# Role Slices: A Notation for RBAC Permission Assignment and Enforcement

J.A. Pavlich-Mariscal, T. Doan, L. Michel, S.A. Demurjian, and T.C. Ting

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

**Abstract.** During the past decade, there has been an explosion in the complexity of software applications, with an increasing emphasis on software design via model-driven architectures, patterns, and models such as the unified modeling language (UML). Despite this, the integration of security concerns throughout the product life cycle has lagged, resulting in software infrastructures that are untrustworthy in terms of their ability to authenticate users and to limit them to their authorized application privileges. To address this issue, we present an approach to integrate role-based access control (RBAC) into UML at design-time for permission assignment and enforcement. Specifically, we introduce a new UML artifact, the *role slice*, supported via a new UML role-slice diagram, to capture RBAC privileges at design time within UML. Once captured, we demonstrate the utilization of aspect-oriented programming (AOP) techniques for the automatic generation of security enforcement code. Overall, we believe that our approach is an important step to upgrading security to be an indispensable part of the software process.

## 1 Introduction

In recent years, the importance of security in software systems has risen to a high level. The typical approach of integrating security into software applications at latter stages of the process can lead to serious security flaws. In order to minimize this problem, security must be considered as a first-class citizen throughout the software process. The issues that must be considered when adding security to a software application include: *security policy definition* to capture the security requirements using tools and artifacts to define and check for consistency in the security rules in order to minimize errors; and, *secure application implementation* to automatically generate security enforcement code that realizes and integrates the security policy with the application code.

In support of security policy definition, we have employed the unified modeling language, UML [17], which is the de facto standard for software modeling. In UML, while there are parallels between security and UML elements, direct support for security specification is not provided. Our ongoing work [9,8,7] has focused on the inclusion of RBAC[12] and MAC[4] by aligning the concept of role

S. Jajodia and D. Wijesekera (Eds.): Data and Applications Security 2005, LNCS 3654, pp. 40–53, 2005.

with actor, and by adding security properties to use-case, class, and sequence diagrams to capture MAC and RBAC characteristics as well as lifetimes (i.e., the legal time intervals of access to UML elements), and translate them into constraints. In support of RBAC, an actor represents one organizational role as defined by the security officer. This organizational role differs from actor-use-case roles in UML, which are used by actors to communicate with each specific use-case. Each security requirement constraint is characterized mainly by the UML elements involved, and the type of the constraints (e.g., Static Mutual Exclusion between actors and other non-actor elements). Intuitively, when the designer creates, modifies, or deletes a design element, s/he has changed the design to a new state with respect to the set of design elements that previously existed. Over time, a UML design can be characterized as the set of all states representing a specific design snapshot. Given a point of design time, a state function returns the information of the design space (UML elements, connections and security requirements) and whether an element is validly applicable at that design time. With the state information, we can perform security analysis to check the validity of the design, thereby providing a degree of security assurance.

Our work to date distributes security definition across use cases, class and sequence diagrams. While this has the advantage of closely associating security with the involved UML elements, it has the disadvantage of having the combination of the security permissions (security policy) not easily understood by designers and programmers. To complement this effort, and to provide a more seamless transition from design to code, we introduce a new artifact, the *role slice*, to visually represent permissions among roles in RBAC. In addition, our role-slice approach can separate the security aspect from the non-security aspects of code, by defining mappings to aspect-oriented programming (AOP) [15] for enforcing the access control policies that have been defined. The role-slice notation uses specialized class diagrams that define permissions and roles, in the form of UML classes and stereotyped packages, respectively, and employs UML stereotyped dependency relationships for representing role hierarchies, relying on model composition [5] for defining the permissions for each role, according to its position in the hierarchy. Since the role-slice diagram utilizes a structure akin to a class diagram, in concept, this security extension to UML occurs at the design level rather than analysis; however, MAC and RBAC defined for actors, use-cases, etc., can all be leveraged as part of the process of defining role slices.

In support of secure application implementation, once the policy has been defined and checked for consistency, the integration of security into an application's code can be greatly improved by an adequate modularization of the security-enforcement code. Using AOP, our intent is to separate application's security and non-security code, providing the means to more easily identify and locate security definitions when changes are required, thereby lessening the impact of these changes on the application. Object-oriented design/programming is centered around the ability to decompose a problem into a solution that captures only one concern (perspective) of an application. AOP addresses this limit by providing the ability to independently specify multiple orthogonal concerns.

