Role Slices and Runtime Permissions: Improving an AOP-based access control schema

[Position Paper]

Jaime Pavlich-Mariscal
Computer Science & Engrg.
Univ. of Connecticut
371 Fairfield Road Unit 2155
Storrs, CT 06269-2155
Phone: (860) 486-3719
jaime.pavlich@uconn.edu

Laurent Michel
Computer Science & Engrg.
Univ. of Connecticut
371 Fairfield Road Unit 2155
Storrs, CT 06269-2155
Phone: (860) 486-3719
ldm@engr.uconn.edu

Steven A. Demurjian
Computer Science & Engrg.
Univ. of Connecticut
371 Fairfield Road Unit 2155
Storrs, CT 06269-2155
Phone: (860) 486-3719
steve@engr.uconn.edu

ABSTRACT

In this paper, we present several issues that need to be addressed to incorporate dynamic permissions –permissions depending on runtime elements– into our current approach to model access control: the *role slice*. We summarize four tasks that conforms to our future research directions: extending the role-slice artifact to represent permissions based on runtime elements; refining the rules that relate role-slice hierarchies with class hierarchies; improving the generation of aspect-oriented enforcement code for dynamic permissions; and, negative permission checking at runtime. These extensions to our work are expected to produce a complete system to model and implement fine-grained access control policies in object-oriented systems.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Design Tools and Techniques—*Formal Methods*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; D.2.4 [**Programming Languages**]: Language Classifications—*Very high-level languages*

General Terms

Design, Languages, Security, Theory

Keywords

Access Control, Aspect-Oriented Programming, Separation of Concerns, UML

1. INTRODUCTION

The integration of security in the software development process has become a very important requirement to provide a degree of security assurance in applications. One of the main elements that must be implemented in a secure application is access control: the design and enforcement of policies that restrict the way that a user can interact with the system. To ensure a correct implementation of access control policies, there are several key elements to consider:

Access control model. There are several models for access control: role-based access control (RBAC) [14], in which the access to the information is constrained according to the *role* that each user has been authorized to interact with the system; mandatory access control (MAC) [5] that restricts the access to the information according to the security levels of the information and the clearance that users must have for its access; and, discretionary access control (DAC) [12], in which users get privileges to access objects, but they are also able to share them with other users, i.e., they can delegate access rights.

Notation to define the policy. UML is the de facto standard to define software artifacts. However, UML does not include specific notation to represent security requirements. To address this issue, concern-specific languages can be used to represent a security policy.

Mechanisms of policy enforcement. The policy that is defined using modeling languages during design must be translated to enforcement code. To ensure consistency between the design models and the implementation, formal mappings must be used to translate from the design notation to code.

Previously, we have shown that access control can be modeled as a separate concern through a UML-based artifact called the *role slice* [21], also providing a mapping from this notation to aspect-oriented enforcement code [22].

The role slice is an artifact to represent RBAC that denotes permission assignment for each role in the system. Visually, a role slice is represented as a specialized class diagram that shows the set of allowed methods in a class diagram. Role hierarchies are represented by using a *composition relation* that determines the way that permissions are inherited. The mapping transforms a role-slice permission assignment into aspect-oriented code, which is applied at specific pointcuts determined by the calls to the methods

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
7th International Workshop on Aspect-Oriented Modeling @ MoDELS'05
Montego Bay, Jamaica
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

that need access-control. This enforcement code uses a database for storing all permissions and roles generated from the role-slice specifications, to query them when necessary. The current permission schema references static properties of the system; it restricts the access to methods of classes to certain roles. One main objective of this paper is the extension of this schema to support dynamic permissions that are dependent on runtime information, to satisfy a wider range of security requirements and to be reactive as an application's security requirements evolve over time.

This paper presents future research directions to improve our model and the generation of enforcement code. Section 2 explores extensions to support dynamic permissions in role slices. Section 3 examines the issues that need to be solved to create enforcement code for dynamic permissions. Section 4 reviews related research as compared and contrasted to our efforts, and Section 5 concludes this paper.

