

Validating the enforcement of access control policies and separation of duty principle in requirement engineering

Khaled Alghathbar *

King Saud University, College of Computer and Information Sciences, P.O. Box 51178, Riyadh 11543, Saudi Arabia

Received 25 October 2004; received in revised form 22 March 2006; accepted 26 March 2006

Available online 11 May 2006

Abstract

Validating the compliance of software requirements with the access control policies during the early development life cycle improves the security of the software. It prevents authorizing unauthorized subject during the specification of requirements and analysis before proceeding to other phases where the cost of fixing defects is augmented. This paper provides a logical-based framework that analyzes the authorization requirements specified in the Unified Modeling Language (UML). It ensures that the access requirements are consistent, complete and conflict-free. The framework proposed in this paper is an extension to AuthUML framework. We refine AuthUML and extend it by expanding its analysis to validate the enforcement of the Separation of Duty (SoD) during the requirement engineering. We enhance and extend AuthUML with the necessary phase, predicates and rules. The paper shows the various types of SoD and how each type can be validated. The extension shows the flexibility and scalability of AuthUML to validate new policies. Also, the extension makes AuthUML spans to different phases of the software development process that widen the application of AuthUML.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Access control policies; Security engineering; Use cases; Semi-formal methods; Separation of duty

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. Security is considered a non-functional requirement (NFR), security does not receive much attention during these early phases, resulting in more vulnerable software [9]. Consequently, the need to formally analyzing and validating security requirements during earlier phases have been proposed before [9] to detect and remove design vulnerabilities, and it has been pointed out that detecting and fixing problem at the early stages of the software development

is much cheaper than later stages [4]. Accordingly, to address this concern, we proposed AuthUML in [2], 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 [5,23]. 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.

AuthUML is a customized version of the Flexible Authorization Framework (FAF) of Jajodia et al. [18] is suitable for the Unified Modeling Language (UML) [23]

* Tel.: +96614678705.

E-mail address: ghathbar@ccis.ksu.edu.sa

based requirement engineering. It is an attempt to advance the application of rule based policies to the requirements specification phase of the software development life cycle. It analyzes access control requirements during requirement engineering to ensure that they are consistent, complete and free of application specified conflicts. To do so, AuthUML uses Prolog style stratified logic programming rules.

In this paper, we enhance AuthUML model by refining its syntax and predicates. In addition, we expand the application of AuthUML to validate the compliance of the authorization requirement with the principle of Separation of Duty (SoD) by introducing new phase, predicates and rules. SoD is a popular and important principle in security that is used to prevent fraud or error [8]. SoD separates the duty of a critical business task to more than one subject. There are many types of SoD. This paper enhances the application of AuthUML by proposing the necessary elements. We introduce the notion of business task that refer to a group of process needed to complete a task. We add new phase, predicates and rules for the SoD validation. We show the different types of SoD and how to express each one in the form of logic rules and when to enforce each type during the software life cycle. We expand the application of AuthUML to cover other phases other than the requirement specification and analysis phases. The extension shows the flexibility and scalability of AuthUML to validate new policies in different stages of the software development process. In this paper, we look at SoD principle from different perspective, from the perspective of integrating it with the software during the requirement engineering process.

The remainder of this paper is organized as follows. Section 2 summarizes the conceptual operations that are used to describe use cases, and a summary of FAF and short background of SoD. Section 3 introduces AuthUML framework and its processes. An example to be used to illustrate the different policies and rules is described in Section 4. Section 5 describes the syntax and semantics of AuthUML. Sections 6–8 describe the first, second and third major phases of AuthUML, respectively. Section 9 presents the new phase of AuthUML to analysis SoD along with the SoD types and the rules. Section 10 discusses the related work and Section 11 concludes the paper.

2. Background

2.1. Conceptual operations

Operation schemas introduced by Sendall and Strohmeier [26] enriches use cases by introducing conceptual operations and specifying their properties using Object Constraints Language (OCL) syntax [29]. 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

The Flexible Authorization Framework (FAF) of Jajodia et al. [18] uses a logic programming style stratified rule based to specify accesses. Despite there being other authorization models [10,30], 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 that can be implemented.

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.

2.3. Separation of duty principle

Separation of Duty (SoD) is a security principle that ensures the distribution of control to complete a critical business task to more than one individual. The purpose of the principle is to minimize fraud and errors by preventing a single individual from performing the whole critical task [25]. For example, order purchasing process has to be completed by at least two employees, such as Purchasing Manger and Account Manger. SoD has a tight liaison with access and flow control policies, SoD principle is employed by both policies to achieve better control of both access to information and flow of information.

SoD has two major types: static and dynamic. Static type can be validated before the software execution, during the development process, however, dynamic types must be validated during the execution because it is based on history that is will be built during the execution time only. The object in both contexts refers to different meaning. With the static types, the object refers to the class of objects such

as “Order” not the instance of class because it is impossible to validate a policy on an instance of a class before the execution. While with the dynamic types, object refers to an instance of class such as “order123”.

3. AuthUML

AuthUML is based on the Flexible Authorization Framework (FAF) of Jajodia et al. [18], 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 during the state of execution, FAF is invoked per each authorization request. Contrastingly, AuthUML is to be used by requirements engineers to avoid conflicts and incompleteness of accesses during the first stage of the development life cycle. 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.

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 (roles) are permitted or prohibited from invoking any given use case. This constitutes the first phase of AuthUML. During the second phase, AuthUML allows use case and role 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 policies to resolve inconsistencies, incompleteness and application dependent conflicts. Unlike the FAF, rules in AuthUML 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 [13,14,22] – some specification methodologies tolerate them. The fourth phase ensures that the specified authorization requirements do not violate the specified SoD principle.

3.1. AuthUML process

AuthUML consists of four main phases where each consists of several steps. As shown in Fig. 1, AuthUML takes

authorizations from requirements specifications and then analyzes them to produce complete, consistent and conflict-free authorizations. The four 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.
4. Ensure the compliance of authorization requirements with the SoD principle.

The first phase transforms a set of access control requirements – that have been specified – into a unified representation in the form of access predicates. Also, the first phase ensures that all specified accesses are consistent and conflict-free. By considering the hierarchy of role and permissions, 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. The fourth phase ensures that the specified authorization requirements do not violate the specified SoD principles. All four 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 role and/or object hierarchies. Any inconsistencies that may occur due to such propagated accesses are resolved using conflict resolution rules. After this, all accesses that are explicit (i.e. given directly in requirement) or implicit (derived) are consistent, but may not be complete. That is, not all accesses for all actors and use cases may be specified. Therefore, using predefined rules and policies (i.e. closed or open policies) the next step (5 in Fig. 1) completes them. Therefore, accesses specified before step 7 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 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