To support this, AOP provides abstractions to define concerns with *aspects*, and a compilation technique, *aspect weaving*, that integrates aspects with the main application code via an AOP compiler. In this paper, we present the role-slice artifact and its mapping to access-control enforcement code via aspects.

The remainder of this paper is organized into four sections. Section 2 explains background concepts on RBAC, model composition, and AOP. Using this as a basis, our presentation on a model for secure design is divided into two parts: Section 3 details the definition of role slices; and, Section 4 describes techniques for mapping these definitions to AOP enforcement code. Section 5 reviews other related research efforts efforts, highlighting the influence to our work, and detailing the commonalities and differences. Section 6 contains the conclusions and reviews ongoing research.

## 2 Background Concepts

In this section, we review background concepts on role-based access control, model composition, and aspect-oriented programming. Our objective is to provide the necessary material to set the context of our work for subsequent sections.

### 2.1 Role-Based Access Control (RBAC)

*Role-based access control, RBAC* [12], is a security policy schema that assumes that the owner of the information in a software system is not the users, but the organization to which they belong. Moreover, RBAC states that the access to that information must be constrained according to the *role* that each user has been authorized and activated to interact with the system. User-role authorization is based on a set of tasks that the user performs inside the organization [11]. Users are authorized to access the system via a specific role, which holds the set of *privileges* that the user will have when interacting with the system.

There are several different interpretations for privileges or *permissions*, (we use both terms interchangeably). Depending on the specific application in which privileges/permissions are used, they can represent different concepts, such as: file access permissions in filesystems; query executions, table access, column or tuple access in database systems; or, instance access, class access, method access or attribute access in object-oriented systems. When using the object-oriented paradigm, there is a class model that represents the main structure and functionality of an application. Our assumption for incorporating RBAC into these kinds of applications, is that permissions are defined over the set of public methods present in the class model. For the purposes of the work on role slices presented herein, we define a permission as the *ability to invoke the method of a class*. We also consider negative permissions, which explicitly deny the right to invoke a method.

### 2.2 Aspect-Oriented Programming (AOP)

Software systems are inherently complex, and as information technologies evolve (e.g., faster CPUs, more memory, etc.), their complexity continues to increase.

Software developers faced challenges in the past when trying to manage that complexity, which they solved via abstraction and modularization mechanisms in programming languages, which has evolved into object-oriented design (UML) and programming (Java, C++, etc). 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 applications. One classic technique is *separation of concerns* that focuses on distinguishing all of the important concerns of an application in modular units, allowing them to be managed independently. According to Tarr et al. [19], 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 into a complete system.

In the object-oriented paradigm, the main composition and decomposition mechanism is the *class*, which while offering a degree of separation of concerns, is limited in its ability to support *crosscutting concerns*, which are requirements of an application that have two common problems:

**Scattering:** Many concerns, which are specified in the requirements, tend to be implemented by using different classes both in the design specification and in the source code. For instance, the code for implementing persistence (e.g., connecting to a database via ODBC/JDBC and issuing queries) in a banking application can be scattered among multiple classes.

**Tangling:** One class can implement several different requirements simultaneously. Using the example above, each class that has code to access the database may also have code which is related to business requirements, such as cash flow calculations, mortage rates, etc.

There are several approaches that address crosscutting concerns [15,19,13]. The key idea for most of them consists of a new form of modularization that decomposes a model into pieces that, if defined and chosen carefully, can be mapped easily from requirements specification into design artifacts and code. Each piece may represent a particular view of the system (the crosscutting concern) consisting of sets of code that (ideally) are designable and implementable separately by independent developers.

One such alternative is *aspect-oriented programming* (AOP). AOP defines a unit of decomposition, called an *aspect*, in order to isolate each crosscutting concern code into one location, and a *weaving mechanism*, to compose the aspect code with the rest of the application as part of the compilation process. Each aspect specifies the way to integrate its code with the rest of the application by using:

**Join Points,** which are points in the execution of the program where the occurence of events of interest to the crosscutting concerns can be observed, and where an aspect *advice* that reacts to such events is inserted.

