A Framework of Composable Access Control Features: Preserving Separation of Access Control Concerns from Models to Code

Jaime A. Pavlich-Mariscal\textsuperscript{a}, Steven A. Demurjian\textsuperscript{b},
Laurent D. Michel\textsuperscript{b}

\textsuperscript{a}Departamento de Ing. de Sistemas y Computacion
Universidad Catolica del Norte
Angamos 0610, Antofagasta, Chile.

\textsuperscript{b}Department of Computer Science & Engineering.
The University of Connecticut.
Unit-2155, 371 Fairfield Road, Storrs, CT 06269-2155, USA.

Abstract

Modeling of security policies, along with their realization in code, must be an integral part of the software development process, to achieve an acceptable level of security for a software application. Among all of the security concerns (e.g. authentication, auditing, access control, confidentiality, etc.), this paper addresses the incorporation of access control into software. The approach is to separate access control concerns from the rest of the design. To assist designers to visualize access control policies separated from non-security concerns, this paper proposes a set of access control diagrams, i.e., extensions to the UML to represent three main access control models: role-based access control (RBAC), mandatory access control (MAC), and discretionary access control (DAC). To better adapt to changing requirements, and assist designers to customize access control policies, this paper proposes a set of access control features, i.e., small components that realize specific capabilities of access control models. Designers can select the features they require, and compose them to yield different access control policies. When transitioning into code, the main focus is to preserve separation of access control concerns. This paper describes an approach to realize access control diagrams and features in code through structure-preserving mappings, describes three different approaches to enforce access control in code, and evaluates the way each of them separate access control from other concerns.

Email addresses: jpavlich@ucn.cl (Jaime A. Pavlich-Mariscal), steve@engr.uconn.edu

Preprint submitted to Elsevier July 4, 2013
1 Introduction

In today’s software applications, capturing security requirements, designing a solution and transitioning it into code, are elements that must be included as primary and indispensable activities of the software development process. Since security requirements tend to change across the entire life-time of a software system, it is very important to have mechanisms that allow the developers or the security administrators to understand and evolve the policies seamlessly. Overall, the general problem that motivates our work is the need of a process for secure software engineering that incorporates security at early stages in the software development process. As such, there are multiple concerns involved: authentication, confidentiality, auditing, access control, among others. The scope of this paper is access control, focusing in the design and coding stages. However, the work presented herein is part of a bigger effort to incorporate multiple security concerns into software.

Access control design and coding encompasses several key dimensions, shown in Figure 1. At the design level, software engineers must create an access control policy, identifying the entities that are involved in access control, and their relationships to one another. There are three main access control models that assist designers to conceptualize access control policies: mandatory access control (MAC) [6,8], discretionary access control (DAC) [11,18], and role-based access control (RBAC) [25,50,61]. Although RBAC, MAC and DAC can satisfy a wide range of requirements, designers often do not need all of the capabilities of access control models, but only some of them. To adequately adapt these models to changing requirements, an access control design must be customizable, i.e., it must provide the tools to select only those capabilities that designers require, and combine them to adapt to changing requirements. To complement all of the above, a visual notation can facilitate the comprehension of access control policies, and their integration with non-access-control concerns. However, standard notations, such as the unified modeling language (UML) [36,37], do not support access control.

Another dimension is separation of concerns [41], an essential principle in software engineering that focuses on collecting, into different modules, the design and code portions that realize different concerns. Separating access control from other concerns can be beneficial for software engineers, because they can better distribute their tasks and work independently on each concern. Strongly related with separation of concerns is traceability, i.e., the ability

(Steven A. Demurjian), ldm@engr.uconn.edu (Laurent D. Michel).
to identify the parts in the design and code that realize different concerns. Traceability facilitates evolution, because it assist designers and programmers to easily locate the portions of the system that need to change to adapt to new requirements. However, access control is a concern that pervades the whole application [15], which makes it difficult to modularize from the rest of the code, and is detrimental for traceability.

Although there exist approaches that incorporate access control into the software process, none of them are comprehensive. The approach of [1] incorporates RBAC into UML, and it uses logic programming to verify access control policies. However, it does not address MAC, DAC, it does not separate access control concerns, and it does not address access control code. Similarly, the approach of [17] enhances UML to support RBAC, MAC, and Lifetimes, but it does not separate access control concerns, and it does not address access control code. The approach of [56] provides a model that integrates RBAC and MAC, but it doesn’t provide a visual notation or address the code level. Approaches such as [15, 22, 40] address separation of access control concerns only at the code level. The works of [5, 30] address both design and coding. However, they concentrate only on one access control model (RBAC); and, they do not fully isolate access control concerns from the rest of the application.

Figure 2 is an overview of the proposed approach to address the above issues. At the design level, the focus is to separate the main design (non-access-control concerns) from the access control design. To assist designers to visualize access control policies, this paper proposes a set of access control diagrams, i.e., extensions to the UML to model access control. Access control diagrams provide a notation that is separated from the standard UML diagrams (e.g. class, sequence, use case diagrams, etc.), thus they promote separation of concerns at the design level. To better adapt to changing access control requirements, each access control diagram comprises a set of access control features, i.e., composable units that
realize specific capabilities of access control models, namely RBAC, MAC and DAC. Designers can select and compose features to achieve the desired behavior in an access control policy. At the code level, this paper proposes an approach to realize the policy design, with a focus on traceability, and assesses several approaches to enforce a policy in an application. The emphasis is to evaluate how well each of them preserve separation of access control concerns from models down to code.

The remainder of the paper details the approach. Section 2 details some background concepts that this paper requires. Section 3 describes a software system utilized as a case study; the experience obtained during its development yielded the essential ideas for this paper. Section 4 describes the design stage of access control, focusing on the utilization of access control diagrams to model a policy. Section 5 explains access control features and the way to compose them to customize access control. Section 6 describes an approach to realize the access control design structures at the code level, with an emphasis on traceability. Section 7 assesses three different approaches of enforcement code, in terms of separation of access control from other concerns. Section 9 details related work. Section 10 concludes.

2 Background

The problem of incorporating access control into software spans through many areas: access control models, separation of concerns, visual notation, etc. This section details several of these concepts that are important to understand the proposed approach. The reader can refer to this section as needed, while reading the rest of the paper. Section 3 shows an example
application that this paper uses to illustrate the concepts of the proposed approach. Section 2.1 introduces access control models (RBAC, MAC, and DAC). Section 2.2 describes aspect-oriented programming, an approach to separate concerns. Section 2.3 details some metrics to assess separation of concerns.

2.1 Access Control Models

Access control models provide the basic structures that designers can use to conceptualize policies. There exist three main models: mandatory access control (MAC), discretionary access control (DAC), and role-based access control (RBAC). Bell and Lapadula [7] proposed the most widely used MAC model, which defines a set of objects, the entities that need to be accessed, and a set of subjects, the entities who need access to objects. There is an ordered hierarchy of subjects, where the highest levels in the hierarchy have more privileges than the lower levels. A central administration gives to each subject a clearance, i.e., a label that represents its position in the hierarchy; and, to each object it gives a classification, i.e., a label that defines the sensitivity of the information they contain. Labels form a partial order that MAC uses to constrain the access to each resource in the system; a subject can access an object only if its clearance dominates the object’s classification.

Discretionary access control (DAC) is “a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control). [18]” In DAC, each subject owns a set of resources; the system allows the subject to perform operations over those resources and to modify permissions to access them. Subjects, at their own discretion, can grant/revoke access to their own resources to other users [24]. Among all of the DAC features, this paper focuses in delegation, i.e., the ability of a subject to transfer to another subject its rights over the system [11,32]. There are two kinds of delegation: delegation authority [32], and pass-on delegation authority [32]. Delegation authority refers to the non-transitive privilege that a subject has to transfer parts of his/her permissions to another user. In other words, a user a can delegate part of his/her permissions to a user b, but the latter cannot subsequently transfer that set of permissions to another user. Pass-on delegation authority is a transitive privilege to transfer permissions from one subject to another, i.e., the authority of a subject to both transfer permissions to another user, and to transfer the ability to delegate those rights subsequently.
Role-based access control (RBAC) has its origins in the way that organizations manage information with the intent to be user-focused in access control rather than the traditional data-focused view of MAC. RBAC establishes access control with respect to the tasks that each user performs inside the organization [23], promoting the definition of *roles* to represent these tasks in an abstract manner. Thus instead of assigning permissions directly to users, RBAC assigns them to a certain role inside the organization. After that, a user can assume that role and utilize permissions for the duration of the authorization. Since in an organization, the set of functions associated to each role is much more stable than the users who assume those roles [50], this approach limits the changes to the access control policy and the impact on end-user authorizations. Historically, there have been many approaches to RBAC [33, 50, 58, 61, 62]; this paper concentrates on concepts related to the NIST standard [25].

### 2.2 Aspect-Oriented Programming

As complexity of software applications continues to increase, there has been an emphasis on providing techniques that reduce complexity while still promoting the ability to construct large-scale application. One such technique focuses on *separation of concerns* to distinguish all important concerns of an application in modular units that could be managed by independent developers. According to [60], in order to achieve this goal, software formalisms may be required to provide: decomposition mechanisms that can partition the software into simpler pieces that are easier to manage; and, composition mechanisms to join all of the component elements together into a final complete application. Thus, there must be support for composition/decomposition in multiple concerns, e.g., performance, scalability, security, etc.

In the object-oriented paradigm, one of the composition and decomposition mechanisms is the *class*. Classes also have *methods* to further decompose the code of the class in smaller parts. While offering a degree of separation of concerns, classes and methods are limited in their ability to modularize every concern in the application. As a result, object-oriented applications tend to suffer from two common problems:

**Scattering:** Certain concerns may be realized in multiple classes and methods in both the design and code. For instance, in an banking application the code to implement auditing (e.g., recording the executions to specific methods in the application) can be scattered
among multiple methods in each class.

**Tangling:** One class or method can implement several different requirements simultaneously. Using the example above, each method that incorporates auditing code may also have code that implements business rules, such as cash flow calculations, mortgage rates, etc.

*Aspect-oriented programming* (AOP) is an approach for isolating *crosscutting concerns*, i.e., requirements orthogonal to the application structure whose implementations are invariably *scattered* and *tangled* throughout the entire application. An *aspect* is a code fragment that modularizes the orthogonal concern. An *aspect weaver* is a compiler that integrates the aspects with the rest of the application. Each aspect specify where and how to inject its own code in the application. Standard terminology for the purposes of this paper includes:

**Advice** is a code fragment that implements a part of an aspect, for our purposes, access control, and is intended to be woven with the main program.

**Join Points** are the locations within a program where the aspect weaver integrates each advice, e.g., security, database interaction, performance, etc.

**Pointcuts** are sets of join points that each share specific static properties. For instance, in AspectJ [4], pointcuts are defined with quantified boolean formulas over method names, class names, control flow or lexical scopes, and capture specific event occurrences such as method calls, access to attributes or exceptions to name a few.

**Aspect Weaving** is a compilation technique that identifies join points in point cuts and modifies the code at that site according to the specified advice.

### 2.3 Separation of Concern Metrics

To adequately manage a software project, it is necessary to quantify its attributes during its life-cycle. Software metrics are an essential tool to achieve this task [27]. This paper focuses on preserving separation of access control concerns in code. This section details a set of metrics that this paper will use to assess different approaches for access control enforcement in terms of separation of concerns.

This paper uses three main metrics for separation of concerns: coupling between components (CBC), concern diffusion over components (CDC), and concern diffusion over operations (CDO).
**Coupling between components (CBC)** is the amount of components to which a particular class or aspect is coupled [12,52]. The CBC for a component \(c\) (class or aspect) counts the number of classes in its attribute declarations. For each method, it counts the number of classes referenced by method parameters, classes referenced by return types, classes of exceptions thrown, local variable classes, and statically referenced classes. CBC for aspects also includes the number of components referenced by aspect introductions, components referenced by poincuts, and components referenced by each advice. CBC counts each component only once. This paper will extend the definition of CBC to measure couplings for packages (sets of components). The CBC for a package will include all of the classes and aspects to which a package is coupled.

**Concern diffusion over components (CDC)** counts the components that realize a specific concern [52]. The more components in the code implement a concern, the more scattered is the concern. CDC counts two kind of elements: the primary components whose main purpose is to realize a specific concern; and, the amount of components that are coupled to them.

**Concern diffusion over operations (CDO)** determines the scattering of a concern across the operations (methods and advices) that realize it [52]. The more operations in the code implement a concern, the more scattered is the concern. CDO counts the operations (methods and advices) whose main purpose is to realize a specific concern; and, the operations that are coupled to the primary operations that realize that concern.

