Access Control Administration with Adjustable Decentralization

by

Amir H. Chinaei

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2007

Abstract

Access control is a key function of enterprises that preserve and propagate massive data. Access control *enforcement* and *administration* are two major components of the system. On one hand, enterprises are responsible for data security; thus, consistent and reliable access control enforcement is necessary although the data may be distributed. On the other hand, data often belongs to several organizational units with various access control policies and many users; therefore, decentralized administration is needed to accommodate diverse access control needs and to avoid the central bottleneck. Yet, the required degree of decentralization varies within different organizations: some organizations may require a powerful administrator in the system; whereas, some others may prefer a *self-governing* setting in which no central administrator exists, but users fully manage their own data. Hence, a single system with adjustable decentralization will be useful for supporting various (de)centralized models within the spectrum of access control administration.

Giving individual users the ability to delegate or grant privileges is a means of decentralizing access control administration. Revocation of arbitrary privileges is a means of retaining control over data. To provide flexible administration, the ability to delegate a specific privilege and the ability to revoke it should be held independently of each other and independently of the privilege itself. Moreover, supporting arbitrary user and data hierarchies, fine-grained access control, and protection of both data (end objects) and metadata (access control data) with a single uniform model will provide the most widely deployable access control system.

Conflict resolution is a major aspect of access control administration in systems. Resolving access conflicts when deriving effective privileges from explicit ones is a challenging problem in the presence of both positive and negative privileges, sophisticated data hierarchies, and diversity of conflict resolution strategies.

This thesis presents a uniform access control administration model with adjustable decentralization, to protect both data and metadata. There are several contributions in this work. First, we present a novel mechanism to constrain access control administration for each object type at object creation time, as a means of adjusting the degree of decentralization for the object when the system is configured. Second, by controlling the access control metadata with the same mechanism that controls the users' data, privileges can be granted and revoked to the extent that these actions conform to the corporation's access control policy. Thus, this model supports a whole spectrum of access control administration, in which each model is characterized as a network of access control states, similar to a finite state automaton. The model depends on a hierarchy of access banks of authorizations which is supported by a formal semantics. Within this framework, we also introduce the *self-governance* property in the context of access control, and show how the model facilitates it. In particular, using this model, we introduce a conflict-free and decentralized access control administration model in which all users are able to retain complete control over their own data while they are also able to delegate any subset of their privileges to other users or user groups. We also introduce two measures to compare any two access control models in terms of the degrees of *decentralization* and *interpretation*. Finally, as the conflict resolution component of access control models, we incorporate a unified algorithm to resolve access conflicts by simultaneously supporting several combined strategies.

Acknowledgements

I would like to warmly thank Frank Wm. Tompa for his outstanding supervision of this course of study: for his valuable guidance, for his everlasting patience, for his continuous support with all means, and for his deep friendship in difficulties. Thank you, Frank, from the bottom of my heart for being far more than just a supervisor!

I would also like to thank Ji-Won Byun, Khuzaima Daudjee, Urs Hengartner, Ninghui Li, Mohammadreza Meybodi, Sylvia Osborn, Mohammadreza Razzazi, Arnie Rosenthal, Hossein Saiedian, Ken Salem, David Toman, Huaxin Zhang, and Grant Weddell, with whom discussions and feedback enhanced my research ideas and skills.

Furthermore, I gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada, Communications and Information Technology Ontario, Open Text Corporation, the Mathematics of Information Technology and Complex Systems, and the University of Waterloo for their financial support.

Last but not least, I would like to thank all friends (let's call them all, "friends") who played important roles in other aspects of my studies, in particular, my parents, my siblings, my wife, as well as Jeremy Barbay, Mohammdreza Khalil-Moghaddam, Zarrin Langari, Hamid Mahvari, Kiarash Narimani, Jesse McCrosky, and Mehran Rad.

To my mother and in memory of my father

Table of Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

The fast development of web applications and information sharing, together with the complex base of sensitive data in many practical systems, poses new challenges for access control administration. Decentralization of access control administration is the problem of our interest in this thesis. The problem is particularly difficult because administration models vary in their degree of decentralization.

In practice, there are certain applications, such as in the military, that require a central access control administration. Yet, there are various environments, such as file-sharing systems, in which a sole centralized administration is not practical. No single existing model supports this variety simultaneously. Most current models provide a centralized administration only. A few recent models support decentralized administration to a limited extent. However, from a system developer perspective, an access control system usually needs to be installed in various organizations with diverse levels of decentralization. Moreover, in a single application, some objects need to be administered centrally whereas other objects demand a decentralized control. Appropriate adjustment of the degree of decentralization has a strong influence on *efficiency* and *self-governance*.

Efficiency. There are several factors that affect efficiency of access control administration: the volume of *data* (i.e., end objects) and *metadata* (e.g., access control data), and consequently the number of administrative requests (at each moment) are so large in such systems that the central administration overhead becomes a bottleneck and efficiency becomes problematic. Centralized administration also imposes longer routings on requests being launched from distributed clients. This is more critical in urgent situations: sensitive objects should be secure all the time yet quickly possible to be accessible to authorized subjects; for example, authorized personal health records should simply be accessible to emergency centers when needed. Yet, administering the access control must be reasonably fast even in non-emergency cases. Furthermore, in applications where data are naturally distributed, centralized administration is often incompatible. For instance, a specialist may need to access several health objects of a visiting patient, possibly filed in different health centers. (Similarly, decentralized administration may become difficult in applications in which accountability is centralized.) On the other hand, access control must be provided for the access control specification (called *metadata* in this thesis) itself; e.g., patients may query who can access their health records. Often systems restrict updating this metadata to a small group of access control administrators (called *security officers* in some systems), and for systems relying on so-called "discretionary access control", it is important to diversify the population who may update metadata. It is also important that access control models provide a single uniform mechanism for administering both data and metadata. Example I in Section 1.3 illustrates the efficiency problem of very large access control systems, in which typically hundreds of thousands of objects exist.

Self-governance. *Self-governance* is an important requirement of many information sharing systems: users in such situations wish to share their data without appealing to administrators; in many environments they should be allowed all operations on particular data (e.g. on their own data or on data for which they are responsible), including delegating privileges to others and revoking them at will; for instance, in healthcare systems, patients often and temporarily have to reveal their personal data to particular appropriate parties, such as physicians, hospitals, and laboratories. Therefore, applications in which several parties need to share data will be simpler to manage if each party fully controls its own (or responsible) data, subject to conformance with the underlying administrative *policy* formulated by the enterprise. (Note: ownership has been interpreted differently in the literature; in this work, "data owner" is a subject who is responsible for the data.) Moreover, objects are usually managed in different contexts that need to be administered differently. Even if different parties use the same underlying technology, they may require different sections of a complex access control policy to be enforced, which would be an error-prone burden for a centralized administration. Example II in Section 1.3 illustrates the self-governance property of access control models.

In conclusion, sharing data, including selectively delegating and revoking administrative privileges on the associated metadata, may be supported more reliably and more efficiently by a flexible decentralized access control administration. The challenge is to ensure that the underlying administrative policies are enforced.

1.1 Access Control Components

Before proceeding further, it is important to distinguish major components of access control systems, requests, policy, enforcement, and administration, depicted in Figure 1.

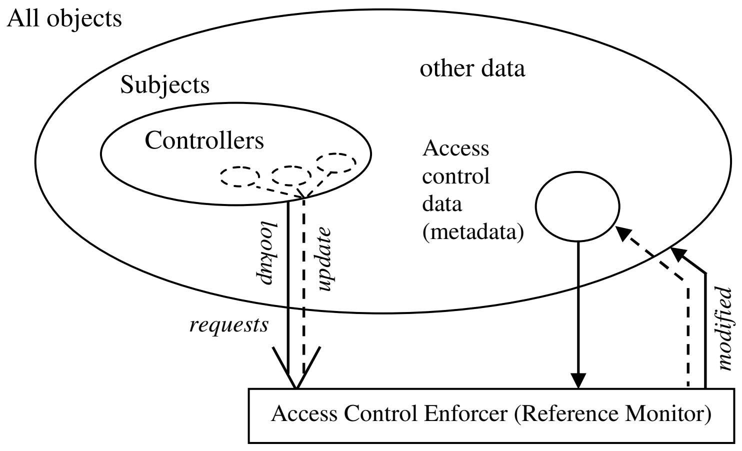


Figure 1. Access control framework.

- The set of objects under access control includes the set of subjects and the access control data itself (the metadata).
- There are two types of access control requests: lookup and update. A *lookup request* is an attempt to retrieve some information from the data or metadata such as whether a given subject (user, application, or group) has privilege to modify a given object (data, application, metadata, etc.). An *update request* (depicted by dashed lines) is an attempt to change metadata. (Note that we are distinguishing here between lookup and update of the metadata itself, not of the end objects for which access is being controlled.) The direction of arrows indicates that when a given subject requests a lookup or an update, the enforcer decides if the

request is valid based on the current state of the metadata, and enforces the request appropriately.

- *Access control policy* is a set of rules that all users in the system must follow; it defines the space of valid metadata states.
- *Access control enforcement* is the mechanism of securing the system against invalid requests, that is, those that are not consistent with access control policy. Because data is one of the critical resources owned by an enterprise, data management systems must enforce access control policies developed by the enterprise administrators. Thus, each operation on each piece of data must be vetted by the system to determine whether the subject attempting the operation has appropriate privileges with respect to that item of data.
- *Access control administration* is the mechanism of handling requests for access control updates. Any subject that can update the metadata is called a controller in our model.

Note that this thesis focuses on decentralizing access control administration. Access control has to be consistently enforced (although perhaps distributed access machines and organizational units), but the administration of access control can often be handled without referring to central administrators. Figure 1 depicts the centralization of access control enforcement and the decentralization of access control administration.

1.2 General Terminology

This section reviews some basic access control terminology that is used in this thesis. In particular, one should distinguish between subject vs. object, delegate vs. grant, decentralized vs. centralized administration, explicit vs. effective privileges, and schema-level vs. instance-level control.

Subject vs. object: Throughout this work, there are two general terms, *subject* and *object*, generalizing the notions of users and data, respectively. Objects, with respect to access control systems, are what are operated on and thus for which particular access is sought. Examples of objects are data, resources, and applications. Subjects, which have sometimes been defined as active objects, are those that (implicitly) request access in order to execute an operation. Examples of subjects are users, applications, and groups. It is important to recognize that the stored information representing a subject is, in fact, data, and we treat a subject as an object for some operations. For formal definitions of subject and object, see Section 3.1.

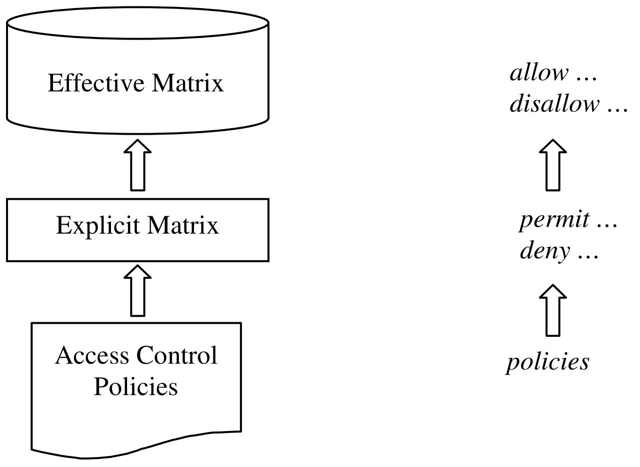
Delegate vs. grant: Some researchers distinguish between two related terms: *grant* and *delegate*. The former often refers to giving a privilege to a subject *permanently*, while the latter usually means giving a privilege to a subject *temporarily*. Similarly, other systems support “revocable” grants by providing a separate revoke operation. It is obvious that a temporary delegation is equally expressive to a permanent grant together with revoke. Hence, throughout this thesis, both terms are used interchangeably to assign a privilege, and we include an explicit revoke operation to remove it.

Decentralized vs. centralized administration: The question is to what extent decentralization can be realized? It is important to recognize the *spectrum* of access control administration. At one end, access control can be absolutely *autocratic*: a powerful administrator exists in the system dictating which subjects have access to which objects; at the other end, it can be completely *self-governing*, which means that no central administrator exists in the system, but users fully manage their own data. Chapter 7 specifies such a model.

Whereas security-conscious enterprises often use central enforcement mechanisms to support their access control policies, closely held access control *administration* may or may not fit their security requirements. In conclusion, access control venders desire to sell their product to various organizations requiring a diverse amount of decentralization. There are applications in which different object types require to be treated differently, centralized or decentralized. Such issues are addressed within Chapters 4 to 6.

Explicit vs. effective privileges: Not every subject-object pair has explicit privileges assigned. Instead, access to objects with no explicit privileges can often be deduced from the set of other explicit privileges. For instance, in a hierarchical structure, access privileges are typically derived from the parent-child relationship: members of a group inherit all the privileges of the group and sub-objects inherit the privileges of the objects in which they are contained. If no privilege or several conflicting privileges can be derived for a given subject-object pair, the access control system should resolve a final privilege for that pair. The ultimate explicitly or implicitly assigned privilege will be called the *effective* privilege.

Access control data (called metadata) −which corresponds to relations of subjects, objects, and privileges− can be conceptually viewed as being represented by an access control matrix, where the rows represent subjects, the columns represent objects, and privileges are stored at the intersections [Lampson 1971]. The term *effective matrix*, is used to represent effective privileges as a three-dimensional Boolean matrix EM, indexed by subject, object, and method. In such a matrix, no cell is null. The value of EM[*s*,*m,o*] is 1 if the corresponding subject *s* is privileged to execute method *m* on object *o*; otherwise EM[*s*,*m,o*] is 0. Correspondingly, an *explicit matrix* implements the idea of condensing the effective matrix by storing explicit privileges only. The explicit matrix can be expanded to the effective one by using propagation and conflict resolution strategies. Figure 2(a) depicts the transitions of access control policies to the explicit matrix and then to the effective matrix. Figure 2(b) illustrates, as an example, that the explicit matrix is represented by predicates *permit* and *deny*, and the effective matrix is represented by predicates *allow* and *disallow* in our **ac**cess control **ad**ministration model (called ACAD), which will be examined in further detail in Chapter 3.



(a) Abstract model. **(b) ACAD predicates.**

Figure 2. Constructing an effective access control matrix from policies.

Schema-level vs. instance-level control: Although access control is inherently a concern at the instance level (i.e., individual objects are subject to control), *policies* are more generally defined at the schema level. This approach simplifies administration, since it provides a convenient means to specify consistent rules on large collections of homogeneous objects at once.

For the sake of generality, this thesis assumes an object-oriented data model, in which the type of an object dictates the set of applicable methods that can be applied to it. It is also assumed that privilege to execute each method is controlled independently: thus the ability to execute the read method requires read privilege, the ability to execute the write method requires write privilege, the ability to execute the append method requires append privilege, the ability to execute the delete method requires delete privilege, the ability to execute the "check into the hospital" method requires "check into the hospital" privilege, and so forth.

1.3 Motivating Examples

This section provides two motivating examples. One is a very large healthcare system that is used in order to highlight the *efficiency* importance of access control models. The other is an ad hoc scenario of document sharing environments that will help to highlight the *self-governance* property of such models.

1.3.1 Healthcare Systems

Example I. Assume a worldwide healthcare system in which there are hundreds of thousands of end objects (such as medical records, personal information, account balances, etc.) and users (such as patients, doctors, technicians, hospitals, etc.). There may be a huge number of access control requests at any moment. Some instances are:

– Patients may wish to authorize other users, e.g. doctors and insurance agents, to read their personal information.

– Some users, e.g., pharmacists, technicians, and patients as well as their family members, may request to see some or all parts of a medical record.

– Some users, e.g. doctors and insurance agents, may request authorization for reading a particular patient's medical history.

– Patients may no longer wish their medical record to be seen by a particular family member.

In such systems, a centralized administration may easily be overwhelmed with too many requests; on the other hand, information accessibility is often vital to such users so the requests should not be backlogged. As an improvement, a significant subset of requests can be handled by different users without referring to administrators. In Chapter 7, we explain the User Managed Access Control (UMAC) specification in which the bottleneck of inefficiency is resolved by applying *decentralized administration*. Once the appropriate corporate policy is set up at configuration time, users can manage and share their data properly at run time.

1.3.2 Document Servers

Example II. Assume a web-based document server within which several users and groups (possibly competitors) share their documents. Each group may have some private documents that are accessible to its organization only, as well as other documents that may be shared with a subset of other users. Once the user's corporate policy is in place, users are allowed to share their data with others so far as they wish and so long as they stay within the policy's guidelines. Moreover, each user may stop sharing information at any point. There is no administrator in the system, and document accessibility is fully managed by users. As a simple instance of such a server, assume user A owns document P. Consider the following requests:

– A authorizes B to delegate read privilege on P to other users.

– B authorizes many others, including user C, to read P.

– A decides to stop C, but not others, from reading P.

Neither A nor B wishes to consult an administrator as long as their requests are within the corporate policy; instead, they wish to govern their own documents independently. Chapter 7 describes how users can manage their own data independently, in the absence of administrators of any kind, and without interfering with one another's decisions with respect to their own data.

Examples I and II, explained above, emphasize the importance of efficiency and self-governance of access control administration. On one hand, a decentralized access control administration is required when several parties, usually without administrative control over one another, need to share their data. Such a model reduces the bottleneck of access administration in situations where data is distributed among various parties. On the other hand, the question is to what extent decentralization can be realized? The main contribution of this thesis is to provide an access control administration model in which the degree of decentralization for the whole system, as well as for each object type, can be adjusted at the configuration time.

