authUML: A Three-phased framework to analyze access control specifications in Use Cases

Khaled Alghathbar
George Mason University, Fairfax, VA 22030, USA,
King Saud University, Riyadh, Saudi Arabia
kalghath@gmu.edu

Duminda Wijesekera
Center for Secure Information Systems
George Mason University, Fairfax, VA 22030, USA
dwijesek@gmu.edu

ABSTRACT
Security requirements of a software product need to receive attention throughout its development life cycle. authUML is a framework based on logic programming that analyzes access control requirements in the requirements phase of the life cycle to ensure that they are consistent, complete, and conflict-free. The framework is a customized version of Flexible Authorization Framework (FAF) of Jajodia et al. [9] suitable for Unified Modeling Language (UML) based requirement engineering. Our approach analyzes requirements on two levels: Use Cases and the conceptual operations [19]. authUML specifies policies to prevent inconsistent, incomplete, and conflicting requirements before the developers proceed to the following phases of the development life cycle.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification – Validation.

General Terms
Security, Languages.

Keywords
Access control policies, Security engineering, Use Cases, and Semiformal methods.

1. INTRODUCTION
Security requirements of a software product need to receive attention throughout its development life cycle. Because security requirements specified at early steps of the software life cycle affect later steps and are likely to feature in the eventual product, it is important to incorporate and analyze it in earlier phases to ensure consistency. However, Devanbu and Stubblebine claim that most security requirements come to light only after functional requirements have been completed [7]. Accordingly, to address this concern, we propose authUML, a framework to formally model and analyze access control requirements within the specification phase of UML based software lifecycle.

In UML, requirements are specified with Use Cases as the first step of the life cycle [3,17]. Use Cases specify actors and their intended usage of the envisioned system. Therefore, access control requirements must some how be introduced into Use Cases of security sensitive software. This is done by specifying which actors are permitted / prohibited from invoking given Use Cases. Sometimes such specifications over-specify or under-specify permissions, leading to inconsistency and incompleteness, respectively. By complete and consistent we respectively mean that the requirements specify any subject either permitted or prohibited, but not both from executing the same Use Case or operation. In order to avoid both extremes, security literature uses conflict resolution and decision meta-policies [13]. Prohibitions taking precedence over permissions are an example of the former, and the closed policy that denies all unspecified permissions is an example of the latter [13].

In addition, in order to avoid fraud and misuse, security models (specially in Role-base Access Control) use application level conflicts such as disallowing an employee to approve her own salary increases – usually referred to as separation of duty principles [6]. Access control requirements modeled by Use Cases can also take such concerns into account. We show how authUML can model and avoid such application specific conflicts.

authUML is based on the Flexible Authorization Framework (FAF) [13] of Jajodia et al., and is an attempt to advance its application to the requirements specification phase of the software development life cycle. Therefore, authUML is a customized version of FAF that is to be used in requirements engineering. Therefore authUML uses similar components of FAF with some modification in the language and the process to suit the Use Case model used in UML. Because FAF specifies authorization modules in computing systems, FAF is invoked per each authorization request. Contrastingly, authUML is to be used by requirements engineers to avoid conflicts and incompleteness of accesses. Therefore, while FAF is used frequently to process each access control request during execution, authUML is to be used less frequently during the requirements engineering phase to analyze the access control requirements. Our ongoing work extends authUML to other phases of the software development life cycle.

authUML uses Prolog style stratified logic programming rules to specify policies that ensure desirable properties of requirements. Because requirements are specified using actors invoking Use Cases, authUML uses predicates to specify which actors (sub-
jects) are permitted or prohibited from invoking any given Use Case. This constitutes the first phase of authUML. During the second phase, authUML allow Use Case and subject hierarchies to be used to allow the benefits of inheriting authorization specifications. As in FAF, authUML resolves inconsistencies, conflicts and incompleteness arising out of this process using policies specified as a finite collection of rules. However, in contrast to FAF, authUML allows policies that ignore conflicts at the discretion of the requirements engineer. During the third phase of authUML, Use Case access permissions are propagated to the conceptual operations that are used to describe the functional aspects of Use Cases. The purpose of this is to allow the application of fine-grain access control policies. Again, authUML allows polices to resolve inconsistencies, incompleteness and application dependent conflicts. Unlike the FAF, rules in Auth-UML can be specified to inform the requirements engineer about the inconsistencies and conflicts that remain unresolved at the end of this phase. This design choice has been made because – as shown in [10,11,16] – some specification methodologies tolerate them.

The remainder of this paper is organized as follows. Section 2 summarizes conceptual operations that are used to describe Use Cases, and a summary of FAF. Section 3 presents the process of applying authUML. Section 4 describes the syntax and semantics of authUML. Section 5, 6 and 7 describe the first, second and third major phases of authUML respectively. Section 8 discusses related work and section 9 concludes the paper.

