Policy-Based Authorization

William R. Cook
Department of Computer Sciences
University of Texas at Austin

Abstract

This paper discusses *policy-based authorization*, an effective intermediate point between MAC and DAC that promises to combine the best features of both models. Policy-based authorization can be viewed as a reformulation of content-based authorization [3] with simplified content-based policies as the central focus. The primary contributions of this paper are a methodology for designing application-oriented authorization policies, a language for expressing the policies, a new approach to separating the duties of creating policy and assigning policies to users, and a discussion of implementation techniques for fine-grained authorization policies in object-oriented and relational contexts. This model has been implemented in a large enterprise application deployed to thousands of users over the last four years.

1 Introduction

An ideal authorization system would enforce fine-grained security policies that automatically adapt to changing situations – yet require little or no effort to manage the system except to adjust high-level policies when needed. At the same time, the system would present an appropriate and understandable view of the authorization model to a wide range of users, including technical staff, policy managers, personnel managers, and end users.

One might argue that Mandatory Access Control (MAC) comes close to this ideal. Mandatory authorization policies are understandable to users whose job involves managing classified information because the policies are tailored to that domain. Users interact with the model by classifying documents and other users, and the system enforces security policy automatically. However, this approach has not been generalized to apply to other domains or software systems that do not seem to have a single unifying set of requirements.

Discretionary Access Control (DAC) allows users or security managers to implement any security policy they want. But there is a cost: they *must* exercise their discretion to apply the proper security to objects, and maintain them over time. When applied at the granularity of rows, instances or individual fields the overhead quickly becomes unmanageable. The constructs available to specify authorization, including roles [25], object hierarchies, and templates [11], do not necessarily match well with corresponding concepts in the user's domain [17]. Often the authorization policies exist only in the heads of the users, who manually update the authorization system to reflect the outcome of their policy decisions.

This paper discusses *policy-based authorization*, an effective intermediate point between MAC and DAC that promises to combine the best features of both models. Policy-based authorization can be viewed as a reformulation of content-based authorization [3] in which content-based policies are the central focus. The primary contributions of this paper are a methodology for designing application-oriented authorization policies, a language for expressing the policies, a new approach to separating the duties of creating policy and assigning policies to users, and a discussion of implementation techniques for fine-grained authorization policies in object-oriented and relational contexts. This model has been implemented in a large enterprise application deployed to thousands of users over the last five years.

Section 2 provides an example of authorization policy within a typical information system. Section 3 presents a methodology and language for specifying authorization policy. Section 4 discusses implementation of the language in object-oriented and relational contexts. Section 5 reviews related work.

2 Example

A course registration application is used to illustrate authorization policies. The system tracks departments, courses, teachers, students, sections, enrollments, and prerequisites. The entity-relationship (ER) diagram in Figure 1 summarizes the entity types, attributes and relationships in the system. This diagram is expressed using the information engineering notation, but it can be interpreted equally well as a class diagram in UML [13].

The boxes represent *entity types*, or classes, and include the name of the entity type above a list of attributes. The *relationships* between entities are labeled with two names corresponding to the two directions in which a relationship can be traversed. For example, the relationship between departments and teachers is defined so that each department has a *faculty* which is a set of teachers, and each teacher has a *department* that he/she belongs to; in this case "department" is both the name of the relationship and the entity type of the related entity. Note that the users of the system (students and teachers) are also represented explicitly in the information model.