3 The University Application Case Study

Some of the essential ideas of this paper are based on the experience of the first author in the development of an university application. This application was the courseware system utilized by the Universidad Católica del Norte (“Northern Catholic University”, located in Antofagasta, Chile) from 2003 to 2007. The access control policy of the system comprises 10 roles, assigned to approximately of 6000 users per semester, and 122 permissions. Roles are organized in a MAC hierarchy that determines the access to course materials. For instance, documents written by teachers can only be read by lower roles (students). In addition, course owners (teachers) can delegate functions to assistants or students, and they can grant or deny people to access their courses or groups within courses. Similar policies apply to forums, workgroups, syllabi, and wikis within the application.
At the design level, the multiple access control requirements (Roles, MAC, delegation) yielded the key ideas for our proposed approach to address a wide range of access control models. At the code level, different approaches were tried to realize those access control policies. Earlier versions of the software provided limited separation of security concerns and were hard to modify and adapt to changing requirements. Later versions utilized more elaborated approaches that provided better separation of concerns.

To better explain the main concepts of our work, this paper utilizes a simpler example based on the original university application. This example is described as follows: The university application is a multi-user system that allows students, assistants, and teachers to access and manage the information about courses and students. Teachers have assigned a set of courses, they can read and write the syllabus, access their prerequisites, and course numbers. Teachers can see the enrolled students in each course, access their information, and read/modify their grades, but they cannot see in which other courses students are enrolled. Assistants can perform the same tasks as teachers, except that they are prohibited from writing the syllabus. Students can see their grades, enrolled courses, the teachers of each course, syllabi, and course numbers, but cannot see which students are enrolled in those courses, or modify any information in the system. Any person external to the university can access catalog information, i.e., courses offered by the university, course numbers, syllabus, and prerequisites. Figure 3 is a class model of the university application. CourseDescription manages all of the course information that is independent of time, i.e., course numbers, syllabi, prerequisites, etc. CourseSection manages the information of each course section per term, i.e., enrolled students, teachers, etc. StudentInformation manages information about students. Catalog manages the publicly-available information on courses offered at a university. Logger records events in the system.

4 Access Control Design

Access control design involves the conceptualization of a policy, the identification of the entities that are involved in access control, and their relationship to one another. Specifically, during design there is the potential to identify: objects (resources within the software application that need protection), subjects (individuals, programs, processes, or other systems that require access to objects), and permissions (authorization or denial to subjects to access objects in the system). There are three main access control models that propose par-
ticular structures for subjects, objects, and permissions, assisting designers to conceptualize access control policies: mandatory access control (MAC) [6, 8], discretionary access control (DAC) [11,18], and role-based access control (RBAC) [25,50,61]. These models are important, because they target a broad range of access control requirements. MAC is well-suited to applications where the protection of information is paramount (i.e., releasing such information would have dire national security or financial consequences). DAC targets applications that are collaborative and dynamic. RBAC is well-suited for organizations where their employees change with more frequency than the tasks and permissions associated to their positions in the organization. To adequately integrate access control into software, designers must be able to create policies based on these three access control models.

RBAC, MAC, and DAC provide an abstract specification of common characteristics of access control policies. While these models help designers to understand the basic properties of access control policies at a high level of abstraction, they do not provide a mechanism to crystallize those properties into a design model and integrate them with other concerns. To represent access-control information, it is crucial to utilize a notation that allows software engineers (designers) to clearly understand the policies that they are defining, and to evolve them easily as requirements change. In this regard, visual languages are very powerful tools; a well-designed visual representation can conceptualize access control information to software engineers (managers, designers, programmers, maintainers, etc.) in an intuitive fashion, facilitate changes, and hopefully reduce errors in the definition of the policy. The standard notation for design models, the Unified Modeling Language (UML) [36, 37], does not support access control. Existing approaches [1,5,17,30,56] propose some extensions to UML to represent access control. However, none of them support RBAC, MAC and DAC at
Another important element that facilitates the definition and evolution of access control design is separation of concerns. A notation separated from the main design provides designers with a collective view of access control policies, which facilitates the conceptualization of access control concerns. However, existing approaches [1, 17] do not adequately separate access control from the rest of the design. In those cases, access control specifications are spread across the entire design, and mixed with the design of other concerns. As a result, it is harder to designers to understand and evolve access control as a whole.

To address these issues, Figure 4 illustrates the proposed modeling process in UML as a set of interacting UML diagrams for roles, delegation, and user authorization, which may be augmented with MAC. The secure subsystem represents the basic information of the access control design as a subset of the application that contains all of the classes, methods, etc., that require access control. All of the access control diagrams must be consistent with the secure subsystem definition, with any changes in the secure subsystem reflected in the access control diagrams (and vice versa). Role-Slice, User, and Delegation Diagrams, are extensions to UML that support different facets of access control. The Role-Slice Diagram [43] is a UML extension to represent roles and their permissions, and is also the basis for the other diagrams. The User Diagram [45] defines authorizations between user and their roles. The Delegation Diagram [45] defines delegation of authority (role) among users. Mandatory Access Control Extensions such as sensitivity levels can be optionally included in each access control diagram [45]. These diagrams and features are dependent upon one another. For example, a Role-Slice Diagram maintains dependency with the User Diagram that includes all of the users of the system, their assignments to roles, and associated constraints. The Delegation Diagram depends on both the role-slice and user diagrams to represent delegation rules between roles and users.

The remainder of this section details the elements of Figure 4. Section 4.1 describes the secure subsystem diagram. Section 4.2 describes the role-slice diagram. Section 4.3 describes the user diagram. Section 4.4 describes the delegation diagram. Section 4.5 describes the MAC extensions to UML. Section 5 describes access control features, and the way to use them to customize access control policies.
4.1 The Secure Subsystem Diagram

An application has a set of operations that subjects can perform when interacting with the system, i.e., the public interface. An access control policy must protect these operations, allowing only authorized subjects to access them. However, only a subset of the public interface needs to be protected. For example, in the university application, the public interface are the methods from CourseSection, CourseDescription, Catalog, and StudentInformation (Logger realizes the internal behavior of the university application, thus it is not in the public interface). The set of operations that require access control conform the secure subsystem. To assist designers to model this information, this paper proposes the secure subsystem diagram. Pictorially, the secure subsystem diagram is a package with the stereotype ≪SecureSubsystem≫, with a class diagram that is a subset of the class model of the application. For instance, Figure 5 shows the secure subsystem diagram of the university application. The secure subsystem diagram includes all of the elements from the class model of Figure 3, with the exception of Logger (because it is not part of the public interface); and, all of the methods related with catalog information (because catalog information is public).
4.2 The Role-Slice Diagram

The role-slice diagram (see Figure 6) depicts roles, their permissions and hierarchy relations. Roles are depicted as role slices, i.e., packages with the stereotype ≪ RoleSlice ≫, with a class diagram that contains all of the methods that are explicitly allowed or denied to that role. Explicitly allowed methods have the stereotype ≪ pos ≫ and explicitly denied methods have the stereotype ≪ neg ≫. For instance, according to the figure, Assistant is explicitly allowed to invoke method getStudents, whereas Student is not. In addition, the role-slice diagram represents role hierarchies. Children roles in the hierarchy point to their parents using arrows with the stereotype ≪ RoleInheritance ≫. In a role hierarchy, children roles inherit all of the permissions from their parents, but they can also override them using negative permissions. This means that the absence of an explicit authorization does not imply that the method is denied. It may happen that a role without an explicitly allowed method is authorized to access that method if one of its parents is already authorized. For example, Student is authorized to invoke getDescription, thus Teacher, who is a child role of Student, can also invoke that method. Negative permissions can override this behavior. For instance, Student is also allowed to invoke getCourses. Even though Teacher inherits getCourses from Student, the explicit denial to access that method, depicted in role-slice Teacher, overrides the permission from its parent role.

A role slice can denote a policy based entirely on method-based permissions, i.e., a policy that focuses on allowing/denying subjects to access operations, regardless of the
objects to which those operations are applied. This may pose some security issues. For example, if a teacher has been assigned permissions to invoke specific methods, such as method.getStudents of class CourseSection, these privileges would be available for every instance of CourseSection. The requirements indicate that a teacher should only have access to the information of his/her own courses. Therefore, permissions defined by a method-based policy are too broad. To solve this problem, the role-slice diagram also authorizes roles based on class instances. In Figure 6, the class CourseSection in role slice Teacher contains an OCL [39] constraint that refers to runtime data, i.e., information from the objects in the system. The constraint allows a teacher to access only those instances of course that belong to him/her. The variable self represents the instance of course being accessed. The variable as represents the subject who is accessing the application.

4.3 The User Diagram

The user diagram (see Figure 7) is another proposed UML extension to represent user-role assignment information and constraints to that assignment. Pictorially, a user is a package with the stereotype << user >>; and, user-role assignments are arrows with the
Figure 7. User Diagram

stereotype \(\text{≪ RoleAssignment }\)\]. In addition, a user diagram can represent separation of duty constraints using a n-ary association with the stereotype \(\text{≪ sod }\)\]. This association connects all of the roles that cannot be simultaneously assigned to the same user. For instance, in the figure, user Bob and Carol are students, and since they have assigned that role, they cannot have assigned role Teacher at the same time, because it is mutually exclusive with Student.

4.4 The Delegation Diagram

The delegation diagram (see Figure 8) is a UML extension that represents all of the rules of delegation between roles. This diagram defines another notational artifact, the delegation slice, to represent all of the delegation rules that can be authorized to certain users. A delegation slice contains all of those roles that a user can delegate utilizing two different mechanisms: delegation authority, which authorizes a user to delegate a role to another user, but the delegated user cannot further delegate that role; and, pass-on delegation authority that, in addition, authorizes the delegated user to further delegate that role. Pictorially, the diagram represents a delegation slice as a package with the stereotype \(\text{≪ DelegationSlice }\)\]. It uses the tag \(\text{da}\) to indicate all of those roles with delegation authority, and the tag \(\text{poda}\) to indicate all of those roles with pass-on delegation authority. Delegation slices can be assigned to users to authorize them to delegate the corresponding roles. The diagram depicts this assignment as an arrow with the stereotype \(\text{≪ DelegationAssignment }\)\]. In the figure, the delegation slice includes delegation authority over role Teacher, and pass-on delegation authority over role Assistant. The diagram assigns that delegation slice to Alice, which
means that she is authorized to delegate \textbf{Teacher}, and \textbf{Assistant}, provided that each of those roles are assigned to her.

4.5 \textit{MAC Extensions}

This last extension comprises a new UML diagram, and a set of notational artifacts that can be incorporated into the previous diagrams to represent MAC rules. Figure 9 shows a \textit{MAC diagram}, which depicts sensitivity levels as packages with the stereotype ≪ \textit{SensitivityLevel} ≫. Each sensitivity level points to their immediately higher levels using arrows with the stereotype ≪ \textit{order} ≫. In the figure, there are four of them (ordered from lower to higher): unclassified (u), confidential (c), secret (s), and top-secret (ts). Figure 10 shows the secure subsystem of the university application that is enhanced with MAC. This figure tags each method in the secure subsystem with an access mode (using tag \textit{am}) that indicates whether the method is read-only, or if it writes some information into the system. The diagram also indicates the classification (using tag \textit{cls}) of each method, that indicates the sensitivity level (u,c,s,ts) of each of them. To complement this information, Figure 11 shows a role-slice diagram (permissions are not shown for simplicity), where each of the roles indicates the clearance (using tag \textit{clr}) that they have to access the operations in the secure subsystem.
4.6 Meta-model of Access Control Diagrams

All of these diagrams are based on an extension to the UML meta-model, shown in Figure 12. The figure represents each entity in the access control diagrams as a meta-class, and represents relations between meta-classes as arrows. Meta-class SecureSubsystem represents the secure subsystem package. SecureSubsystem is associated with many instances of Operation, the latter representing all of the methods in the class model of the application. Meta-class RoleSlice represents the role slices in a diagram. User represents all of the users in a user diagram. Both User and RoleSlice are subtypes of Subject. Subject is an abstract entity, which means that it does not have a visual representation in the role-slice diagram. SoD represents separation of duty relations between role slices. Each instance of SoD can be associated to multiple role slices (representing mutually exclusive roles) and each role slice needs to be associated to only one instance of SoD to indicate that is mutually exclusive with other roles. DelegationSlice represents all of the delegation slices in a delegation diagram. Meta-class SensitivityLevel represents the sensitivity levels of the policy. Operation can have one clearance associated, if required by a MAC policy, or have no clearances associated, otherwise. Subject can have one classification associated,
if required by a MAC policy, or have no classifications associated, otherwise. **AccessMode** represents the access mode (read or write) of an operation. **Operation** can have one access mode associated, if required by a MAC policy, or have no access mode associated, otherwise.

### 5 Access Control Features

The definition of the access control diagrams (secure subsystem, role-slice, user, delegation, and MAC diagrams), provides only part of the information required to realize an access control policy. Suppose, for example, that designers need to create a MAC-based policy for the university application, using diagrams from Figures 9, 10, and 11. These diagrams define sensitivity levels, assign classification, and clearances, but they do not determine the way to grant access to subjects based on that information. To define that missing information, this paper proposes the use of access control features [45]. Access control features are composable units that realize specific capabilities of access control models (see Figure 13). When designing the policy, designers can select access control features to enable the capabilities that are required, utilizing subsets of RBAC, MAC or DAC, according to their needs. At a later
stage, designers can compose all of the selected features to achieve the desired behavior. The result of the composition, the custom access control policy, defines the privileges of every subject who interacts with the system.