2. ROLE SLICES AND DYNAMIC PERMISSIONS

In our work to date [21, 22], role slices support static permissions, where *permissions* are defined as the ability to invoke the methods of a class. However, there are application security requirements that are limited by this definition. For example, suppose that in a health-care application (HCA) the patients' medical records are stored in the class *PatientRecord*. With our current role-slice approach, the access to this class and its methods would be allowed to specific roles in the application. If a physician had been assigned permissions to invoke specific methods, such as *prescribeMedication* or *getMedicalHistory*, these permissions would be available for every instance of *PatientRecord*. In practice, a physician should only have access to the records of his/her own patients. In this case, our role slice permissions are too broad; permissions must reference specific methods in the application for particular instances. Thus, we believe that the definition of permissions to constrain method invocations of a class must be parametrized with the state of the system, represented by the state of all class instances, which includes the instance of the class whose method is invoked; and, the values of the parameters passed to the call. In this section, we explore these issues by considering extensions to role slices and to role and class hierarchies.

2.1 Role-Slice Extensions

In this section, we consider extensions to the role-slice diagram to accommodate a fine-grained representation and management of permissions, for instance-based access-control. We employ a simple and straightforward approach that associates a *constraint* with each method in a role slice. The constraint would refer to static information but also allow the tracking of runtime data that the permissions depend upon. In the role-slice framework, roles specify sets of permissions and role inheritance is used to *relax* permissions. Indeed, the permissions of a child role correspond to the union of its own permissions and the permissions of its parent role. Negative permissions are an exception mechanism that *tighten* the permissions of a role with respect to its parent role through the removal of privileges. More formally, let $p_A(m)$ be the constraint associated to a positive permission on method m in role A, and $n_A(m)$ be the constraint associated to a negative permission on method m[1]. Then, the actual constraint $c_B(m)$ for method m and role B can be computed inductively over the role hierarchy as follows:

$$\begin{aligned} c_B(m) &= p_B(m) \wedge \neg n_B(m) \Leftrightarrow B \text{ has no ancestor} \\ c_B(m) &= [p_B(m) \vee c_A(m)] \wedge \neg n_B(m) \Leftrightarrow B \text{ inherits } A \end{aligned}$$

Figure 1 provides an example of a role-slice diagram with constraints over their permissions. Every constraint is a boolean expression in a formal language (e.g., the Object Constraint Language [19]) that references elements from the class model, and represents the condition under which the method can be executed by the corresponding role. Suppose that to enforce that a physician can only prescribe medications to his/her own patients, $constraint_4$ is defined as $self.owner = Security.activeUser$, where $owner$ is an attribute of PatientRecord referencing the user who owns that record, and $Security.activeUser$ is a static attribute of the class Security (not shown) representing the user that is currently logged on. In Figure 1, the declaration

$$\ll pos \gg \{constraint_4\} prescribeMedication()$$

in role slice *Physician* would mean "allow a user with the role *Physician* to invoke method *prescribeMedication* if the owner of the patient record is that same user". Similarly, the declaration

$$\ll neg \gg \{constraint_3\} getMedicalHistory()$$

in role slice *Nurse* means "do not allow role *Nurse* to invoke method *getMedicalHistory* if *constraint3* is true" (further details of *constraint3* are omitted for space reasons). The