1.4 The Thesis Scope

This thesis addresses the problem of access control administration. Assumptions are that the user's authentication is successfully verified, the corporate policy is appropriate, and access control is correctly enforced by a reference monitor; these aspects are not addressed in the thesis.

There are several contributions in this work. First, an **ac**cess control **ad**ministration model (called ACAD), in which the degree of decentralization is adjustable from a centralized level to a very decentralized extent, is proposed. The model includes a novel configuration mechanism to constrain access control administration for each object type at *object creation time*, as the means of adjusting the degree of decentralization when the system is being installed. ACAD introduces the spectrum of access control administration as well as representing the administration as a network of access control states, similar to a finite state automaton. Each state is a directed acyclic graph in which *access banks of authorizations* and *authorization inheritance relations* map into the nodes and edges, respectively. Furthermore, a User Managed Access Control (UMAC) system, which supports wide-ranging access control features, is designed as an ACAD application. UMAC is a decentralized, conflict-free, and administrator-free model by which all subjects are able to manage their objects by delegating any fine-grained subset of their responsibilities to others yet retaining control to revoke privileges as desired. Moreover, UMAC's delegation and revocation mechanism is distinguished by the following features: the privilege to delegate a privilege can be held independently of holding the privilege itself, the privilege to revoke a privilege can be held independently of the privilege to delegate it, and any privilege can be revoked from any grantee along the delegation path without affecting other grantees. This introduces the concept of self-governance in the context of access control. The thesis also incorporates a widespread framework to resolve access conflicts of environments that require both positive and negative privileges. The thesis validates ACAD by providing guidelines of policy specification, to guarantee the termination of policy reasoning and the well-definedness of effective privileges. Finally, we introduce two measures to highlight ACAD with respect to other significant models .

The rest of this thesis is organized as follows. Chapter 2 reviews the literature of access control. Chapter 3 discusses hybrid models and provides a framework to resolve conflicts by supporting 48 strategies simultaneously. Chapter 4 establishes the ACAD specifications. Chapter 5 introduces creation time policies as a means of constraining access control administration. Chapter 6 describes the formal semantics of the model defined operationally by a relational model. Chapter 7 proposes user-managed access control. Chapter 8 justifies features of ACAD in a comparison with several other noteworthy systems. Finally, Chapter 9 concludes the thesis and describes future work.

Chapter 2

LITERATURE REVIEW

Researchers have investigated a variety of issues concerning the access control. This chapter clusters these issues in six categories, namely access control model, access control administration, role-based access control, decentralized access control, access control granularity, and miscellaneous related topics. Other work that strongly influences parts of the thesis will be cited closer to the points where they are related.

2.1 Access Control Model

An access control model is conceptually viewed as maintaining an access control matrix [Lampson 1971], which was first introduced for operating systems. However, since an access control matrix in practice is often very large and sparse, storage refinements are required. A major approach is to implement the access control matrix implicitly by rules. As examples of rule-based access control, Graham-Denning and Harrison-Ruzzo-Ullman are two similar enhanced models in which protection rules have been proposed [Graham and Denning 1972, Harrison et al. 1976]. Although the latter has a broad expressive power, both models have storage inefficiency. The Take-Grant model [Jones et al. 1976] based on directed graphs is another improved version of the matrix model. This model provides a compact way of representing the access control data as well as supporting the transferring of rights.

2.2 Access Control Administration

Access control administration is responsible for handling update requests on *metadata* (metadata refers to access control data). Mandatory Access Control (MAC) models are an example of fully centralized administration [Bell and Lapadula 1976], in which a subject may access classified objects in accordance with the subject's clearance. The only form of delegation or revocation, then, is to reclassify a subject or an object. MAC has no flexibility, and it is not applicable when subjects or objects may not be classified within a limited number of groups. On the other hand, Discretionary Access Control (DAC) models are more flexible, and the administration model may be centralized or decentralized. For example, Role Based Access Control (RBAC) is a mechanism that typically provides central administration [Sandhu 1993] by defining roles. Although RBAC can be used to model arbitrary DAC systems, the decision of which subjects and objects are to be assigned to which roles is centrally controlled. The next section reviews RBAC models and several supporting extensions.

2.3 Role-Based Access Control

Role Based Access Control (RBAC) models [Ferraiolo et al. 2001] provide a mechanism in which roles usually reflect job titles. However, in traditional role-based models, roles form a hierarchical relationship for the sake of efficiency [Ferraiolo et al. 2001]. For example, a project manager has his special privileges as well as all privileges of the project developers reporting to him. Thus, privileges are propagated through the role hierarchy. Since the first publication of the RBAC model [Ferraiolo et al. 1992], many researchers have investigated various aspects of RBAC, such as exploring properties of the roles hierarchy [Al-Kahtani and Sandhu 2003; Ferraiolo et al. 2003; Jansen 1998] and separation of duties [Botha and Eloff 2001; Joshi et al. 2003; Kuhn 1997]. Wang and Osborn recently proposed to exploit the group hierarchy for user to user and role to role delegations to overcome the shortcomings of RBAC-based delegation models, which suffer from needing to modify the role hierarchy in a very complex structure [Wang and Osborn 2003; Wang and Osborn 2006]. Also, Joshi and Bertino assume the presence of more than one hierarchy among the subjects, and discuss the simplest delegation (with no further delegation) and revocation (no cascade) mechanisms in their work [Joshi and Bertino 2006]. Furthermore, some enhancements have been proposed to the RBAC model for distributed environments [Park and Hwang 2003; Wedde and Lischka 2003].

There have been several restrictions on the use of RBAC in practice. First, it is not normally suggested for applications in which a natural role hierarchy does not exist. Second, delegation and revocation are not sufficiently discussed in the RBAC literature; moreover, the proposals are mostly impacted by the centralized role control of RBAC. Scalability to hundreds of thousands of subjects and millions of objects is also a problem due to the model's central administration. The considerable work on administrative and temporal RBAC models is separately reviewed in the following subsections.

2.3.1 Administrative RBAC

There are several endeavours to decentralize the role administration of RBAC [Sandhu et al. 1999; Sandhu and Munawer 1999; Oh and Sandhu 2002; Kern et al. 2003; Oh et al. 2006]. Nevertheless, none of these models results in a pure decentralized administration. The difficulty is that RBAC has been designed to simulate organizational authorizations, assuming that role hierarchies are essentially centralized and mostly static. This assumption is an innate impediment to developing a successful decentralized RBAC model. Therefore, all these works present similar concepts and rely on the organizational hierarchies exploited by RBAC. In particular, Kern et al. address several shortcomings of ARBAC97 [Sandhu et al. 1999], ARBAC99 [Sandhu and Munawer 1999], and ARBAC02 [Oh and Sandhu 2002; Oh et al. 2006], in which the notion of administrative roles, mobile and immobile role memberships, and the concept of independence of an organisational unit and role hierarchies, respectively, have been introduced. Kern et al. introduce the concept of scopes in their model (ERBAC) to describe the objects over which an administrator has authority. Scopes are principally similar to the concept of domains in ARBAC02; however, scopes are defined as an abstract concept and do not have to mirror an organizational structure. Yet, ERBAC does not address delegation and revocation. Delegation and revocation in ARBAC97, ARBAC99, and ARBAC02 are restricted to existing roles, which limits their flexibility. Moreover, there is no mechanism provided to expand (or decrease) the administrative scopes: scopes themselves are administered centrally. Crampton and Loizou formally define the scoped administration of role-based access control model (SARBAC) using a graph formalism. SARBAC overcomes several shortcomings of ARBAC models. Intuitively, the authors propose several types of updates for the role hierarchy; yet, their model does not address how the scopes can be updated [Crampton 2002; Crampton and Loizou 2003].

Using the logic introduced by Jajodia et al. [Jajodia et al. 2001], Wang et al. propose an attribute-based access control in which they allow hierarchical structuring on any attribute [Wang et al. 2004]. The logic is technically suitable but has no industrial support.

Rosenthal and Sciore propose a collaborative administration model in which they present policies as conjunctions of factors [Rosenthal and Sciore 2004]. Therefore, when a circumstance changes, only the relevant factor(s) need to be revised instead of the whole policy.

2.3.2 Temporal RBAC

Bertino et al. propose a temporal role-based model (TRBAC) to support temporal constraints for role enabling, which is important in time-sensitive applications [Bertino et al. 2001c]. The TRBAC model was generalized, forming GTRBAC, to provide several language constructs to support temporal constraints for role enabling, role activation, role assignments, etc. [Joshi et al. 2001; Joshi et al. 2005]. The authors also address the problem of incompatibility between role hierarchies and temporal constraints [Joshi et al. 2002a, Joshi et al. 2002b]. Moreover, Bhatti et al. propose an XML-based specification of GTRBAC, so called X-GTRBAC [Bhatti et al. 2005a]. The original X-GTRBAC model has no administrative features. Hence, the authors subsequently define a nice administrative model [Bhatti et al. 2004, Bhatti et al. 2005b], which is decentralized in the sense that there is a partially ordered administrative domain hierarchy, and each domain is independently administered by its own team of administrators. The administrative domains map the functional units, and the highest role in each functional hierarchy is called the administrative role. They also propose a policy integration mechanism to resolve possible conflicts existing between domains. The X-GTRBAC administrative model suffers from lack of scalability due to the following reasons: first, since the model is based on the RBAC framework, modifying the role hierarchy is complex; also, modifying administrative domains has not been envisioned in X-GTRBAC; in fact, there is an assumption in the model that the enterprise includes predefined domains which may not change dynamically; furthermore, X-GTRBAC specification imposes capability-list implementation (for the distribution of the abstract access control matrix) and identity-based authentication which affect the applicability of the model in arbitrary large enterprises. It is desirable to support unknown users (as opposed to capabilities and identity-based mechanisms) characterized by certain properties, such as a user's location or role.

2.4 Decentralization

The concept of delegation and revocation has been used as a means of decentralization of administration. Decentralized access control mechanisms were first proposed for System R to permit users to share and control their data in multi-user databases systems [Griffith and Wade 1976]. System R introduced its *grant* and *revoke* commands for decentralized administration. The System R model was later extended to support negative authorizations and more expressive revocation algorithms for relational data management systems [Bertino et al. 1999]. Moffet used the concept of *domains* to specify administrative domains in distributed systems [Moffett 1990]. Ravichandran and Yoon propose to use delegation to distribute the workload among several grouped peer-to-peer communities [Ravichandran and Yoon 2006]. They do not provide a revocation mechanism in their model. Further, similar mechanisms for RBAC models were proposed [Barka and Sandhu 2000a; Barka and Sandhu 2000b; L. Zhang et al. 2002; L. Zhang et al. 2003; X. Zhang et al. 2003]. Barka and Sandhu proposed RBDM (Role-Based Delegation Models) in which the unit of delegation is "role". They also discussed temporary delegation as well as grant-dependent and grant-independent revocations in their work. However, RBDM has not been formalized. L. Zhang et al. proposed RDM2000 (Role-based Delegation Model), which extended RBDM by providing multilevel delegations and formal definitions of delegation features. Later, X. Zhang et al. extended RDM and RBDM models by adding finer-grained delegation to their model, so called PBDM (Permission Based Delegation Model). The PBDM series do not support constraints in delegation, neither do they support decentralized applications. Recently, Ahn et al. proposed an access control model based on RBAC for a collaborative environment [Ahn et al. 2003; Tolone et al. 2005]. Their model exploits the powerful rule-based mechanism of RDM2000 to define constraints. However, it still lacks the ability for further delegation.

In all of these models, the delegation and revocation mechanism is centrally controlled. There is no way of expanding administrative features by individual users and a central security officer (or team) controls the permission flow. Hence, administration becomes a bottleneck in applications where potentially all users wish to administer their data.

2.5 Granularity

Access control granularity, in the context of a hierarchically organized database, refers to the extent to which different levels of access can be defined on objects or parts of objects. Fine-grained access control manages access authorizations on small pieces of objects. Many proposed models [Graham and Denning 1972; Harrison et al. 1976; Jones et al. 1976; Lampson et al. 1976] assume access control at the object level only and ignore any internal structure within objects. However, if objects entail a hierarchy, the distinction between objects and sub-objects becomes meaningful. Jones examined an object-level access control model for client-server object databases [Jones 1997]. He provided a fine-grained access control model, which supports navigating the data structure by inter-object references. Zhang et al. introduced access control vectors and slabs for fine-grained access control based on a code-based scheme to represent a more compact structure for control data [Zhang et al. 2005].

2.6 Miscellany

This section reviews various access control topics, including access control properties, policy analysis, and comparison of access control models..

To evaluate whether an access control instance conforms to an access control policy, one may check the mechanism's properties such as safety, invulnerability, no-information-flow, and non-interference [Bell and Lapadula 1976; Biba 1977; Focardi and Gorrieri 1997; Goguen and Meseguer 1983; Jaeger and Tidswell 2001]. There are also considerable research works on policy analysis [Jajodia et al. 1997; Li et al. 2003; Bertino et al. 2001a; Bertino et al. 2001b; Jajodia et al. 2001; Bertino et al. 2003]. Policies are often analyzed by exploiting logical languages and some ad hoc rules and/or rule properties. Notably Li et al. propose a formal specification and semantics for policy analysis in distributed environments.

Comparison of access control models has also interested researchers. Tripunitara and Li propose a comparison mechanism based on simulation to compare two given access control models [Tripunitara and Li 2004]. The simulation is based on transition networks of access control models, and if model A can simulate all states of model B, A is said to be at least as expressive as B. Using this mechanism, the authors conclude that ARBAC97 [Sandhu et al. 1999] is limited in its expressivity, and also a trust management language [Chander et al. 2001] is at least as expressive as ARBAC97.

Jaeger et al. introduce an access control space in which authorizations are divided into five permission sets, namely permissible, specified, obligated, prohibited, and unknown [Jaeger et al. 2003]. The permissible set consists of authorizations that are known and can be assigned to a given subject S. The specified set consists of those permissible authorizations that have been assigned to a given subject S. The obligated set consists of authorizations that are required (for example by the system) to be assigned to a given subject S. The prohibited set consists of those authorizations that must not be assigned to a given subject S. The unknown set includes those subspaces on which neither permissible rules nor prohibited rules are defined. The authors believe that understanding these subspaces and their intersections assist system administrators to better manage policies.

Chapter 3

DEDUCING EFFECTIVE ACCESS CONTROL

An explicit access control matrix is typically large and sparse since, in practice, authorizations are explicitly defined only for a small proportion of subjects and objects; the rest of the matrix is null. Yet, since the effective access control matrix is required to be well-defined, which means every cell must have an *effective* authorization (no null or conflicting value is allowed), explicit authorizations are propagated throughout the subject and object hierarchies to obtain the effective authorizations.

Conflicts may occur when authorizations are propagated throughout the hierarchies since a particular node may be simultaneously authorized by one of its ancestors for some activity and denied by another ancestor for that same activity. A conflict is also said to occur when the set of ancestors provides neither permission nor denial for some activity.

This chapter addresses propagation of authorizations and resolution of conflicts. In particular, Section 3.1 introduces subjects and objects as well as the hierarchies among them. Section 3.2 describes how conflicts may happen on a given hierarchy. Section 3.3 reviews major conflict resolution policies. Section 3.4 combines the policies to obtain 48 strategy instances. Section 3.5 provides a logical formalism for the combined strategies. Section 3.6 describes a unified parametric algorithm to support all the instances. Section 3.7 demonstrates the experimental results. Section 3.8 describes the propagation of authorization on hierarchies. Finally, Section 3.9 reviews the literature of conflict resolution models.

3.1 Subjects and Object Hierarchies

Definition 1 *(Access Control Universe).* The access control universe *U* is the collection of all objects of any type in the system. (Object types are introduced in Section 4.3.) We assume a simple set language to describe members of the universe. In Figure 1, in Chapter 1, the set All-objects represents the access control universe. Throughout this thesis, sets are depicted with capital letter labels, and those members which are objects are depicted with labels composed of lower-case letters.

Definition 2 *(Subjects).* The set of subjects $Public \subseteq U$, are the collection of all active objects that are able to launch access requests. Figure 1, in Chapter 1, depicts the subset as Subjects to represent all active objects. Throughout this thesis, the subject members are depicted with labels that start with a capital letter.

Definition 3. (*Owner*). A set of subjects $S \subseteq Public$, who are responsible for a given object $O \in U$, is called O's owner. Deciding who is the owner of an object can be expressed by access control policies, which are explained in Section 5.1.

Definition 4 *(Permission).* A permission *p* is the right to execute a specific method on a given object $o \in U$. Note that permissions in ACAD are not limited to a fixed set of rights such as *read* or *write*.

Access control can be defined as a mechanism to dictate the *permissions* that particular sets of *subjects*, i.e., users and applications, are given to access particular sets of *objects*, i.e., data.

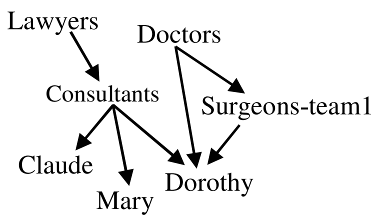
Definition 5 *(Subject Constraint).* A constraint *C* is an expression that defines the domain of applicable subjects who may (or may not) be granted a permission *p*. We assume a simple set language to denote the applicable subset of *Public*.