Composition of features is a very important issue. As described in Section 5.7, access
control models (RBAC, MAC, and DAC) have very specific restrictions that constrain the composition of features; as a result, there are some combinations of features that do not yield meaningful results. To adequately compose access control features, this section details their internal structure, the way that they relate to access control models, and formalize the composition process. Section 5.1 details the notation utilized to describe access control features. Section 5.2 formalizes the elements underlying access control features, specifically: the elements of an application that need protection and the structure of custom access control policies. Section 5.3 formalizes the features associated to roles and their hierarchy. Section 5.4 formalizes the features associated to users and their relation to roles. Section 5.5 formalizes the features for delegation. Section 5.6 formalizes MAC features. Section 5.7 formalizes the composition of access control features, focusing on detailing all of the restrictions to composition of features, and proposing a process to verify that a composition of features is valid.

5.1 Notation

The formal definitions are based on the meta-model of Section 4.6 and utilize set notation to represent the essential elements of the policy. Each meta-class of Figure 12 is represented as a set that contains all of the instances of that meta-class. For instance, the meta-class that represents role slices is denoted as a set RoleSlice. Relationships between meta-classes are represented as relations between the corresponding sets. The transitive and reflexive closure of each relation \( R \) is represented as \( R^* \). N-tuples are denoted as \( \langle a_1, \ldots, a_n \rangle \).

Association ends of Figure 12 are represented as follows: given meta-classes \( A \) and \( C \) connected by a relationship \( R \subseteq A \times C \), an instance \( a \) of a meta-class \( A \) with an association end \( b \) of \( R \) opposite to \( A \), the notation \( a.b \) corresponds to the set of instances \( c \) of \( C \), where \( \langle a, c \rangle \in R \). For example, given an instance \( u \) of User in Figure 12, the expression \( u.roleAssignment \) is the set of all instances of RoleSlice that are assigned to the user \( u \). Similarly, attributes of meta-classes are defined as relations between the meta-class and the type of the attribute; they are denoted in the same way as relationships. For instance, the expression \( r.isDelegatable \), where \( r \) is an instance of RoleSlice, represents the element of the set \( \text{Boolean} = \{ \text{true}, \text{false} \} \) to which \( r \) is associated.
5.2 Formal Model of Access Control Features

To define a custom access control policy through a valid composition of features, it is important to describe features formally. The assumption is that there exist a set of subjects, individuals who interact with the application. An application has two main elements: a set of operations available to subjects, called the public interface and a set of objects that can be modified by those operations. During the interaction, subjects can execute operations in the public interface over objects in the application. An access control policy must restrict to subjects the access to these operations. However, among the operations in the public interface, not all of them require access control. To define access control over an application, one must specify the secure subsystem, the set of operations in the public interface that need access control (see Section 4.1). Designers also select and combine access control features, the components that realize specific capabilities of access control models. The result of the composition of features is the custom access control policy that defines the operations that are authorized to subjects and the objects that subjects are allowed to access through those operations. To precisely represent this information, a custom access control policy is a set of authorizations, where each authorization is the specific privilege of a subject to perform an operation over an object of the application.

**Definition 1** An application APP is a tuple \( \langle \text{PI}, \text{Obj} \rangle \), where \( \text{PI} \) is the public interface, i.e., the set of operations available to subjects; and, \( \text{Obj} \) is the set of objects within the component.

A custom access control policy is a set of authorizations, i.e., tuples \( \langle s, op, obj \rangle \) which mean that subject \( s \) can execute an operation \( op \) over object \( obj \) of the application.

**Definition 2** An authorization is a tuple \( \langle s, op, obj \rangle \), where \( s \in \text{Subject} \) is a subject that interacts with the application; \( op \in \text{SecSubs} \) is an operation of the secure subsystem, \( \text{SecSubs} \subseteq \text{PI} \); and, \( obj \in \text{Obj} \) is an object of the application.

**Definition 3** A custom access control policy \( P \subseteq \text{Subject} \times \text{SecSubs} \times \text{Obj} \) is a set of authorizations.

Access control features are also sets of authorizations. To yield the custom access control policy, designers compose access control features using set operations.

**Definition 4** An access control feature \( sf \subseteq \text{Subject} \times \text{SecSubs} \times \text{Obj} \) is a set of authorizations.
5.3 Role Features

To realize the operations that are authorized to subjects, the feature positive permissions (PP) includes all of the authorizations of the form $\langle s, op, obj \rangle$, where the policy explicitly authorizes $s$ to access $op$, i.e., $op \in s.\text{pos}$. Similarly, the feature negative permissions (NP) realizes the explicit denial to access an operation. This feature includes all of those authorizations of the form $\langle s, op, obj \rangle$, where the policy does not explicitly deny the access to $op$, i.e., $op \notin s.\text{neg}$. To realize object-based permissions in the custom access control policy, the feature runtime permissions (RP) authorizes subjects to access specific objects. RP includes all of those authorizations $\langle s, op, obj \rangle$, where the subject has been explicitly allowed to access $obj$. Finally, the feature role hierarchy (RH) includes all of those authorizations $\langle r_a, op, obj \rangle$, where the role $r_a$ has a parent $r_b$ (i.e., $r_b \in r_a.\text{parents}$) that is already authorized to $op$ and $obj$, according to the policy (i.e., $\langle r_b, op, obj \rangle \in P$).

Definition 5 \( PP = \{ \langle s, op, obj \rangle | op \in s.\text{pos} \} \) is the set of all of those authorizations $\langle s, op, obj \rangle$ where the subject $s$ is explicitly authorized to access operation $op$.

Definition 6 \( NP = \{ \langle s, op, obj \rangle | op \notin s.\text{neg} \} \) is the set of all of those tuples $\langle s, op, obj \rangle$ where $s$ is not explicitly denied to access $op$.

Definition 7 \( RP = \{ \langle s, op, obj \rangle | \exists c \in \text{Class}(op \in c.\text{ownedOperation} \land rc_{s,c}(obj)) \} \) is the set of all of those tuples $\langle s, op, obj \rangle$ where the object $obj$ satisfies the runtime constraint $rc_{s,c}$, where $c$ is a class and $obj$ is an instance of $c$.

Definition 8 RH is the set of tuples $\langle r_a, op, obj \rangle$, where there exist a parent role $r_b$ that is authorized to access $op$ and $obj$, defined as:

$$ RH = \left\{ \langle r_a, op, obj \rangle | r_a \in \text{RoleSlice} \land \exists r_b \in \text{RoleSlice} \left( \langle r_a, r_b \rangle \in H^* \land \langle r_b, op, obj \rangle \in P \right) \right\} $$

5.4 User Features

The feature user-role assignment (URA) assigns roles to users in the custom access control policy. URA includes all of those authorizations $\langle u, op, obj \rangle$, where $u$ is a user that has assigned a role $r$ that is already authorized to access $op$ and $obj$ (i.e., $\langle r, op, obj \rangle \in P$). Mutually exclusive roles are roles that cannot be simultaneously assigned to a user. If a user has assigned two conflicting (mutually exclusive) roles, this user should not be authorized
to any of the operations and objects that have been authorized to these roles. To realize the feature separation of duty (SOD) includes all of those authorizations $\langle u, op, obj \rangle$, where $u$ is a user and there are no roles $r_a$ and $r_b$ assigned to $u$ that are mutually exclusive and that are authorized to $op$ and $obj$ (i.e. $\langle r_a, op, obj \rangle \in P$, $\langle r_b, op, obj \rangle \in P$).

Definition 9 URA is the set of tuples $\langle u, op, obj \rangle$, where there exist a role $r$ that has authorization to access $op$ and $obj$, and user $u$ has assigned role $r$, defined as:

$$URA = \left\{ \langle u, op, obj \rangle | u \in User \land \exists r \in RoleSlice \land (r \in u.roleAssignment \land \langle r, op, obj \rangle \in P) \right\}$$

Definition 10 SOD is the set of tuples $\langle u, op, obj \rangle$, where user $u$ can invoke operation $op$ on object $obj$ if there are no mutually exclusive roles assigned to $u$ that are authorized to invoke $op$ over $obj$, defined as:

$$SOD = \left\{ \langle u, op, obj \rangle | \forall r_a, r_b \in RoleSlice \land \langle r_a \in u.roleAssignment \land \langle r_a, op, obj \rangle \in P \land \langle r_b, op, obj \rangle \in P \Rightarrow \langle r_a, r_b \rangle \notin MR^* \right\}$$

5.5 Delegation Features

Each user has a set of delegated roles, which is realized by the feature user-delegation assignment (UDA). UDA includes all of those authorizations $\langle u, op, obj \rangle$, where there exists a role $r$ delegated to $u$ (i.e., $r \in u.delegatedRole$) that is already authorized to access $op$ and $obj$ (i.e., $\langle r, op, obj \rangle \in P$). The meta-model of Section 4.6 defines the attribute isDelegatable to denote roles that can be delegated. Feature delegatables (D) authorizes users to delegate any delegatable role. The authorizations in $D$ are of the form $\langle u, delegate, \langle u', r \rangle \rangle$, which means that a user $u$ is authorized to perform the delegate operation over $\langle u', r \rangle$. In other words, user $u$ can delegate to user $u'$ the role $r$. Delegation slices that are assigned to users, who obtain the authority to delegate the roles assigned to the delegation slices. For more specific forms of delegation, the feature delegation authority (DA) authorizes users to delegate specific roles, but forbids the delegatees from further delegating those roles, and the feature pass-on delegation authority (PODA) authorizes users to delegate roles and also authorizes the delegatees to subsequently delegate those roles. The DA feature includes all of those authorizations $\langle u, delegate, \langle u', r \rangle \rangle$, where there ex-
ist a delegation slice $ds$ assigned to a user and that delegation slice has assigned a role through the $da$ association end (i.e., $ds.da$). The PODA feature includes all of those authorizations $(u, delegate, \langle u', r \rangle)$, where there exist a delegation slice $ds$ assigned to a user (i.e., $r \in u.delegationAssignment$) and that delegation slice has assigned a role through the $poda$ association end (i.e., $ds.poda$).

**Definition 11** UDA is the set of tuples $(u, op, obj)$ where there exist a role $r$ that is authorized to $op$ and $obj$ and role $r$ is assigned to $u$ by delegation, defined as:

$$UDA = \left\{ \langle u, op, obj \rangle | u \in \text{User} \land \exists r \in \text{RoleSlice} (r \in u.delegatedRoles \land \langle r, op, obj \rangle \in P) \right\}$$

**Definition 12** $D$ is the set of tuples $(u, delegate, \langle u', r \rangle)$ where role $r$ is delegatable, defined as:

$$D = \{ (u, delegate, \langle u', r \rangle) | r.isDelegatable \}$$

**Definition 13** DA is the set of tuples $(u, delegate, \langle u', r \rangle)$, where $u$ has assigned role $r$, and $u$ has assigned a delegation slice $ds$ with delegation authority over role $r$, defined as:

$$DA = \{ \langle u, delegate, \langle u', r \rangle \rangle | r \in u.roleAssignment \land \exists ds \in \text{DelegationSlice} (ds \in u.delegationAssignment \land r \in ds.da) \}$$

**Definition 14** PODA is the set of tuples $(u, delegate, \langle u', r \rangle)$, where user $u$ can delegate a role $r$ to a user $u'$ if role $r$ is already assigned to user $u$, either by explicit user-role assignment, or by delegation; and, user $u$ has assigned a delegation slice $ds$ containing role $r$, defined as:

$$PODA = \{ \langle u, delegate, \langle u', r \rangle \rangle | (r \in u.roleAssignment \land \forall r \in u.delegatedRoles) \land \exists ds \in \text{DelegationSlice} (ds \in u.delegationAssignment \land r \in ds.poda) \}$$

24
5.6 MAC Features

The MAC model defines several security properties to constrain access to the system based on the relation between classifications and clearances [6]. The features that realize MAC security properties are: simple security (SS), simple integrity (SI), liberal-star (LS), strict-star-read (SSR), and strict-star-write (SSW). Note that the MAC model utilizes the term "security" to refer to "access control". For consistency with MAC, the proposed features utilize the same names as defined in [6]. To simplify notation, \( \preceq \) is the partial order relation between security levels (i.e., \( \preceq = \text{HSL}^* \)). SS authorizes to a subject \( s \) a read-only operation \( op \), if the clearance \( clr \) assigned to the subject is greater than or equal to the classification assigned to the operation \( op \). SI authorizes to a subject \( s \) a write-operation \( op \), if the clearance \( clr \) assigned to the subject is greater than or equal to the classification assigned to the operation \( op \). LS authorizes to a subject \( s \) a write-operation \( op \), if the clearance \( clr \) assigned to the subject is lower than or equal to the classification assigned to the operation \( op \). SSR authorizes to a subject \( s \) a read-only operation \( op \), if the clearance \( clr \) assigned to the subject is equal to the classification assigned to the operation \( op \). SSW authorizes to a subject \( s \) a write-operation \( op \), if the clearance \( clr \) assigned to the subject is equal to the classification assigned to the operation \( op \).