2. BACKGROUND
2.1 Conceptual operations
Operation schemas introduced by Sendall and Strohmeier [19] enriches Use Cases by introducing conceptual operations and specifying their properties using Object Constraints Language (OCL) syntax [21]. Operation schema specifies operations that apply to the whole system to be taken as one entity. One of the advantages of operation schemas is that they can be directly mapped to collaboration diagrams that are used later in the analysis and design phases of the software development life cycle. This paper is based on the premise that high-level access control policies should also be applied at this level of detail.

2.2 The Flexible Authorization Framework of Jajodia et al. [13]
The Flexible Authorization Framework (FAF) of Jajodia et al [13] uses a logic-programming style stratified rule based to specify accesses. Despite there being other authorization models [8], FAF is considered flexible because it is not based on any meta-policies and therefore can specify multiple access control policies on the same server.

FAF is based on four steps that are applied in a sequence. They are: propagating authorizations using rules, resolving authorization conflicts, deciding on either to grant or to deny each authorization request and enforce integrity constraints. In the first step, some basic facts, such as, authorization subject, object hierarchies (for example, directory structures) and a set of authorizations along with rules to derive additional authorizations are given. The intent of this step is to use structural properties to derive permissions, called propagation policies. Although propagation policies are flexible and expressive, they may result in over specification. That is, rules could be used to derive permissions and prohibitions. To avoid such inconsistencies FAF uses conflict resolution policies comprising its second step. At the third step, decision policies are applied to ensure the completeness of authorizations, where a decision will be made to either grant or deny (but not both) every access request. This is necessary, as the framework makes no assumptions with respect to undeterminable authorizations. The last step consists of checking for integrity constraints, where all authorizations that violate integrity constraints will be denied. In addition, FAF ensures that every access request is either honored or rejected, thereby providing a built-in completeness property.

3. authUML PROCESS
authUML consists of three main phases where each consists of several steps. As shown in Figure 1, authUML takes authorizations from requirements specifications and then analyzes them to produce complete, consistent and conflict-free authorizations. The three phases of authUML are:
1. Processing access control requirements.
2. Ensuring consistency, completeness and conflict-free accesses to Use Cases.
3. Ensuring consistency, completeness and conflict-free accesses for operations.

The first phase starts with a set of access control requirements, where they are transformed into a unified representation in the form of access predicates. Also, the first phase ensures that all specified accesses are consistent and conflict-free. The second phase ensures that all accesses specified for Use Cases are consistent, complete and conflict-free, without considering the operations used to describe their functionality. During the third phase authUML analyzes the access control requirements on operations. All three phases consists of rules that are customizable to reflect policies used by the security requirements engineers.

From the access control requirements that are provided in the form of authUML predicates, the second phase propagates accesses based on subject and/or object hierarchies. Any inconsistencies that may occur due to such propagated accesses are re-solved using conflict resolution rules. After this, all accesses that are explicit (i.e. given directly in requirement) or implicit (de-rected) are consistent, but may not be complete. That is, not all accesses for all subjects and Use Cases may be specified. Therefore, using predefined rules and policies (i.e. closed or open policies) the next step (5 in Figure 1) completes them. Therefore, accesses specified before step 6 are directly obtained from requirements, propagated due to hierarchy or consequences of applying decision policies. Thus, it is necessary to validate the consistency of the finalized accesses against the original requirements and to check for conflicts between them. If authUML finds any inconsistency or conflict among accesses at
Figure 1. authUML architecture

Authorization Requirements contain:
- Use Case-Operation-Object Relations
- Subject Hierarchy
- Conflict Sets
- Accepted conflicts
- Subject Authorization
this step it will notify the requirement engineer in order to fix it and run the analysis again.

The third phase of authUML applies the same process to operations used to describe Use Cases. This phase does not have a decision step as in the second phase, because each Use Case propagates its accesses to all its operations. As a result, accesses specified during this phase are complete. In addition, access specifications of operations at the end of this phase are consistent because inconsistency resolution step in the operation level will attempt and resolve all inconsistencies. But, if it cannot do so, the process will stop and notify the requirement engineer about the inconsistency to be fixed by manual intervention. Up to this step, accesses are consistent and complete, but may not be free of application specific conflict. Thus, the purpose of the last step of this phase is to detect those conflicts.

There is a difference between the access specifications fed into authUML and those that come out of it. That is, finalized access specifications are consistent, complete and free of application specific conflicts. This outcome is the main advantage of our work. Thus, authUML focuses on access control requirements as early as possible to avoid any foreseeable problems before proceeding to other phases of the development life cycle. As the development process proceed through its life cycle, changes of access control requirement may occur. For example, Use Cases may be changed to invoke different operations, or refined/new operations may be added. Consequently, already accepted accesses may need to be re-analyzed. Therefore, it is necessary to go back and run the authUML again to preserve the consistency, complete and conflict-free accesses. Thus, our framework is flexible enough that allows changes in access control specifications.