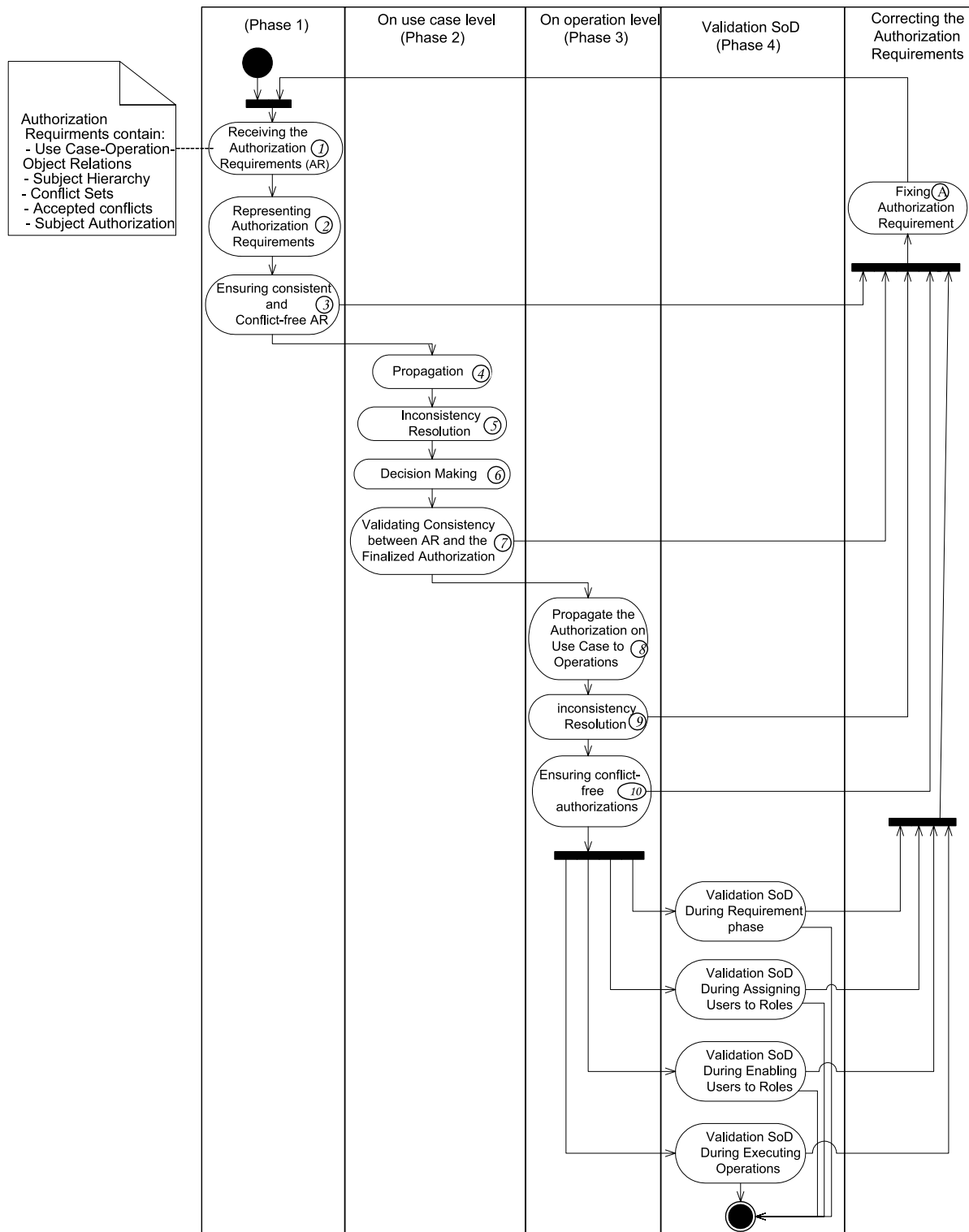


Fig. 1. AuthUML architecture.

not be free of application specific conflict. Thus, the purpose of the last step of this phase is to detect those conflicts.

The fourth phase validates the compliance of the finalized authorization requirements with the specified SoD principle. This phase differs from the previous phases in that the steps of this phase may not be executed in sequence

because each step perform the same objective of the phase but each step specialized in different variant of SoD. We group SoD variants in different steps according to common conditions. For example, the first step validates the compliance with SoD only at the requirement stage. The second step validates the compliance only when the assigning users

to roles. The third step validates the compliance only when user attempts to assume a role to perform a special task. Finally, the fourth step validates the compliance when user attempts to execute an operation. The last two steps are performed during the execution of the software. This phase expands the application of AuthUML to not just the requirement stage but also to the execution stage of the software development life cycle.

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 proceeds 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 reanalyzed. 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. First, AuthUML analyzes accesses in two levels, use cases and operations in order to scrutinize accesses in course-grain and fine-grain levels, respectively. Second, steps 3, 7 and 10 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. Running example

We introduce in this section an example to be used during the rest of the paper. It will be used to illustrate the different features of AuthUML including the different inconsistency resolution policies and user specified conflicts. The example as illustrated in Fig. 2 represents a purchasing process where it starts when a Clerk prepares an order then the Purchasing Officer places the order. Later, the Clerk writes a check for the order and the Manager signs it. The actors (roles) are ordered in a hierarchy where one actor inherits its higher actor authorizations. For example, Purchasing Officer inherits the authorizations of Clerk, thus, Purchasing Officer is allowed (implicitly) to prepare order and write check. The links between actors and use cases is considered permission while the link with Deny on it is considered a denial execution. Each use case consists of one or more operation to achieve the objective of the use case. Sometimes operation can be part of two use cases.

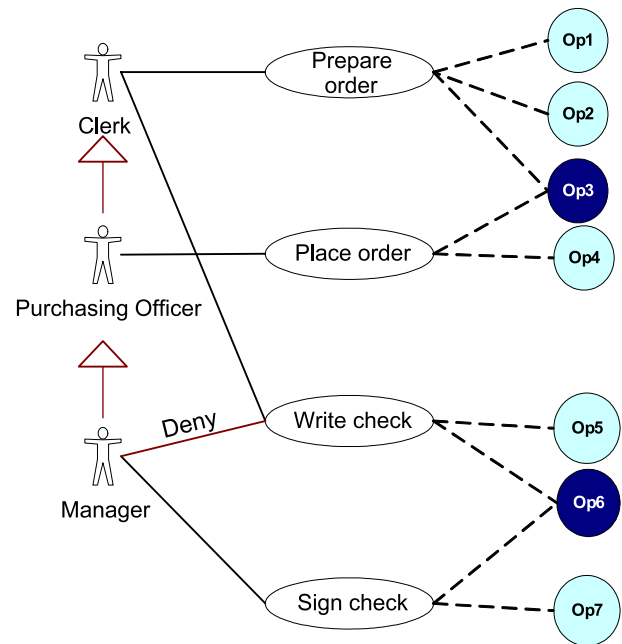


Fig. 2. Purchasing process.

5. AuthUML syntax and semantics

5.1. Individuals and terms of AuthUML

Individuals of AuthUML are use cases, operations, objects, business task, roles and users. 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. Business task refers to a set of use cases or operations that together perform a business task, such as, order purchasing process which consist of several steps, see Fig. 2. Business task is important element to SoD principle where business task identifies the subtasks that need to be assigned to different users or roles to prevent fraud or error. Role refer to actor, role is the same notion that is discussed in Role-based Access control (RBAC) [24]. Role is an element that has a set of permissions to invoke specific use case and operations, also role is the entity that users can assume to gain the permissions assigned to the role. For the rest of the paper, we will use the notion of role instead of actor. We denote UC, OP, OBJ, BT, R and U as set of use cases, operations, objects, business tasks, roles and users, respectively. 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 six individual sorts. Namely, signed use cases, signed operations, (unsigned) objects, (unsigned) business tasks, (signed) roles (when assigning users to roles only, other-

wise, unsigned) and (unsigned) users. They are represented, respectively, as $\pm uc$, $\pm op$, obj , bt , r and u , where variables are represented as $\pm X_{uc}$, $\pm X_{op}$, X_{obj} , X_{bt} , X_r and X_u .

5.2. 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 $UC_OP(X_{uc}, X_{op})$ means operation X_{op} is invoked in use case X_{uc} .
2. A binary predicate $OP_OBJ(X_{op}, X_{obj})$ means operation X_{op} belongs to object X_{obj} .
3. User – role relationship is represented by a binary predicate $User_Role(X_u, X_r)$ that means user X_u can play the role of X_r .
4. The business task – use case relationship is represented by $BT_UC(X_{bt}, X_{uc})$ which means that use case X_{uc} is part of business task X_{bt} . Likewise, $BT_OP(X_{bt}, X_{op})$ represents the relationship between business task and operation.
5. A binary predicate $before(X_{op}, X'_{op})$ means that X_{op} must be invoked before X'_{op} .
6. A ternary predicate $inUCbefore(X_{uc}, X_{op}, X'_{op})$ means use case X_{uc} invokes X_{op} before X'_{op} .

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

1. A binary predicate $in(X_r, X'_r)$, means X_r is below X'_r in the role hierarchy.
2. A binary predicate $dirin(X_r, X'_r)$ mean X_r is directly below X'_r in the role hierarchy.