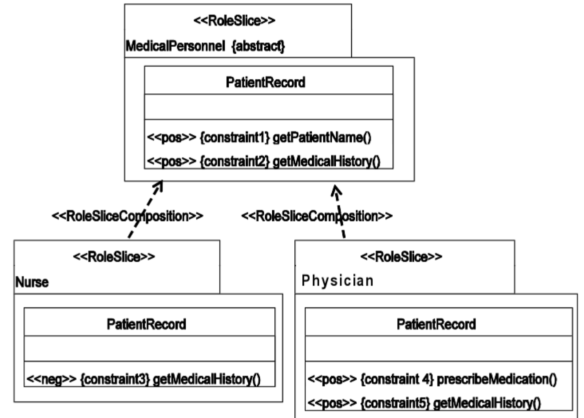

Figure 1: A Role-Slice Diagram for HCA

stereotypes $\ll pos \gg$ and $\ll neg \gg$ are useful to identify the constraints $p_A(m)$ and $n_A(m)$, respectively. Figure 2 shows the role slices *Nurse* and *Physician* after the composition. The constraints are compiled into pre-conditions (constraints) $c_{Physician}$ and c_{Nurse} for the two roles. For instance, $c_{Physician}(getMedicalHistory) = constraint_2 \vee constraint_5$ since it represents the composition of two positive stereotypes, while $c_{Nurse}(getMedicalHistory)$ uses disjunction and negation to compose $constraint_2$ and $constraint_3$

[1]If no positive permission exist for m, one can assume that $p_A(m) = true$ and if no negative permission exist for m, one can assume $n_A(m) = false$.

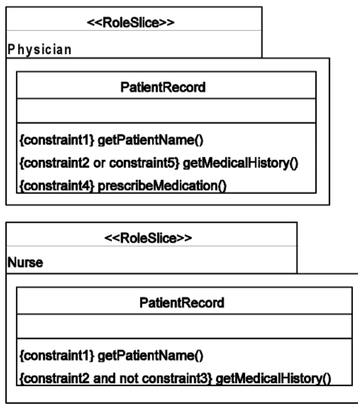

Figure 2: A Composed Role-Slice Diagram

as it aggregates both positive and negative stereotypes from the original role hierarchy. Lastly, we note that this model extension for instance-based security is in an early development stage, and will need to be formalized (both the extension and composition operation) and included as part of our ongoing work [21, 22].

2.2 Role Hierarchies and Class Hierarchies

To augment the extensions of the prior section, we must also consider the impact of the role-slice model with dynamic permissions on class hierarchies. The role-slice diagram defines its own type hierarchy with the composition relation to combine role slices. A role-slice security policy is highly dependent on the class hierarchy it controls access to, since each role slice contains a subset of the class model. In particular, the rules that govern the relation between role slices and classes must be fully defined in depth, to insure that the attainment of dynamic permissions is achieved. Specifically, in Section 2.2.1, we examine abstract classes and methods and their impact on permissions. Next, in Section 2.2.2, we investigate method overriding, and its impact on permissions. Finally, in Section 2.2.3, we explore issues related to static and dynamic typing as related to the material in Sections 2.2.1 and 2.2.2.

2.2.1 Abstract Classes and Abstract Methods.

As conceptualized, role slices do not support the ability to include abstract classes and abstract methods in their permission definitions. However, in practice, applications (e.g., HCA) often leverage abstract classes in the definition of class hierarchies. Thus, it makes sense to consider their inclusion as part of role slices and permission definition. This raises an important question: If an abstract method is included in a role-slice diagram, would it restrict the invocation of every method that implement it in a subclass?

This question has many answers, each with advantages and disadvantages. For example, one alternative would be that the inclusion of an abstract method in a role slice would allow the execution of any method that implement it in the subclasses, even if they are not assigned directly. The only exception would be if the method in the subclass is explicitly prohibited in the role slice; in that case, that method would be the only one that the role would not be allowed to invoke. This alternative is useful when we want to allow

```
abstract class AbstractDB {
    public abstract void executeQuery();
}
abstract class ConcreteDB extends AbstractDB {
    public void executeQuery() { ...  }
}
```

Figure 3: Abstract Methods

a role to invoke methods whose external behavior should be the same regardless of their implementation. For example, Figure 3 shows an abstract class that accesses a database (*AbstractDB*), and a concrete class that implements the access to a particular database (*ConcreteDB*). For this alternative, if the method *executeQuery* of *AbstractDB* is assigned to a role slice, then the role is automatically allowed to invoke every method that implement it. For instance, the role is allowed to run

```
AbstractDB a = new ConcreteDB();
a.executeQuery();
```