**Definition 15** SS is the set of authorizations \( \langle s, op, obj \rangle \), where operation \( op \) is read-only, and the clearance of subject \( s \) is greater than or equal to the classification of \( op \), defined as:

\[
SS = \{ \langle s, op, obj \rangle | s.cls \preceq op.clr \land \text{op.am} = \text{read} \}
\]

**Definition 16** SI is the set of authorizations \( \langle s, op, obj \rangle \), where \( op \) is a write-operation, and the clearance of subject \( s \) is greater than or equal to the classification of \( op \), defined as:

\[
SI = \{ \langle s, op, obj \rangle | s.cls \preceq op.clr \land \text{op.am} = \text{write} \}
\]

**Definition 17** LS is the set of authorizations \( \langle s, op, obj \rangle \), where \( op \) is a write-operation, and the clearance of subject \( s \) is lower than or equal to the classification of \( op \), defined as:

\[
LS = \{ \langle s, op, obj \rangle | s.clr \preceq op.cls \land \text{op.am} = \text{write} \}
\]
Definition 18 SSR is the set of authorizations \( \langle s, op, obj \rangle \), where operation \( op \) is read-only, and the clearance of subject \( s \) is equal to the classification of \( op \), defined as:

\[
SSR = \{ \langle s, op, obj \rangle | s.cls = op.clr ^ \wedge op.am = \text{read} \}
\]

Definition 19 SSW is the set of authorizations \( \langle s, op, obj \rangle \), where \( op \) is a write-operation, and the clearance of subject \( s \) is equal to the classification of \( op \), defined as:

\[
SSW = \{ \langle s, op, obj \rangle | s.cls = op.clr ^ \wedge op.am = \text{write} \}
\]

5.7 Composition of Access Control Features

After designers have selected the required features, they must compose them to yield the custom access control model for their application. This section details the process to compose features and to verify that each combination valid according to the specific restrictions imposed by access control models.

The central idea of composition is that, since each access control feature is a set of authorizations, one can utilize set operations to combine them. Depending on the chosen operations to combine features, the composition can yield different access control policies. Combining two access control features with the union operation yields the set of authorizations that belong to either of the combined features. In other words, the union operation is used to relax the restrictions of an access control model. For example, the expression \( PP \cup SS \) authorizes a subject to access an operation if the subject is explicitly allowed to access the operation, e.g., a role with a positive permission in a role-slice diagram (Definition 5), or if the subject’s clearance is greater than or equal to the classification of a read-only operation (Definition 16). In contrast, combining two features with the intersection operation yields the set of authorizations that belong to both of the features. Therefore, intersection must be used to tighten the restrictions of the access control model. For instance, the expression \( PP \cap SS \) authorizes a subject to access an operation only if the subject is both explicitly authorized to this operation and the subject’s clearance is greater than or equal to this operation’s classification. The set complement operator is not utilized to combine features. The reason is that set complement changes the semantics of access control features. For example, the feature \( NP \) (Definition 6) allows a subject to access all of the operations not explicitly denied. The complement of \( NP \) would authorize a subject to access operations
that are explicitly denied, which is an undesirable situation in terms of security.

Set operations union and intersection allow designers to combine access control features in many different ways. However, not all combinations are valid. There are several cases in which a combination of features is invalid. One of those cases is when the resulting set is empty, which means that subjects are not allowed to do anything in the system. Another case is when the resulting set is equal to the entire set of authorizations, which means that subjects can do anything in the system. Both cases yield undesirable policies. The first case is a too restrictive policy that makes the system unusable [3]. The second case renders access control useless, since it is equivalent to not having any policy in the system [31]. Therefore, those combinations must be avoided. In addition, access control models, specifically MAC, have very specific restrictions that limit the composition of features, which means that some combinations of features do not yield valid combinations, even if the result is a non-empty subset of the entire set of authorizations. Restrictions based on access control models are defined in a case-by-case basis. For instance, the MAC model does not allow to combine more than one feature that references read-only operations or more than one feature that references write-operations [6]. MAC does not impose any explicit restrictions to the composition of more than two features, which means that the validity of a composition expression containing more than two features depends only on the rules that restrict pairwise composition between the operands [6, 7, 51]. This chapter defines a composition expression as valid, as long as there are no forbidden pairs of features in the operands and the resulting set is a non-empty subset of the entire set of authorizations.

All of the above concepts are formalized as follows. There is a set $A = 2^{\text{Subject} \times \text{SecSubs} \times \text{Obj}}$ with all of the possible sets of authorizations and a set $\Omega = \{\cap, \cup\}$ with the set operations to combine them. This defines an algebra $\langle A, \Omega \rangle$, where $A$ is commutative, associative, distributive, and closed under $\Omega$. The theorems are as follows:

**Theorem 20** $A$ is commutative under $\Omega$: $A_i \cup A_j = A_j \cup A_i$, for all $A_i, A_j \in A$ and $A_i \cap A_j = A_j \cap A_i$, for all $A_i, A_j \in A$.

**Theorem 21** $A$ is associative under $\Omega$: $(A_i \cup A_j) \cup A_k = A_i \cup (A_j \cup A_k)$, for all $A_i, A_j, A_k \in A$ and $(A_i \cap A_j) \cap A_k = A_i \cap (A_j \cap A_k)$, for all $A_i, A_j, A_k \in A$.

**Theorem 22** $A$ is distributive under $\Omega$: $(A_i \cup A_j) \cap A_k = (A_i \cap A_k) \cup (A_j \cap A_k)$, for all $A_i, A_j, A_k \in A$ and $(A_i \cap A_j) \cup A_k = (A_i \cup A_k) \cap (A_j \cup A_k)$, for all $A_i, A_j, A_k \in A$. 

27
Theorem 23 \( \mathbb{A} \) is closed under \( \Omega \): For all \( A_i, A_j \in \mathbb{A} \), \( A_i \cup A_j \in \mathbb{A} \) and for all \( A_i, A_j \in \mathbb{A} \), \( A_i \cap A_j \in \mathbb{A} \).

The proofs are straightforward, since \( \mathbb{A} \) is a set and \( \Omega \) contains standard set operations.

The algebra \( \langle \mathbb{A}, \Omega \rangle \) defines a mechanism to compose sets of authorizations, regardless of whether the result is a valid set of authorizations or not. To complete the approach it is necessary to validate combinations of features. Let \( \mathbb{SF} \subset \mathbb{A} \) be the set of all of the access control features defined in Section 5. To obtain the specific restrictions imposed by access control models, \( \phi \subseteq \mathbb{SF} \times \mathbb{SF} \) relates pairs of access control features that cannot be simultaneously included in a composition expression and is defined as follows:

**Definition 24** \( \phi \subseteq \mathbb{SF} \times \mathbb{SF} \) is a relation between access control features in which, given two features \( \mathbb{SF}_i \in \mathbb{SF} \) and \( \mathbb{SF}_j \in \mathbb{SF} \), \( \mathbb{SF}_i \phi \mathbb{SF}_j \) indicates that \( \mathbb{SF}_i \) and \( \mathbb{SF}_j \) cannot be combined in any composition expression.

Table 1 details the information that relation \( \phi \) uses to determine the validity of pairwise combinations. This table determines, on a case-by-case basis, which pairs \( \mathbb{SF}_i, \mathbb{SF}_j \) of features can be simultaneously included in a composition expression. To utilize this table, one must find the row with the first feature (\( \mathbb{SF}_i \)) to combine, and then the column of the second feature (\( \mathbb{SF}_j \)) to combine. If the corresponding cell has an X, it means that both features are not allowed to be included in the same composition expression, i.e., \( \mathbb{SF}_i \phi \mathbb{SF}_j \). If the corresponding cell is empty, or the features do not appear in the table, then they can be included in a composition expression. Only the MAC specification explicitly forbids compositions of features [6], so this table contains only MAC features. The MAC model does not allow to select more than one feature that references read-only operations or more than one feature that references write-operations [6]. Therefore, Table 1 restricts the simultaneous inclusion of MAC features that reference write-operations (\( \text{SI}, \text{LI}, \text{SSW} \)). Similarly, \( \text{SS} \) and \( \text{SSR} \) that reference read-only operations, cannot be included in the same composition expression.

To validate compositions with \( \phi \) expressions, let \( \llbracket A_i \rrbracket \subseteq \mathbb{SF} \) be the set of access control features.
features that designers select and compose to yield a set of authorizations $A_i \in A$. $A_i$ is valid if it is a non-empty subset of $A$ and if $\phi$ does not prohibit any pair of features in $[A_i]$.

**Definition 25** A set of authorizations $A_i \in A$, obtained from the composition of a set of access control features $[A_i] \subseteq SF$, is a valid set of authorizations, denoted $OK(A_i)$, if and only if it satisfies the following properties:

1. $\emptyset \subset A_i \subset \text{Subject} \times \text{SecSubs} \times \text{Obj}$
2. $\forall SF_1 \in [A_i], \exists SF_j \in [A_i]$ s.t. $SF_1 \phi SF_j$

Note that the algebra $\langle A', \Omega \rangle$ obtained from the set of valid authorizations $A' = \{A_i \in A | OK(A_i)\}$ is not closed under $\Omega$, since a combination of valid sets of authorizations can yield a non-valid set of authorizations (see Definition 25). This can be problematic, since there are some valid sets of authorizations that would be undefined under this algebra. For instance, consider the expression $(A_i \cap A_j) \cup A_k$, where $OK(A_i)$, $OK(A_j)$, $OK(A_k)$, and $A_i$ and $A_j$ are disjoint. The combination $A_i \cap A_j$ is invalid, since it yields an empty set. Although the entire expression $(A_i \cap A_j) \cup A_k$ is valid, since $\emptyset \subset A_k \subset \text{Subject} \times \text{SecSubs} \times \text{Obj}$, it is undefined for the algebra $\langle A', \Omega \rangle$, since $A_i \cap A_j \notin A'$. To avoid this problem, designers must use the algebra $\langle A, \Omega \rangle$ (which is closed) to define composition expressions, and later verify the validity of the resulting set of authorizations using $OK$.

To illustrate the use of the algebra $\langle A, \Omega \rangle$, consider the following expression to validate: $PP \cup (SI \cup SS)$. According to Definition 25, one must first check that the result of the composition is a non-empty subset of $A$. Features $SI$ and $SS$ (see Definitions 15 and 16) include all of the authorizations to access operations with classification lower than or equal to the clearance of subjects, with the exception of the delegate operation (see Section 5.5), so their union is a subset of $A$. Since $SI$ and $SS$ are non-empty, the union is also non-empty. The union of $SI \cup SS$ with $PP$ is also a subset of $A$, since $PP$ (see Definition 5) does not include the delegate operation, and is non-empty, since $SI \cup SS$ is not empty. Finally, one must check that there are no forbidden pairs of features in the expression. According to Table 1 all of the pairs are allowed: $PP$ with $SI$, $PP$ with $SS$, and $SI$ with $SS$. Therefore, the expression $PP \cup (SI \cup SS)$ is valid. As another example, consider this expression: $D \cap SI$. Feature $D$ (see Definition 12), only contains references to the delegate operation, while $SI$ does not. Since $SI$ and $D$ are disjoint, the result will be empty. Therefore, $D \cap SI$ is not valid.
6  Mapping the Access Control Design to Code

To complete the integration of access control into software, it is necessary to address access control at the coding stage. A very important dimension in the implementation of access control is traceability, the ability to identify the portions in the code that realize each access control element in the design. Traceability facilitates evolution, because it assist designers and programmers to easily locate the portions of the system that need to change to adapt to new requirements. Similarly, traceability facilitates the inverse process: when a code portion changes, designers can better locate the design portions that must be modified to reflect that change.

To achieve a degree of traceability for the policy it is necessary to provide structure-preserving mappings from models to code. Such a mapping should facilitate the location of specific concerns at the code level, based on the information of the design, since both structures become purposefully similar. A key idea to achieve this kind of mapping is to adequately separate access control from non-access-control concerns. More specifically, preserving separation of access control concerns from models to code is instrumental to achieve a degree of traceability, since it insures that access control concerns could be distinguished from other concerns at both levels.

Based on the above principles, this section presents a set of approaches to realize the access control diagrams and features at the code level. These approaches assume the utilization of an object-oriented language\(^1\) and an application structure of Figure 14. The main code realizes the main concern of the application. Since subjects cannot directly access the methods in the public interface, the assumption is that there is code that is external to the main code (e.g. GUI code) that accesses the operations on behalf of the subjects. When incorporating access control code into the application, there are two main components: the policy code that provides the structures to store and query an access control policy; and, the access control enforcement code that intercepts all of the calls from the external code to the main code and constrain them based on the policy.

