A Formal Enforcement Framework for Role-Based Access Control using Aspect-Oriented Programming

Jaime Pavlich-Mariscal, Laurent Michel and Steven Demurjian

Department of Computer Science & Engineering, The University of Connecticut, Unit-2155, 371 Fairfield Road, Storrs, CT 06269- 2155. `jaime.pavlich@uconn.edu`, `{ldm,steve}@engr.uconn.edu`

Abstract. Many of today's software applications require a high-level of security, defined by a detailed policy and attained via mechanisms such as role-based access control (RBAC), mandatory access control, digital signatures, etc. The integration of the design/implementation processes of access-control policies with runtime enforcement mechanisms is crucial to achieve an acceptable level of security for a software application. Our prior research focused on formalizing the concept of a role slice, which is a unified modeling language (UML) artifact that captures RBAC security requirements by defining permissions in the form of allowable or prohibited methods, and by specifying roles as specialized class diagrams that contain those methods. This paper augments this effort by introducing a formal framework for the security of software applications that supports the automatic translation of a role-slice access-control policy (RBAC requirements) into aspect-oriented programming (AOP) enforcement code that is seamlessly integrated with the application. The formal framework provides the necessary underpinnings to automate the integration of security policies into software. A prototyping effort based on Borland's UML tool Together Control Center for defining role-slice diagrams and the associated AOP code generator is under development.

1 Introduction

Security has become a very important issue in the development of software applications. Definition and realization of access control policies, along with other security requirements, must be an integral part of the development process, to ensure that the proper level of security in an application is attained. Since access-control 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. To realize the integration of security in an application, it is necessary to consider several key elements: the access-control approach, the means to represent the access-control information during the analysis and design of the system, and the access-control mechanisms to translate those specifications to enforcement code during the implementation (or update them after deployment).

In terms of access control, there are several popular approaches: *mandatory access control* (MAC) [1,2], *discretionary access control* (DAC) [3], and *role-based access control* (RBAC) [4,5,6]. In MAC, permissions are assigned to users based on the objects they can access in a system. Each object is labeled with a *classification level* (e.g., *top secret*, *secret*, *confidential*, and *unclassified*) that represents the sensitivity of their information. To constrain the access to information, each user has a *clearance level* that defines the access to objects based on its relative order with the classification level of each object. In DAC, permissions are defined between users and objects, but there are also privileges to delegate rights to other users, i.e. a user can be granted the permission to delegate a subset of its own permissions to another user. RBAC is a more general approach where permissions are grouped in independent units called *roles*, which represent the role that a user assumes in an organization. Thus, roles, rather than permissions, are assigned to users when they initiate an interactive session with the software system. The set of privileges granted to a user is defined by the set of permissions assigned to its corresponding role.

To represent access-control information, regardless of the approach utilized, it is crucial to use a formalism that allows developers to clearly understand the security policies that they are defining, and to evolve them easily as requirements change. In this regard, visual languages can be very powerful tools; a well-designed visual representation can conceptualize the security information to developers in an intuitive fashion, facilitate changes, and hopefully reduce errors in the definition of the policy. A CASE tool that incorporates that notation for modeling, and that can automatically check the consistency of the generated models is also critical to ensure a proper security definition. Ongoing work by Doan et al. [7,8,9] is focusing on creating a framework for the definition of security policies by enhancing UML to support RBAC and MAC, and defining rules for checking the consistency of the models as an application and its security are defined and changed over time. Their focus is on extending use-case, class and sequence diagrams with tagged values representing access-control attributes, such as classification and clearance levels, lifetimes (legal time intervals for accessing elements in the model), etc. Their work associates roles with UML actors, and defines permissions as actor-use-case associations, actor-object associations, and actor-method associations (in sequence diagrams).

While their approach utilizes a visual modeling language to represent security policies, it does not provide a global view of the permissions. A security policy modeled by this method can be hard to understand by developers and security administrators, since such a policy is distributed across many UML elements rather than organized in a single UML artifact. To complement this work, and to provide a more seamless transition from an access-control policy definition to its implementation, a new visual notation was introduced, the *role slice* [10], that can be used to represent roles and their permissions for RBAC. The underlying premise is to define permissions as the *ability to invoke a method of a class*. Roles are represented as stereotyped packages, and their permission assignment is represented by a specialized class diagram containing the assigned methods. Role hierarchies are also supported; they are represented by stereotyped dependency arrows, using model composition[11] to obtain the permissions of each role based on its position in the hierarchy.

