analyzes sequences of snapshot transitions. The ModelRun tool [21] 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 snapshot transition sequences.

Schaad et. al. [19] used Alloy to formally analyze role-based access control policies. Their approach focused on detecting conflicts that could arise out of decentralized administrative actions with respect to the separation constraints. Samuel et. al. [17] proposed a framework for specification and verification of generalized spatio-temporal role-based access control model using Alloy. A security designer who uses these approaches must be familiar with Alloy. Also, if UML is used as the primary modeling language in a development environment, the security designer must manually transform the parts of the UML models that specify policies to Alloy. The approach described in this paper allows a security designer working in a UML-based environment to leverage Alloy’s analytical power without interfacing directly with the Alloy language and analyzer.

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 instances produced by the Alloy Analyzer demonstrate how the property being verified has been violated. These instances are transformed to UML object diagrams which a security designer uses to help locate errors in the UML model. This approach addresses an important usability issue for verification tools: a usable verification tool needs to be integrated with the development processes and notations used by software developers.

The work presented in this paper supports integrated use of a rigorous analysis tool, the Alloy analyzer, and the software modeling language, the UML, to specify and analyze access control policies.

We applied the approach to a LRBAC specification to demonstrate how security developers can use the approach to detect errors in policy models. Since the Alloy Analyzer performs an underlying bounded state-space exploration, we
are developing more access control models as well as user experience studies to evaluate and validate the approach.

A strength of the approach is that it is not limited to specifying and analyzing access control properties, and thus software developers can use the same framework to analyze other functional properties and to analyze the effect access control policies have on other behaviors. One of the objectives of our future work is to produce an analysis framework that allows developers to explore how security policies interact with other desired properties of a software system. We also plan to extend the approach at both the front-end and back-end to support analysis of implementations 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.

ACKNOWLEDGMENT

The work described in this report was supported by the National Science Foundation grant CCF-1018711.

REFERENCES