This section focuses on the policy code, which realizes the formal model for access control (the infrastructure for the UML extensions of Section 4), access control features, and their composition (see Section 5), and also provides methods to keep track of the active subject and to check which methods and objects are authorized/denied for each subject. The

\(^1\) The university application software uses Java
access control enforcement code, which insures that the policy code is adequately isolated from the rest of the application, will be described in Section 7.

Figure 15 is an overview of the mapping to the policy code. Access control diagrams (UML extensions of Section 4) map to a set of annotated interfaces and methods that reflect the original structure of these diagrams. The MOF meta-model of the UML extensions (see Figure 12) maps to a set of classes that utilize the Java Reflection API [59] to access the information mapped from access control diagrams. The access control features and the composition expression map to a set of classes that utilize the MOF meta-model classes to query the policy to yield sets of authorizations. The topmost layer, the session management code, provides methods to specify the active subject (the subject who interacts with the application) and to determine his/her authorization to utilize the system.

The remainder of this section details all of the above components. Section 6.1 describes the code that realizes the infrastructure for role-slice, user, and delegation diagrams, MAC features, and the MOF meta-model of these UML extensions. Section 6.2 explains the code that realizes access control features, which relies on the code from Section 6.1. Section 6.3 describes the code to compose access control features. Section 6.4 details the code to initiate
a session in the system and to query the policy. Section 6.5 assesses the degree of traceability provided by this approach.

6.1 The Code that Realizes Access Control Diagrams

This section specifies the code that realizes the structure of access control diagrams, described in Section 4. Recall that the main focus is to preserve the structure of the access control design, which means that, for each modular structure defined at the design level, there must be an analogous structure in the code. Since the infrastructure of access control diagrams is provided by the MOF meta-model defined in Section 4.6, the simplest approach would be to perform a one-to-one mapping from MOF meta-classes to classes in the implementation language (this paper utilizes Java). Specific instances of each meta-class, which correspond to concrete elements in the access control diagrams, would map to instances of these classes in the code. For instance, role slice Teacher from Figure 6 would map to an instance of class RoleSlice at the code level. This instance could be persisted as a tuple in a relational database [28] or as part of an XML file [10]. This approach has the advantage that it provides a set of classes and methods, consistent to the MOF meta-model, to query the access control policy. However, it has the disadvantage that programmers do not have a clear view of the access control policy at the code level. For instance, understanding the role slice Teacher and its assigned permissions would require that programmers perform calls to methods in the class that realizes meta-class RoleSlice to obtain its name, permissions and hierarchy relations. These actions can only be performed at runtime, since the code does not provide any adequate mechanism to understand the policy before compilation. The only option available to programmers is to look at the places where the policies have been persisted, such as relational databases or XML files, where the policy is difficult to understand as a whole. The approach proposed hereby address this problem by defining additional mappings to realize the structure of access control diagrams into similar syntactic structures at the code level.

Figure 16 describes the approach. They key idea is to take advantage of the similarities between the structure of access control diagrams and the syntax of Java code with annotations. Role slices are similar in structure to Java interfaces, since both can contain classes/interfaces with methods. Therefore, at the code level, role slices map to interfaces that contain inner interfaces with the permissions, the latter represented as annotated methods. Permission inheritance relations map to Java interface inheritance relations, since both
are partial orders. The secure subsystem, similarly to role slices, is represented as an interface that contains annotated interfaces and methods. Delegation slices are represented as Java interfaces that contain annotated interfaces mapped from role slices. Sensitivity levels form a partial order relation that is reproduced in Java through an interface hierarchy. Classifications and clearances are represented as annotations in the secure subsystem. Above this code, there is a layer of classes and interfaces that realize the MOF meta-model of the access control diagrams (see Figure 12), which accesses the policy information through the Java Reflection API [59].

To illustrate the approach, Figure 17 shows the code that maps from the Student role-slice of Figure 6. Role slice Student is mapped to an interface with the annotation @RoleSlice, which indicates that the interface describes the permissions of a role. This interface contains a set of inner interfaces that represent the permissions of the corresponding role slice. Each interface has the annotation @ReferencedClass that indicates the class in the main code to which the role slice is defining permissions. Each method within those interfaces has the tag @pos or @neg to indicate if the method is a positive or negative permission, respectively.

Similarly, Figure 18 shows the code that realizes the Teacher role slice of Figure 6. The interface Teacher is a sub-interface of Students, to reflect that role slice Teacher inherits permissions from Student. In addition, there is the annotation @SoD that indicates that Teacher and Student are mutually exclusive roles, as defined in the user diagram of Figure 7.

Figure 19 shows the implementation of the delegation diagram of Figure 8. The in-
Figure 17. Code that Realizes Role Slice Student.

```java
@RoleSlice
@clr(U.class)
public interface Student {
    @ReferencedClass(univAppl.CourseSection.class)
    public interface CourseSection {
        @pos public String getDescription();
        @pos public int getTerm();
        @pos public int sectionNumber();
        @pos public CourseSection getCourseSection(int courseNumber, int term, int sectionNumber);
    }
    @ReferencedClass(univAppl.StudentInformation.class)
    public interface StudentInformation {
        @pos public List getCourses(int term);
        @pos public List getGrades(int courseNumber, int term);
        @pos public String getName();
        @pos public String getStudentID();
        @pos public String getStudentInformation(String StudentID);
    }
}
```

Figure 18. Code that Realizes Role Slice Teacher.

```java
@RoleSlice
@clr(S.class)
@SoD(Student.class)
public interface Teacher extends Student {
    @ReferencedClass(univAppl.CourseSection.class)
    public interface CourseSection {
        @pos public List getStudents();
    }
    @ReferencedClass(univAppl.CourseDescription.class)
    public interface CourseDescription {
        @pos public void setSyllabus(String syllabus);
    }
    @ReferencedClass(univAppl.StudentInformation.class)
    public interface StudentInformation {
        @neg public List getCourses(int term);
        @pos public void setGrades(int courseNumber, int term, List grades);
    }
}
```

Figure 19. Code that Realizes a Delegation Slice.

```java
@DelegationSlice
public interface Delegations {
    @da interface Teacher {}
    @poda interface Assistant {}
}
```

Figure 20 shows the mapping from the MAC diagram of Figure 9 to a hierarchy of interfaces. The Delegations interface has the annotation @DelegationSlice to indicate that the interface contains delegation information. This interface contains inner interfaces Teacher and Assistant, with the annotations @da and @poda to indicate that the corresponding role slices have delegation authority and pass-on delegation authority, respectively.
The position of each interface in the hierarchy correspond to the order of sensitivity levels specified in Figure 9. To assign these sensitivity levels to role slices, the annotation @clr indicates the clearance of Student and Teacher (see Figures 17 and 18). Similarly, Figure 21 shows the implementation of the secure subsystem diagram into code, where each method contains the annotation @am to indicate the access mode, and the annotation @cls to indicate its classification.

Figure 22 shows the code that realize the MOF meta-model of Figure 12. Each meta-class, namely Subject, User, RoleSlice, DelegationSlice, and Operation, maps to a different interface at the code level. The classes that implement these interfaces utilize the Java reflection API to obtain the policy information mapped from the design. For instance, Figure 23 details the method getClr of the class that implements IRoleSlice, which returns the clearance for the corresponding role slice. The attribute description is an instance of java.lang.Class that corresponds to the interface implementing a role slice. The method getAnnotation obtains the value from the @clr annotation associated to description. The
public interface ISubject {
    public Set<IOperation> getPos();
    public Set<IOperation> getNeg();
    public Class<? extends SensitivityLevel> getClr();
}

public interface IUser extends ISubject {
    public Set<IRoleSlice> getRoleAssignment();
    public Set<IRoleSlice> getDelegatedRoles();
    public IDelegationSlice getDelegationAssignment();
}

public interface IRoleSlice extends ISubject {
    public Set<IRoleSlice> getSOD();
    public Set<IRoleSlice> getParents();
    public boolean isDelegatable();
}

public interface IDelegationSlice {
    public Set<IRoleSlice> getDA();
    public Set<IRoleSlice> getPODA();
}

public interface IOperation {
    public Method getMethod();
    public Class<?> getOwnerClass();
    public Class<? extends SensitivityLevel> getCls();
    public AM getAccessMode();
}

Figure 22. Interfaces that Realize the MOF Meta-Model for Access Control.

public class RoleSliceImpl implements IRoleSlice {
    private Class<?> description;
    public Class<? extends SensitivityLevel> getClr() {
        try {
            clr c = description.getAnnotation(clr.class);
            if (c!= null) {
                return c.value();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
    ...
}

Figure 23. Code of the getClr method.

only information that the code implementing the MOF meta-model does not retrieve through reflection, are users and role assignments. Since these elements change frequently over time, it is better to map that information to structures that could be modified at runtime, such as relational databases [28] or XML files [10]. For space reasons, this section does not detail the remaining code that implement the interfaces of Figure 22.
public class Authorization {
    ISubject s;
    IOperation op;
    Object obj;
    
    public Authorization (ISubject s, IOperation op, Object... obj) {
        this.s = s;
        this.op = op;
        this.obj = obj;
    }
    
    public ISubject getS() { return s; }
    public IOperation getOp() { return op; }
    public Object getObj() { return obj; }
}

Figure 24. The Code that Realizes Authorizations.

6.2 The Code that Realizes Access Control Features

This section examines the code that realizes the access control features of Chapter 5. Recall that the essential elements of access control features and the custom access control policy are the authorizations, which are tuples \((s, op, obj)\), where \(s\) is a subject, \(op\) is an operation, and \(obj\) is an object (see Definition 2). Class Authorization, shown in Figure 24 realizes authorizations. Attributes \(s\), \(op\), and \(obj\) store the subject, operation, and object of the authorization, respectively. Methods \(getS\), \(getOp\), and \(getObj\) return the values of attributes \(s\), \(op\), and \(obj\), respectively.

Access control features are sets of authorizations. The interface AuthorizationSet, shown in Figure 25, realizes sets of authorizations and provides the method contains to determine whether an authorization belongs to the set or not. The policy code realizes each access control feature as a class that implements the interface AuthorizationSet, and utilizes the interfaces from Section 6.1 to implement the method contains. The process to map each access control feature \(SF_i\) to code has the following steps:

1. Create an empty class \(C_i\) that implements interface AuthorizationSet.
2. Add method contains to class \(C_i\), such that for all instances \(c\) of class \(C_i\) and authorizations \(a \in Subject \times SecSubs \times Obj\), \(c\).contains\(a\) \iff \(a \in SF_i\).

The body of method contains of \(CP\) is of the form:

```java
public boolean contains(Authorization a) {
    return <security_feature_expression>;
}
```

where security_feature_expression is the result of using the methods from Section 6.1 to map the expression of \(SF_i\) (see Section 5) into code.

37
public interface AuthorizationSet {
    public boolean contains(Authorization a);
}

Figure 25. The Code that Realizes Sets of Authorizations.

For example, recall that Section 5 defines the simple security (SS) feature as the following set:

$$SS = \{ \langle s, op, obj \rangle | s.cls \preceq op.clr \land op.am = \text{read} \}$$

(1)

To realize (1), class SS, detailed in Figure 26, implements the interface AuthorizationSet. Method contains takes as an argument an instance a of Authorization, and determines whether a belongs to (1) or not. This method utilizes the interfaces defined in Section 6.1 to obtain the clearance of the subject, the classification and access mode of the operation. The method isAssignableFrom (from the Java Reflection API) realizes the \(\preceq\) operation: it compares clearance and classifications according to their position in the interface hierarchy of Figure 20.

```java
public class SS implements AuthorizationSet {
    @Override
    public boolean contains(Authorization a) {
        Class<? extends SensitivityLevel> clearance = a.getS().getClr();
        Class<? extends SensitivityLevel> classification = a.getOp().getCls();
        AM accessMode = a.getOp().getAccessMode();
        if (clearance != null && classification != null && accessMode != null) {
            return accessMode.equals(AM.READ) && clearance.isAssignableFrom(classification);
        }
        return false;
    }
}
```

Figure 26. The Code that realizes the SS feature.

6.3 The Code to Compose Access Control Features

This section examines the code that composes access control features into the custom access control policy. Recall that at the design level, the custom access control policy P includes the access control features selected by designers and the operations used to combine the selected features (see Section 5). To realize the custom access control policy P, the policy code defines class CP, which implements the interface AuthorizationSet. Method contains of CP realizes the composition expression defined at the design level, mapping the union operation to boolean disjunction and the intersection operation to boolean conjunction.
public class CP implements AuthorizationSet {
    private PP pp = new PP();
    private SS ss = new SS();
    private SI si = new SI();
    public boolean contains(Authorization a) {
        return pp.contains(a) && (ss.contains(a) || si.contains(a));
    }
}

Figure 27. The Code that realizes the Custom Access Control Policy.