The purpose of this paper is to detail and formalize the process that translates an access-control policy into code, via an architecture that emphasizes *separation of concerns* to reduce software complexity, significantly extending our prior work on the definition of a role slice [10]. To do so, this paper introduces a specialized formalism for representing security policies which is instrumental in modularizing security concerns at design time. One contribution of this paper is to separate, at development time and through the use of *aspect-oriented programming* (AOP), the security enforcement code from the rest of the application. Without this type of support, access-control enforcement code is often scattered and tangled in the application's code, making it difficult to track an entire policy as a logical entity. For example, using a traditional object-oriented decomposition, access control code may be added at the beginning of every method, which results in modifications to many classes that are otherwise unrelated to security issues. Using aspects, that code can be isolated, resulting in a complete modularization of the security concern. The transition from security specifications to code is automated with aspect-oriented security code generation. In this regard, a second contribution of this paper is the formalization of this compilation process, where AOP generated security code is included as part of an application's software. The formalisms that we present utilize a functional notation based on structural operational semantics [12].

This paper is organized into five sections. Section 2 explains background concepts on RBAC and AOP. Section 3 formalizes the elements needed for implementing access control: the underlying object-oriented and aspect-oriented models, and role slices. Section 4 formalizes the generation of aspect-oriented access control code from a role-slice specification. Section 5 summarizes related work. Section 6 concludes the paper and reviews on-going prototyping and future work.

2 Background on RBAC and AOP

This section provides background information about the integration of access control into software applications by using RBAC, and AOP to create a software architecture that modularizes security. To begin, role-based access control (RBAC) assumes that an organization itself owns the data and not the users (who require access to the data). Access control can be established with respect to the tasks that each user performs inside the organization [13]. Thus, in RBAC permissions are assigned to roles that exist within an organization. A user can assume a role and utilize its permissions for the duration of the authorization.

Since in an organization, the set of functions associated with each role is much more stable than the users who are assigned to those roles[5], the approach limits changes to the security policy and the impact on end-user authorizations. The basic RBAC concepts used in this paper are:

Permissions represent the ability to perform a task over some part of the system. The NIST standard[6,14] deliberately leaves them as uninterpreted symbols, allowing the developers to decide the chosen interpretation according to a particular realization of the security policy. A *positive (negative) permission* explicitly grants (denies) the right to perform an action over the system. Our approach uses negative permissions only to provide overriding capabilities to the role hierarchies.

Methods of an object-oriented application are the unit of permission for our approach, allowing each role to be statically associated with the methods that are positive permissions and negative permissions. The use of methods as the level of privilege assignment has been utilized as part of our foundational security work on the object-oriented paradigm [15], our efforts on security for distributed environments [16,17], and for the integration of MAC and RBAC into UML [7,8,9].

Roles are the entities that represent the set of permissions to perform task in a system. Users represent individuals who interact with a system. To initiate an interaction, a user obtains a role and all its associated permissions. Roles are organized in hierarchies similar to class hierarchies in object-oriented systems; each role is associated to a set of parent roles and inherits all the permissions from them. Role hierarchies can be used for classifying roles, i.e. grouping them according to common sets of permissions. For that purpose *abstract roles* can be used, which in a role hierarchy cannot be assigned to any user. *Concrete roles* represent roles that can be assigned to a user, and are normally associated to organizational roles.

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 AOP *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 includes:

Advices An advice is a code fragment that implements a part of an aspect (e.g., access control), and is intended to be woven with the main program.

Join Points A join point is a location within a program where the aspect weaver integrates an advice.

Pointcuts A pointcut is a set of join points sharing specific static properties. For instance, in AspectJ [18], 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.

3 Formal Definitions

In this section, we detail a formal framework for modeling an object-oriented application (Section 3.1), role slices (Section 3.2), and aspect-oriented concepts (Section 3.3). The formalism employs a functional notation based on [12] to specify the operational semantics of the program transformation. The functional notation used for the program transformation is structure-driven and promotes a concise, yet precise, specification of the compilation process. For uniformity, the following conventions are used throughout the section:

– Most of the definitions use records of the form $\langle l_1 = v_1, l_2 = v_2, ..., l_n = v_n\rangle$, where each l_i is the label of the i^{th} field of the record and v_i is its value.
– The dot operator (".") is used with the label name to project on the corresponding value. For example, for a record $person = \langle name = Joe, age = 20\rangle$, the expression $person.name$ denotes the value Joe

In addition, some definitions, such as the composition function (see Def. 11), or the weaving function (see Def 15), use higher-order functions such as `map` and `foldl`. For completeness, the specification of `map` and `foldl` are: `Foldl` is a higher-order function that takes a function of two arguments, an initial value, and a list, and returns the result of applying the function recursively over every element of the list, in a left-associative way; and, `Map` is a function that takes a rewriting function and a list, and returns a list that consists of all the transformed elements.