**Pointcuts,** which are sets of join points that are defined through static syntactic and semantic conditions on the context surrounding the joint point. For

example, in AspectJ, pointcuts can be defined as all of the call sites to a polymorphic method within a class hierarchy.

**Advices,** which contain the code that is intended to be woven at specific join points specified within a pointcut.

Another approach to separation of concerns that complements AOP is *model composition* [5], which has its roots in *subject-oriented programming, SOP,* [13] and *multidimensional separation of concerns* [19]. Model composition is an extension to UML that decomposes a class model into pieces, called *subjects*, that represent a particular view of the system. Subjects are essentially class models that can be used to represent crosscutting concerns. Later, they are composed into larger class models until the system is finished and has all of the required functionality.

## 3 Role Slices and Secure Design

Role slices are intended to allow a software designer to capture security information in parallel with class design. The role slice provides an abstraction to collect information on the security of a role that cuts across all of the classes in an application, and to organize this information into a role-slice diagram, similar in concept to a UML class diagram. In this section, we introduce role slices, and their placement within the security design process. Specifically, Section 3.1, presents an example used throughout this paper. Next, Section 3.2 explores the role-slice artifact, including both positive and negative permissions. Lastly, Section 3.3 considers other issues related to the usage of role slices for real-world applications.

### 3.1 A Survey Institution Example

To serve as a basis for illustrating the concepts related to role slices and the generation of aspect-oriented enforcement code, we define an example application based on the following scenario:

> *A Survey Institution performs and manages public surveys. After the raw data of the survey is collected, the senior staff person adds a survey header into the database. Then, a senior or junior staff adds questions into that survey, may categorize questions, or add a new question category. Special questions with sensitive content are restricted to senior staff, who are the only ones who can modify them. Every staff person can search for surveys in the system, and, according to their privileges, access them for modification. Some survey results are public, so they can be accessed by anybody who is intersted in viewing the results.*

For simplicity and space limits, we utilize a simple design model that is better suited to explaining the concepts rather than a real-world design.

Given this scenario, Fig. 1 shows the class diagram where: *Public_Survey_Results* holds public data about statistics and questions; *Survey_List* and *Survey_Header* provide an interface to access and modify the information about surveys. Since the class *Public_Survey_Results* holds only public data, we decide to control access only to the subsystem defined by the classes *Survey_List* and *Survey_Header*. We call this set of classes a *secure subsystem*.

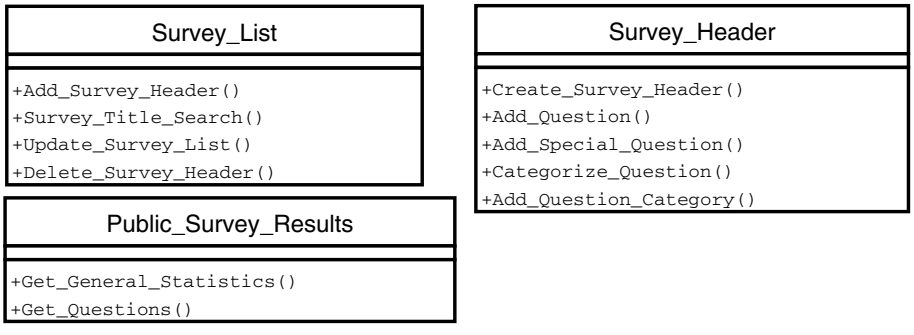


**Fig. 1.** Class Model of a Survey Management Application

### 3.2 Role Slices

A *role slice* is a structure that denotes the set of class methods that a given role can access in an application. Since we may not want to apply security to every class in the class model of the application, we define role-slice permission assignment with respect to a *secure subsystem*; the classes *Survey_List* and *Survey_Header* in Fig. 1. Visually, we represent a role slice as a UML stereotyped package containing a specialized class diagram, which is a subset of the class model of the application. Fig. 2 contains a diagram with roles slices for: *Staff* that contains common privileges; *Senior Staff* for users that have the ability to add a survey header and survey questions; and, *Junior Staff* with more limited access. Each class present in the role slice will have only methods that are assigned to the corresponding role as positive or negative permissions. An *abstract role slice*, *Staff* in Fig. 2, is tagged with the value *abstract*, cannot be assigned to a user, and is intended to be used as a mean to classify roles that have common permissions.