The architecture of authUML differs from the architecture of FAF in two aspects. Firstly, authUML analyzes accesses in two levels, Use Cases and operations in order to scrutinize accesses in coarse-grain and fine-grain levels, respectively. Secondly, steps 2, 6 and 9 are introduced in authUML to detect inconsistencies and conflicts between different levels of accesses that are absent in FAF. Also, authUML receives a bulk of access control requirements but not just one access request at a time. Thus, as we will show later, authUML produces accesses only if there are sufficient rules to resolve all application level conflicts.

4. authUML SYNTAX AND SEMANTICS

4.1 Individuals and terms of authUML

Individuals of authUML are Use Cases, operations, objects and subjects. Use Cases specify actors and their intended usage of the envisioned system. Such usage - usually, but not always - is specified in terms of the interactions between the actors and the system, thereby specifying the behavioral requirements of the proposed software. Each Use Case consists of set of operations that are used to describe the Use Case. Each operation operates on an object, and operations are the only way to query or manipulate objects. Subjects are permitted to invoke a set of Use Cases and thereby all operations describing that Use Cases. We use subjects as actors in UML or role in Role-based Access control (RBAC) [18, 19]. We denote UC, OP, OBJ, and S as set of Use Cases, operations, objects and subjects respectively. An access permission can be permitted or prohibited, that is, modeled as a positive or a negative action, respectively. authUML syntax is built from constants and variables that belong to four individual sorts. Namely, signed Use Cases, signed operations, (unsigned) objects and (unsigned) subjects. They are represented respectively as ±Xuc, ±top, obj, and s, where variables are represented as ±Xuc ± xop ± Xobj and Xs.

4.2 Predicates of authUML

We use FAF predicates with some customizations and some new predicates to model requirements as follows:

Following predicates are used to model structural relationships and called rel-predicates.

1. A binary predicate UC_OP(Xop,Xuc) means operation Xop is invoked in Use Case Xuc.
2. A binary predicate OP_OBJ(Xop,Xobj) means operation Xop belongs to object Xobj.
3. A binary predicate before(Xop,X’op) means that Xop must be invoked before X’op.
4. A ternary predicate inUCbefore(Xuc,Xop,X’op) means Use Case Xuc invokes Xop before X’op.

Following predicates are used to model hierarchies and called hie-predicates.

1. A binary predicate in(Xs,X uc), means Xs is below X uc in the subject hierarchy.
2. A binary predicate dirin(Xs,X uc) mean Xs is directly below X uc in the subject hierarchy.

Following predicates are used to model conflicts and called con-predicates.

1. A binary predicate conflictingSubject(Xs,X uc, X s’) means subject Xs and X s’ are in conflict with each other.
2. A binary predicate conflictingUC(Xuc,X uc’, X uc) means that Use Cases Xuc and X uc’ are in conflict with each other.
3. A binary predicate conflictingOP(Xop,X uc, X op) means operations Xop and X op are in conflict with each other.
4. A ternary predicate ignore(X,Y,Y’) represents an explicit instruction by the requirements engineer to ignore a conflict among X,Y and Y’ where X, Y and Y’ are either subjects, operations or Use Cases.

The following predicates are used in the first phase of authUML to authorize, detect assignment conflict or detect inconsistency in the access control requirements:

1. opInConUC(Xop,X uc,X’op,X uc’) means Xop is an operation in two conflicting Use Cases Xuc and X uc’ and opInConUC(Xop,X uc,X’op,X uc’) means that Xop and X’op are two conflicting operations in Use Case X uc, and flowConInUC(XUC,XOP,X UC’OP) means that Xop and X’op are invoked in a way that violate execution order.
2. A binary predicate candoUC where candoUC (Xs±X uc) means subject Xs can or cannot invoke the Use Case X uc depending on the sign of X uc, positive (+) or negative (−).
3. A binary predicate alertReq(Xs,X uc) to inform the requirements engineer that there is either an inconsistency (between access control requirements) or a conflict (between subjects or Use Cases) on the access of Xs on X uc.
Table 1. Rules defining predicate