$$\begin{array}{ll} \texttt{foldl} & = \lambda f.\lambda v.\lambda l.\,\text{if}\ \texttt{nil}\ l\ \texttt{then}\ \ v\ \texttt{else}\ (\texttt{foldl}\ f\ \ (f\ \ v\,(head\ \ l))\ \ (tail\ \ l)) \\ \texttt{map} & = \lambda f.\lambda l.\,\text{if}\ \texttt{nil}\ l\ \texttt{then}\ \ \texttt{nil}\ \texttt{else}\ (f\,(head\ \ l)) :: (\texttt{map}\ \ f\,(tail\ \ l))) \end{array}$$

3.1 Object-Oriented Definitions

This section formalizes an object-oriented application via an abstraction of a full blown object-oriented language that only retains features that are relevant to the discussion of security concerns. The top-level element *application*, contains classes and inheritance relationships. Each *class* contains a set of methods, and each *method* contains an *implementation*, which is a sequence of *method invocations*. A *subsystem* is a subset of the classes and will be used to separate secure and non-secure portions of the system.

The execution of a program is carried out by an interpreter that chains method invocations on object instances. One important element of the execution schema is that for every method executed, an extra argument representing an *environment* function is passed to every method invocation. The environment keeps the state of variables, such as the return value of a method, the credentials for the authenticated (active) role, the access control policy (available roles and role hierarchy), and the exit value of the program.

Definition 1 (Interpreter Function). *An* interpreter $I : M \to S \to Arg \to \mathcal{N}$ *is a function that, given a method, an environment (see Def. 2), and a method argument, performs a sequence of method invocations (reduction steps) and terminates with the output of an exit status (natural number):*

$$I = \lambda m.\lambda s.\lambda arg.\ (evalCall\ \langle m, s, arg\rangle)\ 'exit'$$

To define its behavior the interpreter uses the auxiliary function `evalCall` that, given a method invocation (see Def. 4), recursively evaluates the implementation of the method, executing control flow statements, performing method invocations, and altering the environment. At the end it returns a new environment. We deliberately avoid a more detailed definition of this function, because it would unnecessarily increase the complexity of our definitions, and because the interpreter and its semantics are not affected by techniques proposed herein.

Definition 2 (Environment Function). *An* environment $S : Id \to T$ *is a function that tracks global information during the execution of the application, by associating an identifier of type* Id *(e.g. a string) to an object of type* T*.*

Note that T is a sum type capable to hold a value of any type that exist within the application. An example of the values that S can assume during the execution of a program is $[`exit' \mapsto 0, `activeRole' \mapsto R, `policy' \mapsto P]$, where ‘exit’ represents the exit value of the execution of the application (see Def. 1), ‘activeRole’ is mapped to the object representing the active role of the application (see Eq. 3 and Eq. 7), and ‘policy’ is mapped to the access control policy (see Def. 10 and Eq. 3). The environment is also used to store the return value of a method after its invocation (e.g. $[`returnValue' \mapsto 5]$). For convenience, the auxiliary function

$$set = \lambda s.\lambda i.\lambda v.\lambda x.\text{if } x = i \text{ then } v \text{ else } s\ x$$

will be used to update the environment. It takes an environment s, an identifier i and a value v, and returns a new environment that contains the association $i \mapsto v$ in addition to the original ones.

Definition 3 (Method). *A* method *is a record* $\langle name, impl\rangle$*, where name is the name of the method and impl is the implementation of the method.*

When the method is evaluated, the interpreter obtains the functional implementation of the method, which is of the form $\lambda s.\lambda arg.b$, where s is the environment function, arg is the argument passed to the method [1] and b is the method implementation. This function returns a new environment containing the changes done by the execution of the method. During compilation time, the implementation of the method is treated as a sequence of the form shown in Def. 5.

Definition 4 (Method Invocation). *A* method invocation *is a record of the form $\langle m, s, arg\rangle$, where m is the invoked method, s is the environment, and arg its argument.*

The interpreter function `evalCall` is responsible for the evaluation of a method and returns a new environment that reflects all the changes and side effects resulting from the execution of the method. For example, a Java method invocation of the form `a.method(p1,p2,p3);` is expressed as $\langle method, s, \langle a, p1, p2, p3\rangle\rangle$.

Definition 5 (Implementation). *An* implementation $(inv_1, ..., inv_n)$ *is a sequence of method invocations.*

This definition *purposefully abstracts away several elements from a real implementation* (e.g., control flow statements, builtin instructions or side-effects operations such as assignments) that would add complexity to the formalization but do not affect the framework.