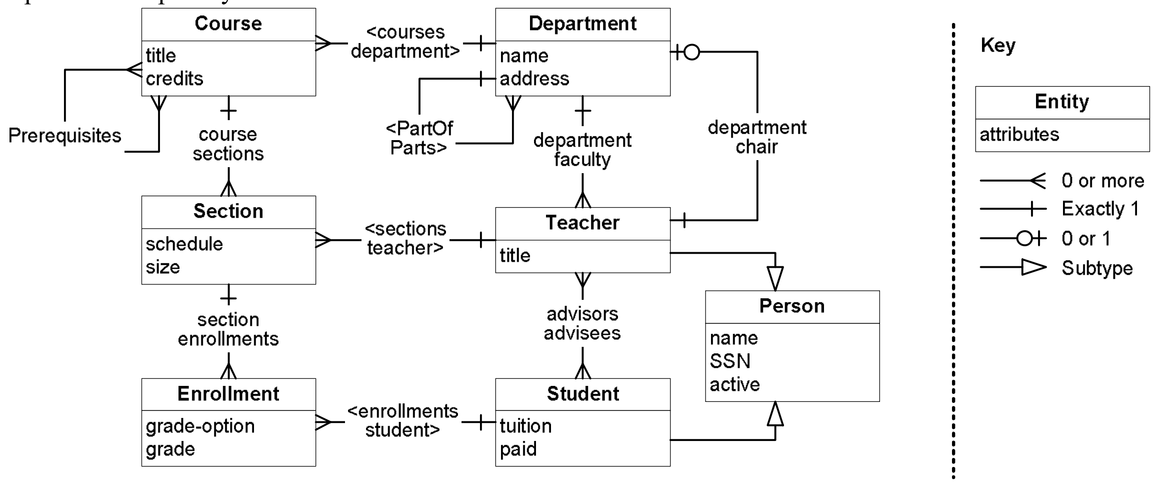


Figure 1: Example entities and relationships in a course registration application

1. Students can view their own enrollments	**allow** read(e : Enrollment) **if** e.student = user
2. Student can enroll themselves in classes during the registration period	**allow** create(e: Enrollment) **if** e.student = user **and** today <= e.section.endRegistration
3. A student can delete their enrollments until the drop deadline	**allow** delete(e: Enrollment) **if** e.student = user **and** today <= e.section.dropDeadline
4. An advisor can view all the enrollments of all their students	**allow** read(e : Enrollment) **if** e.student.advisor = user
5. The teacher of a section can update the grades in the enrollments for the section until the class grading period is closed	**allow** update(e : Enrollment {grade}) **if** e.section.teacher = user **and** today <= e.section.closeDate
6. The same person cannot both enroll in and teach a class	**deny create**(e : Enrollment) **if** e.section.teacher = e.student
7. The number of enrollments must be less or equal to the class size	**deny create**(e : Enrollment) **if size(**e.section.students) > e.section.max
8. To enroll in a course, a student must satisfy all the course's prerequisites	**deny** create(e: Enrollment) **if not forall** prereq **in** e.section.course.prerequisites: **exists** pe **in** user.enrollments: pe.passed **and** pe.section.course = prereq
9. Some teachers can view all the grades for courses in his/her department	**allow** read(e : Enrollment(grade)) **if** user.department = e.section.course.department **and** user.hasPolicy("ViewDeptGrades")
10. Some teachers can update all the grades for courses in his/her department	**allow** update(e : Enrollment(grade)) **if** user.department = e.section.course.department **and** user.hasPolicy("UpdateDeptGrades")
11. Some users can view all grades	**allow** read(e : Enrollment {grade})) **if** user.hasPolicy("ViewAllGrades")
12. Some users can view and update all grades	**allow** update,read(e : Enrollment {grade}) **if** user.hasPolicy("UpdateAllGrades")

Figure 2: Example authorization policies for enrollments in a course registration application

Figure 2 lists some authorization policies within this application relating to enrollments and their grades. The notation in the second column is explained in Section 3 after a discussion of the general properties of these policies. The list of enrollment policies is representative but not necessarily complete. A similar set of policies is needed for each of the other entity types in the information model. Given a complete list of policies for this application, it is convenient to assume that actions not explicitly granted are denied. Negative statements override positive statements.

3 Specifying Authorization Policy

In the example, operations on enrollments are granted to users based on a number of factors, including their relationship to the enrollment, any special rights that they have, and the state of the enrollment. This section introduces a methodology for formalizing and managing the authorization policies in the example.

3.1 Context Roles

A *context role* is a description of an authorization-enabled relationship between a user and an object. A context role is a product of two optional components: a *relationship expression* and a *policy name*. If present, the relationship expression defines how a particular user is related to the object. The policy name identifies what policy must be granted to the user to enable authorizations based on the context role. Policies that are not named are applied to all users. Figure 3 defines some context roles based on the example policies. The relationship expressions are given in the expression grammar of OQL [7]. Context roles are used in the definition of policies; the more familiar use of roles to associate users with sets of privileges is covered in Section 3.2

Context Role	Relationship Expression	Policy Name
Student	user **=** enrollment.student	
Teacher	user = enrollment.section.teacher	
Student's Advisor	user **in** enrollment.student.advisor	
Department Head	user **=** enrollment.seciton.course.department.chair	
Department Auditor	user **in** enrollment.course.department.faculty	ViewDeptGrades
Department Admin	user **in** enrollment.course.department.faculty	UpdateDeptGrades
Global Auditor		ViewAllGrades
Global Admin		UpdatellGrades

Figure 3: Context roles for enrollment

Policies may depend upon the state of an object. States are defined by a Boolean condition again expressed in OQL. States are not required to be mutually exclusive. Figure 4 defines the states in the enrollment example.