<table>
<thead>
<tr>
<th>Phase</th>
<th>Stratum</th>
<th>Predicate</th>
<th>Rules defining the predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 1</td>
<td>0</td>
<td>rel-predicates, hie-predicates, con-predicates, ignore(X,Y,Y')</td>
<td>base relations.</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>opInConUC(X_{op},X_{UC},X'<em>{UC}), conOpInUC(X</em>{op},X'<em>{op},X</em>{UC}), flowConInUC(X_{UC},X_{op},X'_{op})</td>
<td>body may contain hie, ignore, rel-predicates.</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>candoUC(X_{uc} \pm X_{uc})</td>
<td>body may contain hie-, con- and rel-predicates.</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>alertReq(X_{s}, X_{uc})</td>
<td>body may contain literal from strata 0 to 2</td>
</tr>
<tr>
<td>Phase 2</td>
<td>4</td>
<td>overUC(X_{s}, X'<em>{s}, X</em>{uc})</td>
<td>body may contain literals from strata 0 to 3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>dercandoUC(X_{s} \pm X_{uc})</td>
<td>body may contain predicates from strata 0 to 4. Occurrences of dercandoUC must be positive.</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>doUC(X_{s}, X_{uc})</td>
<td>body may contain predicates from strata 0 to 5.</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>doUC(X_{s}, X_{uc})</td>
<td>body contains one literal (\neg do_{UC}(X_{s}, \pm X_{uc})).</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>alertUC(X_{s}, X_{uc})</td>
<td>body may contain literal from strata 0 to 7</td>
</tr>
<tr>
<td>Phase 3</td>
<td>9</td>
<td>dercandoOp(X_{s} \pm X_{op})</td>
<td>body may contain predicates from strata 0 to 7. Occurrences of dercandoOp must be positive.</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>doOp(X_{s}, X_{op})</td>
<td>body may contain predicates from strata 0 to 9.</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>doOp(X_{s}, X_{op})</td>
<td>body contains one literal (\neg do_{Op}(X_{s}, X_{op})).</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>cannotReslove(X_{s}, X_{uc}, X'<em>{uc}, X</em>{op})</td>
<td>body may contain literal from strata 0 to 11</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>alertOP(X_{s}, X_{op})</td>
<td>body may contain literal from strata 0 to 11</td>
</tr>
</tbody>
</table>

Following predicates are used in the second phase of Auth-UML to authorize, detect conflicts and inconsistencies at the Use Case level:

1. A ternary predicate \(\text{over}_{UC}(X_s, X'_{s}, X_{uc})\) meaning \(X_s\) is permission to invoke \(\pm X_{uc}\) overrides that of \(X'_{s}\).
2. A binary predicate \(\text{dercando}_{UC}\) with the same argument as \(\text{cando}_{UC}\). \(\text{dercando}_{UC}(X_s, \pm X_{uc})\) is a permission derived using modus ponens and stratified negation [2].
3. A binary predicate \(\text{do}_{UC}\). where \(\text{do}_{UC}(X_s, \pm X_{uc})\) is the final permission/prohibition for subject \(X_s\) to invoke Use Case \(X_{uc}\) depending on if the sign of \(X_{uc}\) is + or −.
4. A binary predicate \(\text{alert}_{UC}(X_s, X_{uc})\) to inform the requirements engineer that there is an inconsistency (between the access control requirement and the final outcome of this phase) or conflict (between subjects or use cases) on the accesses that involve \(X_s\) and \(X_{uc}\).

Following predicates are used in the third phase of Auth-UML to authorize, detect conflicts and inconsistencies at the operation level:

1. A binary predicate \(\text{dercando}_{Op}(X_s, \pm X_{op})\) is similar to \(\text{dercando}_{UC}\) except the second argument is an operation instead of a Use Cases.
2. A binary predicate \(\text{do}_{Op}(X_s, \pm X_{op})\) is similar to \(\text{do}_{UC}\) but the second argument is an operation.
3. \(\text{cannotReslove}(X_s, X_{uc}, X'_{uc}, X_{op})\) is a 4-ary predicate representing an inconsistency that can not be resolved at the operation level with the given rules.

4. A binary predicate \(\text{alert}_{OP}(X_s, X_{op})\) to inform the requirements engineer that there is a conflict between subjects or operations on the authorization that involve \(X_s\) and \(X_{op}\).

Assumptions

- The subject we used refers to a role (as in RBAC) or an actor (in UML) and not to end user of a software system. The role is a named set of permissions and users may assume a role in order to obtain all of its permissions. We are working on defining a stage that controls user to roles assignment in order to avoid any conflicts.
- Every Use Case must have at least one operation (i.e. \(\forall x \in UC \exists y \in OP \ UC_{OP}(x,y)\)) and that every operation must belong to one and only one object (i.e. \(\forall x \in OP \exists y \in OB \rightarrow UC_{OP}(x,y)\)).
- Each positive access of a Use Case to a subject means that all operations of that use case are also positively authorized to the same subject. This is consistent with [20]. Conversely, a prohibited Use Case to a subject must have at least one prohibited operation to that subject.

As already stated, \(\text{cando}\) represents an access permission obtained from requirements and \(\text{dercando}\) represents an access derived using (to be described shortly) rules. Both \(\text{cando}\) and \(\text{dercando}\) do not represent a final decision, but only an intermediate result. For example, although \(\text{cando}_{UC}(X_s, X_{uc})\) is obtained from requirements does not mean that subject \(X_s\) will be allowed to finally execute Use Case \(X_{uc}\). The reason being that propagation, conflict resolution and decision policies may change the authorization expressed in \(\text{cando}_{UC}(X_s, X_{uc})\). However,
**4.3 Rule of authUML**

An authUML rule is of the form \( L \leftarrow L_1, \ldots L_n \) where \( L \) is a positive literal and \( L_1, \ldots L_n \) are literals satisfying the conditions stated in Table 1.