Definition 6 (Class). *A class is a set of methods. Because attributes are not necessary to explain our approach, we do not include them in the definition of class.*

Definition 7 (Application). *An* application *is a record $\langle C, H\rangle$, where C is the set of classes and $H \subseteq C \times C$ is the inheritance relation between classes in C, with each pair $\langle a, b\rangle \in H$ indicates that a is a subclass of b.*

Definition 8 (Subsystem). *A* subsystem *of an application $\langle C, H\rangle$ is a record $\langle SC, SH\rangle$, where $SC \subseteq C$ and SH is the projection of H onto SC.*

3.2 Role Slices

In this section, we review and formalize the role-slice artifact as it relates to RBAC and permission assignment, using a university application illustrated in Figure 1, that depicts a simplified class model that manages information about courses and students, providing access for different types of users (e.g., teachers, students, administrators, etc.). The `Course` class stores information about syllabus, credits, and enrolled students, while `StudentRecord` stores the information about a student's id number, name, and enrolled courses. The `Catalog` class shows all of the public information on the courses offered. To grant access through RBAC, we define two roles: *Teacher* is able to manage a course, define its syllabus, and obtain the list of enrolled student names; and, *Student* is able to get the basic information on a courses s/he is enrolled in, obtain their syllabus, and the number of credits. A *role slice* is used to define an *access-control policy.* A role

[1] An argument can also be a tuple of values

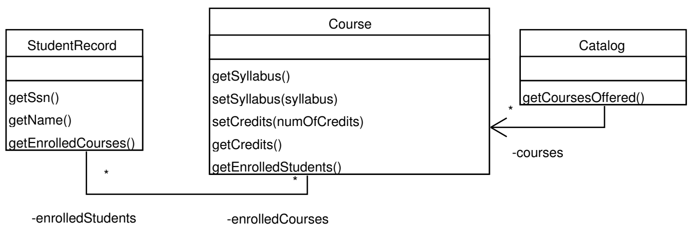

Fig. 1. Class diagram of the Courseware Application

slice denotes the set of class methods that a given role can access, and represents the separate concern that captures permissions for roles. Since a role may not require access to every class, the role-slice permission assignment is defined with respect to a subsystem. Pictorially, a role slice is represented in UML as a stereotyped package containing a specialized class diagram (see Fig. 2), that is a subset of the class model; each class present in the role slice has only the methods that are assigned to the corresponding role as positive or negative permissions. The diagram in Fig. 2 is defined over the subsystem $\langle\{$`Course`,`StudentRecord`$\}, \{\}\rangle$. `Catalog` represents publicly-accessible information and does not appear in any role slice. The two concrete role slices are `Teacher` and `Student`. Each inherit permissions from the abstract role slice `AcademicPeople` that holds the common set of permissions. Note that positive and negative permissions (methods) are represented, respectively, with the stereotype $\ll$ *pos* $\gg$ and $\ll$ *neg* $\gg$. The *role-slice composition relationship* captures inheritance among roles in a role hierarchy. Visually, it is represented as a stereotyped dependency arrow that starts from the child and points to the parent. To obtain the complete set of permissions for a role in a hierarchy, a specialized version of the *composition with override integration* defined by Clarke [11] composes two class diagrams by unifying their classes and methods. For role slices, the names of the classes are matched (i.e., classes with the same name in both role slices compose into one class in the final diagram), and the child overrides any permission definition in the parent. For the role-slice diagram in Fig. 2, a full composition operation produces the diagram shown in Fig. 3. In this new role-slice diagram, only concrete role slices are shown. To illustrate overriding, the method `getEnrolledStudents` is positive in `AcademicPeople`, and negative in `Student`. The composed role slice for `Student` shows this method as negative. Formally, a role-slice is defined as follows.

Definition 9 (Role Slice). *A* role slice *is a record* $\langle PP, NP, abstract\rangle$*, where* PP *is the set of methods with positive permissions,* NP *is the set of methods with negative permissions, and abstract indicates whether the role slice is abstract or concrete (see definition of Roles in section 2).*

Definition 10 (Access-Control Policy). *An* access-control policy *represents the sets of roles and permissions for a specific subsystem containing the classes* *requiring access control, and is a record $\langle RS, CR, S\rangle$, where RS is the set of role slices defined over the subsystem S, and $CR \subseteq RS \times RS$ is the* role-slice composition relation *that defines the role hierarchy. Each pair in CR is of the form $\langle a, b\rangle$, where a is the child role slice and b is the parent role slice.*

Definition 11 (Full composition). Full composition $fc : RS \rightarrow CR \rightarrow RS$ *is a function that takes a role slice and a composition relation as arguments, traversing the role-slice hierarchy to return the role slice composed with all its ancestors. For space reasons, no further details are given for this function.*