The process to create class CP has the following steps:

1. Create an empty class CP that implements the interface AuthorizationSet.
2. For each feature SF\(_i\) selected by designers, add to CP an attribute attr\(_i\) of type C\(_i\), where C\(_i\) is the class that realizes feature SF\(_i\).
3. Add method contains to class CP, such that for all instances cp of class CP and authorizations a ∈ Subject \times SecSubs \times Obj, cp.contains(a) ⇐⇒ a ∈ P. The body of method contains of CP is of the form:

    public boolean contains(Authorization a) {
        return <composition_expression>;
    }

where composition_expression is a direct mapping from the composition expression defined at the design level. Each access control feature SF\(_i\) in the composition expression maps to the sentence attr\(_i\).contains(a), where attr\(_i\) is the attribute that realizes feature SF\(_i\), as defined during step 2. The union operation maps to the disjunction operation ‘\(|\|\)’ and the intersection operation maps to the conjunction operation ‘\(\&\&\)’.

For example, in Section 5, designers selected several access control features and defined the following custom access control policy:

\[
PP \cap (SS \cup SI)
\]  
(2)

Figure 27 shows the code of class CP that reflects the composition of features (2) of the design. Figure 27 defines attributes pp, ss, and si to realize the features of the composition expression (2). Operations ‘\(|\|\)’ and ‘\(\&\&\)’ map from set operations \(\cup\) and \(\cap\), respectively.
6.4 Session Management

A session is an interaction between a subject and an application, where the subject may perform operations over objects of the application. The policy code must check whether a subject can access those operations and objects during a session or not, thus it must keep track of the subject who is interacting with the application (the active subject), and provide mechanisms for subjects to indicate when they initiate/terminate a session with the system. Class Session, shown in Figure 28, manages the interaction with subjects and encapsulates all of the other classes of the policy code from the rest of the application. Note that all of the attributes and methods of Session are static, so this class does not need to be instantiated. Attribute as stores the active subject using an instance of the ThreadLocal class from the Java API. ThreadLocal automatically keeps a different instance of type ISubject per thread in the application. However, depending on specific requirements, one can utilize alternative implementations to keep instances of subject separated by other mechanisms, such as HTTP connections (i.e., keep one instance of type ISubject per browser connected to a web server) [59]. Before starting a session, the instance of type ISubject referenced by as is null, indicating that there are no active subjects in that thread (and that there is no active session in that thread). To initiate a new session with a subject, the application must call method start to indicate the system that this subject is active. While the session is open, the subject can interact with the application and perform different operations over objects. Class Session has an instance of CP to reference the custom access control policy. To check for permissions, the application must call method isAuthorized of Session, which takes as arguments an operation op and an object obj and verifies whether there is an active subject or not. If there is no active subject (as is null) that means that no session has been started, so isAuthorized raises an exception, otherwise isAuthorized uses CP to determine whether the active subject as is authorized or not to op and obj, according to the policy. When the subject finishes the interaction (session) with the system, the application must call method end to remove the active user from the thread and terminate the session.

6.5 Traceability Assessment

This section assesses the degree of traceability provided by the policy code. Traceability is the ability to identify which parts of the code realize each access control design element. As explained at the beginning of Section 6, a structure-preserving mapping is instrumental
public class Session {
    private static CP cp = new CP();
    private static ThreadLocal<ISubject> as = new ThreadLocal<ISubject>() {
        protected ISubject initialValue() {
            return null;
        }
    };
    public static void start(ISubject subject) {
        as.set(subject);
    }
    public static void end() {
        as.set(null);
    }
    public static boolean isAuthorized(IOperation op, Object... obj) {
        if (as.get() == null) {
            throw new RuntimeException("No active subject.");
        }
        return cp.contains(new Authorization(as.get(), op, obj));
    }
    public static ISubject getAs() {
        return as.get();
    }
}

Figure 28. The Code that realizes Session Management.

to achieve an adequate degree of traceability, since software engineers can better identify the parts in the code that realize each specific access control concern. The code described in Section 6.1 preserves the structure of access control diagrams, taking advantage of the similarity between the structure of access control diagrams and Java. Table 2 summarizes these mappings. For each entity of access control diagrams: role slices, secure subsystem, delegation slices, sensitivity levels, and the entities contained within them, exists a modular unit realizing it. Moreover, the relations between design entities are also similar in structure to the corresponding relations in the code. The policy code also preserves the structure of access control features of Section 5. As described in Section 6.2, each access control feature is realized by a unique class at the code level. The custom access control policy is realized by class CP, which references each selected feature as an attribute and maps the composition expression (see Section 6.2) to method contains.

Overall, the most important advantage of the proposed approach is that, whenever designers modify a portion of the access control design, programmers can directly identify the place in the code that must be adapted to maintain consistency between both levels. Consider the case in which a designer wants to modify a role-slice diagram. For instance, if a designer, added/removed a role slice, programmers would know that they would have to add/remove the corresponding interface (labeled with the @RoleSlice annotation) at the code level. Similarly, if designers added/removed permissions from a role slice, programmers would know that they must add/remove the corresponding method from the interface at the code
<table>
<thead>
<tr>
<th>Design-Level Entity</th>
<th>Corresponding Unit at the Code Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Role Slice</td>
<td>Interface with @RoleSlice annotation</td>
</tr>
<tr>
<td>Secure Subsystem</td>
<td>Interface with @SecureSubsystem annotation</td>
</tr>
<tr>
<td>Positive Permission</td>
<td>Method with @pos annotation</td>
</tr>
<tr>
<td>Negative Permission</td>
<td>Method with @neg annotation</td>
</tr>
<tr>
<td>Permission Inheritance Relation</td>
<td>Inheritance relation between @RoleSlice annotated interfaces</td>
</tr>
<tr>
<td>Separation of Duty</td>
<td>@SoD annotation</td>
</tr>
<tr>
<td>Delegation Slice</td>
<td>Interface with @DelegationSlice annotation</td>
</tr>
<tr>
<td>Delegation Authority Role</td>
<td>Interface with the @da annotation</td>
</tr>
<tr>
<td>Pass-on Delegation Authority Roles</td>
<td>Interface with the @poda annotation</td>
</tr>
<tr>
<td>Sensitivity Level</td>
<td>A sub-interface of SensitivityLevel</td>
</tr>
<tr>
<td>Sensitivity Level Partial Order Relation</td>
<td>Inheritance relation between sensitivity level interfaces</td>
</tr>
<tr>
<td>Access Mode</td>
<td>@am annotation at each method in the secure subsystem interface</td>
</tr>
<tr>
<td>Classification</td>
<td>@cls annotation at each method in the secure subsystem</td>
</tr>
<tr>
<td>Clearance</td>
<td>@clr annotation at each @RoleSlice annotated interface</td>
</tr>
<tr>
<td>Access Control Feature</td>
<td>A class that implements AuthorizationSet</td>
</tr>
<tr>
<td>Feature Composition</td>
<td>Method contains of class CP</td>
</tr>
</tbody>
</table>

Table 2
Traceability of Access Control Design Elements

level. Similarly, if the composition of access control features changes, programmers know that class CP must change accordingly. Having a clear link between design and code components benefits an automatic code generator tool, since it facilitates maintaining consistency between both levels and makes it possible to perform traceability analysis at the code level. Another important advantage is that the code that realizes access control features does not change during the software life cycle. The only elements that change over time are the concrete policies that are built from access control diagrams, and the composition of access control features. Only the code that depends on variable elements must be re-created when the policy changes. The remainder of the code can be written only once and used as a library.
Recall the application structure shown in Figure 14. The access control code comprises the policy code and the access control enforcement code. Section 6 described mappings from the access control design to the policy code with a focus on traceability. To take the most advantages of the policy code mapping, it is necessary to insure that the policy code is clearly identifiable from non-access control concerns. In this regard, both at design and coding levels, an essential dimension is separation of concerns [41] that focuses on distinguishing all of the important concerns of an application in modular units, allowing them to be managed independently. At the design level, access control diagrams and features isolate access control concerns. At the code level, it is very important to preserve separation of access control concerns. Having access control concerns adequately isolated in separate code modules makes it easier to cope with changes in requirements, because programmers can better distribute their tasks. Groups of engineers can work independently on access control components, or in the main application, reducing the impact that changes in either of them cause in the other. An adequate separation of concerns should also facilitate code generation, since programmers can be oblivious of the generated code. Any updates to the generated code should not affect other parts of the system.

However, modularization of access control enforcement is not easy to accomplish using traditional approaches. There is a dichotomy between the structure of requirements and the decomposition mechanisms provided by programming languages. While requirements often refer to features, capabilities, and concepts of the end-user domain, programming languages refer to classes, methods, and attributes [13]. As a consequence, access control enforcement is a crosscutting concern [16], a type of requirement that suffers of two problems: scattering, i.e, access control tends to be realized by several components; and, tangling, i.e., one component can realize both access control and other requirements simultaneously.

During the life-cycle of the university application, access control enforcement evolved from earlier versions with limited separation of concerns to later versions that improved separation of access control from the rest of the code. This section describes all of these approaches, assessing the way that each of them separates access control concerns. To facilitate the evolution of the system, the goal is to automate the process of creating access control
code from the design specifications. Code generation should facilitate developing the code of the application, since it saves the work of manually translating the access control policy into the code that realizes it. Therefore, the discussions of each approach will consider the automatic generation of parts of the access control code.

Similarly to the mappings of Section 6, each of the enforcement approaches assume the utilization of an object-oriented language (Java) and the application structure of Figure 14. Recall that the access control enforcement is the intermediary between the policy code and the rest of the application. If a subject who interacts with the application has no permissions to access a method, the enforcement code denies access to that method and raises a security exception, which should be caught by the external code and managed accordingly. For instance, if the external code is the GUI, every caught exception should trigger a notification to the user (e.g. popping up a window) that the access is denied. Similarly, if the external code is another software system that needs access to the public interface of our application, this software needs to know when access is denied to proceed accordingly.

This section addresses three issues related with separation of access control concerns:

1. How changes in main code affect access control code (and vice versa)? To better evolve a system, one should expect that access control and main code should affect each other the less possible. The better separation of access control concerns, the less dependent is the main code from changes in access control.

2. How oblivious of access control are software engineers when working on the main code? The more obliviousness of access control, the better can software engineers evolve other concerns, because there is less information that they have to learn when programming the application. When having a good separation of access control concerns, one should expect that there should be little or no access control code in the main code, thus software engineers should be more oblivious of access control.

3. How difficult is to generate access control code? Code generation is a useful tool to evolve the system, because it can automate many programming tasks. However, the more scattered and tangled is access control with the main code, the harder is to create a code generation tool that can seamlessly update the code when requirements change.

Section 7.1 describes an earlier enforcement strategy utilized in the university application, called the basic approach, which will be used as a base for comparison against other approaches. Section 7.2 describes a more elaborated approach, utilized in later stages to enforce access control in the university application, which uses the command pattern [26].
Section 7.3 describes an approach utilized later to enforce access control using aspect-oriented programming (AOP) [15,44]. Section 7.4 compares all three approaches based on the experience from the development of the university application, and utilizing separation of concerns metrics to addresses questions 1, 2, and 3.

7.1 The Basic Enforcement

To adequately enforce a policy, regardless of the strategy to store and query it, there must be code that intercepts every call to the main code that requires protection, i.e., calls to methods in the secure subsystem, and allow/deny access to those calls accordingly. This section describes the most basic approach for access control enforcement, utilized in the earlier stages of the university application life-cycle. This approach will be used as a base for comparison against the approaches of Sections 7.2 and 7.3.

Figure 29 shows a UML diagram with the structure of the application with basic enforcement. Package Main represents the main code. Package External represents all of the code that needs access to the main code. Package Security contains the policy code (package Policy) and the access control enforcement code (class BasicEnforcement). The basic enforcement approach consists of incorporating access control enforcement code directly at the places that need access control. For example, Figure 30 shows method setSyllabus of class CourseDescription (which belongs to the Main package) after adding basic access control enforcement. Before executing the original code of setSyllabus, there is a call to method enforcePolicy of class BasicEnforcement. BasicEnforcement, shown in Figure 31, uses class Session from package Policy to enforce permissions. Method enforcePolicy takes as arguments a method name (methodName), and an array of objects (obj)

2. If the session has an active subject, enforcePolicy obtains the instance (op) that corresponds to methodName, and it verifies the authorization to access the operation and the objects according to the policy. If the subject is authorized, enforcePolicy returns the control to the method, so it can be executed, otherwise it raises a runtime exception.

2 The Java notation Object... obj means that the method enforcePolicy can be called using a variable list of arguments. In Figure 30, these arguments are this and syllabus. The array obj is constructed from those two arguments.
public class CourseDescription {
    public void setSyllabus(String syllabus) {
        BasicEnforcement.enforcePolicy("setSyllabus", this, syllabus);
        <original code of setSyllabus>
    }
    ...
}

Figure 30. Basic Enforcement on class CourseDescription

public class BasicEnforcement {
    public static void enforcePolicy(String methodName, Object... obj) {
        IOperation op = Policy.findMethod(methodName, obj);
        if (!Session.isAuthorized(op, obj)) {
            throw new RuntimeException("Access Denied: " + op);
        }
    }
}

Figure 31. BasicEnforcement class

7.2 The Command-Pattern Enforcement

The command pattern [26] consists of encapsulating each method of the public interface of the system into command classes. The sole purpose of a command class is to invoke a method of the public interface and store its arguments. One application of the command pattern is to implement “undo” or “redo” of operations in applications [26]. For example, a spreadsheet may use command classes to represent the actions that a user can perform in a sheet. Each time a user executes one of those actions, the system stores an instance of the corresponding command class and its arguments. If the user wants to undo the previous action, the system retrieves the last command class instance and uses its information to undo
the operation.