Object State	Expression
Registration Period	today <= e.section.endRegistration
Section Full	size**(**e.section.students) > e.section.max
Drop Period	today <= e.section.dropDeadline
Grade Period	today > e.section.lastClass **and** today <= e.section.closeDate
Closed	today > e.section.closeDate

Figure 4: Enrollment states

Given a set of context roles and states, the authorization policies for a class can be defined by an *authorization matrix* as show in Figure 5. Although it is similar to the traditional access matrix [21], it is really a concise description of the conditions from which an access matrix can be computed. The authorization matrix is based upon factoring the informal policy conditions into two parts: the conditions that depend only upon the object (states), and conditions that depend upon the object and the user (context roles).

Context Roles / Object States	Student	Teacher	Student's Advisor	Department Auditor	Department Head	Global Auditor
Registration Period	CRD	R	R	R	CRD	R
Section Full	RD	R	R	R	CRD	R
Drop Period	RD	R	R	R		R
Grade Period	R	RU	R	RU		R
Closed	R	R	R	RU		R

Figure 5: Enrollment security by state and context role

Although the example application does not have a full workflow model, with approval and routing of tasks, it does illustrate some of the basic ways that workflow and authorization interact. The workflow around enrollment and grading is encoded by a set of policies that depend upon the state of the objects in the system.

Consistency rules, which prohibit illegal states or transitions, resemble authorization rules but do not depend upon the user. Uniform treatment of consistency and authorization is useful – if a situation arises in which a consistency may be overridden, the consistency rule can be converted into a very restrictive authorization rule by specifying which context role can override the rule.

3.2 User Roles

Not all policies apply to all users. To be configurable, a system should support definition of appropriate roles that invoke appropriate policies. However, different organizations may organize the roles differently; a small organization may have one administrator with many rights, while a large organization may divide the roles more finely. In addition, the skills involved in creating and deploying policies are very different from the skills involved in managing users and assigning them to roles. Since policies are complex enough to require development and testing, they should not be altered on a running application in active use, while roles are easily managed in a live application. As a result, effective management of authorization should separate these tasks.

Given the fine granularity of policies defined above, it is important to be able to group multiple policies together to define a *user role*. These roles are based on the roles in RBAC [25], but specify policies instead of access permissions. Organizing users into *groups* allow many users to be assigned the same set of roles easily. Users can also be automatically assigned to groups by attaching a condition to the group definition [14]. All users who meet the condition are automatically inserted into the group. Finally, groups are associated with roles. As shown in Figure 6, all relationships between user, group, user role, and named policy are many-to-many. This allows a user to be in many groups, a group to have many roles, and a role to enable many policies.

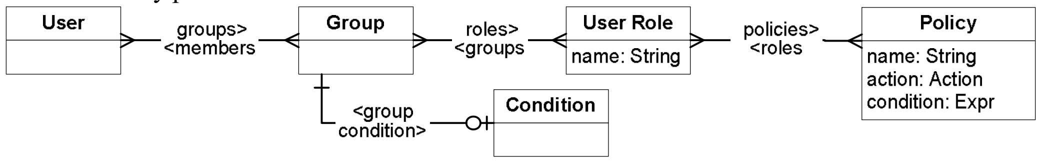


Figure 6: Entities for associating users with named policies

3.3 Policy Language

A simple domain-specific language is used to express authorization policies. Rules contain an action pattern, a condition, and an outcome. The pattern specifies which *actions* the policy controls, while the condition uses all available information from the users, the action and the system state to determine if the policy applies. The outcome is either *allow* or *deny*, where an optional message can be returned for denied actions. Rules are assigned a *strength* to enable conflict resolution when multiple rules apply.

```
policy    ::= strength outcome action [ if expr ]
strength  ::= weak | medium | strong | empty
outcome   ::= allow | deny [ message ]
action    ::= operation ( id : type )
operation ::= create | read | update | delete
type      ::= id | id { id* }
```

The rules for the example application are defined in the second column of Figure 2. Although traditional language constructs like functions, reusable definitions, and modules would be useful in this language, the intent of this presentation is to demonstrate the effectiveness of even a very simple language for authorization policies.