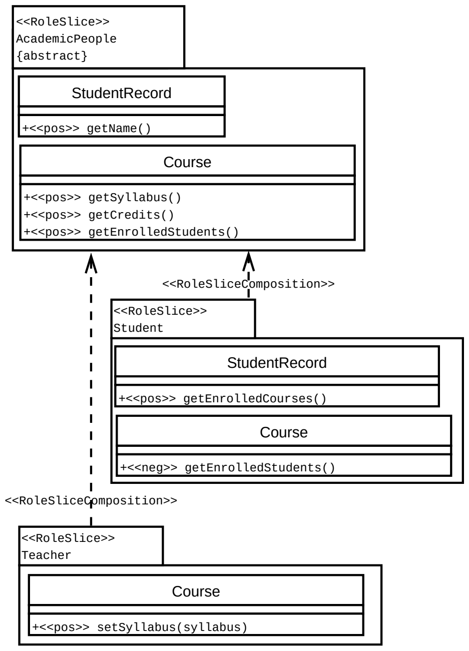

Fig. 2. Role Slice Diagram

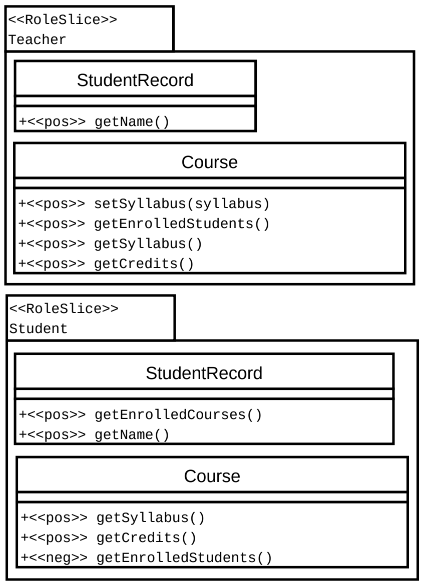

Fig. 3. Composed Role Slice Diagram

3.3 Aspect-Oriented Definitions

This section details the formal definitions of the aspect-oriented elements needed for specifying access-control code. The concepts that are introduced are abstractions of real AOP constructions that only capture the features necessary to describe the compilation of the security design. For example, join points only reference method calls initiated in specific methods, and there are no attribute-based join points and advices only represent the `around` construct.[2]

[2] `before` and `after` constructs can easily be emulated in the rewriting function

Definition 12 (Point Cut). *A* point cut *represents a set of specific locations in the code of the application that are used to integrate the aspect code. It is represented as a record $\langle caller, callee \rangle$, where caller is a method where all invocations of callee must be modified to include the aspect code.*

Definition 13 (Advice). *An* advice *is a record $\langle PC, T \rangle$, where PC is a set of point cuts and T is a rewriting function that modifies the method invocations specified in PC.*

Definition 14 (Aspect). *An* aspect *is a set of advices.*

Definition 15 (Weaving). Weaving $W : App \rightarrow A \rightarrow App$ *is a function that takes an application and an aspect as arguments, and outputs an application with all the advices of the aspect woven to its structure.*

Fig. 4 details the algorithm for weaving using λ-calculus notation. The aspect-weaving function W uses three auxiliary functions. W_C weaves one advice *adv* to the set of classes C of the application. W_M weaves one advice *adv* to the methods of each class. W_{IMPL} modifies the implementation of the current method by weaving the advice *adv* to each method invocation *inv* whenever $\langle m, inv.m \rangle$ is a point cut in the advice, with m the caller obtained from W_M and $inv.m$ the callee. The rewrite simply replaces the invocation *inv* by the invocation of a new function generated by the advice. Typically, the method invoked performs some access control and delegates back to the callee when the access is granted and raises an exception otherwise.

$$\begin{aligned}
W_{IMPL} &= \lambda m.\lambda inv.\lambda adv.\,\texttt{if}\ \ \langle m, inv.m \rangle \in adv.PC \\
&\qquad\qquad\qquad\quad \texttt{then}\ adv.T\ inv \\
&\qquad\qquad\qquad\quad \texttt{else}\ inv \\
W_M &= \lambda m.\lambda adv.\langle m.name, \texttt{map}\ \ (W_{IMPL}\ \ m)\ \ m.impl \rangle \\
W_C &= \lambda c.\lambda adv.\texttt{map}\ \ W_M\ \ c \\
W &= \lambda app.\lambda a.\langle \texttt{foldl}\ \ (\lambda c.\lambda adv.\ \texttt{map}\ W_C\ \ c\ adv)\ \ app.C\ \ a, app.H \rangle
\end{aligned}$$