In this example, the permission of the role to execute queries over a database is independent of the particular database the application connects to; thus, this schema of permission assignment would be adequate. Nevertheless, this alternative has some disadvantages: it assumes that inheritance is used to create different implementations of the same behavior for an abstract class. However, inheritance is also used to specialize classes, and that can lead to some problems in the security design: if a role is allowed to access an abstract method of a class, the role would get access to invoke every implementation in the subclasses, even if the behavior of those implementations is different. In the example above, a role would get access to sensitive data if the method *executeQuery* implemented a different behavior than the expected, e.g., read data from disk.

A second alternative that cope with such problems would be to forbid the inclusion of abstract methods in role slices. By allowing only concrete methods, the permissions could be defined more precisely and without the risk of implicitly giving more permissions than necessary. The drawback in this situation would be to force the security engineer to specify concrete methods from multiple classes in the role slice, as opposed to using a smaller set of abstract methods. In summary, the alternatives to solve the issue of abstract methods need to be evaluated and compared to get a satisfactory solution that balances flexibility with safety.

2.2.2 Method overriding.

In addition to abstract methods, applications also utilize method overriding to alter functionality in a subclass. Such usage has a direct impact on the interpretation of permissions in role slices, raising a number of questions: How is the security policy impacted when a role slice has permission for methods that are overridden? Should these methods be treated (and enforced) in the same way as any other (non-overridden) permission? Should permissions assigned to overridden methods be superseded by permissions assigned to the methods that implement them in subclasses?

The possible solutions are similar to those seen with abstract methods. A first alternative is to implicitly add permissions to overriding methods if a super implementation of a method is assigned to a role slice. The advantages and disadvantages are analogous to the first alternative in Section 2.2.1. A second alternative is to allow overridden methods be treated in the same way as any other permission,

i.e., to be oblivious about the relation between the overridden method and the one that overrides it in a subclass, and treat them as if they were methods from classes without any relation, assigning them to role slices without any special restriction. For example, Figure 4 shows the code of *Class1* whose method *toString* is overridden in *Class2*. In contrast with the approach for abstract methods, if the method *toString* of *Class1* is assigned to a role slice, it would only allow the role to invoke that method and not any other in the subclasses. If we want a role to have access to the method *toString* of *Class2* as in

```
Class1 c = new Class2();
c.toString()
```

The method of *Class2* should be explicitly assigned to the role slice.

Of course, this approach is at an investigatory level, and we are exploring the details of this and other alternatives to determine a solution that again balances flexibility of permission definition with safety of permission enforcement.

```
abstract class Class1 {
    public String toString() { ...  }
}
abstract class Class2 extends Class1 {
    public String toString() { ...  }
}
```

Figure 4: Overriden Methods

2.2.3 Static vs. Dynamic Typing

Abstract methods of Section 2.2.1 and overriding methods of Section 2.2.2 are are both strongly related to whether access control should occur based on the static or the dynamic type of the object on which the method is being invoked. From this point of view, the first alternative proposed for abstract methods relies on static typing. The static type of variable a in the statement *a.executeQuery()* is *AbstractDB*, with the invocation allowed regardless of the type of the object at runtime. In contrast, the second alternative proposed for overriding methods is based on dynamic typing. The call *c.toString()* would be allowed only if the permission to the method of class *Class2* was assigned to the role slice, regardless of the declared (static) type of c. This is because the object assigned to c is of type *Class2*. Our ongoing work is investigating the implications of using static typing, dynamic typing, or both to enforce permission checking.

3. ASPECT-BASED IMPLEMENTATION OF DYNAMIC PERMISSIONS

The extended role slice model for dynamic permissions is not sufficient without a proper mapping to security enforcement code. Since a key premise is to preserve separation of concerns at every level of abstraction (from models to code), it is necessary to extend the mappings to aspect-oriented enforcement code [22]. Two alternatives to enforce runtime permissions are worth considering:

- The creation of an enforcement schema based on preexisting features of AOP languages, specifically static pointcuts (pointcuts dependent on static properties of the system, such as method calls).
- The definition of an enforcement schema using runtime pointcuts, i.e., pointcuts that depend on dynamic properties of the system, such as the state of runtime variables. This requires the usage of the latest languages and research achieved in this area (see Section 4).