Definition 6 *(Access Authorization).* An access authorization *a* is a quadruple $<C, mode, p, O>$ permitting (or denying) permission *p* on a non-empty set of objects $O \subseteq U$ to be assigned to any subset of *Public* who satisfy constraint *C*; an authorization mode is either *permit* or *deny*, as described in detail in Section 3.8.

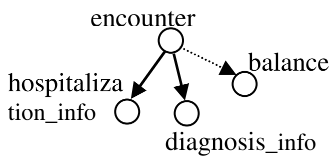
The set of subjects and the set of objects both form hierarchies that can be represented as directed acyclic graphs. Following convention, outgoing edges from a group in the subject hierarchy lead to all members of that group (either subgroups or individual users), and outgoing edges from an object in the object hierarchy lead to all its subobjects. It is important that the hierarchies not be restricted to form trees.

The subject hierarchy when viewed bottom-up maps group membership: if (S_1, S_2) is an edge in the hierarchy, every member of S_2 is also a member of S_1. Note that an individual subject in our model is represented as a group with no child vertex. For example, in Figure 3(a), subjects *Dorothy* and *Claude* are individuals, whereas subjects *Surgeons-team1*, *Doctors*, *Consultants*, and *Lawyers* are groups. *Dorothy* is a direct member of *Surgeon-team1*, *Doctors*, and *Consultants*; *Claude* and *Mary* are direct members of *Consultants* only. Since subject *Consultants* is a member of subject *Lawyers*, subjects *Dorothy*, *Mary*, and *Claude* are members of *Lawyers* too. In general, a group can have zero or more subgroups and zero or more individual nodes; and, a member of a group enjoys all authorizations of that group.

On the other hand, the object hierarchy is a collection of distinct top-down ownership hierarchies that each maps a *has-a* relationship between objects. In general, an object can have zero or more subobjects; and its assigned authorizations are propagated to all of its subobjects. An ownership hierarchy is a sub-graph of all objects owned by the same subject. Ownership hierarchies may be connected to one another through cross-references. For example, in Figure 3(b), object *encounter* includes two nested objects *hospitalization_info* and *diagnosis_info* as well as cross-referring (depicted by the dotted arrow) object *balance* that is owned by another subject. Permissions from higher objects (source vertices) in the object hierarchy are propagated to nested objects (destination vertices) within the same ownership only. For example, in Figure 3(b), permissions on *encounter* are propagated down to *hospitalization_info* and *diagnosis_info* but not to *balance*.



(a) Subject hierarchy maps the group membership.



(b) Object hierarchy maps nested objects.

Figure 3. Example of subject and object hierarchies.

3.2 Conflicts on Hierarchies

Figure 4 illustrates a subject inheritance hierarchy including nine subjects. The arrows represent group membership (e.g., subjects S_4 and S_5 are members of subject/group S_3) and the sign labels represent explicit authorizations (+ indicates positive authorization and – represents denial). For simplicity of exposition, we assume that access to an object is either granted or denied (rather than separately controlling reading, writing, and other operators), and we illustrate only authorizations for a single object. The figure shows that subjects S_2 and S_4 are explicitly labelled to access the object, whereas subject S_5 is explicitly denied from accessing it.

Given the data in Figure 4, assume we are interested in knowing whether or not subject *User* is authorized to access the object. One may interpret the data to mean that the object is accessible to subject *User* since *User* is a descendant of S_2 and thereby inherits S_2's authorizations. However, another may argue that the object should not be accessible to *User* since he is a member of S_5 which is denied access. In fact, there is a conflict in the system. Conflict resolution policies are needed to answer such questions.

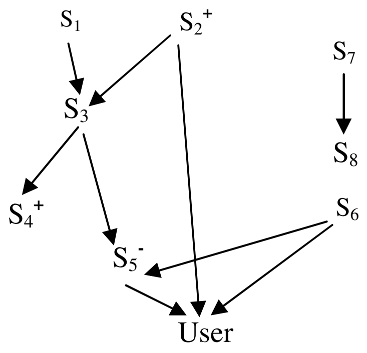


Figure 4. Conflicts on hierarchies.

There exist several conflict resolution policies, such as "denial takes precedence" and "the most specific takes precedence," in the literature of access control models. Yet, adopting one simple policy, such as "the most specific authorization takes precedence," is not sufficient in practice. For instance, such a policy is insufficient where the subject hierarchy is more complex than tree-based structures and therefore, a subject may have more than one "most specific" authorization. For example, in Figure 4, neither S_2 nor S_5 is more specific to *User*, with respect to the other, since both of them are at the minimum distance of 1 from *User*. Furthermore, there are situations in which the highest authority (not the most specific one) should be the final arbiter. For instance, assume a student is authorized by the university athletic office to referee hockey games on campus (which requires more than 20 hours per week for several weeks); however, he is required by the department not to accept heavy non-departmental tasks (in order to comply with his full-time registration status). In such a case, the university administration may override the department by deciding to let him referee. To visualize such a case, assume there is an edge from S_1 to S_2 in Figure 4 and S_1 is labelled positively. Representing the student by *User*, the referees group by S_2, the members of the department by S_5, and the university members by S_1, it is apparent that for this enterprise the most global authorization should take precedence in resolving the conflict.

Some have proposed the "negative takes precedence" policy, but this too is not universally acceptable. For instance, conflicts often are resolved by the "majority takes precedence" rule in voting systems. Additionally, the open policy recommends a default positive authorization for subjects which are not explicitly permitted to access a particular object [Harrison et al. 1976; Lampson 1971]. Therefore, there are applications in which "positive takes precedence".

Even from these simple examples we see that, in many systems, it is required to combine various conflict resolution policies to obtain a comprehensive conflict resolution strategy. Moreover, each policy may encompass several variants, and consequently many strategy instances are possible. If an access control system is to be deployed in a wide range of enterprise settings, many complete strategies must be supported. What are all the legitimate strategy instances? Is there a unified algorithm to support all instances parametrically?

3.3 Conflict Resolution Policies

Definition 7 (*Explicit Access Control Matrix*). The initial access matrix *EACM*, which includes explicit authorizations only is called explicit access control matrix; *EACM* is represented as a set of quadruples <*subject*,*object*,*permission*,*value*>, in which $subject \in Public$, $object \in U$, $permission \in Permissions$ (cf. Definitions 1-3), and *value* is either 1 or 0 representing an explicit permission or denial, respectively.

Definition 8 (*Effective Matrix*). The effective matrix *EM* is a well-defined three-dimensional Boolean matrix indexed by *Public*, *U*, and *Permissions* (cf. Definitions 1,2, and 4). The value of EM[i,j,k]for all $i \in Public$, $j \in U$, and $k \in Permissions$ is either 1 or 0 representing an effective permission or denial, respectively.

Given an explicit matrix, conflict resolution strategies and propagation modes are used to fill in all derived authorizations to determine the effective matrix. Because the explicit matrix is typically very sparse, practical systems will store the explicit matrix (perhaps as capability lists [Dennis and Van 1966] or access control lists [Saltzer 1974]) and compute access control authorizations as needed by executing an authorization propagation and conflict resolution algorithm on an appropriately extracted subset of that matrix. Conflict resolution is required when propagating authorizations results in no decision for a particular <subject, object, operation> triple or when both positive and negative authorizations can be derived for that triple. Commonly used conflict resolution policies are outlined as follows:

Preference Policy. Preferred authorization (with one of two modes: either positive or negative) is determined by the system installer at configuration time. This policy determines which authorization wins when both positive and negative authorizations (or neither negative nor positive authorization) can be derived for a particular triple. Negative authorization is preferred (known as closed policy) in more restricted systems such as the military; positive authorization may be preferred in more open applications such as public information systems.

Locality Policy. The common mode of this distance-based policy states that the most specific authorization takes precedence. It applies to distributed organizations whose local branches may recognize an exception to a general rule. For instance, a department in a university may admit an outstanding applicant although the general admission requirement is not completely met. Thus, for a given subject, when both positive and negative authorizations can be derived from different ancestors, the one that is closer to the subject wins. Note that the distance between two nodes (subjects) in a directed acyclic graph is measured by computing the shortest directed path. The locality policy is not deterministic since no authorization wins when the distances are equal.

As an alternative for the locality policy, some enterprises might choose "globalization," where the most general authorization takes precedence. One application of this policy is in distributed organizations whose headquarters makes the final decision on a pre-approved task by a local office. Similarly, a supreme court may override an appealed decision. For a given subject, when both positive and negative authorizations can be derived from different ancestors, the one that is farther from the subject wins. Similar to the usual locality policy, the distance between two nodes is measured by computing the shortest path, and again this mode of locality is not deterministic since no authorization may win.

Majority Policy. This policy states that the conflict can be resolved based on votes, and the authorization that has the majority wins. The application of this policy is in situations where several parties have different opinions for giving or not giving the authorization to a particular member and the decision is made by votes. For instance, GATT's current members vote to determine if a new applicant can get into the group. By applying this policy, the dominant authorization takes precedence. This policy is also non-deterministic since it can result in a tie.

Default Policy. This policy is applied only to root subjects or objects for which no authorization has been defined. Closed systems, such as in the military, require negative authorization by default; however, open systems, such as public information applications, initially allow any subject to enjoy a positive authorization. There are applications in which the default policy is not appropriate; for instance, one may wish to give priority to the explicit authorization. This policy is deterministic and has three modes (default positive, default negative, or ignore), but applies to root subjects only.

Note that for non-root nodes, only the preference policy is deterministic.

3.4 Combined Strategies

Figure 5 illustrates five conflict resolution strategies based on combining, in different orders, the popular conflict resolution policies summarized above [Chinaei and Zhang 2006]. They are given the mnemonics DLP, DLMP, DP, DMLP, and DMP, in which D, L, M, and P indicate Default, Locality, Majority, and Preference policies, respectively. Two properties are guaranteed: first, none of the policies are redundant, and second, there is no conflict after applying the last step. Note that in this framework the Preference policy is always the last applicable policy, and the other three policies, Default, Locality and Majority, are optional. Moreover, the Default policy, if applicable, is the first policy since otherwise it is meaningless. Note that no other combined strategy can be meaningfully composed from these basic conflict resolution policies. For example, the preference policy cannot be optional and must be considered last, since it is the only policy that is well-defined on every node.

Propagation mode: {*pass through*, *block by*, *override*}

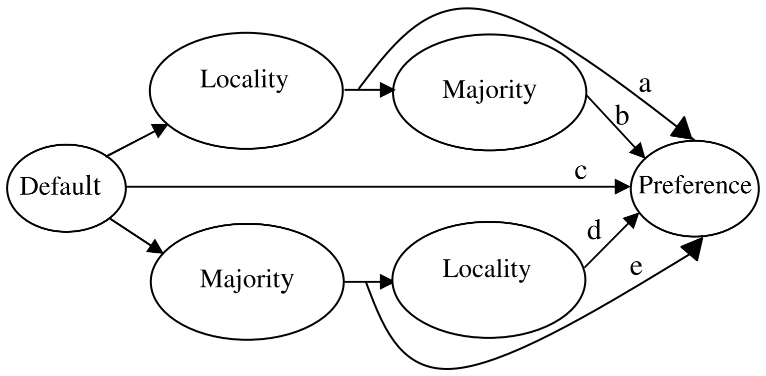


Figure 5. Combined conflict resolution strategies.

Because the default policy can take three modes and the locality and preference policies can take two modes each, there are 48 different strategy instances in total that can be derived from Figure 5 [Chinaei et al. 2007]. (Paths ending with *a*, *b*, and *d* generate twelve instances each, and paths ending with *c* and *e* generate six instances each.) Examples of strategy instances are D^+LP^- (which means, first, apply the positive authorization as default, then apply the locality policy, and finally apply the negative takes precedence if some conflict still exist), D^-GP^- (which means, first, apply the negative authorization as default, then apply the globalization mode of locality, and finally apply the negative takes precedence if some conflict still exist), LP^+ (which means, first, apply the locality policy, and then apply the negative takes precedence), etc. Moreover, the propagation mode can be either *pass through*, *block by*, or *override*, in which propagating authorization may pass through the explicit authorization, be blocked by it, or override it, respectively. For instance, assume Figure 4 does not include the edge $S_2 \rightarrow$User; therefore, to calculate permission for User, either S^+_2 can override S^-_5, or $S^-_{5\text{ can block}}$ S^+_2, or S^+_2 can pass through (but not override) S^-_5. Note that block by and override prioritize permissions along a single path, whereas locality and globalization prioritize permissions even in different paths.

3.5 Logical Formalism

This section provides guidelines for logically implementing the strategy instances explained in the previous section. Like the Authorization Specification Language [Jajodia et al. 2001], policies in ACAD are represented by logic programming rules. The ACAD model introduces four reserved predicates named *allow*, *disallow*, *permit*, and *deny*. Whereas predicates *allow* and *disallow* are used to define the effective access control matrix as defined in Section 1.2 (allow representing EM[*s*,*m,o*]=1 and disallow representing EM[*s*,*m,o*]=0), predicates *permit* and *deny* are used to define an explicit access control matrix and to constrain the propagation of permissions.

Examples:

(a) Explicit permission is given to doctor *Dana* to read Patricia's encounter and its sub-elements:

permit(Dana, methodRead, patricia_encounter)

(b) Robert cannot change consent form *x*:

deny (Robert, methodChange, x)

(c) Effective permission is granted to patient *Patricia* to read her personal information:

allow(Patricia, methodRead, patricia_personal_info)

(d) Effective permission is denied to patient *Patricia* to delete her medical record:

disallow(Patricia, methodDelete, patricia_medical_record)

3.5.1 Propagation Policies

To determine the values of all cells in the effective matrix, authorizations should be propagated within both hierarchies of subjects and objects in order to transform the explicit access control matrix to the corresponding effective matrix. This section selects a few strategy instances, introduced in Section 3.4, and provides the corresponding logical rules to transform an explicit access control matrix to an effective one. First, we propagate all authorizations from the explicit access control matrix to an intermediate matrix which consists of two predicates *maybe*() and *maybeNot()*, regardless of the conflict resolution strategy is. These predicates have five variables, namely, *S*, *M*, *O*, *D*, and *P*, which represent the subject, authorization, object, the corresponding distance (which is later used for the locality rule), and set of propagation paths, respectively.

Assume the propagation mode is "pass through". The following rule,

$$maybe(S,M,O,0,P) \leftarrow permit(S, M,O),\ P=\{S\}. \qquad (1)$$

means that if there is an explicit authorization in the explicit access control matrix stating that subject S is permitted to execute method M on object O, a corresponding tuple is inserted into the intermediate matrix stating that the corresponding distance for the authorization is 0. Set P represents the propagation paths, which is represented by string S in this case. Moreover, the following rule,

$$maybe(S,M,O,D,P) \leftarrow maybe(X, M,O, D',P'), child(S,X),$$
$$D=D'+1, P=P' \| X. \quad (2)$$

means that if subject *S* is a member of group *X* that is granted access to O from distance D' and set of paths P', all possible permissions of *X* are propagated to *S* with distance D'+1 and path P. (We use the notation P' || X to mean that string X is concatenated to all paths in set P'.) Similarly, the following rule

$$maybe(S,M,O,D,P) \leftarrow maybe(S, M,X, D', P'), child(O,X),$$
$$D=D'+1, P=P' \| X. \quad (3)$$

means that if subject *S* is granted access to object *X* from distance D' and set of paths P', its permission is extended with distance D'+1 to all sub-elements of X and path P.

If the propagation mode is "block by", rules (2) and (3) are replaced by the following rules:

$$maybe(S,M,O,D,P) \leftarrow maybe(X, M,O, D',P'), child(S,X), \neg\, deny(S, M, O),$$
$$D=D'+1, P=P' \| X. \quad (4)$$

and

$$maybe(S,M,O,D,P) \leftarrow maybe(S, M,X, D',P'), child(O,X), \neg\, deny(S, M, O),$$
$$D=D'+1, P=P' \| X. \quad (5)$$

respectively.

Similar to rules 1-3, when the propagation mode is pass through, the following three rules,

$$maybeNot(S,M,O,0,P) \leftarrow deny(S, M,O), P=\{S\}. \quad (6)$$

$$maybeNot(S,M,O,D,P) \leftarrow maybeNot(X, M,O, D',P'), child(S,X),$$
$$D=D'+1, P=P' \| X. \quad (7)$$

maybeNot(S,M,O,D,P) $\leftarrow$ *maybeNot(S, M,X, D',P'), child(O,X),*

D=D'+1, P=P' || *X.* (8)

propagate all negative authorizations within both hierarchies of subjects and objects and insert them into the intermediate matrix.

Now, recall that the effective matrix is constructed using predicates *allow* and *disallow*. The following rules

disallow(S, M,O) $\leftarrow$ *maybeNot(S,M,O,_,_),* ¬ *maybe(S,M,O,_,_).* (9)

allow(S,M,O) $\leftarrow$ ¬ *disallow(S,M,O).* (10)

corresponds to the P^+ strategy instance, where _ indicates a *don't-care* term..

Now, consider the D^-P^+ strategy instance, which means the default and preferred authorizations are negative and positive, respectively. The following rules together with rules (1) to (3) and (6) to (7) represent this policy:

allow(S, M, O) $\leftarrow$ *maybe(S, M, O, _,_).* (11)

disallow(S, M, O) $\leftarrow$ ¬ *allow(S, M, O).* (12)

These rules state that as long as there is a corresponding positive authorization in the intermediate matrix, S is effectively allowed to execute M on O; and, once all positive authorizations are propagated and transformed, all non-filled cells of the effective access matrix are treated as denial of permission.