Fig. 4. Weaving Algorithm

4 Enforcing RBAC using AOP

Once an access-control policy is defined by using role slices, it is necessary to translate that specification to enforcement code. This process is done automatically by a code generator, currently under development at UConn. This program takes as input a role-slice access-control policy (see Def. 10) and outputs:

- A *policy database*, containing the access control policy, and an authorization schema to store user instances and their assigned roles. The assumption is that every user is assigned only one role per session with the system. In our example, this information is accessed through the environment using the id string 'policy'.
- An *access-control aspect* that intercepts every call to the set of classes which access needs to be controlled and grants or deny access depending on the permissions stored in the policy database.

Formally, to implement access control for an application *app*, a subsystem *subs* is defined for controlling access, i.e., for the university application in Sec. 3.2 $subs = \langle \{StudentRecord, Course\}, \{\} \rangle$. The access-control aspect is defined as:

$$ac = \{adv_{\text{login}}, adv_{\text{enf}}\} \tag{1}$$

To enforce access control, the aspect uses the active role of the user currently logged in. The method that obtains the active role remains application dependent. In this example assume that when a user initiates a session in the system, a `login` method is invoked to obtain a tuple $\langle u, r \rangle$ representing an instance of the logged user (u) and his/her active role (r). The adv_{login} advice intercepts the login method and stores the active role in the environment.

$$adv_{\text{login}} = \langle \{\langle m, \texttt{login} \rangle : m \in (\bigcup_{c \in app.C} c) \backslash \{\texttt{login}\}\}, T_{\text{login}} \rangle \tag{2}$$

The pointcut of adv_{login} references all calls to the `login` method that do not occur within the `login` method itself. T_{login} is the rewriting function that retrieves the user's role from the return value of the `login` method, applies full composition to it (see Def. 11), and stores it into the environment as the 'activeRole'.

$$\begin{aligned} T_{\text{login}} = \lambda inv.\langle\, (\lambda s.\lambda arg.\texttt{let}\ y = ((inv.m\ s\ arg)\ `returnValue')\ \texttt{in} \\ \texttt{set}\ s\ `activeRole'\ (fc\ y.r\ (s\ `policy').CR)), inv.s, inv.arg \rangle \end{aligned} \tag{3}$$

The adv_{enf} advice enforces the security policy. It intercepts external calls to the subsystem *subs* (calls to methods in *subs* originating outside *subs*).

$$adv_{\text{enf}} = \langle \{\langle a, b \rangle : a \in M_{\text{ext}}, b \in M_{\text{in}}\}, T_{\text{enf}} \rangle \tag{4}$$

M_{ext} is the set of methods outside the subsystem *subs*, and M_{in} is the set of methods within subsystem *subs*.

$$M_{\text{ext}} = \bigcup_{c \in (app.C \backslash subs.SC)} c \tag{5}$$

$$M_{\text{in}} = \bigcup_{c \in subs.SC} c \tag{6}$$

T_{enf} is the method invocation rewriting function that checks positive permissions. It receives a method invocation *inv* and produces a new invocation record whose first member m is a new function that performs the access control and possibly delegates to the original function implementation when the access is granted.

$$\begin{aligned} T_{\text{enf}} = \lambda inv.\langle(\lambda s.\lambda arg.\, & \texttt{if } \ inv.m \in (s \text{ `activeRole'}).PP \\ & \texttt{then } (inv.m \ s \ arg) \\ & \texttt{else Exception}), inv.s, inv.arg\rangle \end{aligned} \tag{7}$$

Notice that negative permissions are not checked explicitly, because they are implicitly enforced by this implementation; the main purpose of negative permissions is to provide overriding when doing role slice composition.

To illustrate the ideas discussed above, we first model the university application and a secure subsystem:

$$app = \langle\{StudentRecord, Course, Catalog\}, \{\}\rangle \tag{8}$$

$$subs = \langle\{StudentRecord, Course\}, \{\}\rangle \tag{9}$$

The security policy for the subsystem *subs* is defined by the composed role slices *Teacher* and *Student* (see Fig. 3); the *Student* role slice is:

$$\text{Student} = \left\langle \left\{ \begin{array}{c} getEnrolledCourses, getName \\ getSyllabus, getCredits \end{array} \right\}, \{getEnrolledStudents\} \right\rangle \tag{10}$$