Named policies are conjoined with the rest of the policy condition as a call to a special function hasPolicy, which determines if the current user has the corresponding policy.

The interpretation of *type* influences the granularity of the authorization model. In this case, types can either be the class names of the system, or restrictions to particular sets of fields within a class. These field sets allow a policy to control reading or updating certain fields of an object without affecting the user's access to other fields. The latter case is only meaningful for read and update operations. The types in the rule definitions enable the conditions to be type-checked statically. Specific relational and object-oriented operations are considered in Section 4.

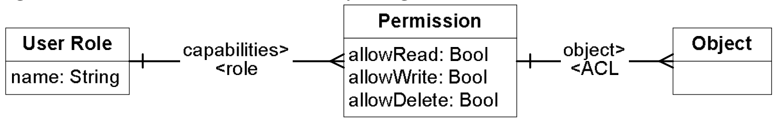
The conditions can refer to relationships defined in the information model. The expression grammar can be borrowed from a variety of sources. Because the underlying data model has a strong foundation in sets, set comprehensions are important. The expression grammar of Object Query Language (OQL) [7], UML's Object Constraint Language (OCL) [13] would be suitable. As above, OQL grammar is used here.

3.4 Variations

Traditional forms of access control can be expressed using explicit content-based policies [18]. A simple form of mandatory authorization can be defined if each user and object has a *security level* attribute:

allow read(obj : Object) **if** obj.level ≤ user.level
allow create(obj : Object) **if** obj.level ≥ user.level

A form of role-based access control lists can be encoded within the policy language. To do so requires adding additional entities to the entity diagram:



This rule defines a simple form of access ACL without sophisticated conflict resolution or role inheritance.

allow read(x : Object) **if exists** ace **in** x.ACL **where** (user **in** ace.role.members) **and** ace.allowRead

In the author's experience building an enterprise application using policy-based authorization, such explicit authorizations are usually tied to other behaviors in the application domain. For example, if there is a requirement for an explicit list of users who can access an object, like a contract, then that list of users is relevant to other aspects of the behavior of contracts within the application. As such, the list is not just an ACL introduced for the purposes of access control, but is instead an integral part of the application which is relevant to authorization.

These traditional forms of access control can be blended with each other and with other content-based policies.

3.5 Semantics

Authorization determines whether to allow or deny a subject's request to perform an action. An authorization configuration is a value of type *Authorization*:

$Authorization = Subject \times Action \times State \to Bool$

The form of an action will depend upon the domain of application. For database authorization, actions are create, read, update, or delete on a set of rows in a table. For object-oriented authorization, actions are method calls and object instantiation. For web service authorization, the actions are XML requests. The semantics of the policy language maps a set of policies P in language *Policies* to an authorization configuration $Auth : Policies \to Authorization$

$$P : Policies \subseteq Level \times Outcome \times Method \times id \times Type \times expr$$

$$Auth[\![P]\!] = Auth_{MaxLevel}[\![P]\!]$$

$$Auth_l[\![P]\!]x = \neg Auth_l^{\mathsf{deny}}[\![P]\!]x \wedge (Auth_l^{\mathsf{allow}}[\![P]\!]x \vee Auth_{l-1}[\![P]\!]x)$$

$$Auth_l^d[\![P]\!]\langle s, o.m, \sigma\rangle = \bigvee_{\substack{\langle l,d,m,v,t,e\rangle \in P \\ typeof(o)=t}} Eval[\![e]\!]\sigma[v \mapsto o, \text{USER} \mapsto s]$$

The expression language is assumed to supply a semantic function:

$Eval[\![e]\!]\sigma : Expr \to State \to Bool$

A potentially serious problem with this approach is that subjects may be able to infer information about data that is hidden from them. If a policy condition depends upon hidden fields, but has some effect upon visible information, then inference is possible. A trivial example of an ineffective policy would be an attempt to hide failing grades. If a user saw a list of grades where some were missing, they could infer that the missing grades were failing grades. This is one reason why creating policies is something that must be done carefully and not exposed to ordinary users.