The approach in this section, used in later stages of the university application lifecycle, utilizes command classes as an intermediary between the external code and the main code. Instead of adding access control at the methods in the public interface, it enforces access control at the command classes that encapsulate them. Figure 32 shows the structure of the application after incorporating access control. The main code includes, in addition to the original code from the application without access control, a set of command classes (package Command) that wrap all of the methods in the public interface (the figure shows only a few command classes). PublicOperation, shown in Figure 33, is the superclass of every command class. PublicOperation has a constructor that takes as parameters the instance over which the operation will be invoked, and the arguments to pass to that operation. Method execute delegates into executeWrappedMethod. Subclasses implement executeWrappedMethod with their own behavior (i.e., to invoke the corresponding encapsulated method). For example, Figure 34 shows the command class GetCoursesOffered. Its constructor takes as arguments an instance of Catalog. Method executeWrappedMethod invokes getCatalog over that instance.

The example code of Figure 35 shows the way to use command classes. There is an
public abstract class PublicOperation<T, C> extends OperationImpl implements IOperation {
    protected C instance;
    protected Object[] args;
    public PublicOperation(C instance, Object... args) {
        this.instance = instance;
        this.args = args;
    }
    public T execute() {
        return executeWrappedMethod();
    }
    protected abstract T executeWrappedMethod();
}

Figure 33. PublicOperation class

public class GetCoursesOffered
    extends PublicOperation<Set<CourseDescription>, Catalog> {
    public GetCoursesOffered(Catalog instance, Object... args) {
        super(instance, args);
    }
    @Override
    protected Set<CourseDescription> executeWrappedMethod() {
        return instance.getCoursesOffered();
    }
}

Figure 34. Command Class GetCoursesOffered

Catalog c = new Catalog();
...
PublicOperation op = new GetCoursesOffered(c);
op.execute();

Figure 35. Using a Command Class

instance c of Catalog. In another place of the code, there is an instance of GetCoursesOffered, initialized with the instance c. Instead of directly invoking method getCoursesOffered of Catalog, programmers must invoke method execute from op. Class GetCoursesOffered will execute getCoursesOffered over c and return the result.

SecureOperation, a subclass of PublicOperation (see Figure 36) provides access control enforcement. Every method in the secure subsystem must be encapsulated by a subclass of SecureOperation instead of PublicOperation. SecureOperation overrides method execute to add access control. First, it checks if there is an active subject in the session. If not, it raises an exception. Otherwise, it checks whether that subject is authorized or not to invoke the operation over the objects\(^3\). If the subject is authorized, it invokes executeWrappedMethod, otherwise it raises an exception. Figure 37 shows the secure

\(^3\) Method concatenate from class Util creates an array that includes the instance where the operation is invoked, and its arguments. Class Session uses this array as the object to check for authorizations.
implementation for SetSyllabus. The parameter instance of the constructor corresponds to the instance of CourseDescription where the method is being invoked. The parameter args correspond to the arguments passed to the invocation of SetSyllabus.

```java
public abstract class SecureOperation<T,C>
    extends PublicOperation<T,C> {
    public SecureOperation(C instance, Object... args) {
        super(instance, args);
    }
    public T execute () {
        if (!Session.isAuthorized(this, Util.concatenate(instance,args))) {
            throw new RuntimeException("Access Denied: ' + this);
        }
        return executeWrappedMethod();
    }
}
```

Figure 36. SecureOperation Class

```java
public class SetSyllabus extends SecureOperation<Object,CourseDescription> {
    protected Object executeWrappedMethod() {
        instance.setSyllabus((String)arg[0]);
        return null;
    }
    public SetSyllabus(CourseDescription instance, Object... args) {
        super(instance, args);
        ...
    }
}
```

Figure 37. Command Class SetSyllabus

7.3 The Aspect-Oriented Enforcement

This approach uses aspect-oriented programming (see Section 2.2) to separate access control concerns from the rest of the code [15, 44]. Figure 38 shows the structure of the application after incorporating aspect-oriented enforcement. The AccessControl aspect modularizes all of the access control checks. Since the code from AccessControl is woven during the compilation process, the main code does not have any couplings to the custom access control code. Instead, the aspect references operations in the main code, thus it is coupled to the main code. Figure 39 shows part of the AccessControl aspect. Pointcut secureSubsystem references all of the methods in the secure subsystem. An advice is woven before the execution of all of these methods. That advice obtains the instance (op) corresponding to the method being executed, and the arguments passed to it (args). Both are obtained from the join point into which the advice is woven (thisJoinPoint). Using the above information, the advice checks whether there exist or not an active subject in the session. If no subject is active, it raises an exception. Otherwise, it checks whether the
public aspect AccessControl {
    pointcut secureSubsystem() :
        execution(* uniAppl.CourseSection..(..)) ||
        execution(* uniAppl.StudentInformation..(..)) ||
        execution(* uniAppl.CourseDescription.getCourseNumber(..)) ||
        execution(* uniAppl.CourseDescription.setSyllabus(..));

    before(Object obj) : target(obj) && secureSubsystem() {
        Method thisMethod = ((MethodSignature) thisJoinPoint.getSignature()).getMethod();
        Object[] args = thisJoinPoint.getArgs();
        IOperation op = Policy.findMethod(thisMethod);
        if (!Session.isAuthorized(op, concatenate(obj, args))) {
            throw new RuntimeException("Access Denied: " + thisMethod);
        }
    }
}

Figure 39. Access Control Aspect

operation is authorized or not, according to the policy. If the subject is not authorized, the
code raises a runtime exception, and denies access to the method and the object.

7.4 Comparison between Access Control Enforcement Approaches

Sections 7.1, 7.2 and 7.3 described three approaches to access control enforcement. This section assesses the way each approach affects separation of concerns of the system, based on the experience of the development of the university application, the development
of a proof-of-concept tool for the research herein (see Section 8), and a set of metrics. Recall
the three main questions described at the beginning of Section 7:
(1) How changes in main code affect access control code (and vice versa)?
(2) How oblivious of access control are software engineers when working on the main code?
(3) How difficult is to generate access control code?

The three above questions are strongly related with separation of concerns. An adequate separation between access control and other concerns can reduce the impact that changes in access control over the main code of an application, improve the obliviousness of software engineers, and facilitate code generation.

In particular, the three questions are strongly related to specific factors that affect separation of concerns. The first factor is coupling (see Section 2.3) from the main code to access control code. The more elements in the access control code the main code is coupled to, the more likely is that the main code will be affected by changes in access control code; and, the less oblivious of access control are software engineers (because they must know more about access control to evolve the system).

The second factor is scattering, i.e., when the code realizes a concern in many different places. The more scattered is a concern across code, the more places in the code are coupled to that concern. A higher scattering of access control concerns means that the main code can suffer more changes when access control changes; and, also that software engineers are less oblivious of access control, because there are more places in the main code coupled to access control.

The third factor is tangling, i.e., when a specific module realizes more than one concern. Isolating access control into specific modules that only realize access control concerns can make it easier to write a code generator for access control, because the generator should only modify those specific modules, without worrying of affecting other concerns. A lower tangling also makes software engineers more oblivious of access control, because the non-access-control code is less mixed with access control.

To answer all of the three questions, this section discusses the way each enforcement approach affected the overall structure of the university application. In addition, this section adapts metrics from Section 2.3 to measure coupling, scattering and tangling. To quantify coupling, this section uses coupling between components (CBC), which counts, for a certain class, aspect, or package, the amount of other components to which it is coupled. The higher the CBC for the components of a system, the worse is the separation of concerns. To focus measurements on access control, this section will restrict the CBC metric to count only
Table 3
Measurements for the Main Code

<table>
<thead>
<tr>
<th>Metric</th>
<th>Basic</th>
<th>Command-Pattern</th>
<th>Aspect-Oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBC</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>CDC</td>
<td>$</td>
<td>C^{\text{SecSubs}}</td>
<td>$</td>
</tr>
<tr>
<td>CDO</td>
<td>$</td>
<td>\text{SecSubs}</td>
<td>$</td>
</tr>
</tbody>
</table>

The couplings to components in the access control enforcement code. To measure scattering of access control, this section uses concern diffusion over components (CDC) and concern diffusion over operations (CDO) that count the components and operations that realize a specific concern, respectively. To measure tangling of access control into the main code, this section applies CDC and CDO only to the main code. The assumption is that elements in the main code that realize access control are also tangled to the main concern. Therefore, the higher the values for CDC and CDO for access control concerns in the main code, the higher is the tangling (and the worse the separation of access control concerns), because there are more elements in the main code that realize both access control and the main concern.

The application of each enforcement approach during the development life-cycle, affected in a different way the university application structure. To quantitatively support the discussion of each approach, Table 3 summarizes all of the measurements to elements in the main code for the three approaches. For simplicity, the discussion makes the values independent of the specific dimensions of the university application, utilizing the following parameters:

$C^{\text{SecSubs}}$: Set of classes in the secure subsystem.
$\text{SecSubs}$: Set of methods in the secure subsystem.

The values in Table 3 reflect the way each approach affected the structure of the university application. Consider the basic approach: the incorporation of basic enforcement to the university application added the package Security, but it did not add any new classes or methods to the main code (see Figure 29). Instead, the basic enforcement added a line of code at the beginning of each method in the secure subsystem to enforce permissions, thus the main code became directly coupled to the access control enforcement code.

The main code became coupled to BasicEnforcement from Security, thus its CBC is 1. The CDC for access control in the main code includes all of the classes coupled to BasicEnforcement, which are the classes that belong to the secure subsystem ($C^{\text{SecSubs}}$).
Similarly, the CDO for access control in the main code includes only the methods in the secure subsystem (SecSubs).

The external code code of the university application, namely GUI code, did not get coupled to the access control enforcement code, because the enforcement code was hidden beneath the implementation of methods in the public interface. However, it had dependencies to other parts of the access control code: methods start and end from class Session. The external code had to call these methods whenever a user initiated/ended a session in the system. This part of the code did not significantly change during the life-cycle of the university application, thus it is not relevant for the discussion.

The application of the command pattern enforcement changed the university application to a different structure. The main code incorporated a set of command classes (see Figure 32) that encapsulated the public interface from the external code. As a result, the size of the main code increased. The main code became coupled to class SecureOperation, thus the CBC is 1. All of the command classes that encapsulate methods from the secure subsystem became coupled to SecureOperation (they are subclasses of SecureOperation), thus CDC = |SecSubs|. However, there were no methods in the main code that were directly coupled to access control, i.e., not referencing any component from Security, thus the CDO is 0.

The aspect-oriented approach (see Figure 38) did not add any new classes or methods to the main code. The AccessControl aspect eliminated dependencies from the main code to access control: no components in the main code became coupled to access control, thus the CBC measurement is 0. Consequently, the scattering of access control in the main code also disappeared, thus CDC and CDO are 0.

Using this information, one can answer the three questions defined at the beginning of this section. The first question refers to the way changes in access control affect the main code. During the development of the university application, namely the utilization of the basic and, afterwards, the command-pattern approaches, the coupling from the main code to access control was constant; both approaches were coupled to only one modular unit in the enforcement code. When using the basic enforcement, the main code was coupled to the enforcePolicy method. When using the command-pattern enforcement, the command classes were coupled to the SecureOperation class. From this point of view, the reduced amount of coupling was not significatively detrimental for the evolution of the university application. Nevertheless, the utilization of the aspect-oriented approach better separated
access control from the rest of the code, because it eliminated the coupling from the main code to the access control enforcement code. There was still coupling from the enforcement code to the main code (i.e., the `AccessControl` aspect referenced all of the methods from the secure subsystem in its pointcut), so changes in the public interface (adding/removing methods from the public interface), or in the policy (adding/removing methods from the secure subsystem), triggered changes in the aspect. However, when using the proof-of-concept tool to automatically generate the aspect from the design models, these dependencies have a minimal impact to evolve the system.

Scattering is also an important criteria to answer the first question. The higher the scattering, the more places in the main code are affected when access control changes. The utilization of the basic approach scattered access control across all of the classes in the secure subsystem (\(\text{CDC}=|C^{\text{SecSubs}}|\)). Afterwards, when utilizing the command-pattern enforcement, access control became scattered across command classes that inherited from `SecureOperation`, one per operation in the secure subsystem (\(\text{CDC}=|\text{SecSubs}|\)). Since there are less classes in the secure subsystem than methods (i.e., \(|C^{\text{SecSubs}}| \leq |\text{SecSubs}|\)), the scattering of access control across components (CDC) is smaller for the basic approach than for the command-pattern approach. In the development of the university application, this difference was compensated by the difference in size of the classes involved, since a class tied to access control in the basic approach had significantly more methods than a command class.