An Example:

\[
\text{candoUC}(\text{supervisor}, +\text{authorize payment}) \leftarrow \text{doUC}(\text{Xs}, +\text{Xuc}) \leftarrow \text{in}(\text{Xs}, \text{X's})
\]

Rule 1 says that supervisor can access Use Case “authorize payment”. Rule 2 specifies the inheritance of authorizations in the subject hierarchy. Rule 3 expresses the permissions take precedence policy of resolving conflicts.

**4.4 authUML Semantics**

Table 1 shows the stratification of rules used in authUML. Rules constructed according to these specifications forms a local stratification. Accordingly, any such rule based form has unique stable model and that stable model is also a well-founded model, ala Gelfond and Lifschitz [12]. As done in FAF, we can materialize authUML rules also, thereby making the authUML inference engine efficient.

**5. PHASE I: ANALYZING INFORMATION IN REQUIREMENTS DOCUMENTS**

This section goes through steps 1 and 2 of authUML and shows how the authUML processes the access control requirements.

**5.1 Representing authorizations**

Following [13], we assume that requirement engineers already specify access control requirements and it is not in the scope of this paper to go further on that. Authorization requirements consist of:

1. Permissions for subject to invoke Use Cases
2. Subject hierarchy.
3. Structural relationships (Use Case - Operation - Object Relations).
4. Conflicting subjects, Use Cases and operations sets.
5. Conflicts of interest.

All of the above must be written in this step in the form of AuthUML rules in order to be used during subsequent steps. They are represented as follow:

1. At this step access permissions are written in the form of \( \text{candoUC} \) rules representing explicit authorization obtained from the requirement specification. Rule 4 and 5 are examples:

\[
\text{candoUC}(\text{clerk}, +\text{recordInvoiceArrival}) \leftarrow \text{Rule (4) permits the clerk to invoke the “recordInvoice-Arrival” Use Case. Rule (5)}
\]

\[
\text{candoUC}(\text{supervisor}, +\text{alert}) \leftarrow \text{Rule (5) prohibits the supervisor to invoke the “writeCheck” Use Case.}
\]

2. Subject hierarchy is represented using the \( \text{in} \) predicate to indicate which subject inherits what. For example, \( \text{in(purchasingOfficer,clerk)} \) means that the purchasing officer is a specialized subject of clerk that inherits all its permissions.

3. Structural relationships represent the relations between use case and its operations, operations and its object and the flow between operations in a use case. \( \text{UC_OP(Xuc,Xop)} \) says that \( \text{Xop} \) is an operations invoked in the Use Case \( \text{Xuc} \). \( \text{OP_OBJ(Xop,Xobj)} \) says that operation \( \text{Xop} \) belongs to object \( \text{Xobj} \). In addition, \( \text{inUC-before(Xuc,Xop,X'op)} \) means that \( \text{Xop} \) must be executed before \( X'op \) is executed and \( \text{inUC-before(Xuc,Xop,X'op)} \) means that Use Case \( Xuc \) calls for executing \( X'op \) before \( X_op \).

4. Application definable conflicts occurring among subject, Use Case and operations are represented respectively by \( \text{conflictingSubjects(Xs,X's)} \), \( \text{conflictingUC(Xuc,X'uc)} \) and \( \text{conflictingOP(Xop,X'op)} \).

5. Requirement engineers may decide to accept some conflicts as in [10,11,16]. authUML uses \( \text{ignore}(X,Y,Y') \) to accept a conflict between \( Y \) and \( Y' \). The main goal of the \text{ignore} predicate is to only allow specified conflicts, but not others between access.

**5.2 Ensuring consistent and conflict-free access control specifications**

Access control requirements may specify inconsistencies where one requirement permits and another requirement denies the same permission. In addition, two conflicting subjects may be permitted to invoke the same Use Case or operation, or a subject may be permitted to invoke two conflicting Use Cases or operations. Latter kinds of permissions may violate the Separation of Duty principle [6].