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

1. A binary predicate $conflictingRole(X_r, X'_r)$ means roles X_r and X'_r are in conflict with each other.
2. A binary predicate $conflictingUC(X_{uc}, X'_{uc})$ means that use cases X_{uc} and X'_{uc} are in conflict with each other.
3. A binary predicate $conflictingOP(X_{op}, X'_{op})$ means operations X_{op} 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 roles, operations or use cases.

Following predicates are used to specify history and called history predicates.

1. A binary predicate $done(X_u, X'_{op})$ means that user X_u has executed operation X_{op} before.

2. A binary predicate $assumed(X_u, X_r)$ means user X_u has already assumed role X_r .

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. $opInConUC(X_{op}, X_{uc}, X'_{uc})$, means X_{op} is an operation in two conflicting use cases X_{uc} and X'_{uc} , and $conOpInUC(X_{op}, X'_{op}, X_{uc})$ means that X_{op} and X'_{op} are two conflicting operations in use case X_{uc} , and $flowConInUC(X_{uc}, X_{op}, X'_{op})$ means that X_{op} and X'_{op} are invoked in a way that violate execution order.
2. A binary predicate $cando_{UC}$, where $cando_{UC}(X_r, \pm X_{uc})$ means role X_r can or cannot invoke the use case X_{uc} depending on the *sign* of X_{uc} , positive (+) or negative (–).
3. A binary predicate $alertReq(X_r, X_{uc})$ to inform the requirements engineer that there is either an inconsistency (between access control requirements) or a conflict (between roles or use cases) on the access of X_r on X_{uc} .

Following predicates are used in the second phase of AuthUML to authorize, detect conflicts and inconsistencies at the use case level:

1. A ternary predicate $overUC(X_r, X'_r, \pm X_{uc})$ meaning X_r 's permission to invoke $\pm X_{uc}$ overrides that of X'_r .
2. A binary predicate $dercando_{UC}$ with the same argument as $cando_{UC}$. $dercando_{UC}(X_r, \pm X_{uc})$ is a permission derived using modus ponens and stratified negation [3].
3. A binary predicate do_{UC} , where $do_{UC}(X_r, \pm X_{uc})$ is the final permission/prohibition for role X_r to invoke use case X_{uc} depending on if the *sign* of X_{uc} is + or –.
4. A binary predicate $alert_{UC}(X_r, X_{uc})$ to inform the requirements engineer that there is either an inconsistency (between the access control requirement and the final outcome of this phase) or conflict (between roles or use cases) on the accesses that involve X_r and X_{uc} .

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

1. A binary predicate $dercando_{OP}(X_r, \pm X_{op})$ is similar to $dercando_{UC}$ except the second argument is an operation instead of a use cases.
2. A binary predicate $do_{OP}(X_r, \pm X_{op})$ is similar to do_{UC} but the second argument is an operation.
3. $cannotReslove(X_r, X_{uc}, X'_{uc}, X_{op})$ 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 $alert_{OP}(X_r, X_{op})$ to inform the requirements engineer that there is a conflict between roles or operations on the authorization that involve X_r and X_{op} .

The last section of predicates are describing the necessary predicates to detect violations of SoD principle, all this predicates are used in the fourth phase.

1. A binary predicate $\text{canAssume}(X_u, \pm X_r)$ means that whether user X_u can or can not assume role X_r , where the sign refers to permission or denial.
2. A binary predicate $\text{canExecute}(X_u, \pm X_{op})$ means that user X_u can or can not execute operation X_{op} .
3. A temporary predicate $\text{valid}_{\text{SoD}}$ to be used for closure rules.
4. A predicate $\text{alert}_{\text{SoD}}$ informs the requirements engineer that there is a violation of SoD policy.

Assumptions

- The *role* 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.
- Every use case must have at least one operation (i.e. $\forall x \in \text{UC} \exists y \in \text{OP UC_OP}(x, y)$) and that every operation must belong to one and only one object (i.e. $\forall x \in \text{OP} \exists! y \in \text{OBJ OP_OBJ}(x, y)$).
- Each positive access of a use case to a role means that all operations of that use case are also positively authorized to the same role, this is consistent with [27]. Conversely, a prohibited use case to a role must have at least one prohibited operation to that role.

As already stated, *cando* represents an access permission obtained from requirements and *dercando* represents an access derived using (to be described shortly) rules. Both *cando* and *dercando* do not represent a final decision, but only an intermediate result. For example, although $\text{cando}_{\text{UC}}(X_r, +X_{uc})$ is obtained from requirements does not mean that role X_r 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 $\text{cando}_{\text{UC}}(X_r, +X_{uc})$. However, $\text{do}_{\text{UC}}(X_r, +X_{uc})$ if derived represents the final authorization decision.

5.3. Rule of AuthUML

An AuthUML rule is the form of $L \leftarrow L_1, \dots, L_n$, where L is a positive literal and L_1, \dots, L_n are literals satisfying the conditions stated in Table 1. The following are rules examples:

$$\text{cando}_{\text{UC}}(\text{Manager}, +\text{"Sign a check"}) \leftarrow \quad (1)$$

$$\text{dercando}_{\text{UC}}(X_r, +X_{uc}) \leftarrow \text{cando}_{\text{UC}}(X'_r, +X_{uc}), \quad \text{in}(X_r, X'_r) \quad (2)$$

$$\text{do}_{\text{UC}}(X_r, +X_{uc}) \leftarrow \text{cando}_{\text{UC}}(X_r, +X_{uc}), \quad \text{cando}_{\text{UC}}(X_r, -X_{uc}) \quad (3)$$

Rule 1 says that Manager can access use case "Sign a check". Rule 2 specifies the inheritance of authorizations in the role hierarchy. Rule 3 expresses the permissions take precedence policy of resolving conflicts.

Table 1
Rules defining predicate

Phase	Stratum	Predicate	Rules defining the predicate
Phase 1	0	rel-predicates	base relations
		hie-predicates	base relations
		con-predicates	base relations
		history predicates	base relations
	1	$\text{opInConUC}(X_{op}, X_{uc}, X'_{uc})$	body may contain hie, ignore, rel predicates
		$\text{conOpInUC}(X_{op}, X'_{op}, X_{uc})$	
		$\text{flowConInUC}(X_{uc}, X_{op}, X'_{op})$	
	2	$\text{cando}_{\text{UC}}(X_r, \pm X_{uc})$	body may contain hie-, con- and rel-predicates
	3	$\text{alert}_{\text{Req}}(X_r, X_{uc})$	body may contain literal from strata 0 to 2
Phase 2	4	$\text{over}_{\text{UC}}(X_r, X'_r, \pm X_{uc})$	body may contain literals from strata 0 to 3
	5	$\text{dercando}_{\text{UC}}(X_r, \pm X_{uc})$	body may contain predicates from strata 0 to 4 Occurrences of $\text{dercando}_{\text{UC}}$ must be positive
	6	$\text{do}_{\text{UC}}(X_r, +X_{uc})$	body may contain predicates from strata 0 to 5
	7	$\text{do}_{\text{UC}}(X_r, -X_{uc})$	body contains one literal $\neg \text{do}_{\text{UC}}(X_r, +X_{uc})$
	8	$\text{alert}_{\text{UC}}(X_r, X_{uc})$	body may contain literal from strata 0 to 7
Phase 3	9	$\text{dercando}_{\text{OP}}(X_r, \pm X_{op})$	body may contain predicates from strata 0 to 7 Occurrences of $\text{dercando}_{\text{OP}}$ must be positive
	10	$\text{do}_{\text{OP}}(X_r, +X_{op})$	body may contain predicates from strata 0 to 9
	11	$\text{do}_{\text{OP}}(X_r, -X_{op})$	body contains one literal $\neg \text{do}_{\text{OP}}(X_r, +X_{op})$
	12	$\text{cannotReslove}(X_r, X_{uc}, X'_{uc}, X_{op})$	body may contain literal from strata 0 to 11
	13	$\text{alert}_{\text{OP}}(X_r, X_{op})$	body may contain literal from strata 0 to 11
Phase 4	14	$\text{canAssume}(X_u, \pm X_r)$	body may contain literal from strata 0 to 13
	15	$\text{canExecute}(X_u, \pm X_{op})$	body may contain literal from strata 0 to 13
	16	$\text{alret}_{\text{SoD}}$	body may contain literal from strata 0 to 13

5.4. 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 [16]. As done in FAF, we can materialize AuthUML rules also, thereby making the AuthUML inference engine efficient.