Security enforcement of the university application is implemented by an access control aspect as shown in Eq. 1. For space reasons, we only give details of the advice adv_{enf} (Eq. 4), defined with respect to the sets of external and internal methods, as:

$$M_{\text{ext}} = \{getCoursesOffered\} \tag{11}$$

$$\begin{aligned} M_{\text{in}} = \{ & getSsn, getName, getEnrolledCourses, getSyllabus, setSyllabus, \\ & getCredits, setCredits, getEnrolledStudents\} \end{aligned} \tag{12}$$

To illustrate the effects of the weaving function, we show the details of the method $getCoursesOffered$:

$$\left\langle \text{getCoursesOffered}, \left(\begin{array}{c} \langle getSyllabus, s, \langle thecourse\rangle\rangle, \\ \langle getCredits, s, \langle thecourse\rangle\rangle \end{array} \right) \right\rangle \tag{13}$$

For brevity, assume that its implementation has only two method invocations, which are executed over an instance of *Course*, called *thecourse*. Since the method $getCoursesOffered$ is external, all its invocations to internal methods are woven to advice adv_{enf}. The functional implementation of the invocation to *getSyllabus* after the weaving is:

$$\begin{aligned} \langle(\lambda s.\lambda arg.\, & \texttt{if } \ getSyllabus \in (s \text{ `activeRole'}).PP \\ & \texttt{then } (getSyllabus \ s \ arg) \\ & \texttt{else Exception}), s, \langle thecourse\rangle\rangle \end{aligned} \tag{14}$$

This woven invocation now calls *getSyllabus* only if the active role has permissions to do it.

5 Related Work

There have been previous attempts to use AOP for enforcing access control. One such approach is [19], which contains an example of composition of access-control behavior into an application by using aspect-oriented modeling techniques, with the aim of integrating security into a class model that allows designers to verify its access-control properties. Their approach takes a generic security design and instantiates it in a model tied to the domain of the application. In contrast, our code generation also requires the instantiation of the design, but only the access control aspect has dependencies with the domain class model. In addition, the role-slice notation provides a language to represent the policy that can be implemented using the aspect-oriented paradigm.

Another effort is [20] that provides a general framework for incorporating security into software via AOP, presenting a particular example access control via aspects. Their approach is similar to ours in the way they constrain method invocations based on permissions, but it differs in permission definition; in theirs, each permission is represented as a specific method tied to a framework of server objects that define them and a set of client objects that invoke them, while in ours, permissions are defined over any method in the class diagram, with a formal mapping between policy definition and code to set the base for automatic code generation. In terms for formalizing AOP, [21] proposes a monadic formal model for dynamic join points and AOP. Their notation is complete and general enough for representing AOP. Our approach is simpler (sufficient for our needs) and with the specific purpose of representing access-control enforcement.

Regarding the UML notation, [22] has proposed a Network Enterprise Framework using UML for representing RBAC requirements without separation of duty. Permissions are represented using UML packages and interfaces; role hierarchies are achieved by interface inheritance. This approach inspired the role-slice model, which in contrast uses classes, supports permission overriding, and role hierarchies, which are defined over a special grouping unit (the role slice). Another effort that relates to role slices is [23], which defines a metamodel to generate security definition languages. SecureUML [23] is an instance defined by this approach; a platform-independent security definition language for RBAC. The syntax of SecureUML has two parts: an *abstract syntax* independent from the modeling notation; and, a *concrete syntax* which can be used as an extension to a modeling language, such as UML. The abstract syntax defines basic elements to represent RBAC: *roles*, which can be assigned to *users* or *groups* of users; *permissions*, which are assigned to roles based on specific associated *constraints*; and, *actions*, which are associated with *permissions*, where a role can have a permission to execute one or more actions. SecureUML's concrete syntax is defined by mapping elements in the abstract syntax to concrete UML elements [23]. We note that our role-slice diagram and associated concepts can be an instance of the concrete-syntax of the SecureUML notation, and that our syntax and associated mappings to UML elements differ from their approach.

We also note that the role-slice diagram is only one component of our overall research. Specifically, our usage of composition in the role-slice diagram and the subsequent transition of the composed diagram into AOP enforcement code, is significantly different than the approach in SecureUML.

6 Conclusions and Future Work

This paper has formalized a compilation mechanism for security specification, that is able to support the automatic transition of a new UML artifact, the *role slice* (based on our previous work), into aspect-oriented code for security enforcement. Based on background on RBAC and AOP in Section 2, we have presented a formal functional model that captured an object-oriented application, aspect-oriented modeling, and role slices (see Section 3). This model facilitates the formalization of aspect-oriented access control generation from role slices, as presented in Section 4. Overall, we believe that our efforts to formalize the security definition and enforcement processes can be instrumental in attaining precise and accurate security specifications that can be evolved over time.