To state the locality policy, we define two temporary predicates *negativeCloser() and positiveCloser()*, each of which has four variables, namely, *S*, *M*, *O*, and *D*, which represent the subject, authorization, object, and the corresponding distance, respectively; furthermore, we define two similar predicates *negativeFurther()* and *positiveFurther()* to state the globality rule:

negativeCloser(S,M,O,c) $\leftarrow$ *maybeNot(S,M,O,j,_), j<c.* (13)

positiveCloser(S,M,O,c) $\leftarrow$ *maybe(S,M,O,j,_), j<c.* (14)

negativeFurther(S,M,O,c) $\leftarrow$ *maybeNot(S,M,O,j,_), j>c.* (15)

positiveFurther(S,M,O,c) $\leftarrow$ *maybe(S,M,O,j,_), j>c.* (16)

Now consider D^-LP^- in which negative default, locality, and negative preference are indicated. If the propagation mode is again pass through, this can be represented by rules (1) to (3), (6) to (8), (12), (13), and the following rule,

allow(S,M,O) $\leftarrow$ *maybe(S,M,O,i,_)*, ¬ *negativeCloser(S,M,O,i).* (17)

To state the strategies in which the majority rule is in place, we define two other temporary predicates *negativeBigger() and positiveBigger().* These predicates again have four variables, namely, *S*, *M*, *O*, and C, which represent the subject, authorization, object, and a counter, respectively:

negativeBigger(S,M,O,c) $\leftarrow$ *maybeNot(S,M,O,_,P), |P|>c.* (18)

positiveBigger(S,M,O,c) $\leftarrow$ *maybe(S,M,O,_,P), |P|>c.* (19)

$maybe(S,M,O,D,P) \leftarrow maybe(S,M,O,D,P_1),$

$maybe(S,M,O,D',P_2), P = P_1 \cup P_2.$ (20)

$maybeNot(S,M,O,D,P) \leftarrow maybeNot(S,M,O,D,P_1),$

$maybeNot(S,M,O,D',P_2), P = P_1 \cup P_2.$ (21)

Each of rules (20) and (21) combines multiple paths into a single set, for positive and negative authorizations, respectively; the sets' cardinality determine majority.

For instance, to represent the D^-MP^- strategy instance, rules (1) to (3), (6) to (8), (12), and (18) to (21), as well as the following rule are applied:

allow(S,M,O) $\leftarrow$ *positiveBigger(S,M,O,c)*, ¬ *negativeBigger(S,M,O,c).* (22)

Finally, to keep the priority among majority and locality rules, we define another temporary predicate *willAllow()* which has three variables namely, *S*, *M*, and *O*, which represent the subject, authorization, and object, respectively. To represent the sophisticated strategy instance of D^+MLP^-, one can apply rules (1) to (3), (6) to (8), (12), (14), and (18) to (21), as well as the following rules:

disallow(S,M,O) $\leftarrow$ *negativeBigger(S,M,O,i),*

¬ *positiveBigger(S,M,O,i).* (23)

willAllow(S,M,O) $\leftarrow$ *positiveBigger(S,M,O,i),*

¬ *negativeBigger(S,M,O,i).* (24)

disallow(S,M,O) $\leftarrow$ *maybeNot(S,M,O,i,_),*

¬ positiveCloser(S,M,O,i), ¬ willAllow(S,M,O). (25)

whereas, to represent D^+LMP^-, rules (23) to (25) should be replaced by the following rules:

disallow(S,M,O) $\leftarrow$ *maybeNot(S,M,O,i,_), ¬ positiveCloser(S,M,O,i).* (26)

willAllow(S,M,O) $\leftarrow$ *maybe(S,M,O,i,_), ¬ negativeCloser(S,M,O,i).* (27)

disallow(S,M,O) $\leftarrow$ *negativeBigger(S,M,O,i),*

¬ positiveBigger(S,M,O,i), ¬ willAllow(S,M,O). (28)

One can formalize other strategy instances similarly.

3.5.2 Propagation Policies Alternatives

Propagation policies explained in Section 3.5.1 can be substituted with alternatives if required by the enterprise. For example,

(a) Assume an enterprise does not wish to propagate authorizations through the object hierarchy, therefore only rules (1), (2), (6), and (7) are applied. (Rules (3) and (8) do not apply here.)

(b) If the enterprise also does not require the permission propagation through the subject hierarchy, rules (2) and (7) should be removed as well.

(c) If the enterprise requires an open policy in which authorizations are allowed unless otherwise explicitly denied, the following rules are applied:
allow(S,M,O) $\leftarrow$*¬ deny (S,M,O,_,_).*
disallow(S, M, O) $\leftarrow$ *¬ allow(S, M, O).*

3.5.3 Policy Soundness

Sections 3.5.1 and 3.5.2 introduced the guidelines for logical implementation of conflict resolution strategies depicted in Figure 5. Notice that, rules (1) to (3) and (6) to (7) demonstrate the propagation phase when the mode is "pass through", rules (4) and (5) illustrate how propagation rules can be enhanced to support other propagation modes, rules (10) and (12) guarantee the effective matrix is well-defined, and the rest of the rules provide guidelines for implementing various conflict resolution strategies introduced by Figure 5. In particular, rules (13) to (16) are applicable when the locality policies are in place; for instance, as shown in Section 3.5.1, rule (13) is applicable to D^-LP^-. Similarly, rules (18) and (19) are applicable when the majority policy is in place. Finally, rules (24) and (27) demonstrate how to handle the priority when both majority and locality policies are in place.

By design, these rules are stratified which means none of the *intensional* predicates are in a negative recursive definition [Garcia-Molina et al. 2002]. Figure 6 illustrates the graph of *intensional* predicates applicable to strategy instance D^+LMP^-, one of the most sophisticated possible instances. This instance requires rules (1) to (3), (6) to (8), (12), (13), (18) to (21), and (26) to (28). Since there is not a cycle with a negative labelled arc in the graph, the set of rules are safe. In fact, the only restriction in the formalism is that rules (10) and (12) cannot be applied simultaneously. Moreover, since the default and preference policies are deterministic, our policy reasoning is sound, which means first it will eventually terminate and also be effective: if at least one of the rules (10) and (12) (and not both) is used, the effective matrix is well-defined.

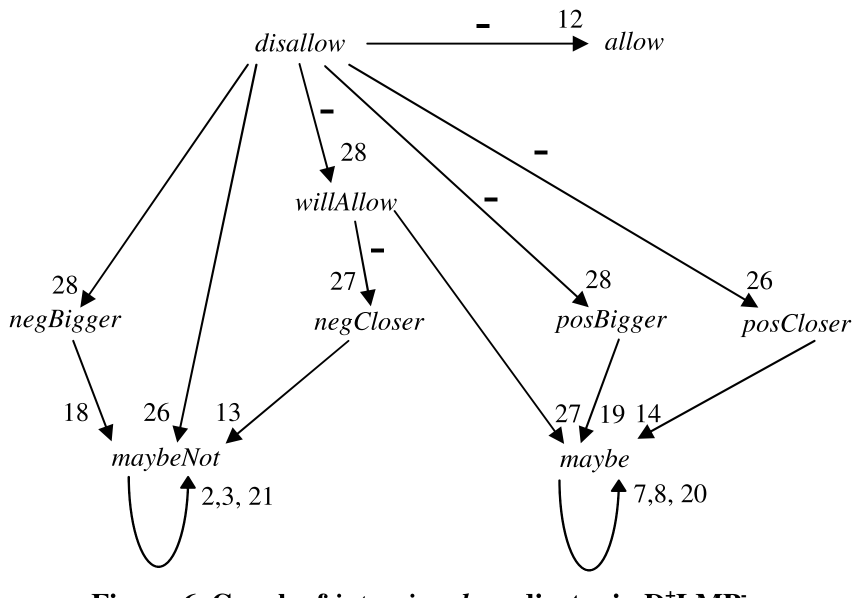


Figure 6. Graph of *intensional* predicates in D^+LMP^-.

Notice that these propagation and conflict resolution rules are independent of the rest of the access control model (in particular of the creation time policies specified in Chapter 5); these same rules define the corresponding effective matrix for *any* given explicit access control matrix following the particular conflict-resolution policy defined. Furthermore, if the effective matrix is not materialized (that is, it is interpreted as a view over the explicit access control matrix), its content automatically reflects any changes made to the explicit access control matrix.

3.6 Unified Algorithm

This section describes an algorithm that propagates explicit authorizations through the subject and object hierarchies, and resolves the possible conflicts based on any of the 48 strategy instances illustrated in Section 3.4. To determine whether a given object, o_j, is effectively accessible to a given subject, S_i, with respect to a given right, r_k, the idea is to apply the following four-step procedure:

Step 1: Consider the maximal sub-graph (called H_1) of the subject hierarchy in which S_i is the sole sink and all other nodes are its ancestors. Similarly, consider the maximal sub-graph (called H_2) of the object hierarchy in which O_j is the sole sink and all other nodes are its ancestors.

Step 2: Assign a letter "d" to all root subjects in H_1 that are unlabeled with respect to right r_k and ancestors of o_j. Similarly, assign a letter "d" to all root objects in H_2 that are unlabeled with respect to right r_k and ancestors of S_i.

Figure 7 illustrates the result of Steps 1 and 2 for subject *User*, object *obj*, and right *read*, illustrated in Figure 4 as the example of conflicts on hierarchies.

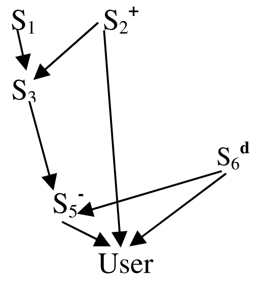


Figure 7. Sub-graph of subject *User*.

Step 3: Propagate all authorization labels down every path to subject *User* and store the distance of each propagated authorization from its source node to its destination node (*User*). For instance, the distance of label - (on S_5) to node *User* is 1; also, there are two distances for label "d" (on S_6) to node *User*: one (with value 1) directly from S_6 to *User*, and one (with value 2) via S_5.

Table 1 illustrates the result of authorization propagation in the "pass through" mode for subject *User*, object *obj*, and right *read* as represented by Figure 7.

Step 4: Apply a particular conflict resolution strategy to resolve any conflicts and derive a final *effective* authorization for the triple $<S_i, o_j, r_k>$.

Table 2 illustrates the result of applying each of the 48 strategy instances (explained in Section 3.4) to Table 1.

Table 1. All read authorizations of *User* on *obj*.

subject	object	right	dis	mode
User	obj	read	1	-
User	obj	read	1	d
User	obj	read	2	d
User	obj	read	1	+
User	obj	read	3	+
User	obj	read	3	d

For example, D^+LMP^+ is the strategy instance in which first the default policy is applied and every root subject which is null is initialized to +; then, if there is a conflict, the Locality policy ("the most specific authorization takes precedence") is applied; then if there is still a conflict, the Majority policy is applied; and finally, if the conflict is not resolved, the preference policy in which the positive (+) authorization takes precedence is applied. Let's see what the result of this strategy instance is on Table 1: by applying the default policy D^+, all mode d's are replaced by +; by applying the locality policy, the conflict is not resolved since there are conflicting modes + and - from the shortest distance 1; however, by applying the majority rule, mode + wins over mode - since there are more + entries than - entries. Note that the preference policy is not applicable to this case since the conflict is resolved before this rule is triggered; however in other hierarchies the conflict may not yet have been resolved.

Table 2. Resolved authorization for each combined strategy.

strategy	mode	strategy	mode	strategy	mode	strategy	mode
$D^+L\mathbf{MP}^+$	+	$D^+L\mathbf{P}^+$	+	$LM\mathbf{P}^+$	+	$D^+\mathbf{M}LP^+$	+
$D^+L\mathbf{MP}^-$	+	$D^+L\mathbf{P}^-$	-	$LM\mathbf{P}^-$	-	$D^+\mathbf{M}LP^-$	+
$D^-L\mathbf{MP}^+$	-	$D^-L\mathbf{P}^+$	+	$\mathbf{G}MP^+$	+	$D^-\mathbf{M}LP^+$	-
$D^-L\mathbf{MP}^-$	-	$D^-L\mathbf{P}^-$	-	$\mathbf{G}MP^-$	+	$D^-\mathbf{M}LP^-$	-
$D^+\mathbf{G}MP^+$	+	$D^+\mathbf{G}P^+$	+	$\mathbf{MP}^+$	+	$D^+\mathbf{M}GP^+$	+
$D^+\mathbf{G}MP^-$	+	$D^+\mathbf{G}P^-$	+	$\mathbf{MP}^-$	+	$D^+\mathbf{M}GP^-$	+
$D^-GM\mathbf{P}^+$	+	$D^-G\mathbf{P}^+$	+	$L\mathbf{P}^+$	+	$D^-\mathbf{M}GP^+$	-
$D^-GM\mathbf{P}^-$	-	$D^-G\mathbf{P}^-$	-	$L\mathbf{P}^-$	-	$D^-\mathbf{M}GP^-$	-
$D^+\mathbf{MP}^+$	+	$D^+\mathbf{P}^+$	+	$\mathbf{G}P^+$	+	$\mathbf{M}LP^+$	+
$D^+\mathbf{MP}^-$	+	$D^+\mathbf{P}^-$	-	$\mathbf{G}P^-$	+	$\mathbf{M}LP^-$	+
$D^-\mathbf{MP}^+$	-	$D^-\mathbf{P}^+$	+	$\mathbf{P}^+$	+	$\mathbf{M}GP^+$	+
$D^-\mathbf{MP}^-$	-	$D^-\mathbf{P}^-$	-	$\mathbf{P}^-$	-	$\mathbf{M}GP^-$	+

For each strategy instance in Table 2, we use a bold font to show which policy has determined the effective authorization when applied to our example. For example, in the last strategy instance, MGP^-, by applying the first policy (Majority), the positive authorization wins since there are two +'s (rows 4 and 5) as opposed to only one - (row 1) in Table 1. Therefore, the localization and preference policies of the MGP^- instance are not applicable to this case.

3.6.1 Algorithm *Resolve()*

This section defines our conflict resolution algorithm. Figure 8 illustrates Algorithm *Resolve()* which computes the derived authorization mode of a given subject with respect to a given object and right. The algorithm parameters are *s*, *o*, *r*, *dRule*, *lRule*, *mRule*, and *pRule;* and the result is either + or -. Parameters *s*, *o*, and *r* designate a particular subject, object, and right, respectively, on which the caller is interested to know whether or not the object is accessible to the subject with respect to the specified right. Parameters *dRule*, *lRule*, *mRule*, and *pRule* determine the conflict resolution strategy, based on which the final right of the subject on the object must be derived. In particular, parameter *dRule* represents the default authorization policy and takes either of the three values "+", "-", or "0", which respectively states that the unlabelled root ancestors of the subject and object are to be initialized to positive authorization, negative authorization, or remain null (no default authorization policy). Parameter *lRule* represents the locality policy; its value is either *min*(), *max*(), or *identity*(), which represent “the most specific authorization takes precedence, ” “the most general authorization takes precedence, ” or “no locality policy” modes, respectively. Parameter *mRule* takes three values *before*, *after*, or *skip*, which determines whether the majority policy is applied before the locality policy, after it, or not at all, respectively. Finally, parameter *pRule* represents the preference policy and determines whether positive or negative authorization is preferred in the case of remaining conflicts. (We assume that the subject and objects hierarchies (SDAG and ODAG) and the explicit access control matrix (EACM) are globally defined in the algorithm.)

In Line 1, relation *allRights* is created by calling Function *Propagate().* The details of this function are explained in the next section, but the effect is to apply the first three steps of the procedure described in the introduction to Section 3.6.

Line 2 checks whether the caller is interested in applying the default policy (*dRule* = “+” or “-“) or not (*dRule* = “0”). In the latter case, only those rows of relation *allRights* are considered in which *mode* <> “d” (see Line 3). In the former case (Line 4), those rows of relation *allRights* in which *mode*=“d” are updated with the value of *dRule* (“+” if positive authorization is to be the default policy, “-” otherwise).

In Line 5, if the majority policy should be applied before the locality policy, we count the number of positive authorizations (Line 6) and negative authorizations (Line 7) that exist in relation *allRights*; however, as stated in Line 8, if the majority policy should be applied after the locality policy, we first apply the locality on relation *allRights*, and then count the number of positive (Line 9) and negative authorizations (Line 10). In either of these cases (Line 11), the algorithm returns the authorization which is in majority (Lines 12 and 13).