6. Phase I: Analyzing information in requirements documents

This section goes through steps 1, 2 and 3 of AuthUML and shows how the AuthUML processes the access control requirements.

6.1. Receiving the authorizations

This step is simple, it just receives the authorization requirements from the requirement engineers.

6.2. Representing authorizations

Following [18], 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 role to invoke use cases
2. Role hierarchy.
3. Structural relationships (use case – Operation – Object Relations).
4. Conflicting roles, 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 $cando_{uc}$ rules representing explicit authorization obtained from the requirement specification. The following two rules are examples of Fig. 2:

$$cando_{uc}(Clerk, +\text{"Prepare order"}) \leftarrow \quad (4)$$

$$cando_{uc}(Manager, -\text{"Write heck"}) \leftarrow \quad (5)$$

Rule 4 permits the clerk to invoke the Prepare order use case. Rule 5 prohibits the Manager to invoke the Write Check use case.

2. Role hierarchy is represented using the in predicate to indicate which role inherits what. For example, $in(purchasing\ Officer, Clerk)$ means that the purchasing officer is a specialized role 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. $UC_OP(X_{uc}, X_{op})$ says that X_{op} is an operations invoked in the use case X_{uc} . For example, $UC_OP(\text{"Prepare order"}, op1)$. $OP_OBJ(X_{op}, X_{obj})$ says that operation X_{op} , belongs to object X_{obj} . In addition, $before(X_{op}, X'_{op})$ means that X_{op} must be executed before X'_{op} is executed and $inUCbefore(X_{uc}, X_{op}, X'_{op})$ means that use case X_{uc} calls for executing X_{op} , before X'_{op} . For example, $inUCbefore(\text{"Prepare order"}, op1, op2)$ if we assume operation $op1$ must be performed before operation $op2$.
4. Application definable conflicts occurring among role, use case and operations are represented respectively by $conflictingRoles(X_r, X'_r)$, $conflicting_{UC}(X_{uc}, X'_{uc})$ and $conflicting_{OP}(X_{op}, X'_{op})$. For example, $conflicting_{UC}(\text{"Prepare order"}, \text{"Place order"})$ as representation of conflicting use cases.
5. Requirement engineers may decide to accept some conflicts as in [13,14,22]. AuthUML uses $ignore(X, Y, Y')$ to accept a conflict between Y and Y' . The main goal of the ignore predicate is to only allow specified conflicts, but not others between access. For example, we may write this predicate $ignore(op_3, \text{"Prepare order"}, \text{"Place order"})$ to allow the conflict of performing an operation that belongs to two conflicted use cases, such conflict may be propagated as a result of the relation between operation and use cases.

6.3. 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 roles may be permitted to invoke the same use case or operation, or a role may be permitted to invoke two conflicting use cases or operations. Latter kinds of permissions may violate the SoD principle [8].

In small systems, discovering inconsistency and 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 $cando$ rules and automatically applies inconsistency and conflict detection rules to identify their existence, as follow:

$$\text{alert}_{\text{Req}}(X_r, X_{uc}) \leftarrow \text{cando}_{UC}(X_r, +X_{uc}), \quad \text{cando}_{UC}(X_r, -X_{uc}) \quad (6)$$

$$\text{alert}_{UC}(X_r, X_{uc}) \leftarrow \text{cando}_{UC}(X_r, +X_{uc}), \quad \text{cando}_{UC}(X_r, +X'_{uc}), \quad \text{conflicting}_{UC}(X_{uc}, X'_{uc}), \quad \neg \text{ignore}(X_r, X_{uc}, X'_{uc}) \quad (7)$$

$$\text{alert}_{UC}(X_r, X_{uc}) \leftarrow \text{cando}_{UC}(X_r, +X_{uc}), \quad \text{cando}_{UC}(X'_r, +X_{uc}), \quad \text{conflictingRole}(X_r, X'_r), \quad \neg \text{ignore}(X_{uc}, X_r, X'_r) \quad (8)$$

$$\text{opInConUC}(X_{op}, X_{uc}, X'_{uc}) \leftarrow UC_OP(X_{uc}, X_{op}), \quad UC_OP(X'_{uc}, X_{op}), \quad \text{conflicting}_{UC}(X_{uc}, X'_{uc}), \quad \neg \text{ignore}(X_{op}, X_{uc}, X'_{uc}) \quad (9)$$

$$\text{conOpInUC}(X_{uc}, X_{op}, X'_{op}) \leftarrow UC_OP(X_{uc}, X_{op}), \quad UC_OP(X_{uc}, X'_{op}), \quad \text{conflicting}_{OP}(X_{op}, X'_{op}), \quad \neg \text{ignore}(uc, X_{op}, X'_{op}) \quad (10)$$

$$\text{flowConInUC}(X_{uc}, X_{op}, X'_{op}) \leftarrow UC_OP(X_{uc}, X_{op}), \quad UC_OP(X_{uc}, X'_{op}), \quad \text{before}(X_{op}, X'_{op}), X_{op} \neq X'_{op}, \quad \neg \text{inUCbefore}(X_{uc}, X_{op}, X'_{op}) \quad (11)$$

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 role then an alert message will be raised to the security engineer that identifies those that leads to the inconsistency. For example, the following authorization requirements are inconsistent $\text{cando}_{UC}(\text{Clerk}, + \text{“Prepare order”})$, $\text{cando}_{UC}(\text{Clerk}, - \text{“Prepare order”})$. Rule 7 says that if a role 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 roles, 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 violates 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.

7. 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 4, 5, 6 and 7) applies policies that are specified using AuthUML rules relevant to use cases. Such policies may add new permissions or change existing ones.

7.1. 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 dercando_{OP} rules (derived authorizations). Some examples of propagation policies are listed in [18] and represented as AuthUML rules in Table 2. The following examples are the output of the propagation rules, they represent implicit authorizations that were derived from the role hierarchy.

$$\begin{aligned} &\text{dercando}_{UC}(\text{Purchasing Officer}, +\text{Prepare order}) \\ &\text{dercando}_{UC}(\text{Manager}, +\text{Prepare order}) \\ &\text{dercando}_{UC}(\text{Manager}, +\text{Place order}) \\ &\text{dercando}_{UC}(\text{Manager}, +\text{Write check}) \end{aligned} \quad (12)$$

7.2. Inconsistency resolution policies

In complex systems with many use cases, permission propagation may introduce new permissions that in turn

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

Propagation policy	Rules
No propagation	$\text{dercando}_{UC}(X_r, +X_{uc}) \leftarrow \text{cando}_{UC}(X_r, +X_{uc})$ $\text{dercando}_{UC}(X_r, -X_{uc}) \leftarrow \text{cando}_{UC}(X_r, -X_{uc})$
No overriding	$\text{dercando}_{UC}(X_r, +X_{uc}) \leftarrow \text{cando}_{UC}(X'_r, +X_{uc}), \text{in}(X_r, X'_r)$ $\text{dercando}_{UC}(X_r, -X_{uc}) \leftarrow \text{cando}_{UC}(X'_r, -X_{uc}), \text{in}(X_r, X'_r)$
Most specific overrides	$\text{dercando}_{UC}(X_r, +X_{uc}) \leftarrow \text{cando}_{UC}(X'_r, +X_{uc}), \text{in}(X_r, X'_r), \neg \text{over}_{UC}(X_r, X'_r, +X_{uc})$ $\text{dercando}_{UC}(X_r, -X_{uc}) \leftarrow \text{cando}_{UC}(X'_r, -X_{uc}), \text{in}(X_r, X'_r), \neg \text{over}_{UC}(X_r, X'_r, -X_{uc})$ $\text{over}_{UC}(X_r, +X_{uc}, X'_r) \leftarrow \text{cando}_{UC}(X''_r, -X_{uc}), \text{in}(X_r, X''_r), \text{in}(X'_r, X'_s), r'' \neq r'$ $\text{over}_{UC}(X_r, -X_{uc}, X'_r) \leftarrow \text{cando}_{UC}(X''_r, +X_{uc}), \text{in}(X_r, X''_r), \text{in}(X'_r, X'_s), r'' \neq r'$
Path overrides	$\text{dercando}_{UC}(X_r, +X_{uc}) \leftarrow \text{cando}_{UC}(X_r, +X_{uc})$ $\text{dercando}_{UC}(X_r, -X_{uc}) \leftarrow \text{cando}_{UC}(X_r, -X_{uc})$ $\text{dercando}_{UC}(X_r, +X_{uc}) \leftarrow \text{cando}_{UC}(X'_r, +X_{uc}), \neg \text{cando}_{UC}(X_r, -X_{uc}), \text{dirin}(X_r, X'_r)$ $\text{dercando}_{UC}(X_r, -X_{uc}) \leftarrow \text{cando}_{UC}(X'_r, -X_{uc}), \neg \text{cando}_{UC}(X_r, +X_{uc}), \text{dirin}(X_r, X'_r)$