To represent role hierarchies, we define the *role-slice composition relationship*, which represents a hierarchical relationship between a *child* role slice and a *parent* role slice. The child role slice inherits the permissions from the parent role slice. Visually, we represent this relationship as a stereotyped dependency arrow that starts in the child and points to the parent. This relationship is shown in Fig. 2 with *Senior Staff* and *Junior Staff* as children of *Staff*. To obtain the complete set of permissions for a role in a hierarchy, we utilize the *composition with override integration* defined in [5], which composes two class diagrams by unifying their classes and methods into one diagram. For role slices, we match

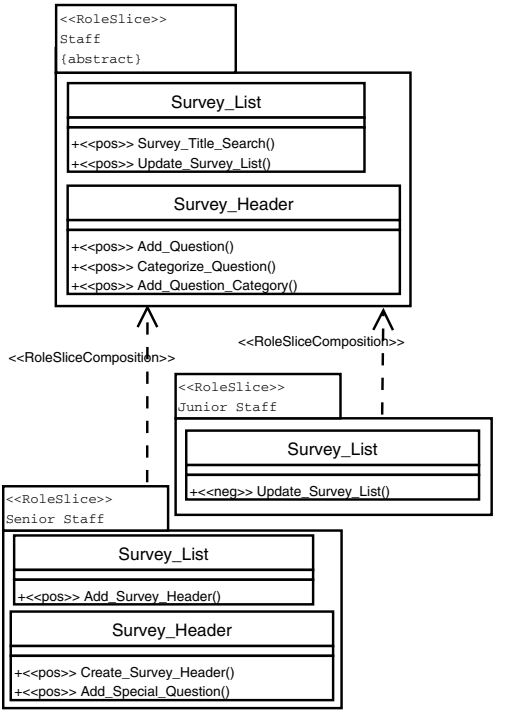


**Fig. 2.** Role-Slice Diagram

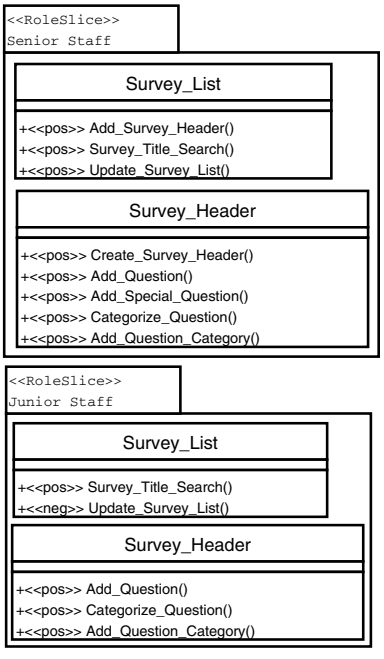


**Fig. 3.** Composed Role-Slice Diagram

the names of the classes (i.e., classes with the same name in both role slices compose into one class in the final diagram), and make the child override any permission definition in the parent.

We define permissions for the roles in Fig. 2 as follows: *Staff* is abstract and cannot be assigned to a user; and, *Senior Staff* and *Junior Staff*, which are non-abstract roles and assignable to users. The *Staff* role defines a set of common permissions: *Survey_Title_Search*, *Update_Survey_List*, *Add_Question*, *Categorize_Question* and *Add_Question_Category*. For Senior Staff, the assigned methods are: *Add_Survey_Header*, *Create_Survey_Header*, and *Add_Special_Question*. For Junior Staff, no permissions are directly assigned, but the permission to call *Update_Survey_List* is explicitly denied. Note that the method stereotypes $\ll pos \gg$ and $\ll neg \gg$ are used in the UML role-slice diagram for representing positive and negative permissions, respectively. The final set of permissions for each non-abstract role is defined through the composition of every non-abstract role slice with their ancestors, as shown in Fig. 3. Each final role slice has the union of all of the permissions from the ancestors (in this case, the parent Staff) and the respective child (Senior Staff or Junior Staff), with the exception of *Update_Survey_List*, which was overriden and restricted (negative) by Junior Staff.