The first alternative is simply to delegate the check of runtime properties to advices; they should not only intercept the methods that need access control, but additionally check runtime constraints to determine whether a role can execute a method or not on the particular instance. Those advices would be woven at places in the code where runtime properties must be checked. For example, in HCA, to ensure that a physician only accesses the records of his/her own patients, an access control advice can be woven before every call to the methods in the class *PatientRecord*. The advice would check the permission to invoke each method based on the instance of the role of the user who is logged on, and the instance of *PatientRecord* whose method is being invoked. Access would be granted only if the patient record is among the authorized instances for that role instance, and if the method is assigned to the corresponding role slice.

The second alternative, which is a potentially good venue for research, has the advantage of better modularizing the access control concern, since it isolates in the pointcut definition all of the code that checks security preconditions. For example, for HCA, an AspectJ pointcut that references all calls to *prescribeMedication* that satisfy $constraint_4$ would be as follows:

```
pointcut physicianPrescription() :
    call(* PatientRecord.prescribeMedication(..))
    && if (this.owner == Security.getActiveUser())
```

This isolation also facilitates the implementation of weaving algorithms that optimize the checking of runtime properties, such as [17], which can reduce the overhead of permission checking code. Our work continues to explore and assess alternatives, and once a viable alternative is chosen, the formal basis for the mapping from role slices to a language that supports runtime pointcuts, could be specified.

3.1 Negative Permissions and Call Graphs

Negative permissions can be a useful tool to enforce the *least privilege principle* [23]. Negative permissions can be used to explicitly restrict access to a method, and by doing so, also restrict the access to any method that directly or indirectly call it within their implementation. This *conservative* approach has several advantages. First, the technique is relatively safe; there is no possibility to call a method that calls a prohibited one, since it automatically becomes prohibited. Second, the implementation is simple; the propagation of the prohibited method (i.e., deny access to any method that calls a prohibited method) through the call graph can be done statically (at compile time). Lastly, the checking is lightweight; The denial of access occurs as early as possible, which means that it requires less time to control access, as fewer advices are woven into the code.

However, there are some drawbacks, which can be illustrated with the example code of Figure 5. In the figure, there is a method called *method1* that as part of its implementation calls a method explicitly prohibited by a negative permission (*prohibitedMethod*). Only when *condition* is false, the prohibited method is executed. Under a conservative approach, *method1* would also be prohibited, since it can potentially call to *prohibitedMethod*. This propagation of negative permissions can grow very complex, espe-

cially when there are heavily reused methods that are prohibited; propagation can deny access to more methods than expected, requiring the use overriding positive permissions to compensate.

Additionally, propagation is hard to manage, since it is necessary for the developer to keep track of both the call graph and the class model to determine the origin of the propagation. Another problem is that code duplication may occur: if access to *method1* was given to a role slice that has denied the access to *prohibitedMethod*, then another method with the same behavior of *method1* but without invoking *prohibitedMethod* must be written and assigned to that role slice.

This raises the issue of whether the enforcement code should behave conservatively, or if a more *relaxed* schema could be implemented to detect situations such as above, allowing the execution of the caller in case that the execution does not flow to the call of the prohibited method. The relaxed approach has two advantages. First, code duplication is avoided; a method that calls a prohibited one can still be assigned to the role slice, with the execution schema deciding whether to allow or deny it depending on runtime conditions. Second, the propagation is limited; propagation of negative permissions only occurs when there is total certainty that a method will call a prohibited one within its implementation.

```
class MyClass {
    ...
    public void method1() {
      if (condition) { ...  }
      else { myInstance.prohibitedMethod(); }
    }
}
```

Figure 5: Invocation of a Prohibited Method

There are two alternatives to implement a relaxed schema:

- Define an execution schema able to predict the execution path of the code in a method, i.e., determine in advance if the prohibited method in the example above will be executed or not, and then allow or deny the access to the caller accordingly.