If neither positive nor negative labels is in the majority or the majority policy is not designated at all, we apply the locality policy to relation *allRights* to select its relevant rows (Line 14); if *lRule* = *min()*, only rows in which the value of column *dis* is equal to the minimum distance (the most specific authorizations) are selected; similarly, if *lRule* = *max()*, only rows in which the value of column *dis* is equal to the maximum distance (the most general authorizations) are selected; however, if *lRule=identity()*, all rows are selected (this is equivalent to no locality policy being designated).

```
Algorithm I: Resolve(s, o, r, pMode, dRule, lRule, mRule, pRule)
¤ To compute the effective accessibility of subject s on object o w.r.t. right r
¤ Propagation mode pMode ∈ {"pass through", "block by", "override"}
¤ Default rule dRule ∈ {"+", "-", "0"}
¤ Locality rule lRule ∈ {max(), min(), identity()}
¤ Majority rule mRule ∈ {"before", "after", "skip"}
¤ Preferred rule pRule ∈ {"+", "-"}
¤ The subject and object hierarchies (SDAG and ODAG), and the explicit access control
    matrix (EACM) are globally defined.
Output: either "+" or "-"
1. allRights ← Propagate (s, o, r, pMode, SDAG, ODAG, EACM);
2. if dRule = "0"
3.         $allRights \leftarrow \sigma_{mode<>\text{"d"}} allRights$
4. else update allRights set mode=dRule
           where mode="d";
5. if mRule = "before"
6.         $c_1 \leftarrow \Pi_{count()}\left(\sigma_{mode=\text{"+"}} allRights\right)$;
7.         $c_2 \leftarrow \Pi_{count()}\left(\sigma_{mode=\text{"−"}} allRights\right)$;
8. if mRule = "after"
9.         $c_1 \leftarrow \Pi_{count()}\left(\sigma_{mode=\text{"+"},\ dis=lRule(dis)} allRights\right)$;
10.        $c_2 \leftarrow \Pi_{count()}\left(\sigma_{mode=\text{"−"},\ dis=lRule(dis)} allRights\right)$;
11. if mRule <> "skip"
12.        if $c_1 > c_2$ return "+";
13.        if $c_2 > c_1$ return "-";
14. $Auth \leftarrow \Pi_{mode}\left(\sigma_{dis=lRule(dis)} allRights\right)$;
15. if count(distinct Auth) = 1
16.        return Auth;
17. return pRule;
```

Figure 8. Algorithm *Resolve().*

Next, the values of column *mode* of corresponding rows are projected to form a set called *Auth*, which may be empty or contain positive and negative authorizations. If only one type of authorization is present (Line 15), it is returned (Line 16); otherwise, the preferred authorization (*pRule*) is returned and the algorithm ends.

Table 3. Trace of *Resolve()*.

Strategy	c_1	c_2	*Auth*	*mode*	Line
$D^+L\mathbf{M}P^+$	2	1	*n/a*	+	6
$D^-GM\mathbf{P}^-$	1	1	+,-	-	9
$D^-\mathbf{M}P^-$	2	4	*n/a*	-	6
$D^-L\mathbf{P}^+$	*n/a*	*n/a*	-,+	+	9
$D^+\mathbf{G}P^-$	*n/a*	*n/a*	+	+	8
$\mathbf{G}MP^-$	1	0	*n/a*	+	6
$\mathbf{P}^-$	*n/a*	*n/a*	-,+	-	9
$\mathbf{M}GP^-$	1	0	*n/a*	+	6

Table 3 illustrates the result of Algorithm *Resolve()* applied to our motivating example for several illustrative strategies. In particular, we trace the algorithm for eight strategy instances (selected from Table 2) namely D^+LMP^+, D^-GMP^-, D^-MP^-, D^-LP^+, D^+GP^-, GMP^-, P^-, and MGP^-. Table 3 shows values of c_1, c_2, *Auth*, and the effective mode derived by the algorithm, as well as its corresponding return line number. In the table, *n/a* means that the algorithm does not use the corresponding variable for the conflict resolution.

For instance, if one chooses the strategy instance D^-GMP^-, all default values of relation *allRights* are replaced with "-" (Line 3). Since the global mode of the locality policy is in place and there are one positive and one negative authorization from distance 3 in Table 1, both c_1 and c_2's values are assigned the value 1 (Lines 5). Then, since neither positive nor negative is in majority, the algorithm continues to Line 7, and *Auth* is assigned the value {+,-}. Finally, since there is a conflict in *Auth*, Line 9 of the algorithm returns the value of preference policy, which is "-" (indicated by P^- in the strategy instance), as the final derived decision with respect to triple <*User*, *obj*, *read*>.

As another example, if one chooses strategy instance MGP$^-$, in Line 2, only those rows of relation *allRights* in which the mode is not "d" are selected. Then, since the globalization policy is in place and there is one explicit positive authorization from distance 3 in relation *allRights*, the value of c_1 is set to 1 and the value of c_2 is set to 0 in Lines 4. Finally, the algorithm returns "+" from Line 6 as the final derived decision with respect to triple <*User*, *obj*, *read*>.

3.6.2 Function ***Propagate()***

In this section, we explain the details of Function *Propagate()*, which returns all corresponding authorizations of a given subject, object, and authorization, shown as <*s*, *o*, *r*> when called from Line 1 in Algorithm *Resolve()*. The idea is to extract the part of the subject hierarchy in which *s* is the only sink and the part of the object hierarchy in which *o* is the only sink. (Note that, to avoid possible ambiguities, propagation modes are applied to the subject hierarchy only, and in the object hierarchy pass through is always applied.) Then, using top-down breadth-first propagation, all authorizations from root nodes are propagated towards *s*. If a root node has no authorization assigned in the explicit matrix, a letter "d" is assigned to it to represent the default authorization. Moreover, the distance of each authorization to *s* is computed, so that it can be exploited by Algorithm *Resolve()* if the locality policy is applied. Note that authorizations are propagated from *all paths* starting from the source node and ending at the destination.

Figure 9 illustrates Function *Propagate()*, which inputs parameters *s*, *o*, and *r* (representing the subject, object, and authorization on which the conflict should be resolved), as well as *pMode* (representing the propagation mode); also, the function inputs *SDAG, ODAG,* and *EACM*, which represent the subject and object hierarchies as well as the explicit access control matrix, respectively.

In Line 1, we extract from *SDAG* the maximal connected sub-hierarchy *SDAG'*, in which *s* is the sole sink. Similarly, the maximal connected sub-hierarchy *ODAG'*, in which *o* is the sole sink, is extracted.

In Line 2, we create a relation *P*, which consists of five columns namely, *subject*, *object*, *right*, *dis*, and *mode*. Values for columns *subject*, *object*, *right*, and *mode* are taken from the corresponding ones in relation *EACM* (as explained in Section 3.3). Column *dis* represents the distance of the explicit authorization from the subject. Thus, the *dis* value for explicit authorizations is 0, for an authorization inherited from a parent is 1, and for an authorization inherited from a grandparent is 2.

```
Function Propagate (s, o, r, pMode, SDAG, ODAG, EACM);
¤ To obtain all authorizations, with respect to triple <s, o, r> by propagating explicit
  authorizations in EACM through subject and object hierarchies (SDAG and ODAG)
¤ EACM has attributes <subject, object, right, mode>
¤ SDAG has attributes <subject, child> , ODAG has attributes <object, child>
¤ $ancestors(s) = \{s\} \cup \{x | \exists y <y,s> \in SDAG \wedge x \in ancestors(y)\}$
¤ $ancestors(\mathrm{o}) = \{\mathrm{o}\} \cup \{x | \exists y <y,o> \in \mathrm{O}DAG \wedge x \in ancestors(y)\}$
Output: table allRights
1. $SDAG' \leftarrow \sigma_{subject \in ancestors(s), child \in ancestors(s)} SDAG$; $ODAG' \leftarrow \sigma_{object \in ancestors(o), child \in ancestors(o)} ODAG$;
2. $P \leftarrow \Pi_{subject, o, right, depth(object, o), mode} (SDAG' \bowtie \sigma_{right=r, object \in ancestor(o)} EACM)$;
3. $RootSubjects \leftarrow \Pi_{subject} SDAG' - \Pi_{child} SDAG' - \Pi_{subject} P$;
4. $RootObjects \leftarrow \Pi_{object} ODAG' - \Pi_{child} ODAG' - \Pi_{object} P$;
5. $P \leftarrow P \cup RootSubjects \times \{<o, r, 0, \text{“d”}>\}$;
6. $P \leftarrow P \cup \Pi_{subject, object, right, depth(object, o), mode} s \times RootObjects \times \{< r, \text{“d”}>\}$;
7. $P' \leftarrow \sigma_{subject \neq s} P$;
   repeat
8.        $P' \leftarrow \Pi_{child, object, right, i+1, mode} P' \bowtie SDAG'$;
9.        $P'' \leftarrow P \bowtie_{P.subject = P'.subject, P.object = P'.object, P.right = P'.right, P.mode <> P'.mode, P.i = 0} P'$;
10.       If $pMode$= “block by” then $P' \leftarrow P' - \Pi_{P.subject, P.object, P.right, P'.i, P'.mode} P''$;
11.       If $pMode$= “override” then $P \leftarrow P - \Pi_{P.subject, P.object, P.right, P.i, P.mode} P''$;
12.       $P \leftarrow P \cup P'$;
13.       $P' \leftarrow \sigma_{subject \neq s} P'$;
14.       until $P' = \varnothing$ ;
15. return $\sigma_{subject=s} P$;
```

Figure 9. Function *Propagate()*.

Before completing the Function *Propagate()*, relation *P* will record all relevant authorizations propagated from all subjects and objects in the sub-graphs to all other nodes.

In Line 3, we store all unlabelled root subjects of *SDAG'* into a relation called *RootSubjects*. For instance, *RootSubjects* contains $\{S_1, S_6\}$ if applied to our motivating example. Similarly, in Line 4, we store all unlabelled root objects of *ODAG'* into a relation called *RootObjects*.

In Line 5, for each root subject with no explicit authorization with respect to *r* and ancestors of *o*, we insert an additional row into relation *P* to assign it the default authorization with distance 0. Similarly, in Line 6, for each root object with no explicit authorization with respect to *r* and ancestors of *s*, we insert an additional row into relation *P* to assign *s* the default authorization with appropriate distance (which is the distance of the root object from the object). In Line 7, we select as *P'* all identified authorizations other than those on the sink node *s*.

In Lines 8 to 15, we iteratively propagate all of the newly identified authorizations to all of the children of the corresponding nodes, stopping when no more nodes exist in *P'*. This involves copying the authorizations from each node to its children (with the increased distance) (Line 8), blocking or overriding the rows based on the propagation mode (Line 9 to 11), inserting the new authorizations into P (Line 12), and re-determining which authorizations still need to be propagated further (Line 13). Finally, Line 15 selects and returns authorizations that correspond to subject *s*.

3.6.3 Computational Analysis

The performance of the *Resolve()* algorithm depends on the structure of the subject hierarchy, on the placement of the explicit authorizations in the explicit access control matrix, and on the choice of subject, object, and right. We will examine the performance in practice in the next section, but here we summarize its asymptotic behaviour in the worst case.

Consider first the structure of the subject hierarchy as represented by *SDAG*. Let ***r*** be the number of roots of the graph and let ***n*** be the total number of subjects in the hierarchy. We assume that at most one authorization is explicitly given for every subject-object-right triple; duplicates are meaningless and contradicting authorizations can be assumed to be disallowed. Thus, when selected subjects from *SDAG* are matched with explicit authorizations for a given object and right (Line 2 of Function *Propagate()*), at most one explicit authorization is joined to each subject in the subject hierarchy. Let

p be the number of paths to the given subject of interest *s* from subjects assigned explicit authorizations for the given object-right pair. Finally, let ***d*** be the sum of the path lengths for *all* paths leading from a root or an explicitly authorized subject to *s*.

Algorithm *Resolve()* first calls Function *Propagate()*. Lines 1 through 7 take time $O(\boldsymbol{n})$ to select a subset of the subjects, attach the explicit authorizations, and set the defaults in the remaining roots. Each authorization (default or explicit) is then pushed down each path to the node representing the given subject. The loop from Lines 8 though 14 of Function *Propagate()* thus require $O(\boldsymbol{d})$ time in total. Finally relation *P* contains all these propagated authorizations, but only those associated with the given subject s are returned; this returned relation includes exactly one tuple for each explicit authorization and at most one tuple for each root. In summary, Function *Propagate()* takes time $O(\boldsymbol{n}+\boldsymbol{d})$ and returns a structure of size $O(\boldsymbol{p})$.

The remainder of Algorithm *Resolve()* repeatedly examines subsets of the relation *allRights*, and thus each line requires time at most $O(\boldsymbol{p})$. Thus the total time for executing Algorithm *Resolve()* is $O(\boldsymbol{p}+\boldsymbol{n}+\boldsymbol{d})$. Unfortunately, since the number of paths in a directed acyclic graph can grow exponentially in the number of nodes in the graph, ***d*** is $O(\boldsymbol{n}2^{\boldsymbol{n}})$ in the worst case (***p*** is $O(2^n)$). We shall see that in practice, however, the algorithm is typically much better behaved, as the authorization rate is often significantly low and also data hierarchies seldom contain the repeated diamond patterns that cause the number of paths to explode.

3.7 Experiments

We tested our algorithm first on synthetic data. We constructed several random complete directed acyclic graphs. In particular, KDAG(n) includes n nodes, one of which is a root and one of which is a sink, and $\binom{n}{2}$ edges (i.e., an edge between every pair of nodes), directed in such a way as to prevent cycles. Thus such graphs contain many more edges and paths than would be expected in typical applications, and constitute good stress tests for our algorithm.

We executed our algorithm on random KDAGs of three different sizes. For each graph, we assigned explicit authorizations to subjects at random, choosing subjects proportionally to the number of members. In particular, 0.5% to 10.0% of the graph's *edges* were selected at random and their source nodes were assigned explicit authorizations. We ran our experiments on a Sun UltraSPARC-II with a 450 MHz processor and 2048 Megabytes of RAM. The program is in C and SQL and was complied by gcc version 3.3.4 and DB2 version 7. We then measured the CPU time for computing the result of the Function *Propagate()* (that being the dominant part of the algorithm) for each of the resulting SDAG-EACM pairs, and averaged over 20 random repetitions with the same parameters. Our experiments show that for small authorization rates (which often occur in practical cases), the running time is linearly proportional to the authorization rates (see Figure 10).

We also evaluated our algorithm on the subject hierarchy extracted from an installation of Livelink, Open Text's enterprise content management system[1]. In Livelink, groups can be arbitrarily structured and nested to arbitrary depth. In the environment we tested, the subject hierarchy has over 8000 nodes and 22,000 edges. There are 1582 sinks (individual users), each of which represents a real-world sample for our experiments. The depths of the induced sub-graphs range from 1 to 11.

We measured the time of our algorithm for each of the sinks in the Livelink subject hierarchy, using an authorization rate of 0.7% of the edges as above. The results are presented in Figure 11, plotting the CPU time as a function of ***d***, the sum of the lengths of all paths from explicit and default authorizations to the selected sink.

Figure 11 also compares the execution time of the *Resolve()* algorithm to that of the *Dominance()* algorithm, presented in Chinaei and Zhang's work [2]. The latter algorithm was designed to evaluate the D^-LP^- strategy as efficiently as possible under the assumption that there are relatively few explicit positive and negative authorizations (i.e., that the authorization rate is low). Thus the comparison sheds some light on the overhead imposed by adopting a unified conflict resolution algorithm. It is important to note that the propagation of *Dominance()* algorithm is dependent on the placement of negative authorizations whereas the *Resolve()* algorithm is not. To account for this, we calculated the average of three trials for each data point for the *Dominance()* algorithm: one where 1% of the explicit authorizations are negative, one where half of them are negative, and one where all explicit authorizations are negative.

The *Dominance()* algorithm is occasionally very fast due to visiting an early negative authorization in the hierarchy, but it is not as efficient as *Resolve()* for objects that have few negative authorizations. Figure 11 shows that the run time for the *Dominance()* algorithm can fall anywhere

[1] http://www.opentext.com/

below the time for the *Resolve()* algorithm, and occasionally it can be higher. On average, over all graph sizes and shapes in these experiments, *Resolve()* required 1260 ms to compute whether or not a leaf subject was authorized to access an object, whereas the *Dominance()* average is 920 ms. Thus the flexibility to compute the value for any strategy comes at a cost of 27% overhead.

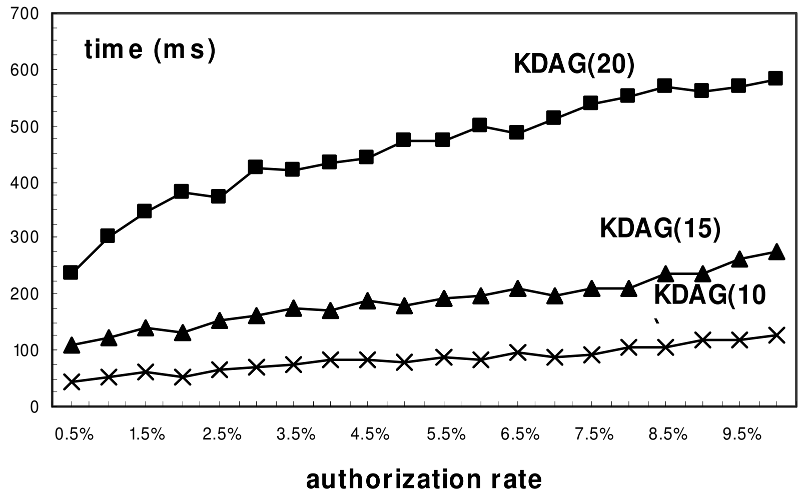


Figure 10. Function *Propagate()* on synthetic data examples.

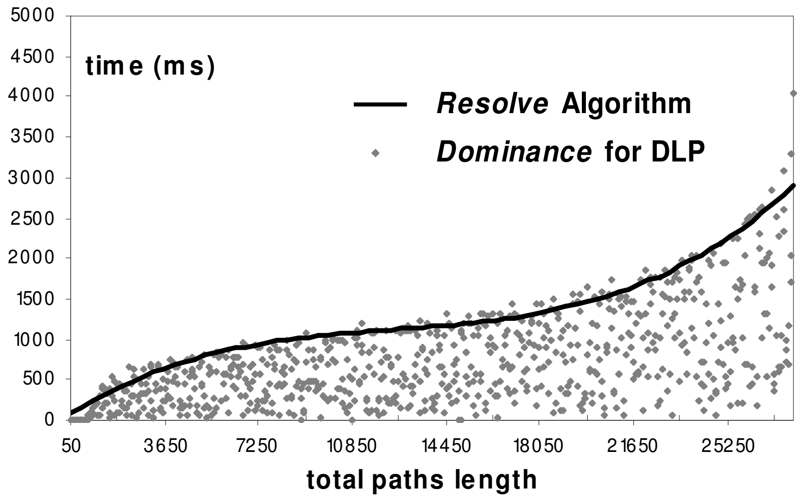


Figure 11. Algorithms *Resolve()* and *Dominance()* on *Livelink* data.