### 3.3 Considerations for Real-World Scenarios

The main objective of the role-slice model is to represent a complex access control policy in a diagram that is easy to modify by security officers, and easy to understand by software designers and developers. From a practical standpoint, some issues must be taken into account:

- Any reasonably-sized application contains hundreds of classes, and the first critical decision in the security-policy definition process is to determine the subset of these classes to be included in the secure subsystem. A good approach is to only include the classes in the domain model that require access control and to exclude classes related to other concerns (e.g., I/O libraries, GUI components, etc.), since their presence would clutter the definition of role slices.
- The conceptualization of permissions during software development must be both be comprehensive and easy to understand by security officers and designers. To facilitate this, the composition relationship can be used to not only generate the final set of permissions for a security policy, but also to represent the permissions of each role at any point during the software process. This is especially useful when designing large role hierarchies, since the permissions of a concrete role can be difficult to visualize when spread across a significant portion of the role hierarchy.

Overall, issues related to the definition of security policies, their realization via role-slice diagrams, and the interplay of role-slice diagrams and application classes, are all critical to fully integrate the approach into the software process.

## 4 Mapping Role Slices to an Aspect-Oriented Application

This section details the transformation of role-slice definitions (as given in Section 3) into the application's code using aspect-oriented programming (AOP). Recall that the main purpose of a role slice is to define the access-control policy of an application regarding the authorized or prohibited methods (permissions) for each user (playing that role) interacting with the application. To map this information to aspect-oriented code and control the access to a method, it is necessary to check whether that method is denied for the active role (the role that the current user has when logged in) and raise an exception if that occurrs; otherwise, the method is allowed to execute. This process is achievable with a set of AOP advices. All of the information for security permissions (role slices) are stored in a database. When a user logs into the system, an access-control aspect obtains its role-slice permissions by intercepting the login method in the class model and retrieves from the database the pertinent role slice for the user based on his/her credentials. For method permissions, an advice intercepts every call to methods in the secure subsystem (the classes *Survey_List* and *Survey_Header* in Fig. 1), made from methods external to the subsystem (every call that originates from *Public_Survey_Results* in Fig. 1), and allow their execution if and only if they are defined as a positive permission in the corresponding role slice.

The process of mapping from a role-slice diagram to aspect-oriented enforcement code will ultimately be automated with a code generator as shown in Fig. 4. This tool, currently under development at UConn, takes a role-slice specification (diagram and composed slices) as input, and outputs:

- A *policy database* that contains all of the information on roles and permissions (as defined in the composed role slices), and an authorization schema to store user instances and their assigned roles. We assume that a user is only permitted to play a single role at any given time (but can switch roles).
- An *access-control aspect* with the following characteristics:
  - The role-slice specification, particularly the secure subsystem definition, identifies the method invocations subject to access control. From this information, pointcut definitions for the access-control aspect are obtained.
  - The advice code that is woven at the pointcuts defined previously, must have access to the policy database, and be able to grant or deny access to a user invoking access controlled methods, based on his/her active role, and the call site.

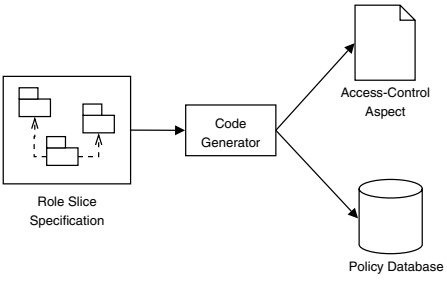


**Fig. 4.** Code Generator Scheme

We now explore an example aspect code, generated by our prototype, that enforces access control for the survey management application. Different portions of this aspect, implemented in AspectJ, are shown in Figs. 5, and 6.

Fig. 5 illustrates the portion of the access-control aspect that obtains the current active user. The `login` pointcut references a call to a method in the class `SecurityAdmin`, which returns the authenticated user. In this example, assume a multi-threading environment where each thread serves only one user. The advice using the pointcut stores the active user identification in a thread's local storage area.