may result in new inconsistencies. Inconsistency resolution policies resolve such inconsistencies. Examples are listed in [18] 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 is granted for such role. 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. For example, note that in rules 12 Manager inherits a positive authorization to invoke Write check use case while in the Fig. 2 Manager has an explicit negative authorization to invoke the same use case. Therefore, step 5 is dedicated to detect such inconsistency and resolve it.

7.3. Decision policies

Decision policies complete authorizations so that every role must have either permission or a prohibition to execute each use case and operation. For example there is no positive or negative, explicit or implicit authorization to Clerk to invoke *Place order*, does that mean a permission or a denial? 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 roles that are consistent with each other and complete. They are written in the form of do_{UC} rules. AuthUML like FAF ensure the completeness of access control decision by enforcing the following.

$$do_{UC}(X_r, -X_{uc}) \leftarrow \neg do_{UC}(X_r, +X_{uc}) \quad (13)$$

7.4. 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. For example, if we chose to apply positive

over negative in case of inconsistency in step 5, then we end in a granting positive authorization for Manager to invoke *Write check* use case which contradicts the original authorization requirement which explicitly denies such invocation. Also, there may not be sufficient rules to resolve application specific conflicts. This step uses the $alert_{UC}$ predicate to inform the requirements engineer of such changes or problems.

$$\begin{aligned} alert_{UC}(X_r, X_{uc}) &\leftarrow cando_{UC}(X_r, +X_{uc}), do_{UC}(X_r, -X_{uc}) \\ alert_{UC}(X_r, X_{uc}) &\leftarrow cando_{UC}(X_r, -X_{uc}), do_{UC}(X_r, +X_{uc}) \end{aligned} \quad (14)$$

Rule 14 says that an alert message will be raised if there is an access control requirement and a contradicting final authorization for the same role on the same use case.

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.

8. Phase III: Applying policies to operations

The previous phase produces consistent and conflict-free use cases. This phase (step 8, 9 and 10) analyzes operations to ensure consistent, conflict-free and complete permissions to invoke operations.

8.1. Propagating permissions to operations

This phase applies fine grain access control policies to operations. Recall that use cases are described using operations and some execution order among them. Because any use case contains one or more operations, permission to invoke a use case propagate to its operations. Following rules specify such propagation policies.

$$\begin{aligned} dercando_{OP}(X_r, -X_{op}) &\leftarrow UC.OP(X_{uc}, X_{op}), do_{UC}(X_r, -X_{uc}) \\ dercando_{OP}(X_r, +X_{op}) &\leftarrow UC.OP(X_{uc}, X_{op}), do_{UC}(X_r, +X_{uc}) \end{aligned} \quad (15)$$

Rule 15 says that if an operation is part of a use case then the permission of the use case propagates to that operation. For example, the following are outcome of the previous rule on just the operations of the *Place order* use case.

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

Inconsistency	Decision	Rules
Denial take precedence	open	$do_{UC}(X_r, +X_{uc}) \leftarrow \neg dercando_{UC}(X_r, -X_{uc})$
Denial take precedence	closed	$do_{UC}(X_r, +X_{uc}) \leftarrow dercando_{UC}(X_r, +X_{uc}), \neg dercando_{UC}(X_r, -X_{uc})$
permission take precedence	open	$do_{UC}(X_r, +X_{uc}) \leftarrow dercando_{UC}(X_r, +X_{uc})$
		$do_{UC}(X_r, +X_{uc}) \leftarrow \neg dercando_{UC}(X_r, -X_{uc})$
permission take precedence	closed	$do_{UC}(X_r, +X_{uc}) \leftarrow dercando_{UC}(X_r, +X_{uc})$
Nothing take precedence	open	$do_{UC}(X_r, +X_{uc}) \leftarrow dercando_{UC}(X_r, -$
Nothing take precedence	closed	$do_{UC}(X_r, +X_{uc}) \leftarrow dercando_{UC}(X_r, +X_{uc}), \neg dercando_{UC}(X_r, -X_{uc})$
Additional closure rule		$do_{UC}(X_r, -X_{uc}) \leftarrow \neg do_{UC}(X_r, +X_{uc})$

$$\begin{aligned}
&\text{dercando}_{\text{OP}}(\text{Purchase Officer}, -\text{op3}) \\
&\text{dercando}_{\text{OP}}(\text{Purchase Officer}, -\text{op4}) \\
&\text{dercando}_{\text{OP}}(\text{Manager}, -\text{op3}) \\
&\text{dercando}_{\text{OP}}(\text{Manager}, -\text{op4})
\end{aligned} \tag{16}$$

8.2. 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 15 may introduce conflicts. Therefore, conflict resolution must be applied to operations. For example, in Fig. 2, operation *op6* is part of use cases *Write check* and *Sign check*, because Manager has an implicit (inherited) positive authorization to the first use case and an explicit negative authorization to the second use case, thus, Manager has an inconsistent authorization for the *op6*.

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 role, resulting from that, role 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 role only if another operation belonging to the prohibited use case already has a prohibition for the same role. In doing so, we preserve the assumption that as long as there is at least one prohibition on operation for a role in a use case, then that use case has a prohibition for the same role. For example, *op5* and *op6* have negative authorization for Manager role due to the *Deny* authorization for Manager to access *write check*, however, *op6* is also part of *sign check* use case which grant manager the right to access it. According to our policy in this situation which we discussed it shortly, rule 17 will solve the conflict of *op6* because the other operation of *Write check* still has a negative authorization, thus Manager can not completely execute *Write check* use case, but can execute *Sign check*.

Rule 17 specifies this conflict resolution policy as an AuthUML rule:

$$\begin{aligned}
&\text{do}_{\text{OP}}(X_r, +X_{\text{op}}) \leftarrow \text{dercando}_{\text{OP}}(X_r, +X_{\text{op}}), \\
&\quad \text{dercando}_{\text{OP}}(X_r, -X_{\text{op}}), \\
&\quad \text{UC.OP}(X_{\text{uc}}, X_{\text{op}}), \text{do}_{\text{UC}}(X_r, -X_{\text{uc}}), \\
&\quad \text{UC.OP}(X_{\text{uc}}, X'_{\text{op}}), \text{dercando}_{\text{OP}}(X_r, -X'_{\text{op}}), X'_{\text{op}} \neq X_{\text{op}}
\end{aligned} \tag{17}$$

8.3. Completing accesses for operations

Therefore, after the application of rules 17, AuthUML ensures the following:

1. There is no operation with contradictory authorizations for the same role.
2. For every role, all operations of a use case are permitted iff the use case is permitted.