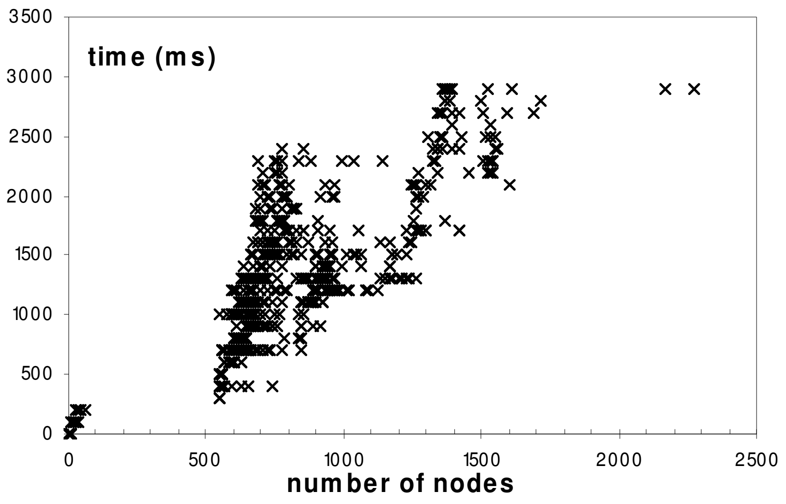


Figure 12. Total paths lengths vs. number of nodes in *LiveLink* data.

Finally, Figure 12 restates the behaviour of Algorithm *Resolve()* against the number of nodes in the sub-graph rather than the total length of all paths in the sub-graph. The results show that graphs with very many subjects do not necessarily require much more time to resolve than do small graphs. From this data we conclude that it is unlikely that the asymptotic worst case performance will be problematic in practice.

3.8 Propagation of Authorizations

For the of the remainder of the thesis, we assume that ACAD uses a closed system, in which permissions are denied by default, and provides positive explicit authorizations only. We also use "stoppers" on arbitrary nodes to limit the propagation of positive authorizations: a positive authorization may not be propagated across a node that has a corresponding stopper. Thus, the propagation of explicit (positive) authorizations is controlled with "stoppers": associating a stopper for authorization p with any subject in the subject hierarchy and any object in the object hierarchy prevents the propagation of p as an effective authorization to sub-levels in the subject hierarchy. Whereas some other researchers have argued that more general negative authorizations are often required in practice [Bertino et al. 1999], we maintain that controlling the propagation of positive authorizations with “stoppers” can produce the same effect, without incurring the cognitive overhead inherent in conflict resolution [Rosenthal and Sciore 2001]. In fact, our use of stoppers corresponds exactly to the strategy instance D^-LP^+ with block by propagation, which intuitively means a given node is effectively labelled positive only if there is at least one path to the node from at least one of its ancestors that assigned an explicit permission with no corresponding negative authorization nodes along the path. Note, however, that negative authorizations are fundamentally required in the effective matrix, where each subject-object-method triple is explicitly granted or denied.

3.8.1 Propagation and Stoppers

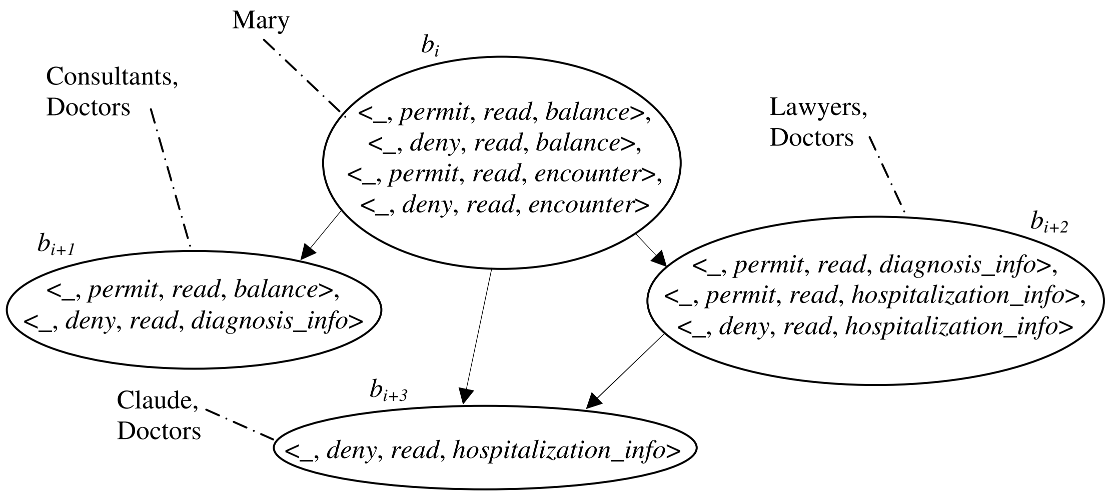
Propagation of permissions may be stopped by a *stopper* authorization at any level in the subject hierarchy. Using wildcard “_” to indicate a “whatever” value, each permission *p* which is represented as <_, *permit*, *p*, _> has a corresponding stopper that is represented in ACAD as <_, *deny*, *p*, _>. For example, in Figure 13(a), a subject who is assigned to an access bank that includes stopper <_, *deny*, *read*, *diagnosis_info*> cannot read *diagnosis_info* or its descendants merely because he has read permission on *encounter*. (Access banks are a collection of access authorizations and are formally defined in Section 4.2.) That is, the propagation of the read permission is stopped at *diagnosis_info*. However, stoppers do not override an explicit permission. For example, in Figure 3(b), a subject who has both *read* permission and *read* stopper on *encounter* can still read this object.

Stoppers are unlike traditional negative authorizations in models that include both positive and negative authorizations, such as proposed by Bertino et al. [Bertino et al. 1999]. A negative authorization typically represents a denial of access, whereas a stopper is a mechanism to stop the propagation of a positive authorization. In ACAD, if any authorization can be found connecting a subject to an object, the corresponding permissions are valid. Because there is no need to check whether an authorization is overridden elsewhere, the ACAD approach is easier to administer. We will therefore use this mechanism in the remainder of the thesis.

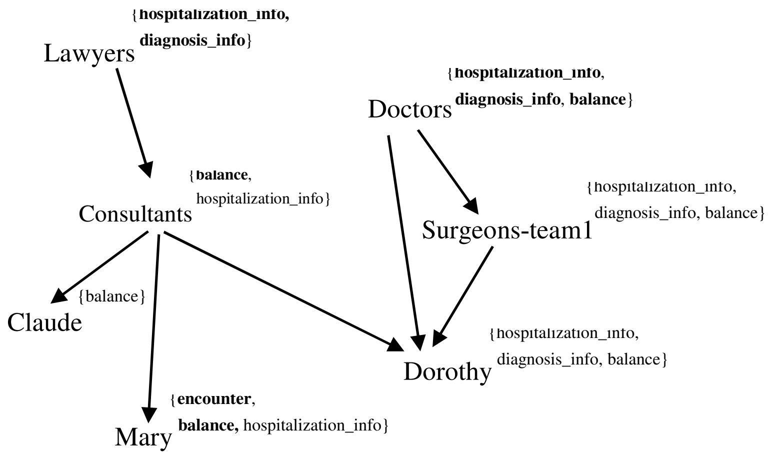
3.8.2 Propagation Example with Stoppers

This section provides an enhanced example of propagation in the context of motivating example I, explained in Section 1.3.1. Figure 13 illustrates the example, in which for simplicity only read operations are depicted. Bank b_i includes authorizations <_, *permit*, *read*, *balance*>, <_, *deny*, *read*, *balance*>, <_, *permit*, *read*, *encounter*>, and <_, *deny*, *read*, *encounter*>. Note that b_i also includes the explicit authorizations on sub-objects of *encounter, hospitalization-info* and *diagnosis_info,* which are not depicted in the figure for simplicity. Bank b_{i+1} includes authorizations <_, *permit*, *read*, *balance*> and <_, deny, *read*, *diagnosis_info*>. Bank b_{i+2} includes authorizations <_, *permit*, *read*, *diagnosis_info*>, <_, *permit*, *read*, *hospitalization_info*>, and <_, *deny*,*read*,*hospitalization_info*>; and, b_{i+3} includes authorization <_,*deny*,*read*, *hospitalization_info*>.

Now assume that subjects *Mary*, *Consultants*, *Lawyers*, and *Claude* are assigned to banks b_i, b_{i+1}, b_{i+2}, and b_{i+3}, respectively; and subject *Doctors* is assigned to banks b_{i+1}, b_{i+2}, and b_{i+3}. Figure 13(a) illustrates the bank hierarchy with these subjects assigned. Because Dorothy is a member of *Doctors* and *Consultants*, and hence *Lawyers*, her effective permissions include reading objects *hospitalization_info*, *diagnosis_info*, and *balance*. Even if *Surgeons-team1* were stopped from inheriting a permission from *Doctors*, Dorothy would still be permitted that operation because of her direct membership in *Doctors*. Mary's explicit permissions include reading objects *encounter* and *balance*, and because of the stopper for *Consultants*, she cannot inherit permission to read *diagnosis_info* from *Lawyers*, although she does inherit the ability to read *hospitalization_info*.

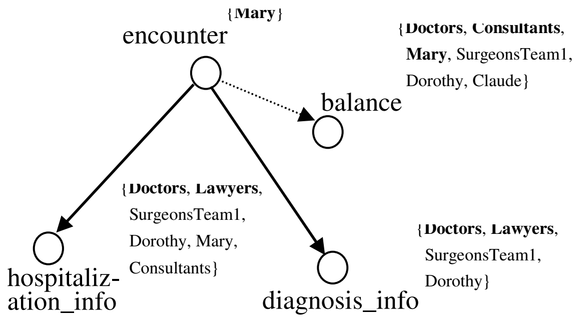

Mary
b_i
<_, permit, read, balance>,
<_, deny, read, balance>,
<_, permit, read, encounter>,
<_, deny, read, encounter>
Consultants, Doctors
Lawyers, Doctors
b_{i+1}
<_, permit, read, balance>,
<_, deny, read, diagnosis_info>
b_{i+2}
<_, permit, read, diagnosis_info>,
<_, permit, read, hospitalization_info>,
<_, deny, read, hospitalization_info>
Claude, Doctors
b_{i+3}
<_, deny, read, hospitalization_info>

(a) Example of the access bank hierarchy with assigned subjects.



Lawyers
{hospitalization_info, diagnosis_info}
Doctors
{hospitalization_info, diagnosis_info, balance}
Consultants
{balance, hospitalization_info}
Surgeons-team1
{hospitalization_info, diagnosis_info, balance}
Claude
{balance}
Dorothy
{hospitalization_info, diagnosis_info, balance}
Mary
{encounter, balance, hospitalization_info}

(b) Subject hierarchy and effective read authorizations to objects.



encounter
{Mary}
balance
{Doctors, Consultants, Mary, SurgeonsTeam1, Dorothy, Claude}
hospitaliz-ation_info
{Doctors, Lawyers, SurgeonsTeam1, Dorothy, Mary, Consultants}
diagnosis_info
{Doctors, Lawyers, SurgeonsTeam1, Dorothy}

(c) Object hierarchy and the subjects that effectively can read it.

Figure 13. Example of propagation of authorizations.

Finally, Claude's effective permissions include reading object *balance* only: although he is a *Lawyer*, he is explicitly stopped from inheriting permission to read *hospitalization_info*.

Figure 13(b) summarizes these and other effective permissions of subjects, using bold type to depict explicit permissions. For instance, subject *Consultants* is explicitly permitted to read object *balance*, and implicitly permitted to read object *hospitalization_info*. Similarly, Figure 13(c) summarizes effective permissions for objects. For instance, object *balance* is explicitly readable by subjects *Doctors*, *Consultants*, and Mary and implicitly readable by *Surgeons-team1*, Dorothy, and Claude.

3.9 Related Work

Bertino et al. propose an authorization mechanism for relational models in which conflicts are mainly resolved based on "the negative authorization takes precedence" policy [Bertino et al. 1999]. They also introduce the concept of weak and strong authorizations, which is equivalent to using our combined strategy instance $D^{-}LP^{-}$ when the propagation mode is block by.

Jajodia et al. use Datalog programs to model access controls of hybrid authorizations with a wide range of conflict prevention/resolving policies [Jajodia et al. 2001]. Their modeling stores the raw authorizations and computes the effective authorizations for a <*subject, object*> pair in time linear to the size of the Datalog program (rules and ground facts). However, their ground facts include the transitive closure of the subject hierarchy (which cannot be computed in linear time) plus all the raw authorizations. The potentially large number of ground facts implies that even a linear time solution may not be efficient in practice. To answer access control queries efficiently, they suggest materializing the entire effective access control. The accessibility check for a given <*subject, object*> pair is thus equivalent to checking the materialized effective access control table (constant time). However, considering the formidable size of the effective access controls, which is the product of the number of objects and the number of subjects, this approach is not practical for very large systems. Moreover, the materialized effective access controls are not self-maintainable with respect to updating the explicit authorizations, and even a slight update to the explicit authorizations could trigger a drastic modification to the effective ones, making the maintenance task very expensive.

Propagation of authorizations has been addressed in many works [Ferraiolo et al. 1992; Bertino et al. 1999; Osborn and Guo 2000]. However, traditional role-based models often use only the role hierarchy for authorizations inheritance. Bertino et al. propose the propagation of authorizations within the subject hierarchy in their role-based proposal for relational access control; they assumpe that there is no hierarchy among the objects, which are mostly base tables. Osborn and Guo augment RBAC models by suggesting to support a group hierarchy independent from the role hierarchy as well as authorization inheritance within group members. ACAD is distinguished from other models due to propagating authorizations within the subject and object graphs simultaneously, and utilizing the bank hierarchy as a means of retaining control for object owners (see Chapter 4).

Some existing solutions for computing effective authorizations assume that the explicit authorizations are propagated on tree-structured data [Damiani et al. 2002; Moses 2005; Yu et al

2002; Zhang et al. 2005]. This trivializes conflict resolution since there is only one path between any ancestor and a leaf. Moreover, the number of ancestors for a leaf is bounded by the depth of the tree, which is usually a small value in real world data [Mignet et al. 2003]. Unfortunately, real world subject hierarchies are mostly DAG-structured rather than trees: the UNIX file system allows a user to be member of several groups at the same time, and in role-based access control systems, a user can be assigned several roles and each role can be assigned to multiple parent roles [Ferraiolo and Kuhn 1992]. When explicit authorizations are propagated on a DAG subject hierarchy, a leaf subject potentially has all subjects as its ancestors, and each ancestor may have several paths reaching to that leaf. Therefore, none of the approaches for tree-structured data are appropriate in this setting.

Cuppens et al. propose a conflict resolution model for documents containing sensitive information [Cuppens et al. 1998]. They address the problem of downgrading the classification of these documents when their contents become obsolete. Their approach is to impose a strict order of preference between rules and does not include any hierarchy among subjects.

Koch et al. provide a systematic graph-based conflict detection and resolution algorithm based on two properties namely, *rule reduction* and *rule expansion* [Koch et al. 2002]. Using these properties, they transform a conflicting graph into a conflict-free one. However, their approach is applicable only to the rules that are related to one another, whereas our approach addresses independent policies.

Finally, our approach is also different from the combining algorithms in XACML [Moses 2005], in which the resolution model relies on the data hierarchy rather than the also considering subject hierarchy.

Chapter 4

FLEXIBLE ADMINISTRATION OF ACCESS CONTROL

The spectrum of access control administration was introduced in Chapter 1 to illustrate to what extent decentralization can be realized. This spectrum covers a wide range of systems from absolutely *autocratic* to completely *self-governing*. This chapter presents the **ac**cess control **ad**ministration (ACAD) model. The goal is to be a policy-neutral and uniform model in which various degrees of decentralization are supported. It is important to notice that whereas there must be a consistent enforcement mechanism to support the enterprises' access control policies, closely held access control may or may not fit the security requirements of various organizations. Furthermore, no matter how decentralized the access control administration is, the system must remain secure: its policy must be enforced entirely regardless of how restrictive or open the policy is.

ACAD consists of three layers; each is a directed acyclic graph corresponding to subjects, objects, and a layer of access banks. The subject and object were formally defined in Definitions 1 and 2, respectively. The access banks are defined in this chapter (Definition 9). Section 4.1 describes why a uniform access control mechanism is important. Section 4.2 formalizes authorizations of ACAD. Section 4.3 defines applicable authorizations for an object based on the object type. Section 4.4 introduces *omnibank* as a means of empowering an object's owner with all applicable authorizations. Section 4.5 illustrates how subjects can manage access control through various types of delegation and revocation features in light of customized access banks. Section 4.6 illustrates how omnibanks can include several customized omnibanks. Section 4.7 compares access banks to access roles introduced in RBAC models. Finally, Section 4.8 addresses some aspects of the related work.