Fig. 6 illustrates the code of the aspect that controls the access to methods from call sites outside the secure subsystem. The `externalCall` pointcut identifies all of the calls made to classes in the secure subsystem (i.e., `Survey_List` and `Survey_Header`), that originate from exogenous call sites. The advice code associated to this pointcut definition obtains the user's active role, and checks if s/he has a positive permission for to the intercepted method call. If not, an

```
public aspect AccessControl {
    ...
    pointcut login() : call(User SecurityAdmin.logIn(..));

    User around():login() {
        User u = proceed();
        activeUser.set(u);
        return u;
    }

    private ThreadLocal activeUser = new ThreadLocal() {
        protected Object initialValue() {
            return null;
        }
    };

    private User getActiveUser() {
        return (User)activeUser.get();
    }
    ...
}
```

**Fig. 5.** Obtaining Active User

exception is raised. Due to Java's semantics for exception handling, only runtime exceptions can be raised from this aspect. In summary, this example as given in Figs. 5 and 6, clearly illustrates the basic elements of the mapping from role slices to AOP enforcement code. Note that we are currently in the process of formalizing and implementing the role-slice code generator, as part of our overall prototyping work using Borland's UML tool Together Control Center [9,8,7].

```
public aspect AccessControl {
     ...
     pointcut externalCall() : (call(* Survey_List.*(..)) || call(* Survey_Header.*(..)))
                               && !within(Survey_List) && !within(Survey_Header);
     before() : externalCall() {
          Role r = getActiveUser().getActiveRole();
          if (!r.hasPosPermission(thisJoinPointStaticPart)) {
               throw new org.aspectj.lang.SoftException(new PermissionDeniedException());
          }
     }
     ...
}
```

**Fig. 6.** Checking of Permissions from Outside Calls

## 5 Related Work

In terms of related research, role slices are based on [10], which proposes a Network Enterprise Framework using UML to represent RBAC requirements for a specific framework given in [20]. Permissions are represented as methods of an interface-like artifact called *object handle*. Object handles are grouped in *keys*, which are stereotyped UML packages; role hierarchies are achieved by interface inheritance. In our approach, permissions are also represented as methods but, in contrast, they are grouped in role slices, which define specific rules of composition

for them. Role slices also add negative permissions and permission overriding by descendent role slices. Our approach aims to be implementation-independent for object-oriented systems.

Another effort that relates to role slices is [3], which defines a metamodel to generate security definition languages. SecureUML [3,16] 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. Actions can be *atomic*, which means that they can be mapped directly to an action in the target platform, or *composite* actions, which are higher-level actions that may not be mapped directly to the target platform, and may contain lower-level actions within them. SecureUML's concrete syntax is defined by mapping elements in the abstract syntax to concrete UML elements [3]. 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.

Another related approach, AuthUML [1,2] focuses on a process and a modeling language to express RBAC policies using *only* use cases. Permissions are defined by allowing or denying to actors the execution of use cases, and at a lower level, the execution of finer-grained conceptual operations that describe use-case behavior. Prolog-like predicates are used to represent the information and to check its consistency. In contrast, our approach uses classes to group permissions (methods), and role slices to group the entire set of permissions for a role. We do not define a specific process to develop software, so the decision of the way to utilize role slices to represent security information depends on the designers and developers. If the design of a particular application mapped each use case to a class, and each conceptual operation of a use case to a method, then both approaches would represent the same information about permissions.

The UMLsec approach [14] is another effort in security modeling related to our research. UMLsec 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/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. As currently structured, the UMLSec model is not tightly tied to RBAC, but the information it represents can be used to outline access control policies.

Regarding the aspect-oriented paradigm, [18] 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 similar effort [6], provides a general framework to incorporate security into software using AOP. Similarities to our work include: the management of authentication; and, the interception of method invocations to constrain them based on permissions. The main difference is related to permissions. In their work, 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. In contrast, in our role-slice approach, permissions are definable over any method in the class diagram, regardless of its structure.

## 6 Conclusions and Future Work

This paper has presented our efforts to define a new UML artifact to capture RBAC, the role slice and an associated diagram, and has detailed the transition from a role-slice diagram to security enforcement code, based on aspect-oriented programming (AOP). We believe that the role-slice notation, as presented in this paper, can assist designers and developers in the conceptualization of security policy, and facilitate its evolution throughout the design process. In addition, the automated mapping from a role-slice diagram (composed) to AOP enforcement code can provide a seamless transition from a security specification to code, and greatly facilitate the separation of concerns at the implementation level.