- Define an execution schema able to rollback the execution of a method if it cannot continue due to a call to a prohibited method.

The first alternative differs from the second in that it should determine the flow of the program by predicting the values of the minimum set of variables required (in the example, the variables appearing in the boolean expression *condition*). The second approach would execute all of the code, and undo the changes if a prohibited method is reached.

For both alternatives, especially the second one, the main issue is one of tractability, to insure the undo and more importantly, that no other portion of the code has seen the changes before the undo can occur. In addition, the amount of overhead these schemas would add in terms of the time and space complexities of their implementations must be carefully calculated and assessed. In both cases, the state of the program from a method entry point to the call site of a prohibited method must be tracked.

4. RELATED WORK

Regarding aspect-oriented modeling, the approach shown in [15] has elements in common with our work. They propose a schema for composition of models towards aspect-oriented development, defining a process with two levels of abstraction to drive the composition. Their approach is intended to be used as a general-purpose modeling approach, where some specific applications for access control have been developed: [24] shows 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 access-control properties. Their approach takes a generic security design and instantiates a model tied to the domain of the application. Our model in contrast does not instantiate a generic security design, but leaves the decision of the design (i.e. the schema to store users, roles and permissions) to the software developer, providing the tools to define the policy (based in the application's class model), and the mappings to generate enforcement code.

There have been several efforts to add support for runtime pointcuts to aspect-oriented languages. A formalization for dynamic point cuts in a general-purpose aspect language is proposed in [25]. The work in [6, 20] proposes specific AspectJ extensions to support pointcuts based on runtime properties. [7] proposes state-based join points, which are intended to reference points in the program based on runtime properties that can be extracted from the state of the program in any given moment of the execution. Another effort is [16] that proposes a runtime crosscut language that intends to reduce coupling between aspects and the main program. In [17], a weaving approach is proposed that aims to keep overhead of runtime pointcuts as low as possible. Finally, AspectJ version 5 [3] improves runtime join point support; the pointcut designator "if" can now reference runtime properties.

On the UML front, SecureUML [4] defines several levels of abstraction to create an access control notation, where the lowest level is a concrete notation to represent security. The approach of [1, 2] is based mainly on use cases. UMLSec [18] aims to represent and check general security properties not tied specifically to access-control. Another effort [13] (that does not support runtime permissions) inspired the role-slice framework. Our work differs mainly in the emphasis on method-based permissions, the direct relation between role slices and class diagrams, and permission composition across role hierarchies.

5. CONCLUSION

In this paper, we reviewed the main issues that will drive our future research in the area of runtime permission modeling and security enforcement by using aspect-oriented code. We have explored open questions on the incorporation of runtime permission in role slices and the enforcement of runtime permissions via aspect-oriented programming. We have investigated specific topics related to our approach, including the relation between role-slice hierarchies and class hierarchies, and the relaxed schema of enforcement of negative permissions. We are continuing our efforts in the topics as presented in Sections 2 and 3 towards our objective of creating a robust and versatile system to represent and implement a wide-variety of access control policies.

The incorporation of dynamic permissions in the process of defining security in an application, together with the current efforts made at UConn to provide security assurance [8, 9, 10, 11], are a strong starting point towards satisfying numerous access control requirements. To provide empirical verification of these two ongoing security efforts at UConn, we have prototyped an extension to Borland's UML tool Together Architect 1.1 to support security definition for [8, 9, 10, 11] coupled with aspect-oriented security enforcement code generation for [21, 22]. Our intent is to provide a means to concretely demonstrate the design, development, and deployment of real-world secure software applications. Our prototyping efforts are ongoing to include the work presented herein.