4.1 Uniformity

It is important as an overarching feature that access control systems be based on a single uniform model with respect to both data and metadata (access control data). The goal is to address the needs of software companies to provide Enterprise Content Management capabilities[2] that fit a wide spectrum of security needs for their diverse customer base. Some data needs very tight control; other data can be managed by their creators or by the groups within which the creators work. To support collaboration, access privileges often have to be granted and revoked in unstructured ways. In particular, since the access privileges themselves are typically stored as data, operations to alter those privileges (called metadata), whether to update them or to assign privileges with respect to newly created subjects or objects, must also be subject to access control.

Some existing models include separate implicit access control methods for controlling authorizations to update the metadata, and other models ignore such authorizations. Yet, the broad view of content in an ECM, as depicted in the access control framework of Figure 1, supports the idea that information about the subjects and information about access control can (and should) be treated just like any other data. Therefore it is important that there is one uniform mechanism to manage data whatever its form, and we take that as axiomatic.

4.2 Access Banks

Definition 9 *(Access Banks).* Access banks, which are themselves a subset of the data universe U, are collections of access authorizations. Henceforth, we always mean an *access bank of authorizations* when we use the term "bank" or "access bank", and access banks are depicted by ovals with italic labels in all figures. Banks connect a set of subjects to the authorizations for each subject in the set. A given bank $b \in U$ is a triple $<b_n, Subj, Auth>$ where

—b_n is a unique bank name,

—*Subj* is a set of subjects,

—*Auth* is a non-empty set of access authorizations, and

[2] defined by the Association for Information and Image Management <http://www.aiim.org/about-ecm.asp>

— $\forall$<*C, mode, p, O*> $\in$ *Auth*, *Subj* $\subseteq$ *C*. That is, all subjects associated with a bank must meet all the constraints specified in the bank. Through bank b = <b_n, *Subj, Auth*>, any subject that is a member of *Subj* receives all authorizations of *Auth*.

Definition 10 *(Fertile Bank).* A fertile bank is an access bank in which there exists at least one access authorization which permits the corresponding subjects to create new access banks.

We assume that when an authorization is to be enforced, subjects in the system have already been authenticated, that is identified and associated with all banks in which the subject is in *Subj*. The method of authentication is outside the scope of this thesis; interested readers should consult alternative sources [Burrows et al. 1990; Ellison 1996; Maughan et al. 1998]. Thus, whenever subjects request access to any piece of data (or metadata), the enforcement mechanism is activated to check if their access bank list includes a bank that contains the corresponding permission.

4.2.1 Access Banks Hierarchy

Banks of authorizations are organized as hierarchies in which source banks export authorizations to destination banks. Furthermore, the constraint of an authorization in a destination bank must be at least as restrictive as the constraint of the corresponding authorization in the source bank.

Definition 11 (*Bank Hierarchy*). The bank hierarchy is a directed acyclic graph <*V*, *E*>, where *V* is the set of access banks, and *E* is a set of edges $v_1 \rightarrow v_2$ where $\forall$<v_2, $Subj_2$, $Auth_2$>$\in V$, $\forall$<C_2, *mode*, *p*, o_2> $\in Auth_2$ ($\neg\exists$v ($v \rightarrow v_2 \in E$) $\vee$ ($\exists$ <v_1, $Subj_1$, $Auth_1$>$\in V$, $\exists$<C_1, *mode*, *p*, o_1> $\in Auth_1$ ($v_1 \rightarrow v_2 \wedge C_2 \subseteq C_1 \wedge o_2 \subseteq o_1$)).

Thus, for every non-root bank in the hierarchy, its authorizations must all be derived from those of its parents, possibly with tighter constraints or fewer objects. For example, in Figure 14, the licensing hierarchy between access banks (indicated by arrows) represents delegated privileges, which are imported from source banks. If b_{i+3} imports <C_3*, mode, p,* O_3> from <C_2*, mode, p,* O_2> in b_{i+2}, then C_3 and O_3 must be subsets of C_2 and O_2, respectively. Note, however, that the restriction that all subjects assigned to a bank must satisfy all constraints in that bank does *not* imply that the subjects assigned to a bank must be a subset of the union of subjects assigned to its parents.

Recall the ACAD model also includes two other layers: the subject hierarchy (SDAG) and the object hierarchy (ODAG) described in Section 3.1 and depicted in Figure 3. A typical bank hierarchy

(BDAG) is depicted in Figure 15. For readability of Figure 15 an abbreviated language is used: for instance, M_3(d2) corresponds to authorization <_,permit,M_3,{d2}>; as another example, M_1(d1, d2)+ corresponds to authorizations <_,permit, M_1, {d1, d2}> and <_,permit, DelegateM_1, {d1, d2}>. Therefore, in Figure 15, bank b_3 contains a privilege to run method M_1 on document d_1 with grant option (+). BDAG is an intermediate layer between subjects and objects; on one hand, a set of subjects (either individual users or groups) from SDAG is assigned to each node of BDAG (indicated by dot-dashed lines). On the other hand, each access bank contains a set of authorizations on a set of objects of ODAG (indicated by dashed lines). We note that objects can be of any type (with arbitrary associated methods), including documents or document fragments, groups in the subject hierarchy, or even access banks (since an access bank is itself a special object containing a set of privileges on some set of objects).

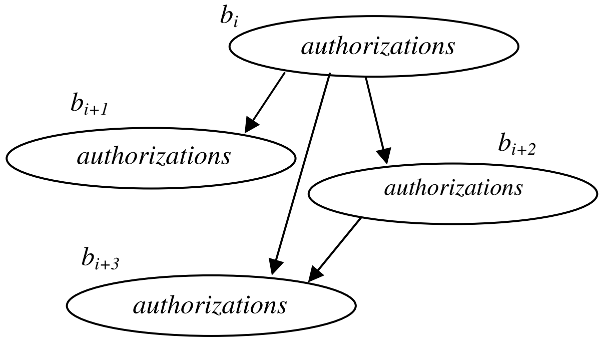


Figure 14. Hierarchy of access banks maps authorization inheritance.

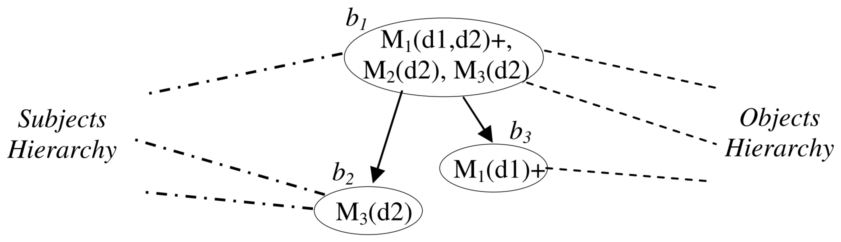


Figure 15. Access banks as an intermediate layer between SDAG and ODAG.

4.2.2 Bank Operations

We predefine six bank operations, namely *read*, *assignTo*, *removeFrom*, *moreConstraints*, *import*, and *deleteBank*. Operation *read* is the ACAD means of reading content of a given bank. Recall from Definition 9, a bank is represented as a triple <b_n, *Subj, Auth*> ; therefore, reading a bank discloses its name, the set of subjects assigned to it, and its set of access authorizations. Operation *assignTo* is the method for assigning a subject to a given bank. In fact, *assignTo* adds one or more subjects to the *Subj* component of a bank. Similarly, *removeFrom* is an operation to remove a subject from a given bank, which means one or more subjects are removed from the *Subj* part. Operation *moreConstraints* is the method to add more constraints to the *C* part of any access authorization in the *Auth* component of a given bank. (Recall from Definition 6 that the *C* part of an access authorization constrains the set of subjects that may hold the authorization.) Operation *import* creates a new bank as a child vertex of one or more existing source banks. It takes as parameter pairs of a subset of authorizations and the source vertex from which the authorizations are imported to the destination vertex. For example, *import*(b_3, *{<<_, permit, M,{d1}>,* b_1 *>, <<_, permit, M,{d2}>,* b_2 *>})* creates bank b_3 and edges $b_1 \rightarrow b_3$ and $b_2 \rightarrow b_3$ as depicted in Figure 16(c). Finally, operation *deleteBank* is the method to delete a bank from the bank hierarchy.

4.2.3 Banks and the Explicit Matrix

Access banks in ACAD are used to assign a set of subjects to a set of access authorizations. Furthermore, access authorizations associate access rights to a set of objects. In fact, an access bank can be briefly represented as <S, {<r_i, O_i>}> in which *S* and O_i are given sets of subjects and objects, respectively, and r_i is an access right. The following predicates can be used to formalize these concepts using Datalog: *bank(b,s,a)* associates subject *s* to authorization *a* in the bank with name *b*, and *auth(a,c,m,p,o)* associates constraint *c*, mode *m*, permission *p*, and object o in the authorization with name *a*. Although Definition 9 is defined in terms of sets of subjects and sets of authorizations, these predicates deal with individual elements to simplify the Datalog expressions. Hence, an access bank serves as several cells (predicates *permit* and *deny*) of an explicit access control matrix. The following rules demonstrate this service.

permit(s, r, o) $\leftarrow$ *bank(_, s, a), auth(a, _,'permit', r, o).*

deny(s, r, o) $\leftarrow$ *bank(_, s, a), auth(a, _,'deny', r, o).*

in which "_" means a "whatever" value. Note that this formalism assumes a unique authorization name for access authorizations; authorization names serve no purpose within the rest of the thesis. Moreover, note that predicates *permit* and *deny* are now considered to be *intensional,* built on top of *extensional* predicates *bank* and *auth*.

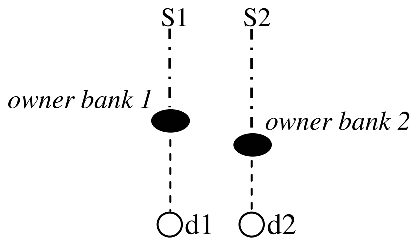
4.2.4 Classes of Banks

In practical environments, there are several characteristics that lead to useful refinements of the ACAD model, which are described here.

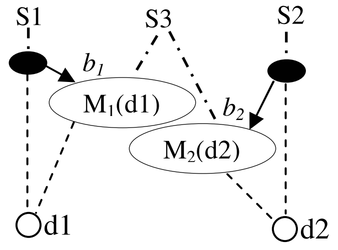
ACAD assumes that every object belongs to an owner, and thus an owner access bank is always required for each object. The owner of an object is not necessarily the object's creator, but it is initially determined at the time the object is created (see Chapter 5). As a result, as individual users create objects (e.g. personal documents), the number of access banks in a system may become quite large (e.g. if creators own the object they create), yet many of them are similar. This can be initialized parametrically. ACAD provides *generic banks* to provide such a parametric bank. As another characteristic, the model should also provide subjects with the feature of customizing access banks. ACAD provides *customized banks* to provide such features. Furthermore, banks may need to be combined to provide more sophisticated capabilities for the corresponding subjects. ACAD provides *combined banks* for this purpose. The *generic*, *customized*, and *combined banks* are depicted in Figure 16 and described in more detail as follows:

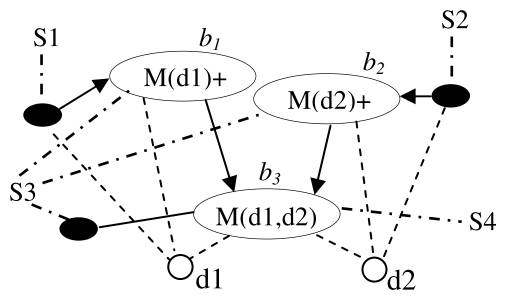
Generic bank: This facility allows the specification of a template for a set of possible access banks. To derive an effective access bank from such a template, a specific event acts as a trigger. For example, the *owner bank* (indicated by black ovals in figures) is a generic bank, which is instantiated as soon as the first object belonging to a subject is created; the owner id and object id are the parameters of the instantiation, and the concrete owner bank includes all authorizations on the object determined by a creation time policy (see Chapter 5). Similarly, *omnibanks* (introduced in Section 4.4) are generic banks that include all related authorizations on specified objects. Consider an application in which every subject who creates an object is its owner; hence, every creator is assigned to a concrete owner bank. Figure 16(a) illustrates this example, where subjects S1 and S2 are assigned to the owner banks when they create objects d1 and d2, respectively.

Customized bank: Besides the generic bank, subjects who have at least one import privilege are able to create their own customized banks. Hence, every owner can choose an arbitrary subset of the authorizations in its owner bank to create a customized bank, and then assign arbitrary subjects to this bank. More generally, any subject assigned to a fertile bank may create a new sub-bank with a subset of the authorizations and assign other subjects to that new bank. Customized banks thus form the bank hierarchy, in which owner banks are the roots. For instance, in Example 1 in Chapter 1, any physician may customize the privileges assigned to the physicians group to create a particular bank for nurses. In Figure 16(b), owners S1 and S2 have created banks b_1 and b_2 to access objects d1 and d2, respectively. They both have assigned user S3 to these banks. Therefore, S3 has access to both d1 and d2. S1 or S2 can remove S3's access at any time, independently.



(a) Owner bank as a Generic





(c) Customized bank **(d) Combined bank**

Figure 16. Various access banks in ACAD.

Combined bank: Lastly, subjects can create a combined bank if they have importable authorizations in more than one bank. In other words, a combined bank is a bank that has more than one immediate senior bank. For instance, in Example 1 in Chapter 1, an accountant may combine several banks to make several medical objects accessible to an insurance representative. In Figure 16(c), owners S1 and S2 have created fertile banks b_1 and b_2 to access objects d1 and d2, respectively; both S1 and S2 are willing to export authorizations on their documents. They have also assigned subject S3 to their banks. Therefore, S3 can create b_3 and inherit (import) access authorizations on both d1 and d2. Moreover, S3 can assign other subjects (e.g. S4) to b_3. Both S1 and S2 can remove access privileges for S3 and its dependents (e.g. S4) from their objects by severing the inheritance chain without disrupting one another. For example, S1 (the owner of b_1) can remove access for S3 and S4, from d1 simply by removing both edges S3→ b_1 and b_1→ b_3. Therefore, b_3 no longer inherits anything from b_1.

4.3 Object Types and Authorizations

Recall from Definition 4, authorizations in ACAD are not limited to a fixed set of rights such as *read* or *write*. Instead, various authorizations on objects are defined based on the object types that are present in the application. In addition, certain predefined object types and authorizations are required by the model.

The predefined types include a basic type *TData,* which includes all objects, as well as two subtypes *TBank* and *TSubject* which represent access banks and subjects (individuals and groups), respectively. Authorizations on *TData* are applicable to all objects in *U*, while the subtype authorizations are applied to objects that are banks or subjects, respectively. We predefine two operations *read* and *create* applicable to type *TData*, which therefore define corresponding operations for every type through inheritance. Thus, permission to read (or create) an object is a permission to call a method *read* (or *create*) on an object of any type. (For readability, we give authorizations the same names as their corresponding methods.)

We also define eight specific authorizations for objects of *TBank* and *TSubject* types: *assignTo, removeFrom, import*, *addConstraint*, and *deleteBank* are applicable to type *TBank* only; and *subscribeTo, unsubscribeFrom*¸ and *deleteSubject* are applicable to type *TSubject* only. The *TBank* authorizations correspond to the bank operations: For instance, a subject enjoying permission *assignTo* on bank *b* is permitted to assign subjects to the *Subj* component of *b*. For authorizations applicable to type *TSubject*, a subject enjoying permissions *subscribeTo*, *unsubscribeTo*, and *deleteSubject* on subject *S* is permitted to subscribe other subjects to (group) *S*, unsubscribe other subjects from *S*, and delete subject *S*, respectively. Hence, in contrast to most other models, any user can potentially update the group hierarchy in our model, depending on the permissions assigned to that user.

4.4 Object Creation and Omnibanks

This section introduces the omnibank, which is the aggregation of all owner banks for a particular subject. Consequently, it is the collection of all authorizations for a subject (whether a user, a group, or an application) on its own objects. The omnibank also serves as the root of the bank hierarchy associated with the owner. In fact, the omnibank is the root of the owner's whole object hierarchy.

As stated in Section 4.2.4, an owner bank is a generic bank, which is instantiated when an object is created. This section specifies how various types of authorizations are combined in an omnibank for subject S when various types of objects are created, whether by S or by other subjects, and owned by S. For simplicity, we assume that an object's owner is initially assigned all possible permissions on that object; this is relaxed in Chapter 5. Therefore,

Axiom 1. Every subject who owns at least one object has an associated omnibank.

Figure 17(a) illustrates that an omnibank initially includes twelve access authorizations: six permissions and six corresponding stoppers. Recall from Definition 6 that each access authorization consists of four components: a constraint (C_j^i), a mode (permit/deny), a permission (p_m), and a set of objects (o_n). C_j^i constrains the set of subjects S_j who may be explicitly permitted to hold (or stopped from holding) permission p_m on the set of objects o_n. In the absence of stoppers, all members of the subject set S_j, assigned to the bank, implicitly enjoy the authorization on the set of objects o_n and all their descendants. Since subjects assigned to a bank are given all the authorizations within it, they must satisfy all constraints by being a subset of the intersection of all C_j^i for all i, i.e., $S_j \subseteq C_j = \bigcap_i C_j^i$.

Using this notation, authorization <C_0^1, *permit, read,* b_0> in omnibank b_0 means that subjects assigned to the bank are permitted to invoke method *read* on object b_0 and its descendants. For example, assume an application in which initially the creator is assigned to such a bank; therefore, it means that creators can read all banks in their bank hierarchy. In applications where the creator is not necessarily the object owner, the creator is not automatically assigned to such bank and consequently the object owner retains control.