Ongoing and future research is focusing on achieving security policy composition via AOP, with the potential to also consider other, similar paradigms. We are interested in enhancing our model with additional security concerns, including: *mandatory access control* for security of methods based on classification and clearance; and, *delegation* for the ability to pass on authority (role) from one user to another. With three separate concerns (RBAC, MAC, and delegation), we must have the ability to compose any combination, which may require dynamic weaving of more than one set of constraints for access control, and the definition of different policies for separated secure subsystems. To facilitate this work on analysis and security extensions, we are formalizing role slices and their mapping to access-control aspects.

Another planned topic of research is to refine the definition of permissions, so they can support a wider-range of requirements. Specifically, we are interested in defining instance-based permissions, where roles would be authorized to invoke a method based on the instance of its class, and the value of their parameters. For example, different Senior Staff members in our example might be in charge

of different surveys; even if their roles are the same, we would want the role parameterizable by instance so that they are restricted to particular survey instances. This research is related to aspect compilers, since it needs an aspect language that could support dynamic (runtime) join points that can be selected according to instance data (class instances, parameters), so that access control can be implemented seamlessly.

Lastly, we continue our joint implementation effort, focusing on integrating the work described herein with our other UML research [9,8,7]. Our objective is to provide a complete modeling framework from analysis and design through coding, which will also include the implementation of a role-slice diagramming tool, and the mapping from role slices to AOP security enforcement code. We are utilizing Borland's UML tool Together Control Center in support of this effort.

## References

1. K. Alghathbar and D. Wijesekera. authUML: a three-phased framework to analyze access control specifications in use cases. In *FMSE '03: Proceedings of the 2003 ACM workshop on Formal methods in security engineering*, pages 77–86. ACM Press, 2003.
2. K. Alghathbar and D. Wijeskera. Consistent and complete access control policies in use cases. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 373–387. Springer, 2003.
3. D. Basin, J. Doser, and T. Lodderstedt. *Model driven security, Engineering Theories of Software Intensive Systems*. 2004.
4. D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations model. Technical report, Mitre Corporation, 1975.
5. S. Clarke. *Composition of object-oriented software design models*. PhD thesis, Dublin City University, January 2001.
6. B. De Win, B. Vanhaute, and B. De Decker. Security through aspect-oriented programming. In *Proceedings of the IFIP TC11 WG11.4 First Annual Working Conference on Network Security*, pages 125–138. Kluwer, B.V., 2001.
7. T. Doan, S. Demurjian, R. Ammar, and T.C. Ting. 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, December 2004.
8. T. Doan, S. Demurjian, T.C. Ting, and A. Ketterl. MAC and UML for secure software design. In *Proc. of 2nd ACM Wksp. on Formal Methods in Security Engineering*, Washington D.C., October 2004.
9. T. Doan, S. Demurjian, T.C. Ting, and C. Phillips. RBAC/MAC security for UML. In C. Farkas and P. Samarati, editors, *Research Directions in Data and Applications Security XVIII*, July 2004.
10. P. Epstein and R. Sandhu. Towards a UML based approach to role engineering. In *Proceedings of the fourth ACM workshop on Role-based access control*, pages 135–143, 1999.
11. D. Ferraiolo and R. Kuhn. Role-based access controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

12. D. Ferraiolo, R. Sandhu, S. Gavrila, R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, 2001.
13. W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428, 1993.
14. J. Jürjens. UMLsec: Extending UML for secure systems development. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 412–425. Springer-Verlag, 2002.
15. G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.
16. T. Lodderstedt, D.A. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 426–441. Springer-Verlag, 2002.
17. OMG. OMG-unified modeling language, v.1.5. UML Resource Page `http://www.omg.org/uml`, March 2003.
18. E. Song, R. Reddy, R. France, I. Ray, G. Georg, and R. Alexander. Verifiable composition of access control features and applications. In *Proceedings of 10th ACM Symposium on Access Control Models and Technologies (SACMAT 2005)*, 2005.
19. P. Tarr, H. Ossher, W. Harrison, and M. Sutton, Jr. Stanley. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. IEEE Computer Society Press, 1999.
20. D. Thomsen, D. O'Brien, and J. Bogle. Role based access control framework for network enterprises. In *Proceedings of 14th Annual Computer Security Application Conference*, pages 50–58, Phoenix, AZ, December 7-11 1998.