In small systems, discovering conflicts can be easy, because of the small number of entities and engineers writing those requirements. However, detecting conflicts and inconsistencies between access control requirements in large system is more problematic. Therefore, authUML can specify rules that detect inconsistencies between the requirements that are specified by many security engineers. Detecting inconsistencies and conflicts at this stage prevent them from spreading to the following stages of the life cycle. This step of authUML takes access control requirements in the form of \( \text{cando} \) rules and automatically applies inconsistency and conflict detection rules to identify their existence, as follow:

\[
\text{alert}(X,X') \leftarrow \text{candoUC}(X,X'), \text{candoUC}(X',X'), \text{conflictingUC}(X,X'), \text{conflictingOP}(X,X'), \text{ignore}(X,X'), \text{ignore}(X',X')
\]

\[
\text{alert}(X,X') \leftarrow \text{candoUC}(X,X'), \text{conflictingUC}(X,X'), \text{conflictingOP}(X,X'), \text{ignore}(X,X'), \text{ignore}(X',X')
\]

\[
\text{alert}(X,X') \leftarrow \text{candoUC}(X,X'), \text{conflictingUC}(X,X'), \text{conflictingOP}(X,X'), \text{ignore}(X,X'), \text{ignore}(X',X')
\]

\[
\text{alert}(X,X') \leftarrow \text{candoUC}(X,X'), \text{conflictingUC}(X,X'), \text{conflictingOP}(X,X'), \text{ignore}(X,X'), \text{ignore}(X',X')
\]

\[
\text{alert}(X,X') \leftarrow \text{candoUC}(X,X'), \text{conflictingUC}(X,X'), \text{conflictingOP}(X,X'), \text{ignore}(X,X'), \text{ignore}(X',X')
\]
Rule 6 says that if there are two requirements where one grants and the other denies the invocation of the same Use Case to the same subject then an alert message will be raised to the security engineer that identifies those that lead to the inconsistency. Rule 7 says that if a subject is permitted to invoke two conflicting Use Cases that are not explicitly allowed by the ignore predicate, then an alert message is triggered in order to facilitate manual intervention. Rule 8 says that if a Use Case is permitted to be invoked by two conflicting subjects, then a manual intervention need to be sought. Rule 9 and 10 are related to the conflicting assignments of operations to Use Cases. Rule 9 detects having operations in two conflicting Use Cases and rule 10 detects having two conflicting operations in the same Use Case. Rule 11 says that if two operations used in one Use Case violate the order in which they are to be called. The first two conflicts can be ignored if the requirement engineer explicitly uses the “ignore” predicate.

Notice that detectable conflicts that appear at this step are structural in nature. That is, they are conflicts or inconsistencies independent of the permissions or prohibitions assigned to execute them.

6. PHASE II: APPLYING POLICIES TO USE CASES

Previous phase analyzes statically given access control requirements without using any policies and produce consistent and conflict-free accesses. This phase (steps 3, 4, 5 and 6) applies policies that are specified using authUML rules relevant to Use Cases. Such policies may add new permissions or change existing ones.

6.1 Propagation Policies

Most systems use some hierarchies to benefit from inheritance. This step may generate new permissions according to chosen propagation policies. All explicit or derived permissions are transformed to the form of derived rules (derived authorizations). Some examples of propagation policies are listed in [13] and represented as authUML rules in Table 2.

6.2 Inconsistency Resolution Policies

In complex systems with many Use Cases, permission propagation may introduce new permissions that in turn may result in new inconsistencies. Inconsistency resolution policies resolve such inconsistencies. Examples are listed in [13] and represented as authUML rules in Table 3. The rules in Table 3 define inconsistency resolution policies. For example, for denial take precedence with open policy, if there are no denial then permission if granted for such subject. However, in case of closed policy, the previous definition is not enough, as there must be a permission in the absence of a prohibition. The last rule completes the rule base prohibiting every access that is not permitted.

6.3 Decision Policies

Decision policies complete authorizations so that every subject must have either permission or a prohibition to execute each Use Case and operation. Following are some decision policies that have been suggested:

Closed Policy: Accesses without permissions are prohibited.
Open Policy: Accesses without prohibitions are permitted.

This is the last step that finalizes all accesses of Use Cases to subjects that are consistent with each other and complete. They and written in the form of derived rules. authUML like FAF ensure the completeness of access control decision by enforcing the following.

6.4 Alerting the requirements engineer of changes to Use Case accesses

As stated, final accesses of the last step are consistent with each other, but it may have changed the original requirements. Also, there may not be sufficient rules to resolve application specific conflicts. This step uses the alertUC predicate to inform the requirements engineer of such changes or problems.

Once informed by authUML the requirements, the engineer can revisit potential problems, and hopefully resolve them before proceeding to apply fine grain policies that specify operation level accesses.
7. PHASE III: APPLYING POLICIES TO OPERATIONS

The previous phase produces consistent and conflict-free Use Cases. This phase (step 7, 8 and 9) analyzes operations to ensure consistent, conflict-free and complete permissions to invoke operations.

7.1 Propagating Permissions to Operations

This phase applies fine grain access control policies to operations. Recall that Use Cases are described using operations and some additional closure rule.

7.2 Inconsistency resolution for operations

Because an operation can be called on behalf of more than one Use Case, and thus can inherit permissions from more than one Use Case, applying rules such as (14) may introduce conflicts. Therefore, conflict resolution must be applied to operations. as we stated it before, we assume that each positive permission of a Use Case is inherited by all its operations. Conversely, a prohibited Use Case must have at least one prohibited operation.