In addition to checking user actions, the policy rules can also be evaluated for speculative purposes. For example, user interfaces should reflect the authorizations of the current user. This involves determining whether a subject could perform a given action. When the operation the user may perform is incompletely specified, the policy conditions must be evaluated to see if the operation is known to be denied given only partial information. When a workflow system assigns tasks to users, it must determine the set of users who can perform the task based on the policies for those users. While the semantics of these operations are well-defined, specialized evaluation strategies would be needed for these operations to be executed efficiently.

3.6 Delegation, Dependency, and Reflexivity

Delegation covers a wide range of situations in which one user (grantee) is given authority to act on behalf of another (grantor): delegation can apply to a task, a role, or complete responsibility. Delegation in policy-based authorization is complicated by policies that depend upon relationships. Simply adding user roles to the grantee is not sufficient for delegation, because the grantee will not always be in the same context roles as the grantor. Instead, the system must evaluate the policy conditions for the grantee using the identity of the grantor. To do so, delegation involves recording the grantee, grantor, and the set of roles and/or objects being granted. When testing authorization, the system must evaluate the policy conditions for each of the delegations that apply to the current user.

Allowing authorization of one object to depend upon authorization on another is also useful, especially when one object is logically a component of another. For example, the authorization to create and update a section could be derived from the authorization on the course to which the section belongs. This technique is called "implicit authorization" [4] in the context of RBAC. To support dependencies between object authorizations, the authorizations on an object must be exposed via a function in the expression grammar; these are the same speculative authorization needed for user interfaces. The dependence of sections on courses can then be expressed as

allow update(s : Section) **if** can-update(s.course)

Reflexive application of the authorization model provides security on the authorization system. Authorization policies are defined on all the entities related to authorization, including user roles, groups, and policies. Each of these may objects be managed by different roles. In addition, the set of roles, groups, and policies may be partitioned at an instance level using content-based policies. Evaluating a reflexive condition requires extending *Eval* to support calls to *Auth*. For the resulting system to be well-defined there must be no cycles in the chain of dependencies from one authorization to another. To prevent cycles, we requires that the relationships used in reflexive calls form an acyclic graph.

4 Implementation

Authorization is implemented by a *reference monitor* that sits between a client and an application, where it acts to either allow or deny requests from the client to the application. We assume that a secure mechanism for authentication results in a validated user identity. Each authenticated user is given a session object, in which information about the user can be cached. Note that the reference monitor also has trusted access to user and application properties. Finally, security is not enforced during evaluation of policy conditions.

4.1 Object-Oriented Implementation

An authorization system for an object-oriented interface must intercept every message sent by the client to an object. This can be achieved by wrapping every object with an individual reference monitor. In this model, policies are defined for method calls instead of the generic create, read, update, and delete operations. The policies, written in OQL, are automatically converted to appropriate code in the target architecture, for example, Java or C# methods. The wrapper instances must be initialized with an object representing the current user. Additional method calls can be added to the wrappers for clients to test authorization rights without actually performing an operation.

Policies restrict access to individual methods or entire objects. If an entire object is restricted, it must not be returned from any method, included in any array, or returned as a value from any iterator. As a result, policies can have a significant impact on the behavior of an application.

One of the problems with this approach is that some operations can be inefficient and difficult to optimize, especially when the objects are managed by a persistent object system. For example, if a client requests all objects from a very large set but is only authorized to see a fraction of the complete list, every object must be loaded and explicitly tested for access. In addition, testing for authorization may require additional objects to be loaded beyond the ones requested by the client. For bulk updates it would be advantageous to test if the operation is authorized on all objects before beginning to perform the actual operations. Optimizing these cases may require breaking encapsulation boundaries to combine the policy conditions with lower-level data access code.

4.2 Relational Implementation

Database languages like SQL provide a modular interface where authorization can be inserted. Row-level security in databases is traditionally implemented by granting access to specialized *views* [15]. However, this approach cannot express the policy language, because the policy conditions depend not just upon the data being viewed, but also upon the attributes of the user. In addition, multiple conditions must be combined based on the set of policies that apply to a user. Our approach is to create dynamic views for each user based on their attributes and policies. In what follows we discuss a straightforward implementation based on query modification a version of which has been deployed in production in a large enterprise application. We have not yet explored other important options, like implementing policy-based authorization inside the database engine.