The next two rules ensure that all permission of a roles 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.

$$\text{do}_{\text{OP}}(X_r, +X_{\text{op}}) \leftarrow \text{dercando}_{\text{OP}}(X_r, +X_{\text{op}}), \neg \text{dercando}_{\text{OP}}(X_r, -X_{\text{op}}) \tag{18}$$

$$\text{do}_{\text{OP}}(X_r, -X_{\text{op}}) \leftarrow \neg \text{do}_{\text{OP}}(X_r, +X_{\text{op}}) \tag{19}$$

8.4. Alerting the requirements engineer of irreconcilable conflicts

Continuing with the example given at the end of Section 9.2, if there is no X'_{op} prohibiting X_r , then rule 17 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 20. For example, it is possible that Clerk ends to be authorized to invoke operations *op1* and *op6* which may be specified to be in conflict with each other.

$$\begin{aligned}
&\text{cannotReslove}(X_r, X_{\text{uc}}, X'_{\text{uc}}, X_{\text{op}}) \leftarrow \text{dercando}_{\text{OP}}(X_r, +X_{\text{op}}), \\
&\quad \text{dercando}_{\text{OP}}(X_r, -X_{\text{op}}), \\
&\quad \neg \text{do}_{\text{OP}}(X_r, +X_{\text{op}}), \\
&\quad X_{\text{uc}} \neq X'_{\text{uc}}, \text{UC.OP}(X_{\text{uc}}, X_{\text{op}}), \text{UC.OP}(X'_{\text{uc}}, X_{\text{op}})
\end{aligned} \tag{20}$$

$$\begin{aligned}
&\text{alert}_{\text{OP}}(X_r, X_{\text{op}}) \leftarrow \text{do}_{\text{OP}}(X_r, +X_{\text{op}}), \text{do}_{\text{OP}}(X_r, +X'_{\text{op}}), \\
&\quad \text{conflictingOP}(X_{\text{op}}, X'_{\text{op}}), \\
&\quad \neg \text{ignore}(X_{\text{op}}, X_r, X'_r)
\end{aligned} \tag{21}$$

$$\begin{aligned}
&\text{alert}_{\text{OP}}(X_{\text{op}}, X_r) \leftarrow \text{do}_{\text{OP}}(X_r, +X_{\text{op}}), \text{do}_{\text{OP}}(X'_r, +X_{\text{op}}), \\
&\quad \text{conflictingRole}(X_r, X'_r), \\
&\quad \neg \text{ignore}(X_{\text{op}}, X_r, X'_r)
\end{aligned} \tag{22}$$

Rule 21 triggers an alert message if it finds a role X_r that has an authorization to invoke two operations that are conflicting with each other. Rule 22 triggers an alert message if it finds two conflicting roles 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 roles to all operations.

9. Phase IV: Validating authorization requirements compliance with separation of duty principle

Although, most of the previous sections – except some enhancement and refinement to AuthUML – has been pub-

lished before in Alghathbar and Wijesekera [2], presents a solid foundation to this section and how this section integrate with the AuthUML and extend it. This phase validates the compliance of authorization requirements with SoD. It is not possible to enforce all types of SoD at the same time. The phase consists of four steps – not necessary sequenced – where each step focuses on special variants of SoD based on the time condition of validation. For example, for the static SoD some SoD types can be validated during the requirement stage where others can not be validated until a user is assigned to roles. For the dynamic SoD, all dynamic variations can not be enforced in the requirements phase but during the system execution because the validation is based on the history of completed operations or assumed roles.

The rest of this section will group SoD types in to two sections: static and dynamic SoD variations and inside each section the variations will be further grouped into steps that group common SoD types together. Each step will show the rules used to ensure the proper enforcement of SoD principle. Note that the steps are not sequenced, it depends on the type of SoD that the requirement engineer needs be validated. Some type of SoD applies to use cases and operations. For simplicity, we write rules for use cases only for all types that can be applied on use cases or operations, however, to show an example of applicability, we will write rules for use case and operations for only the **Strict Static Separation of Duty (SSSoD)**. The following sort of SOD policies is not always considered in all systems, it depends on the requirement of the systems and sensitivity of it, thus, the implementation of it returns to the requirement engineers.

9.1. Static separation of duty

These types of SoD relies on static information, it requires only the role and its permissions to invoke use cases or operations, the following SoD types are the one that can be validated in this stage.

Stage 1: At the requirement phase.

At this stage only the role and its permissions to invoke use cases or operations are defined, thus, only the SoD types that require just this information can be validated now, the following SoD types are the one that can be validated in this stage:

Per-Role Conflicting Operation Static Separation of Duty (RCOpSSoD): No role can be allowed to perform two conflicting use cases or operations.

$$\text{alert}_{\text{SoD}} \leftarrow \text{do}_{\text{UC}}(X_r, +X_{uc}), \text{do}_{\text{UC}}(X_r, +X'_{uc}), \\ \text{conflictingUC}(X_{uc}, X'_{uc}) \quad (23)$$

The rule alerts the requirement engineer if a role has authorization to invoke two conflicting use cases.

Per-Role Business Task-based Static Separation of Duty (RBTSSoD): No role can be authorized to perform more than one use case or operation of a business task.

$$\text{alert}_{\text{SoD}} \leftarrow \text{do}_{\text{UC}}(X_r, +X_{uc}), \text{do}_{\text{UC}}(X_r, +X'_{uc}), \\ \text{BT_UC}(X_{bt}, X_{uc}), \text{BT_UC}(X_{bt}, X'_{uc}) \quad (24)$$

The rule alerts the requirement engineer if a role has authorization to invoke two use cases that belong to the same business task.

Per-Role Operation Static Separation of Duty (ROpSSoD): No roles can be authorized to perform all use cases or operations of a business task.

$$\text{alert}_{\text{SoD}} \leftarrow \neg \text{do}_{\text{UC}}(X_r, -X_{uc}), \text{BT_UC}(X_{bt}, X_{uc}) \quad (25)$$

The rule alerts the requirement engineer if there is no negative authorization to invoke a use case of business task. If there is at least on negative then we are ensured that the role can not execute the whole business task.

Stage 2: When users are assigned to roles.

At this stage, beside roles and their permissions, roles' users are also assigned. Knowing the assigned users of roles allow the validation of the following SoD types:

Static Separation of Duty (SSoD): No user can be assigned to two conflicting roles.

$$\text{alert}_{\text{SoD}} \leftarrow \text{User_Role}(X_u, X_r), \text{User_Role}(X_u, X'_r), \\ \text{conflictingRole}(X_r, X'_r) \quad (26)$$

The rule alerts the requirement engineer if there is a user assigned to two conflicting roles.

Strict Static Separation of Duty (SSSoD): No user can be assigned to two conflicting roles and no two conflicting roles can be authorized to perform the same use case or operation.

For use case:

$$\text{alert}_{\text{SoD}} \leftarrow \text{User_Role}(X_u, X_r), \text{User_Role}(X_u, X'_r), \\ \text{conflictingRole}(X_r, X'_r) \\ \text{alert}_{\text{SoD}} \leftarrow \text{do}_{\text{UC}}(X_r, +X_{uc}), \text{do}_{\text{UC}}(X'_r, +X_{uc}), \\ \text{conflictingRole}(X_r, X'_r) \quad (27)$$

The rule alerts the requirement engineer if there is a user assigned to two conflicting roles, and if there are two conflicting roles that have authorizations to invoke the same use case.

For example, let us assume in our running example that Manager and Clerk (without the inheritance) are in conflict with each other, the rule says that no user can assume both roles and those role can not invoke write check use case, yet one of them can if the other role has negative authorization on Write check.

For operation:

$$\text{alert}_{\text{SoD}} \leftarrow \text{User_Role}(X_u, X_r), \text{User_Role}(X_u, X'_r), \\ \text{conflictingRole}(X_r, X'_r) \\ \text{alert}_{\text{SoD}} \leftarrow \text{conflictingRole}(X_r, X'_r), \text{do}_{\text{OP}}(X_r, +X_{op}), \\ \text{do}_{\text{OP}}(X'_r, +X_{op}) \quad (28)$$

Conflicting User Static Separation of Duty (CUSSoD): No conflicting users can be assigned to the same role.

$$\text{alert}_{\text{SoD}} \leftarrow \text{conflictingUser}(X_u, X'_u), \text{User_Role}(X_u, X_r), \\ \text{User_Role}(X'_u, X_r) \quad (29)$$

The rule alerts the requirement engineer if there are two conflicting users who are members of the same roles.