An operation may be called in two Use Cases with contradicting permissions for the same subject, resulting from that, subject will have been granted a permission and a prohibition to execute the same operation. One policy that can resolve this contradictory situation is to retain the permission to execute the operation for the subject only if another operation belonging to the prohibited Use Case already has a prohibition for the same subject. In doing so, we preserve the assumption that as long as there is at least one prohibition on operation for a subject in a Use Case, then that Use Case has a prohibition for the same subject. Rule 15 specifies this conflict resolution policy as an authUML rule:

Table 2. Rules for enforcing propagation policies for use cases on subject hierarchy.

<table>
<thead>
<tr>
<th>Propagation policy</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>No propagation</td>
<td>dercando_{UC}(X_s+X_{uc}) ← cando_{UC}(X_s+X_{uc})</td>
</tr>
<tr>
<td></td>
<td>dercando_{UC}(X_s- X_{uc}) ← cando_{UC}(X_s- X_{uc})</td>
</tr>
<tr>
<td>No overriding</td>
<td>dercando_{OP}(X_s+X_{op}) ← cando_{OP}(X_s+X_{op}), in(X_s, X_{op})</td>
</tr>
<tr>
<td></td>
<td>dercando_{OP}(X_s- X_{op}) ← cando_{OP}(X_s- X_{op}), in(X_s, X_{op})</td>
</tr>
<tr>
<td>Most specific overrides</td>
<td>dercando_{OP}(X_s+X_{op}) ← cando_{OP}(X_s+X_{op}), in(X_s, X_{op}), -over_{UC}(X_s, X_{uc})</td>
</tr>
<tr>
<td></td>
<td>dercando_{OP}(X_s- X_{op}) ← cando_{OP}(X_s- X_{op}), in(X_s, X_{op}), -over_{UC}(X_s, X_{uc})</td>
</tr>
<tr>
<td></td>
<td>over_{UC}(X_s,X_{op}) ← cando_{OP}(X_{op} - X_{op}), dirin(X_s, X_{op})</td>
</tr>
<tr>
<td></td>
<td>over_{UC}(X_s,X_{op}) ← cando_{OP}(X_{op} - X_{op}), dirin(X_s, X_{op})</td>
</tr>
<tr>
<td>Path overrides</td>
<td>dercando_{OP}(X_s+X_{op}) ← cando_{OP}(X_s+X_{op})</td>
</tr>
<tr>
<td></td>
<td>dercando_{OP}(X_s- X_{op}) ← cando_{OP}(X_s- X_{op})</td>
</tr>
<tr>
<td></td>
<td>dercando_{OP}(X_s+X_{op}) ← cando_{OP}(X_s+X_{op}), -cando_{OP}(X_{op} - X_{op}), dirin(X_s, X_{op})</td>
</tr>
<tr>
<td></td>
<td>dercando_{OP}(X_s- X_{op}) ← cando_{OP}(X_s- X_{op}), -cando_{OP}(X_{op} - X_{op}), dirin(X_s, X_{op})</td>
</tr>
</tbody>
</table>

Table 3. Rules for enforcing inconsistency resolution and decision policies for use cases.

<table>
<thead>
<tr>
<th>Inconsistency</th>
<th>Decision</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denial take precedence</td>
<td>open</td>
<td>d_{OP}(X_s+X_{op}) ← -d_{OP}(X_s- X_{op})</td>
</tr>
<tr>
<td>Denial take precedence</td>
<td>closed</td>
<td>d_{OP}(X_s+X_{op}) ← -d_{OP}(X_s- X_{op}), -d_{OP}(X_s+X_{op})</td>
</tr>
<tr>
<td>permission take precedence</td>
<td>open</td>
<td>d_{OP}(X_s+X_{op}) ← -d_{OP}(X_s+X_{op})</td>
</tr>
<tr>
<td>permission take precedence</td>
<td>closed</td>
<td>d_{OP}(X_s+X_{op}) ← -d_{OP}(X_s+X_{op})</td>
</tr>
<tr>
<td>Nothing take precedence</td>
<td>open</td>
<td>d_{OP}(X_s+X_{op}) ← -d_{OP}(X_s+X_{op})</td>
</tr>
<tr>
<td>Nothing take precedence</td>
<td>closed</td>
<td>d_{OP}(X_s+X_{op}) ← -d_{OP}(X_s+X_{op})</td>
</tr>
<tr>
<td>Additional closure rule</td>
<td></td>
<td>d_{OP}(X_s+X_{op}) ← -d_{OP}(X_s+X_{op})</td>
</tr>
</tbody>
</table>

Rule (14) says that if an operation is part of a Use Case then the permission of the Use Case propagates to that operation.