4.2.1 Condition Translation and Caching

The conditions in authorization policies are executed by translating them into SQL and including then in client queries.

In translating conditions, all references to user rights and user properties must be substituted with the particular values for the user. These values can be cached when the user session is started. Changes to user properties must be monitored so that that cache can be flushed if the effect of using cached security information is considered significant. One other advantage of caching user properties is that they may be used for partial evaluation of the authorization conditions. For example, if a policy does not apply to a user then the corresponding condition does not need to be added to the query. For example, if the user does not have the policy ViewDeptGrades, then the test for being in the same department is not needed.

The table-based translation described below can result in redundant sub-queries which may or may not be optimized by the SQL optimizer. The actual system based on this model includes a more sophisticated translation engine that eliminates these redundancies.

4.2.2 Read

Since SELECT queries in SQL allow reading information from the database, read policies must be applied to ensure that only authorized information is returned to the user. Read policies can prevent access to entire rows or individual fields of a row. The authorized fields in one row may be different from authorized fields in another row. Queries must be modified so that hidden fields cannot be used for sorting, grouping and testing rows, or in sub-selects.

Clients only see data that they are authorized to see, but are not provided any explicit information about data that may be hidden from them. Thus every user sees a view of the database as if it only contained exactly the data that they are authorized to see. Alternatives, like rejecting complete operations that involve unauthorized data or providing an explicit error value for unauthorized data would give clients more information. Note that clients cannot depend upon any required fields in the database, because any information could be hidden.

To implement authorization, each table reference in a query is wrapped by a policy-enforcing view based on the policies and attributes of the current user. If a table T appears anywhere in the query, it is replaced by

SELECT *output*$_{field}$, … **FROM** T [**JOIN** additional-tables] **WHERE** *row-condition*

The additional tables result from the translation of policy conditions that refer to attributes of related objects. The outputs are defined as follows:

output$_{field}$ = **CASE WHEN** *field-condition*$_{ifeld}$ **THEN** *T.field* **ELSE** NULL **END AS** *field*

The row-condition is derived from the policies that apply to the current user's access to this table, based on the semantics in Section 3. Only those fields which are actually used in the body of the original query need to be included in the output list. One advantage to this approach is that the authorization policies and underlying data access are combined in a single query, where they are optimized together.

Note that this approach will work for aggregate queries, but does not provide a fully satisfactory solution: aggregates will only include values for which the user has rights to access the detailed data.

4.2.3 Update and Delete

Update and delete operations may operate on multiple objects but are allowed or denied atomically. An update operation updates a set of fields on a set of objects defined by an update condition. If any of the rows that meet the update condition violate the field-level policies, then the entire update operation is rejected. A single policy query can aggregate the value of the field-level and row-level polices for all rows meeting the update condition. To return a useful error, it is also possible to determine exactly which policies were violated. The same approach works for delete operations. It is unfortunate that an additional query is required for every update or delete operation. If they policies were implemented directly in the database engine, then more efficient evaluation strategies are possible.

4.2.4 Insert

There are two forms of insert statement: inserting rows returned from a select statement, and inserting a new row based on a set of literal values. The first form can be handled by the same approach used for update and delete. The select query in the insert statement is taken as a view specifying new objects for the target table, and a policy check query is executed against this view to evaluate and aggregate the create policy conditions for these new rows.

The second form poses a problem because we must insert the values into the database before SQL can evaluate the policy conditions. However, inserting the row into the database before policies have been checked violates the basic purpose of the authorization system to prevent unauthorized actions. To solve this problem, the insert is performed within a transaction; the insert is committed to the database only if the creation policies are satisfied. This depends upon transactions for isolation and reading uncommitted data.

5 Related Work

A recent survey [8] of approaches to policy specification includes a section on authorization policies. Although they point out that “in many systems there is no real policy specification”, the following systems all support definition of authorization policies that express the essential dependence on content and user attributes. The systems are reviewed in chronological order.

Moffet & Sloman [22] considered policies of the form “All users may perform Pay on those Transaction objects where the value of the user's Authorisation_Level attribute is greater than the value of the Amount attribute in the Transaction object.” They conclude that while content-based access control is clearly desirable, it was not practical because access to attributes may not be trusted and the implementation would be inefficient.