6. REFERENCES

[1] K. Alghathbar and D. Wijesekera. authuml: a three-phased framework to analyze access control specifications in use cases. In *FMSE '03: Proc. of the 2003 ACM Wksp. on Formal methods in security engineering*, pages 77–86. ACM Press, 2003.

[2] K. Alghathbar and D. Wijesekera. Consistent and complete access control policies in use cases. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th Intl. Conf., San Francisco, CA, USA, October 2003, Proc.*, volume 2863 of *LNCS*, pages 373–387. Springer, 2003.

[3] AspectJ-Team. The AspectJ programming guide, 2003.

[4] D. Basin, J. Doser, and T. Lodderstedt. *Model driven security, Engineering Theories of Software Intensive Systems.* 2004.

[5] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations model. Technical report, Mitre Corporation, 1975.

[6] C. Bockisch, M. Mezini, and K. Ostermann. Quantifying over dynamic properties of program execution. In *Dynamic Aspects Wksp.*, 2005.

[7] N. Boucké and T. Holvoet. State based join-points: Motivation and requirements (position paper), 2005.

[8] T. Doan, S. Demurjian, R. Ammar, and T. 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.

[9] T. Doan, S. Demurjian, T. 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.

[10] T. Doan, S. Demurjian, T. 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.

[11] T. Doan, L. Michel, S. Demurjian, and T. Ting. Stateful design for secure information systems. In *Proc. of 3rd Intl. Wksp. on Security in Information Systems (WOSIS05)*, 2005.

[12] DoD. *Trusted Computer System Evaluation Criteria. 5200.28-STD.* DoD, December 1985.

[13] P. Epstein and R. Sandhu. Towards a uml based approach to role engineering. In *Proc. of the fourth ACM Wksp. on Role-based access control*, pages 135–143, 1999.

[14] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, 2001.

[15] G. Georg, R. France, and I. Ray. Composing aspect models. In F. Akkawi, O. Aldawud, G. Booch, S. Clarke, J. Gray, B. Harrison, M. Kandé, D. Stein, P. Tarr, and A. Zakaria, editors, *The 4th AOSD Modeling With UML Wksp.*, 2003.

[16] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proc. of the 2nd Intl. Conf. on Aspect-oriented software development*, pages 60–69, 2003.

[17] S. Hanenberg, R. Hirschfeld, and R. Unland. Morphing aspects: incompletely woven aspects and continuous weaving. In *AOSD '04: Proc. of the 3rd Intl. Conf. on Aspect-oriented software development*, pages 46–55, New York, NY, USA, 2004. ACM Press.

[18] J. Jürjens. Umlsec: Extending uml for secure systems development. In *Proc. of the 5th Intl. Conf. on The Unified Modeling Language*, pages 412–425. Springer-Verlag, 2002.

[19] Uml 2.0 object constraint language (ocl) specification, 2003.

[20] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *Proc. of ECOOP 2005*, 2005.

[21] J. A. Pavlich-Mariscal, T. Doan, L. Michel, S. A. Demurjian, and T. C. Ting. Role Slices: A Notation for RBAC Permission Assignment and Enforcement. In *Proc. of 19th Annual IFIP WG 11.3 Working Conf. on Data and Applications Security*, 2005.

[22] J. A. Pavlich-Mariscal, L. Michel, and S. A. Demurjian. A formal enforcement framework for role-based access control using aspect-oriented programming. In L. Briand and C. Williams, editors, *ACM/IEEE 8th Intl. Conf. on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, 2005.

[23] J. H. Saltzer and M. Schroeder. The protection of information in computer systems. *Proc. of the IEEE*, 1975.

[24] E. Song, R. Reddy, R. France, I. Ray, G. Georg, and R. Alexander. Verifiable composition of access control features and applications. In *Proc. of 10th ACM Symposium on Access Control Models and Technologies (SACMAT 2005)*, 2005.

[25] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In G. T. Leavens and R. Cytron, editors, *FOAL 2002 Proc.*, April 2002.