7.3 Completing accesses for operations

Therefore, after the application of rules 15, authUML ensures the following

1. There is no operation with contradictory authorizations for the same subject.
2. For every subject, all operations of a Use Case are permitted iff the Use Case is permitted.
The next two rules ensure that all permission of a subjects to invoke operations will be represented as do predicates, and therefore either granted or denied, but not both. These rules were used in FAF also.

\[
\text{do}_{\text{OP}}(X_s, +X_{op}) \leftarrow \text{dercando}_{\text{OP}}(X_s, +X_{op}), \\
\text{¬do}_{\text{OP}}(X_s, X'_{op}) \leftarrow \text{¬do}_{\text{OP}}(X_s, +X_{op}), \\
\text{do}_{\text{OP}}(X_s, -X_{op}) \leftarrow \text{¬do}_{\text{OP}}(X_s, -X_{op}), \\
\text{¬do}_{\text{OP}}(X_s, X'_{op}) \leftarrow \text{¬do}_{\text{OP}}(X_s, -X_{op}).
\]

(16)

(17)

7.4 Alerting the requirements engineer of Irreconcilable Conflicts

Continuing with the example given at the end of section 7.2, if there is no \(X'_{op}\) prohibiting \(X_{op}\) then rule 15 can not resolve the inconsistency. Hence, authUML will raise a conflict message to the requirements engineer informing its inability to resolve the contradiction, as stated in rule 18.

\[
\text{cannotReslove}(X_s, X_{op}, X_{uc} \rightarrow X_{op}) \leftarrow \text{dercando}_{\text{OP}}(X_s, +X_{op}), \\
\text{dercando}_{\text{OP}}(X_s, -X_{op}), \\
\text{¬do}_{\text{OP}}(X_s, +X_{op}), \\
\text{do}_{\text{OP}}(X_s, X_{uc} \rightarrow X_{op}), \\
\text{UC}_{\text{OP}}(X_{uc} \rightarrow X_{op}), \\
\text{UC}_{\text{OP}}(X_{uc} \rightarrow X_{op}).
\]

(18)

(19)

(20)

Rule 19 triggers an alert message if it finds a subject \(X_s\) that has an authorization to invoke two operations that are conflicting with each other. Rule 20 triggers an alert message if it finds two conflicting subjects that have authorizations to invoke the same operation. Both rules will not hold if the requirement engineer explicitly allows that conflict by using ignore predicate.

At the end of phase 3, from the finalized authorization we can generate an access control list (ACL) of all positive and negative permissions of all subject to all operations.

8. RELATED WORK

Our work aims at analyzing security requirements during the early steps of the software life cycle. There are number of related works that address the same goal. But, most of them focus on modeling security requirement using UML. Our main focal point is analyzing (not necessarily modeling) access control requirements during the early stages of the software development life cycle.

In the area of modeling security requirements, Fernandez-Medina et al [9] propose an extension to the Use Case and Class models of UML. The extensions of Use Case diagram they introduced were stereotypes: \(<\text{safe-UC}>\) and \(<\text{accredited -actor}>\) as an indication of a secure Use Case and authorized actor.

Brose et al [4] extended UML to support the automatic generation of access control policies in order to configure a CORBA-based infrastructure. They specify permissions and prohibitions on accessing system’s objects (such as read, write, execute, etc) explicitly by writing notes that are attached to actors in the Use Case diagrams.

Jurjens’s work in [14] extends UML to integrate standards concepts from formal methods regarding multi-level secure system and security protocols. In addition, Lodderstedt et al [15] propose a methodology to model access control policies and integrate them into a model-driven software development process.

We proposed in [1] an extension to the UML’s metamodel to specify and enforce authorization and flow control policies. All cited works model security requirements in the design phase. We assume that all authorization requirements are consistent, conflict-free and complete. However, This current work takes a step back and focuses on analyzing access control requirements before proceeding to the design modeling to ensure consistent, conflict-free and complete requirements.

In the area of access control enforcement language, Brose [5] presented an access control language that allows security administrators to specify access control policies in CORBA. However, the language does not detect inconsistency or conflict of access control policies. Additionally, Jajodia et al [13] proposed FAF that we described in sections 1 and 2.2.

9. CONCLUSIONS

There is a need to consider security requirements in early phases of the software development life cycle. Our work differs form others work in this area by focusing on analyzing access control requirements at the requirement specification stage rather than modeling them with extra syntactic enrichments to UML. We have proposed authUML, a framework that analyze access control requirements to ensure consistency, completeness and conflict-free between access control requirements. The framework propagates access permissions on subject hierarchies and solves inconsistencies between authorizations by enforcing predefined policies that are written using the logical language of authUML.

To assure fine-grain analysis of access control requirements, authUML considers access control requirements for both Use Case and its operations. This is our preliminary work in bridging the gap between Logic programming and formal security engineering.

Our ongoing work addresses expanding authUML to accommodate other aspects of security and extending authUML to other phases of the software development life cycle.

10. REFERENCES