Template-based authorization [11] creates parameterized roles, but these must be instantiated to express content dependencies. Gross [16] proposed modification policies as conditions over functional abstractions of relational data. The functions complicate execution of the rules, but accommodate radical changes to the underlying data model.

Bertino and Weigand [3] define content-dependent policies for object-oriented databases, as an adjunct to explicit access control. They only sketch a user-level version of the language, and do not separate the management of roles and policies. They define extensive mechanisms for implicit authorization, where authorization of one object or user may depend upon others. These techniques could be investigated further in the context of pure policy-based authorization.

Pandey and Hashii [23] define a policy language for java programs. Authorization can depend on the attributes of the object, but there is no explicit notion of user or role. Since policies are enforced by byte-code editing, different users cannot have different sets of policies.

Barkley et. al. [2] allow relationships tests to be plugged in to an RBAC model. Didriksen [10] allows “fragmentation” of database tables based on a form of SQL condition that may refer to table content or user attributes. No formal semantics or implementation details are given.

Steen and Derrick [26] formalize ODP enterprise policies using a policy language based on OCL conditions; the semantics is defined by an informal translation to Object-Z. Good example policies are discussed but no implementation is given. They also consider obligation policies, which are outside this scope of this paper.

LasCO [18] is a graph-based language for defining security policies over operation invocation. The mapping of policies to roles is embedded within the policy definition, making management more difficult. It is also not clear that the graphical notation helps readability, since most of the graphs are simply one or two arcs.

Ponder [9] is a policy specification language targeted at distributed systems and network security. The language assumes a hierarchical organization of subjects and objects, and allows constraints defined in a subset of OCL. Ponder also supports reuse of parameterized policy definitions.

Many languages use role activation conditions to allow roles to depend upon object content [1]. Our approach evaluates the conditions at each application of policy without requiring role activation.

SPL [24] is an expressive policy language very similar to the model presented here. The model provides notation for reusing and combining policies. The biggest problem with the language is scalability, although this may be due in part to the use an ACL-based implementation.

Goodwin, Goh and Wu [14] extend template-based RBAC to allow object groups defined by implicit conditions and support a relationship test between the subject and the object. The use of relationships is limited; for example, rather than use a relationship test for membership in an appropriate organization they use template instantiation. They only allow a single direct relationship rather than a generalized relationship-based condition. Zhao [28] support a specific set of ownership relationships for supply-chain applications.

The eXtensible Access Control Markup Language (XACML) is a recent standard for access control that includes an XML-based notation for defining authorization conditions. XACML uses an XML grammar to describe policy conditions, which can depend upon object or user attributes; the management of policies and roles is not specified.

The relationship between the expressive power of these different approaches is a significant open problem.

Commercial software products have experimented with a number of novel mechanisms, most of which have never been studied formally. Oracle has a row-level security model called Virtual Private Databases in which a security procedure appends a condition to a view prior to execution. The model is not declarative and does not provide the full generality of SQL joins. Enterprise applications, like PeopleSoft, Siebel, and SAP have also defined proprietary approaches to authorization which may express a form of content-dependent policies.

6 Conclusion

Policy-based authorization is a reformulation of content-based authorization with a focus on explicit policies represented as rules. Policies are defined in terms of the properties of the subject, the properties of the object(s) and the relationship between the subject and the object. This authorization model combined elements of discretionary and mandatory authorization. The policies are mandatory in the sense that they are defined centrally, but are discretionary in the application of policies to users and control over the relationships on which authorization is based.

By abstracting away from the semantics of the underlying application, in the definition of the abstract sets of subjects, objects, and rights, traditional access control models lose the ability to effectively model the intention of an authorization policy. On the other hand, policy-based authentication does not work unless the underlying application has suitable policies to implement: a generic file serve does not have enough behavioral constraints to support useful policies.