The second question, which refers to the obliviousness of access control when software engineers work in the main code, is also highly related to coupling and scattering. The higher the coupling to access control, the more elements about access control software engineers must know. The higher the scattering, the more places in the code software engineers will find with access control code. The basic and command-pattern approaches, with higher coupling and scattering, made software developers less oblivious of access control. However, if the command classes were automatically generated with the proof-of-concept tool, the higher CDC value for the command-pattern approach would have minimal impact to evolve the system: changes in access control would only affect command classes; and, software engineers could be oblivious of access control, because they would not need to modify command classes. The aspect-oriented approach has no couplings to access control code, thus there is no scattering. The enforcement is isolated into one specific unit (the `AccessControl` aspect), so programmers can modify the main code without worrying about access control code.
The third question, which is about the difficulty of automatically generating access control code, was not directly addressed during the development of the university application, but during the prototyping efforts of the proof-of-concept tool that realizes access control diagrams, features and automatic mappings to code. Code generation is mostly related to tangling. The more tangled is access control with other concerns, the harder is for a code generator to insert/modify access control code, because it must be careful to not interfere with non-access-control code. In this regard, comparing scattering accross operations (CDO), provides important information, because it indicates which methods implement both access control and non-access-control concerns. In the basic approach the value CDO=$|\text{SecSubs}|$ means that every method of the secure subsystem has access control tangled with the main concern. In the command-pattern approach, method `executeWrappedMethod` of each command class separates the implementation of the main concern from the code that is coupled to access control (method `execute` in the `SecurityEnforcement` class). Therefore, there is no tangling at the method level (CDO=0). The aspect-oriented approach eliminates all of the couplings to access control, thus it has no tangling (CDO=0). Tangling in the basic approach makes it harder to implement code generation. For instance, if the secure subsystem needs to add/remove methods, a code generator must change the code of all of those methods to add/remove the enforcement code. During that process, the code generator must not alter the portions of those methods that implement other concerns. As the size and complexity of the software increases, this process may become harder and error-prone. The reason is that programmers must still manually modify those methods to evolve the main concern, which may lead to errors. For instance, assume that after generating the enforcement code for the university application, method `getSyllabus` from `CourseDescription` is removed from the secure subsystem. To update the code accordingly, the code generator must be able to identify the enforcement code and remove it from `getSyllabus`. Since programmers have manually modified `getSyllabus` to implement its behavior, they may have accidentally changed the enforcement code, so the code generator tool may not recognize it.

In summary, based on our experience in the development of the university application, each approach has different levels of advantages to implement access control. At earlier stages of the development of the university, when the team utilized the basic enforcement approach, maintenance was difficult: because of the high tangling software engineers could not be oblivious of access control when modifying the main code. During the following iterations, the team redefined the architecture to incorporate the command pattern enforcement, which significantly reduced tangling and improved obliviousness of access control. At the latest stages of its life-cycle, the architecture was redesigned to incorporate the aspect-oriented
enforcement to significantly reduce the amount of classes in the system. In practice, this last change reduced the amount of code to maintain by removing the layer of command classes that acted as intermediaries between the GUI and the domain classes, while maintaining a reduced tangling with access control. Overall, the aspect-oriented and the command-pattern approach provided better separation of access control concerns than the basic approach.

8 Prototyping Effort and Applicability

Section 8.1 discusses a proof-of-concept prototype that realizes most of the ideas of this paper, namely access control diagrams, features and enforcement. Section 8.2 explores the applicability of the proposed framework for authentication and auditing concerns.

8.1 Prototyping Effort

The development of the university application yielded the initial ideas for modeling and code generation presented in this paper. Part of the work to realize these ideas was the development of a proof-of-concept meta-modeling tool to visually create meta-models, notations and code templates. This tool has three main components: the diagram creation tool, which assists designers to create access control diagrams (notation, meta-models and instantiation of those meta-models); the access control feature composer, which allows designers to select and compose access control features to create a custom access control policy; and, the code generator, that transforms access control diagrams and features into code using the command-pattern and aspect-oriented approaches. The code generation templates were based on the access control code of the different versions of the university application. Code generation was tested against a version of the university application stripped from its access control code. The tool was able to re-generate the access control enforcement of the original university application with minimal changes. For more information about this tool, the reader can refer to [42].
8.2 Applicability to other Security Concerns

The motivation for our work is to address most security concerns of an application. Although this paper focused on access control, our ongoing work is addressing a broader range of security concerns, including authentication and auditing, among others. This section briefly reviews the applicability of our framework to authentication and auditing.

Authentication is the means that a subject has to prove his/her identity to the system. Our ongoing work in this area is to enhance the meta-model from Section 4.6 to represent authentication mechanisms. A secure subsystem incorporates an attribute that defines its authentication type, so that subjects, before accessing the operations in this subsystem, should identify themselves utilizing that kind of authentication. At the code level, each enforcement strategy must map those authentication types to the corresponding code. The command pattern approach can incorporate authentication code either at the SecureOperation class or, to preserve separation of concerns, at a subclass. The aspect-oriented approach can incorporate the authentication code at the AccessControl aspect or in a separate aspect (the latter requires specifying the weaving order of the aspects). In both cases, whenever a subject tries to access a method without being authenticated, the authentication code must be executed before controlling access to the method.

Auditing is to record the actions that each subject performs when interacting with the system. At the model level, designers need to define the different parts of the system that need auditing. Ongoing work is to define an auditing slice, a UML extension very similar to a role slice. An auditing slice is a subset of the secure subsystem that includes all of the operations that need auditing, and can be associated to multiple role slices to audit the access of each role to those operations. Realizing auditing into code is also similar to access control. For the command pattern approach, SecureOperation or a subclass can include the auditing code. For the aspect-oriented approach, an aspect can perform the auditing of method accesses. Auditing is an important issue, since one have to define rules of precedence between auditing and other concerns. Should one audit before or after controlling access? Which methods need auditing before/after access control? Part of our ongoing work is to address those issues.
9 Related Work

In the UML area, there have been many research efforts. SecureUML [5] is a modeling language to design security in distributed systems. This language is based in UML, extending its semantics and notation to support RBAC, and certain authorization constraints. The focus of this approach is to utilize UML to specify access control as part of the main design of the application, and automatically generate access control infrastructures based on the design models. The approach defines a meta-model for SecureUML’s and details a methodology to integrate it into several design modeling languages. Our approach also provides a notation to represent RBAC and mechanisms to generate code from that specification. Both approaches can represent similar RBAC policies. However, our approach also provides support for DAC and MAC, and a framework to compose their features, broadening the amount of artifacts to represent access control models.

UMLSec, described in [30] and improved by Zisman et al. [64] and Popp et al [47], is an extension to UML that defines several new stereotypes towards formal security verification of elements such as: fair exchange to avoid cheating for any party in a 2-party transaction; secrecy and confidentiality of information (accessible only to the intended people); secure information flow to avoid partial leaking of sensitive information; and, secure communication links like encryption. UMLSec also provides a notation to represent RBAC policies in a manner similar to ours. Similarly to SecureUML, they do not support DAC, MAC and composition of features. However, their focus on other security concerns can complement our approach.

The network enterprise framework described in [19], utilizes UML to specify RBAC policies. Permissions are represented as UML packages and interfaces; role hierarchies are achieved by interface inheritance. This approach inspired the role-slice notation, which in contrast utilizes classes, supports permission overriding, and role hierarchies, which are defined over a special grouping unit (the role slice).

AuthUML [1] models RBAC policies using use cases and Horn clauses to represent the access control information and to check its consistency. In contrast, the work proposed herein augments UML with RBAC, MAC, and DAC capabilities.

Verifiable composition, described in [57], composes access-control behavior into an application by utilizing aspect-oriented modeling techniques, with the aim of integrating
security into a class model that allows designers to verify access-control properties. Their approach takes a generic security design and instantiates a model tied to the domain of the application. The work of [48] uses the same technique to compose RBAC and MAC into an hybrid access control model, utilizing UML class and collaboration diagrams. There are three important differences with our work. First, they do not provide a visual notation for access control models; they describe policies at the conceptual level utilizing MOF [38]. Second, the granularity of the policy elements that they compose is much coarser than our approach, which may limit the range of possible policy combinations. Third, their approach only supports RBAC and MAC, in contrast to ours that also supports DAC.

The work of Mouheb et al [34] proposes a high-level design approach to define and weave security aspect models into a base model. Their approach is similar to Song et al and Ray et al, since both approaches explicitly define weaving operations to combine security models to non-security models. An important difference is that Mouheb’s approach does not explicitly support high-level access control models, such as RBAC, MAC or DAC, focusing the utilization of a pointcut model to define places to add security in the design. Their work can complement ours, since Mouheb’s pointcut model can be utilized to specify the places in the main design (sequence diagrams) where the behavior must change to enforce a policy specified by our access control diagrams.

The work by Doan et al. [17] extends use-case, class, and sequence diagrams with tagged values representing access-control attributes, such as classification and clearance levels, etc. Their approach addresses RBAC and MAC policies. Doan’s approach does not provide a collective view of access control at a model level. As a result, access control definitions (i.e., objects, subjects and permissions) are scattered throughout the design. In contrast the approach proposed in this paper isolates access control into separate units from the rest of the design, and also provides support for DAC policies.

At the code level, there are several related approaches. Farias [22] proposes a domain-specific security language for Java applets that collects security enforcement into a separate module. The language defines a set of security preconditions for sensitive methods that are incorporated into code using program transformation. Evans et al. [21], in their Naccio approach, propose a mechanism to transform code to enforce policies in the Java virtual machine and Win32 executables. Pandey and Hashii [40] define a declarative textual language to define constraints on access to local resources and the conditions under which they apply. The approach utilizes program transformation to introduce access constraint checking into a
Java program. Erlingsson and Schneider [20] propose an approach called SASI to implement a security automaton that enforces access control. SASI stands for Security Automata SFI Implementation. SFI stands for software-fault isolation that is a technique to implement fault-isolation (i.e., avoiding that the failure of a software module corrupt others) within a single address space. The basis of this work is that SFI protects the code that enforces access control from the rest of the program. Their prototyping effort focuses in two platforms: the Gnu C compiler, and the Java Virtual Machine Language (JVML). Mourad et al [35] propose a high-level aspect-oriented framework to enforce security. The main element of their approach is a language, SHL, utilized to systematically harden security into the code by generating aspect code. Alhadidi et al. [2], Bodkin [9], Dantas [14], De Win [15], Huang et al [29] Shah et al. [54], Slowikowski et al. [55], Sewe [53], and Viega et al. [63] also utilize aspect-oriented programming to intercept methods and constrain access based on permissions.

All of the code-level approaches mentioned above have a key element in common: they incorporate access control into code by inserting enforcement code into method calls. The enforcement code checks for preconditions imposed by the access control policy and allows the execution of the method only if those preconditions are satisfied. However, none of them explicitly focus on providing a traceable link from RBAC, MAC, and DAC models to code. Our proposed approach goes a step further, providing two additional elements: a set of structure-preserving mappings to syntactic structures in the policy code, which allows programmers to understand and evolve the access control policy as a whole, and a framework to assess separation of access control concerns.

10 Conclusions and Future Work

This paper presented a set of composable access control features that realize the most widespread access control models, i.e., RBAC, MAC, and DAC. The framework tightly integrates these features into the UML, through a set of access control diagrams. Access control features also improve customizability, because designers can select features and combine them in different ways to adapt to different requirements. The use of UML as the base language for access control diagrams should make it easier the integration with standard practices for software development.

This paper also proposed an approach to map access control to two components at
the code level: the policy code, which realizes access control diagrams and features, and the enforcement code, to restrict access to methods based on information of the policy code. This paper assessed the degree of traceability of the mappings and analyzed three different access control enforcement approaches, showing that the Command-Pattern approach, and the Aspect-Oriented approach provide a better separation of concerns than the basic, more traditional, approach to enforce access control. This benefits programmers, because it can make them oblivious of access control when working on non-access-control concerns; it can reduce the impact of changes in access control over other concerns; and, it simplifies the creation of code generation tools for access control.

The specification of RBAC, MAC and DAC is sufficiently general to satisfy a broad range of requirements. However, there are more specific access control capabilities that are not included in these models or in the access control features specifications. One of the future courses of research is to define new features and diagrams to realize the following capabilities: lifetime constraints, which define the time intervals when subjects can access to the protected resources of the system [46]; role templates, i.e., roles with privileges that are parameterized from elements in the design; and, administrative RBAC (ARBAC) [49], which extends RBAC to support roles that can perform operations that modify an RBAC policy. Ongoing work is adding support for other security concerns, auditing and authentication, at both the design and coding levels.

To further improve this framework, future research considers the integration with Doan et al. work [17], which addresses access control at the requirements and design level with a focus on use cases, class, and sequence diagrams. The expected result is a comprehensive framework that integrates access control from requirements specification down to code; and, it provides security assurance both at the design and coding levels.

References


[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.


64