Per-User Conflicting Operation Static Separation of Duty (UCOpSSoD): No user can have privileges to perform two conflicting use cases or operations.

$$\text{alert}_{\text{SoD}} \leftarrow \text{conflictingUC}(X_{uc}, X'_{uc}), \text{User_Role}(X_u, X_r), \\ \text{User_Role}(X_u, X'_r), \\ (\text{do}_{UC}(X_r, +X_{uc}), \text{do}_{UC}(X'_r, +X'_{uc})) \quad (30)$$

The rule alerts the requirement engineer if there is a user who is a member of two roles that have authorization to invoke two conflicting use cases.

Per-Operation Conflicting User Static Separation of Duty (OpCUSSoD): No conflicting users can have a privilege to perform the same use case or an operation.

$$\text{alert}_{\text{SoD}} \leftarrow \text{conflictingUser}(X_u, X'_u), \text{User_Role}(X_u, X_r), \\ \text{User_Role}(X'_u, X'_r), \\ \text{do}_{UC}(X_r, +X_{uc}), \text{do}_{UC}(X'_r, +X'_{uc}) \quad (31)$$

The rule alerts the requirement engineer if there is two conflicting users who are a member of two roles and both roles have an authorization to invoke the same use case.

For example, in our running example, we want to know if two conflicting users (Alice, Bob) can invoke Prepare order use case and that could happen if Alice belongs to Clerk role and Bob belongs to Manager role.

Business Task-based Static Separation of Duty (BTS-SoD): No two roles with common users can be authorized to perform more than one use case or operation of a business task.

$$\text{alert}_{\text{SoD}} \leftarrow \text{User_Role}(X_u, X_r), \text{User_Role}(X_u, X'_r), \\ \text{do}_{UC}(X_r, +X_{uc}), \text{do}_{UC}(X'_r, +X'_{uc}), \\ \text{BT_UC}(X_{bt}, X_{uc}), \text{BT_UC}(X_{bt}, X'_{uc}) \quad (32)$$

The rule alerts the requirement engineer if there is a user who is a member of two roles and each role has an authorization to invoke a different use case that belong to the same business task.

Per Business Task Operation Static Separation of Duty (BTOpSSoD): No two roles with common users can be authorized to perform all operations of a business Task.

$$\text{valid}_{\text{SoD}} \leftarrow \text{User_Role}(X_u, X_r), \text{User_Role}(X_u, X'_r), \\ \text{do}_{UC}(X_r, -X_{uc}), \text{do}_{UC}(X'_r, -X'_{uc}), \\ \text{BT_UC}(X_{bt}, X_{uc}) \quad (33)$$

$$\text{alert}_{\text{SoD}} \leftarrow \neg \text{valid}_{\text{SoD}}$$

The rule alerts the requirement engineer if there is a user who is a member of two roles and there is no negative authorization for one of the roles to invoke a use case of a business task.

9.2. Dynamic separation of duty variants

Stage 3: When enabling roles to users.

At this stage, when users are in the process of assuming roles to invoke a particular operation, the following SoD can be validated:

Dynamic Separation of Duty (DSoD): A user who is a member of two conflicting roles can not assume both roles at the same time.

$$\text{canAssume}(X_u, +X'_r) \leftarrow \text{User_Role}(X_u, X_r), \text{User_Role}(X_u, X'_r), \\ \text{conflictingRole}(X_r, X'_r), \neg \text{assumed}(X_u, X_r) \quad (34)$$

The rule says that a user can assume a role if the user has not already assumed another role that is in conflict with the role that needed to be assumed.

Per-Business Task Operation Dynamic Separation of Duty (BTOpDSoD): Any subset of roles enabled for the same user can not be authorized to perform all operations of a business task.

$$\text{canAssume}(X_u, +X_r) \leftarrow \text{User_Role}(X_u, X_r), \text{BT_OP}(X_{bt}, X_{op}), \\ (\text{do}_{OP}(X_r, -X_{op})); \\ (\text{assumed}(X_u, X'_r), \text{do}_{OP}(X'_r, -X_{op})) \quad (35)$$

The rule says that a user can assume a role only if one of the following condition is true: 1) the role that the user wants to assume has a negative authorization for one of the operations of the business task, or 2) one of the assumed roles has a negative authorization for one of the operations of the business task.

Stage 4: When executing operations.

At this stage, when users are in the process of executing operations after a successful assumption of role, the following SoD can be validated:

Object-based Dynamic Separation of Duty (ObjDSoD): No user can be allowed to perform an operation on object if that user has performed another operation on the same object.

$$\text{canExecute}(X_u, -X_{op}) \leftarrow \text{User_Role}(X_u, X_r), \\ \text{do}_{OP}(X_r, +X_{op}), \text{done}(X_u, X'_{op}), \\ \text{OP_OBJ}(X_{op}, X_{obj}), \text{OP_OBJ}(X'_{op}, X_{obj}) \quad (36)$$

$$\text{canExecute}(X_u, +X_{op}) \leftarrow \neg \text{canExecute}(X_u, -X_{op})$$

The first rule will deny any request of a user to execute an operation on an object if the user has executed another operation on the same object. If there is no denial, then the execution is allowed. For example, let us assume that operation op1 and op3 can perform some activity on Obj1, then we have to deny the user who invoked op1 from executing op3 which may grant him another access to Obj1.

Business Task Dynamic Separation of Duty (BTDSoD): No user can be allowed to perform an operation on busi-

ness task if that user has performed another operation on the same business task.

$$\begin{aligned} \text{canExecute}(X_u, -X_{op}) &\leftarrow \text{User_Role}(X_u, X_r), \\ &\quad \text{do}_{OP}(X_r, +X_{op}), \text{BT_OP}(X_{bt}, X_{op}), \\ &\quad \text{BT_OP}(X_{bt}, X'_{op}), \text{done}(X_u, X'_{op}) \\ \text{canExecute}(X_u, +X_{op}) &\leftarrow \neg \text{canExecute}(X_u, -X_{op}) \end{aligned} \quad (37)$$

The first rule will deny any request of a user to execute an operation on a business task if the user has executed another operation on the same business task before. If there is no denial, then the execution is allowed.

History-based Dynamic Separation of Duty (HDSOD):

The same user can not be allowed to perform all operations of a business task on the same object.

$$\begin{aligned} \text{canExecute}(X_u, +X_{op}) &\leftarrow \text{BT_OP}(X_{bt}, X_{op}), \text{BT_OP}(X_{bt}, X'_{op}), \\ &\quad \text{OP.OBJ}(X_{op}, X_{obj}), \\ &\quad \text{OP.OBJ}(X'_{op}, X_{obj}), \neg \text{done}(X_u, X'_{op}) \end{aligned} \quad (38)$$

The rule says that a user can execute an operation only if there another operation that acts on the same object and has not been executed before by the same user.

Note the previous rules are applied in general, though, it is possible to limit the validation to a particular element. For example, OpCUSSoD can be written as follow to prevent accessing use case XYZ by conflicting users.

$$\begin{aligned} \text{alert}_{\text{SoD}} &\leftarrow \text{conflictingUser}(X_u, X'_u), \text{User_Role}(X_u, X_r), \\ &\quad \text{User_Role}(X'_u, X'_r), \\ &\quad \text{do}_{UC}(X_r, +\text{"XYZ"}), \text{do}_{UC}(X'_r, +\text{"XYZ"}) \end{aligned} \quad (39)$$

The rule alerts the requirement engineer if there is two conflicting users who are a member of two roles and both roles have an authorization to invoke use case XYZ.

10. 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. [11] propose an extension to the use case and Class models of UML. The extensions of use case diagram they introduced were stereotypes: «safe-UC» and «accredited-actor» as an indication of a secure use case and authorized actor. Also, [12] proposed a methodology for the design of secure databases, they have defined some models to include security information in the database model, and a constraint language to define security constraints.

Brose et al. [6] 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 [19] extends UML to integrate standards concepts from formal methods regarding multilevel secure system and security protocols. In addition, Lodderstedt et al. [20] 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 meta-model 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 [7] 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. [18] proposed FAF that we described in Sections 1 and 2.2.