References

1. J. Bacon, K. Moody, and W. Yao. A model of OASIS role-based access control and its support for active security. *ACM Transactions on Information and System Security* (TISSEC), 5(4):492-540, November 2002.
2. J. Barkley, K. Beznosov and J. Uppal. Supporting Relationships in Access Control Using Role Based Access Control. In Proc. of the Fourth ACM Workshop on Role-Based Access Control, October 1999, 55-65.
3. E. Bertino and H. Weigand. An approach to authorization modeling in object-oriented database systems. *Data & Knowledge Engineering*, 12(1), February 1994, 1-29.
4. E. Bertino, W. Kim, F. Rabitti, and D. Woelk. A model of authorization for next-generation database systems. *ACM Transactions on Database Systems* (TODS), 16(1), 1991.
5. R.A. Botha. *CoSAWoE – A Model for Context-sensitive Access Control in Workflow Environments*. PhD Thesis, Rand Afrikaans University, November 2001.
6. R.K. Burns. Report on the Homework Problem. *Research Directions in Database Security*, Springer-Verlag, 1992, 109-123.
7. R.G.G. Cattell and D.K. Barry. *The Object Database Standard (ODMG) 3.0*. Morgan Kaufmann, 2000.
8. N. Damianou, A.K. Bandara, M. Sloman, E.C. Lupu. A Survey of Policy Specification Approaches. Unpublished manuscript, 2002.
9. N. Damianou, N. Dulay, E. Lupu, M. Sloman. The Ponder Specification Language. *Workshop on Policies for Distributed Systems and Networks* (Policy2001), January 2001, 29-31.
10. T. Didriksen. Rule based database access control - a practical approach. *ACM Workshop on Role-Based Access Control*, 1997, 143-151.
11. L. Giuri, P. Iglio. Role templates for content-based access control. *ACM Workshop on Role-Based Access Control* 1997, 153-159.
12. S. Godik, T. Moses (eds.). *eXtensible Access Control Markup Language* (XACML) OASIS Standard 1.0, 2003.
13. Object Management Group. *Unified Modeling Language* (UML) version 1.5, 2003.
14. R. Goodwin, S. Goh, F. Wu. Instance-level access control for business-to-business electronic commerce. *IBM Systems Journal* 41(2), 2002, 303-321.

15. P. Griffiths, B.W. Wade. An authorization mechanism for a relational database system. *ACM Transactions on Database Systems* (TODS), 1(3), 1976, 242–255.
16. T. Gross. Creating Abstract Discretionary Modification Policies with Reconfigurable Data Objects. In *Proc. of the IFIP Working Conference on Database Security* (DBSec) 1994, 335-351.
17. B. Hilchenbach. Observations on the real-world implementation of role-based access control. In *Proc. of 20th National Information Systems Security Conf.*, 1997, 341-352.
18. J. Hoagland, *Specifying and implementing security policies using LaSCO, the language for security constraints on objects*, Ph.D. dissertation, 2000.
19. S. Jajodia, P. Samarati, M. L. Sapino, and V. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2).214--260, June 2001.
20. S. Jajodia, P. Samarati, V.S. Subramanian, E. Bertino, A Unified Framework for Enforcing Multiple Access Control Policies. In *Proc. of the 1997 ACM International SIGMOD Conf. on Management of Data*, 1997.
21. B. Lampson. Protection. Proc. 5th *Princeton Conf. on Information Sciences and Systems*, 1971. Reprinted in *ACM Operating Systems* Rev. **8**, 1 (Jan. 1974), pp 18-24.
22. J.D. Moffett, M.S. Sloman. Content-dependent access control. *Operating Systems Review*, 25(2), April 1991, 63-70.
23. R. Pandey and B. Hashii. Providing Fine-Grained Access Control for Java Programs. In Proc. *European Conf. on Object-Oriented Programming* (ECOOP), LCNS 1628, June 1999, 449-473.
24. C. Ribeiro A. Zuquete, P. Ferreira and P. Guedes. SPL: An Access Control Language for Security Policies and Complex Constraints. In *Proc. Network and Distributed System Security Symposium*, 2001.
25. R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, 29(2), February 1996, 38-47.
26. M.W.A. Steen and J. Derrick, , Formalising ODP Enterprise Policies. In *Proc. Third Int. Enterprise Distributed Object Computing Conf.* (EDOC), 1999, 84-93.
27. J.A. Talvitie. An Object-Oriented Application Security Framework. In *Proc. of the OOPSLA Workshop on Security for Object-Oriented Systems*, 1993, 55-75.
28. J.L. Zhao, H.J. Wang, S.S. Huang, and G. Chen, Relationship Driven Access Control in a Supply Web. In *Proc. of the 12th Workshop on Information Technology and Systems,* 2002.