In terms of ongoing research, the effort presented in this paper is occurring concurrently with work underway at UConn to extend UML with MAC and RBAC [7,8,9], as mentioned in the introduction. As part of this effort, a team of graduate students has been integrating both that work and the work presented herein as part of Borland's UML tool Together Control Center (TCC). TCC has an open API and plug in architecture that has allowed us to extend UML diagrams to support security definition, and to define a new role-slice diagram. In addition to this prototyping effort, we are continuing our research into the role-slice model as presented in this paper. Specifically, we are interested in enhancing our model with additional security concerns, including: MAC to be able to handle security against methods based on classification and clearance; delegation to provide the ability to pass on authority (role) from one user to another; and, instance-based security that expands our work to control access to methods based on object instances in addition to our current class-based approach. Our intent is to extend the formalisims of Sections 3 and 4 with each additional access control capability.

References

1. Bell, D., LaPadula, L.: Secure computer systems: Mathematical foundations model. Technical report, Mitre Corporation (1975)
2. Biba, K.: Integrity considerations for secure computer systems. Technical report, Mitre Corporation (1977)
3. DoD: Trusted Computer System Evaluation Criteria. 5200.28-STD. DoD (1985)
4. Ting, T.C.: A user-role based data security approach. In Landwehr, C., ed.: Database Security: Status and Prospects. (1988)
5. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. IEEE Computer **29** (1996) 38–47

6. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. ACM Trans. Inf. Syst. Secur. **4** (2001) 224–274
7. Doan, T., Demurjian, S., Ting, T., Phillips, C.: RBAC/MAC security for UML. In Farkas, C., Samarati, P., eds.: Research Directions in Data and Applications Security XVIII. (2004)
8. Doan, T., Demurjian, S., Ting, T., Ketterl, A.: MAC and UML for secure software design. In: Proc. of 2nd ACM Wksp. on Formal Methods in Security Engineering, Washington D.C. (2004)
9. Doan, T., Demurjian, S., Ammar, R., Ting, T.: UML design with security integration as a first class citizen. In: Proc. of 3rd Intl. Conf. on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA'04), Cairo (2004)
10. Pavlich-Mariscal, J.A., Doan, T., Michel, L., Demurjian, S.A., Ting, T.C.: Role slices: A notation for rbac permission assignment and enforcement. In: Proceedings of 19th Annual IFIP WG 11.3 Working Conference on Data and Applications Security. (2005)
11. Clarke, S.: Composition of object-oriented software design models. PhD thesis, Dublin City University (2001)
12. Plotkin, G.: A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, CS Department, University of Aarhus (1981)
13. Ferraiolo, D., Kuhn, R.: Role-based access controls. In: 15th NIST-NCSC National Computer Security Conference. (1992) 554–563
14. Sandhu, R., Ferraiolo, D., Kuhn, R.: The NIST model for role-based access control: Towards a unified standard. (2000) 47–64
15. Demurjian, S.A., Ting, T.C.: Towards a definitive paradigm for security in object-oriented systems and applications. Journal of Computer Security **5** (1997)
16. Phillips, C., Demurjian, S., Ting, T.: Security assurance for an rbac/mac security model. In: Proc. of 2003 IEEE Info. Assurance Workshop, West Point, NY (2003)
17. Phillips, C., Demurjian, S., Ting, T.C.: Safety and liveness for an rbac/mac security model. In di Vimercati, S., Ray, I., eds.: Database and Applications Security XVII: Status and Prospects. (2004)
18. AspectJ-Team: The aspectj programming guide. http://dev.eclipse.org/viewcvs/indextech.cgi/ checkout /aspectj-home/doc/progguide/index.html (2003)
19. Song, E., Reddy, R., France, R., Ray, I., Georg, G., Alexander, R.: Verifiable composition of access control features and applications. In: Proceedings of 10th ACM Symposium on Access Control Models and Technologies (SACMAT 2005). (2005)
20. Win, B.D., Vanhaute, B., Decker, B.D.: Security through aspect-oriented programming. In: Proceedings of the IFIP TC11 WG11.4 First Annual Working Conference on Network Security, Kluwer, B.V. (2001) 125–138
21. Wand, M., Kiczales, G., Dutchyn, C.: A semantics for advice and dynamic join points in aspect-oriented programming. In Leavens, G.T., Cytron, R., eds.: FOAL 2002 Proceedings. (2002)
22. Epstein, P., Sandhu, R.: Towards a uml based approach to role engineering. In: Proceedings of the fourth ACM workshop on Role-based access control. (1999) 135–143
23. Basin, D., Doser, J., Lodderstedt, T.: Model driven security, Engineering Theories of Software Intensive Systems. (2004)