The SoD principle has been studied before from different directions, from defining it to applying it. Clark and Wilson [8] identified the need for SoD to defy fraud and error. Several other works identified different kinds of SoD [25,21,28,15]. In addition, Gligor et al. [17] formalized most different variations of SoD.

In this paper, we look at SoD principle from different perspective, from the perspective of integrating it with the software during the requirement engineering process. We propose Prolog style stratified logic programming rules to validate the correctness of enforcement of the principle during the early stages of software development. We adapt, enhance, and introduce different types of SoD. We enhance AuthUML [2] by refining its predicates and syntax. In addition, we extend AuthUML [2] by introducing new phase, predicates and rules to validate the proper enforcement of SoD principle.

11. Conclusions

There is a need to consider security requirements in early phases of the software development life cycle. Our work differs from others work in this area by focusing on validating the enforcement of access control rather than modeling them with extra syntactic enrichments to UML. Validation during early phases of the software development life cycle reduces the propagation of errors and cost of fixing those errors during later phases. We have proposed to extend AuthUML, a framework that

analyze access control requirements to ensure consistency, completeness and conflict-free between access control requirements. We expanded the capability of AuthUML to validate the proper application of SoD principle. SoD is an important principle used in real-world to prevent fraud and errors. The extension introduces new phase, predicates and rules. The introduction of those elements enhances AuthUML and expands its application to other phases of the software development life cycle. This is our preliminary work in bridging the gap between Logic programming and formal security engineering. As well as, bridging the gap between functional requirements and non functional requirements.

The application of AuthUML provides several advantages. First, it validates the requirement as early as possible during the software development life cycle, results in lowering the cost and effort of fixing errors and inconsistency at later phases of the development process. Second, AuthUML is based on logic-based programming that eases the automation of validation. Third, the framework is designed in a flexible way that allows the validation of different types of access control policies. In addition, the policies are written in a unified representation that can be reused on different requirements.

AuthUML holds potential advantages and we are continuing to address expanding AuthUML to accommodate other aspects of security and extending AuthUML to other phases of the software development life cycle. In addition, we are currently working on combining the validation of both access and flow control policies to achieve better results.

Acknowledgements

The author thanks Duminda Wijesekera for his contribution of the previous work AuthUML which this paper extends. Also, the author thanks the referees and auditors for their valuable comments, which have significantly improved the readability of this paper.

References

- [1] K. Alghathbar, D. Wijesekera, Modeling dynamic role-based access constraints using UML, in: Proceedings of the 1st International Conference on Software Engineering Research & Applications (ICSERA'03), San Francisco, CA, June (2003).
- [2] K. Alghathbar, D. Wijesekera, AuthUML: a three-phased framework to analyze access control specifications in Use Cases, Washington, DC, in: Proceedings of the Workshop on Formal Methods in Security Engineering (FMSE), ACM Press, New York, 2003.
- [3] K. Apt, H. Blair, A. Walker, Towards a theory of declarative knowledge, in: J. Minker (Ed.), Foundations of deductive databases, Morgan Kaufman, San Mateo, 1988, pp. 89–148.
- [4] B. Boehm, Software engineering economics, Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [5] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, Reading, MA, 1999.
- [6] G. Brose, M. Koch, K.-P. Löh, Integrating access control design into the software development process, in: Proceedings of the sixth biennial world conference on the Integrated Design and Process Technology (IDPT), Pasadena, CA, June (2002).
- [7] G. Brose, A typed access control model for CORBA, in: F. Cuppens, Y. Deswarte, D. Gollmann, M. Weidner (Eds.), Proceedings European Symposium on Research in Computer Security (ESORICS), Lecture Notes in Computer Science, vol. 1895, Springer, Berlin, 2000, pp. 88–105.
- [8] D.D. Clark, D.R. Wilson, A comparison of commercial and military computer security policies, in: Proceedings of the IEEE Symposium on Security and Privacy, 1987, pp. 184–191.
- [9] P.T. Devanbu, S. Stubblebine, Software engineering for security: a roadmap, in: A. Finkelstein (Ed.), The Future of Software Engineering, ACM Press, New York, 2000.
- [10] J. Dobson, J. McDermid, A framework for expressing models of security policy, in: Proceedings of the IEEE Symposium on Security and Privacy, 1989, pp. 229–241.
- [11] E. Fernandez-Medina, A. Martinez, C. Medina, M. Piattini, Integrating multilevel security in the database design process, in: Proceedings of the sixth biennial world conference on the Integrated Design and Process Technology (IDPT), Pasadena, CA, June (2002).
- [12] E. Fernandez-Medina, M. Piattini, Designing secure databases, Information and Software Technology 47 (7) (2005) 463–477.
- [13] D. Gabbay, A. Hunter, Making inconsistency respectable: a logical framework for inconsistency in reasoning, phase1 – a position paper, Proceedings of Fundamentals of Artificial Intelligence Research, Springer, Berlin, 1991, pp. 19–32.
- [14] D. Gabbay, A. Hunter, Making Inconsistency Respectable: A Logical Framework for Inconsistency in Reasoning, Phase2, in: Symbolic and Quantitative Approaches to Reasoning and Uncertainty, Lecture Notes in Computer Science, Springer, Berlin, 1992, pp. 129–136.
- [15] Gail-J. Ahn, Ravi Sandhu, Role-based authorization constraints specification, ACM Transactions on Information and System Security 3 (4) (2000) 207–226.
- [16] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: Proceedings of the 5th International Conference and Symposium on Logic Programming, Seattle, Washington, 1988, pp. 1070–1080.
- [17] V. Gligor, S. Gavrila, D. Ferraiolo, On the formal definition of separation of duty policies and their composition, in: Proceedings of the IEEE Symposium on Security and Privacy, Berkeley, CA, May (1998).
- [18] S. Jajodia, Pierangela Samarati, Maria Luisa Sapino, V.S. Subrahmanian, Flexible support for multiple access control policies, ACM Trans. on Database Systems 26 (2) (2001) 214–260.
- [19] J. Jurjens, Towards development of secure systems using UMLsec, in: H. Hussmann (Ed.), Fundamental Approaches to Software Engineering, 4th International Conference, Proceedings, Lecture Notes in Computer Science, Springer, Berlin, 2001, pp. 187–200.
- [20] T. Lodderstedt, D. Basin, J. Doser, SecureUML: a UML-based modeling language for model-driven security, Proceedings of the 5th International Conference on the Unified Modeling Language, Dresden, Germany, Springer, Berlin, 2002, pp. 426–441.
- [21] M. Nash, K. Poland, Some conundrums concerning separation of duty, In: Proceedings of the IEEE Symposium on Security and Privacy, May 1990, pp. 201–207.
- [22] B. Nuseibeh, S. Easterbrook, A. Russo, Making respectable in software development, Journal of Systems and Software 56 (11) (2001).
- [23] Object Management Group, OMG Unified Modeling Language Specification, Version 1.4, 2001 <<http://www.omg.org/technology/documents/formal/uml.htm>>.
- [24] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, C.E. Youman, Role-based access control models, IEEE Computer 29 (2) (1996) 38–47.
- [25] R.S. Sandhu, Transaction control expressions for separation of duties, in: Proceedings of the Fourth Computer Security Applications Conference, 1988, pp. 282–286.
- [26] S. Sendall, A. Strohmeier, From use cases to system operation specifications, UML (2000) 1–15.

- [27] S. Sendall, Specifying Reactive System Behavior, Ph.D. thesis, Swiss Federal Institute of Technology – Lausanne (EPFL), May (2002).
- [28] R. Simon, M. Zurko, Separation of duty in role-based environments, in: Proceedings of the 10th Computer Security Foundations Workshop, Rockport, Massachusetts, June (1997).
- [29] J. Warmer, A. Kleppe, The Object Constraint Language: Precise Modeling with UML, Addison-Wesley, Reading, MA, 1999.
- [30] T. Woo, S. Lam, Authorizations in distributed systems: a new approach, *Journal of Computer Security* 2 (2–3) (1993) 107–136.