Similarly, the third and fifth authorizations mean that subjects assigned to the bank are permitted to invoke method *assignTo* (*removeFrom*) to add (remove) subjects to (from) the *Subj* component of bank b_0, and of its descendants. Owners can thus delegate and revoke all their rights to other subjects, including the rights of delegation and revocation.

Authorization <C_0^7, *permit, import(Rights)*, b_0> in omnibank b_0 means that subjects assigned to the bank are permitted to invoke method *import* to import authorizations from source bank b_0. This authorization essentially means that the importer is able to create a child bank for the source bank. Hence, as soon as a subject invokes such a permission, a customized omnibank is created and assigned to the subject. Customized omnibanks, elaborated in Section 4.6, are sub-elements of the original omnibank. In order to control permissions on child banks, the exporting subject can limit authorizations of the customized omnibank by parameter *Rights* of the *import* permission. This feature is our means of enhanced delegation to retain control for the exporter (as explained in Section 4.5).

<C_0^1, *permit, read*, b_0>,
<C_0^2, *deny, read*, b_0>,
<C_0^3, *permit, assignTo*, b_0>,
<C_0^4, *deny, assignTo*, b_0>,
<C_0^5, *permit, removeFrom*, b_0>,
<C_0^6, *deny, removeFrom*, b_0>,
<C_0^7, *permit, import(omniRights)*, b_0>,
<C_0^8, *deny, import*, b_0>,
<C_0^9, *permit, addConstraint*, b_0>,
<C_0^{10}, *deny, addConstraint*, b_0>,
<C_0^{11}, *permit, deleteBank, children*(b_0)>,
<C_0^{12}, *deny, deleteBank, children*(b_0)>,

(a) Initial predefined authorizations in omnibank b_0

<C_0^{13}, *permit, read*, S_2>,
<C_0^{14}, *deny, read*, S_2>,
<C_0^{15}, *permit, subscribeTo*, S_2>,
<C_0^{16}, *deny, subscribeTo*, S_2>,
<C_0^{17}, *permit, unsubscribeFrom*, S_2>,
<C_0^{18}, *deny, unsubscribeFrom*, S_2>,
<C_0^{19}, *permit, deleteSubject*, S_2>,
<C_0^{20}, *deny, deleteSubject*, S_2>,

(b) Additional predefined authorizations when subject S_2 is created

<C_0^{21}, *permit, read*, d_1>,
<C_0^{22}, *deny, read*, d_1>,

(c) Additional predefined authorizations when object d_1 is created

Figure 17. Omnibanks and different types of objects.

Authorization <C_0^9, *permit, addConstraint,* b_0> in omnibank b_0 means that subjects assigned to the bank are permitted to invoke method *addConstraint* to add more constraints to any authorization of bank b_0, and its descendants. In other words, object owners can update the *C* part of any authorization in their bank hierarchy.

Finally, let *children*(b_j) denotes all children of bank b_j. Authorization <C_0^{11}, *permit, deleteBank, children*(b_0)> in omnibank b_0 means that subjects assigned to the bank are permitted to invoke method *deleteBank* to delete any children of b_0 (but not b_0 itself). For simplicity, we assume that only leaf banks can be deleted. Therefore, with this initialization, another right that object owners initially enjoy is the right of deleting leaf banks from their bank hierarchy.

Figure 17(a) also illustrates six initial stopper authorizations in omnibank b_0 to stop the propagation of *read*, *assignTo*, *removeFrom*, *import*, *addConstraint*, and *deletBank* permissions for the subject assigned to bank b_0. Although stoppers are not functioning in omnibanks due to coexistence of explicit permissions, their presence allows them to be exported to other banks in the hierarchy. In this way, we can maintain the property of the hierarchy that authorizations in descendent nodes always have corresponding authorizations in some ancestor node (cf. Definition 11).

Figure 17(b) illustrates eight other predefined authorizations (including four stoppers) that are added to the omnibank as soon as a group or individual (i.e. an object of type *TSubject*) is created. For instance, when a subject S_1 creates subject S_2, corresponding authorizations, illustrated in Figure 17(b), will be added to the owner's omnibank (which is assumed to be b_0 in this example). Authorization <C_0^{13}, *permit, read,* S_2> in omnibank b_0 means that S_2's owner is permitted to invoke method *read* in order to read the value of S_2 and any of its descendants. Similarly, permission *deleteSubject* in omnibank means that the owner can delete S_2 or any of its descendants. As is true for all objects, it is assumed again that only leaf subjects (individuals or empty groups) can be deleted. Moreover, Figure 17(b) illustrates that the owner enjoys permission *subscribeTo* (*unsubscribeFrom*), and therefore is permitted to add (remove) members to (from) S_2. Note that a subject is represented as a member of S_2 by linking it as a child vertex of S_2. Moreover, Figure 17(b) includes four stopper authorizations in omnibank b_0 to stop the propagation of *read*, *subscribeTo*, *unsubscribeFrom*, and *deleteSubject* permissions, respectively. As before, although stoppers are not functioning in omnibanks due to coexistence of explicit permissions, their presence is important so that owners can export them to other banks in the hierarchy when needed.

Finally, Figure 17(c) illustrates the case in which an object d_1 (i.e. any other object of type *TData*) is created. Authorization $<C_0^{21}$, *permit, read,* $d_1>$ in omnibank b_0 means that the owner is permitted to invoke method *read* in order to read the value of d_1 and any of its descendants, and the *read* stopper in Figure 17(c) is for export purposes. Depending on the data type and on the policies in force (see Chapter 5), other authorizations will also be added to the omnibank when d_1 is created.

In summary, Figure 17 illustrates all initial authorizations for a particular subject, S_1, who is considered to be the owner of an object of type *TSubject*, S_2, and an object of type *TData*, d_1. When an object is created, the omnibank of the owner is updated to include new authorizations for all methods of the new object if not already included. Based on these authorizations, S_1 can extend his bank hierarchy by creating other banks and importing arbitrary authorizations from his omnibank, b_0. In this way, he can share his data with other subjects.

Figure 18 illustrates a system with *m* objects (o_1 to o_m), *n* subjects (S_1 to S_n), all of which own one or more objects, and therefore *n* omnibanks ($omni_1$ to $omni_n$). Recall that in all figures of this thesis, subject-bank assignments are depicted with dashed-dot lines, bank-object assignments are depicted with dashed lines, and omnibanks are depicted with black ovals; also, subject names start with a capital letter, and bank names are italic.

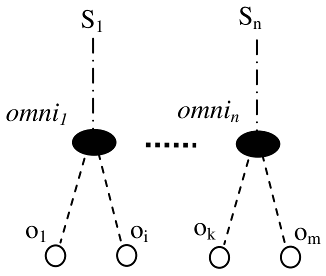


Figure 18. Omnibanks in a system with *n* subjects.

4.5 Access Control Administration

In the last section, we explained that subjects are initially authorized for their own objects within omnibanks. Now we explain how users manage the access authorizations for their data. As the running example, we assume user S_x owns a newly created object d_1 of type *TData*; and therefore, his omnibank $omni_x$ includes authorizations similar to those depicted in Figures 17(a) and 17(c).

For the rest of this section, we explain how S_x can utilize the corresponding bank hierarchy (depicted with triangles in Figures 19-22) to delegate its authorizations in any of three forms, namely *simple delegation*, *delegation by agent*, and *enhanced delegation*, and to retain control by revoking authorizations in two ways, namely *weak* and *strong* revocation.

4.5.1 Simple Delegation

In the *simple delegation* mechanism, subject S_x creates bank b_i importing authorization $<C_i, permit, p, d_1>$ from his bank hierarchy and delegates permission p to other subjects or group of subjects (such as S_a and S_b in Figure 19) by assigning them to b_i. C_i constrains who is eligible to have authorization p. Thus, in order to be assigned to b_i, S_a and S_b must be subsets of C_i, and once they are assigned, they are eligible to invoke method p on object d_1 and its descendants.

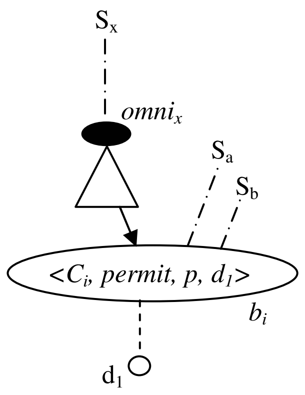


Figure 19. Simple delegation in ACAD.

4.5.2 Delegation by Agent

In the *delegation by agent* mechanism, subject S_x creates banks b_i and b_{i+1} importing authorizations $<C_i, permit, p, d_1>$ and $<C_{i+1}, permit, assignTo, b_i>$, respectively, from his bank hierarchy. Moreover, S_x assigns other subjects (such as S_c and S_d in Figure 20) to b_{i+1} as his agents so that they can delegate permission p on d_1 to others (such as S_e and S_f in Figure 20) by assigning them to b_i as long as the latter subjects meet constraint C_i. Therefore, if C_i excludes S_d as an eligible subject for invoking method p on d_1, S_d cannot be assigned to b_i. In such a case, S_d can only assign other subjects to b_i on S_x's behalf, but he has no other permission on d_1 or on b_i itself. In this form of delegation, S_c and S_d can delegate permission *p* on S_x's behalf, however grantees have no right to delegate it further.

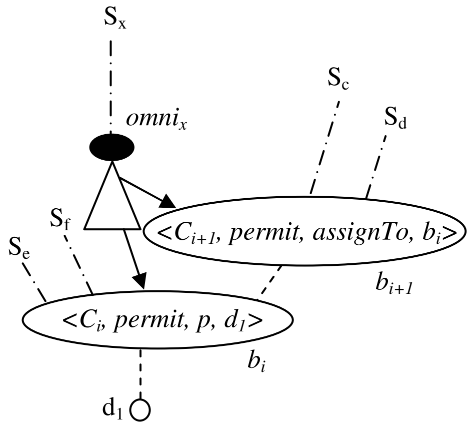
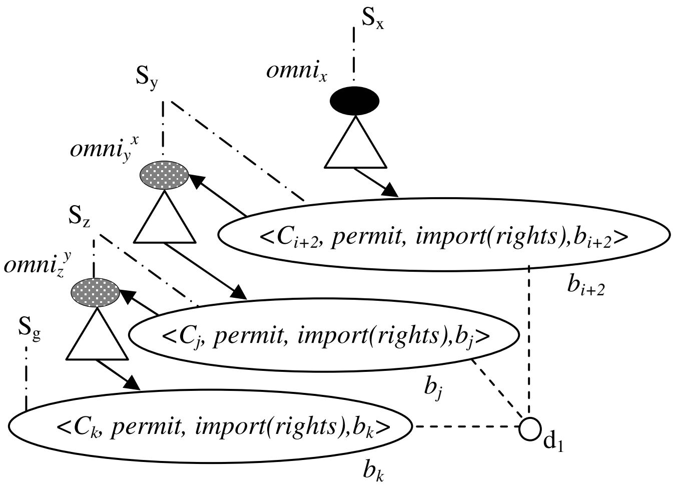


Figure 20. Delegation by agent in ACAD.

4.5.3 Enhanced Delegation

In the *enhanced delegation* mechanism, subject S_x creates bank b_{i+2} importing permission *import* from his bank hierarchy, and assigns S_y to it so that S_y can create new children for b_{i+2} on S_x's behalf. However, merely because S_y creates such banks, does not mean that he has full control over them; instead, S_x can adjust S_y's authorizations on such banks through the *rights* argument of *import*. For instance, if S_x wants to prevent S_y from deleting such banks, he may include authorization *<Public-{S_y}, permit, deleteBank, b_c>* in the *rights* argument. Note that *Public-{S_y}* means every subject in the universe but S_y, and b_c refers to the child bank being created. Also, in order for S_y to be able to invoke method *p* on d_1 or delegate permission *p* to others, S_x must include authorization $< C^{m}_{i+2}$*, permit, p,* d_1> in the *rights* argument. Now, S_y has at least two options: one is to create a child for b_{i+2} containing permission *p* only; and the other is to create a bank, say $b_{j,}$, including the *import* permission. Both are mechanisms to delegate the permission on S_x's behalf. The former is without and the latter is with the option for further delegation. Of course, S_y is not obliged to use his delegation right. Figure 21 shows the case that S_y has delegated the import permission to S_z through b_j, and similarly S_z has delegated it to S_g through b_k.



S_x
$omni_x$
S_y
$omni_y^x$
S_z
$omni_z^y$
S_g
$<C_{i+2}$, permit, import(rights),$b_{i+2}>$
b_{i+2}
$<C_j$, permit, import(rights),$b_j>$
b_j
$<C_k$, permit, import(rights),$b_k>$
b_k
d_1

Figure 21. Enhanced delegation in ACAD.

It is important to notice that S_x has necessary controls on all banks b_{i+2}, b_j, and b_k, while S_y has control on banks b_j and b_k, and S_z has control on bank b_k only. For example, if S_y wants to prevent S_z from invoking method *p* on d_1 or prevent him from delegating the import permission to S_h, he can adjust the *rights* argument of b_j with authorizations *<Public-{S_z}, permit, p, d_1>* or *<Public-{S_h}, permit, import, _>*, respectively.

It is also important to notice that S_x does not have to be the owner of d_1 in order to exploit the above delegation mechanisms. In general, once banks b_i, b_{i+1} and b_{i+2} exist, any subject G which holds authorizations *<{G}, permit, assignTo, b_i>*, *<{G}, permit, assignTo, b_{i+1}>*, and *<{G}, permit, assignTo, b_{i+2}>* can invoke *simple delegation*, *delegation by agent*, and *enhanced delegation* mechanisms, respectively, to delegate permission *p* on d_1.

Other than preventing grantees from doing particular operations in the future, there is often a need to undo or revoke some given permissions. For the rest of this section, we explain how subjects retain control over their own objects even though many authorizations have been delegated. We provide two levels of revocation, *weak* and *strong*.

4.5.4 Weak Revocation

Assume bank b_i with authorization $<C_i$, *permit, p,* $d_1>$. Assigned subjects to a bank containing authorization $<C$, *permit, removeFrom,* $b_i>$ can revoke permission p on d_1 for any subject enjoying it through b_i by removing the subject from the set assigned to the bank. We call this mechanism a weak revocation since if the revoked subject is reassigned to b_i or if the subject enjoys the permission through other banks, he still can invoke method p on d_1.

4.5.5 Strong Revocation

Strong revocation is feasible by exploiting authorization constraints. Assume bank b_i with authorization $<C_i$, *permit, p,* $d_1>$. Any subject who is able to further constrain b_i can strengthen constraint C_i so that specific subjects are revoked from b_i and all its descendants, since constraints for a child bank must be subsets of constraints of its parents. No subject can reassign them to b_i or its descendant since the assignment must be in accordance with the constraint. However, if a subject enjoys permission p through banks other than b_i or its descendants, he still can invoke method p on d_1. But a given subject S_x can strongly revoke a given subject S_h from invoking method p on S_x's documents by excluding S_h at a very top bank of the hierarchy, usually in S_x's omnibank. Note that if this is done, S_k cannot access any documents of S_x.

Figure 21 illustrates a delegation chain from S_x to S_g through banks b_{i+2}, b_j, and b_k. Since these banks are in S_x's bank hierarchy, S_x has full control on all of them. In general, through the delegation chain, grantors retain control over grantees. Therefore, owners have the most control since they initiate the chain. For instance, S_x can (weakly or strongly) revoke S_z from bank b_j in order to prevent him from further importing permission p on d_1. This is a selective revocation since it does not impact any other subject in the delegation chain, including S_g who was assigned to the chain by S_z. However, in a similar manner S_x can revoke S_g too from the chain if he wishes to cascade his previous revoke. Moreover, to completely cascade revocation, S_x does not need to know explicitly which grantees have received the import permission from S_z since he can revoke them all by setting C_j to empty.

4.6 Another Look at Omnibanks

As explained in the last section, exporters retain control over their objects (including banks) by customizing a sub-element of the omnibank of importers. Figure 22 illustrates the omnibank of subject S_y that has been granted import permissions delegated by S_x and $S_{x'}$.

Bank $omni_y$ includes two sub-elements $omni_y^x$ and $omni_y^{x'}$ that are customized by S_x and $S_{x'}$, respectively. S_y enjoys all authorizations included in omnibank $omni_y$ as well as customized omnibanks $omni_y^x$ and $omni_y^{x'}$. In this way, S_y can create combined banks importing from both $omni_y^x$ and $omni_y^{x'}$, and provide simultaneous access to both S_x and $S_{x'}$ hierarchies. However, as illustrated in the figure, customized omnibanks import their authorizations from the bank hierarchy of the corresponding exporters not from the parent omnibank. Hence, S_x and $S_{x'}$ retain control over descendants of $omni_y^x$ and $omni_y^{x'}$, respectively.

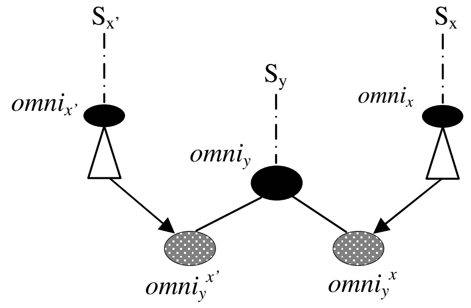


Figure 22. Omnibanks and customized omnibanks.

4.7 Access Banks vs. Access Roles

Access banks in ACAD are similar to access roles in Role Based Access Control (RBAC) [Ferraiolo et al. 2001] in which roles usually reflect job titles. Similar to roles in RBAC models, access banks in ACAD map a many-to-many relationship between subjects and objects. Therefore assigning either a new subject or object to a bank is one operation corresponding to multiple permissions being granted. However, in traditional role-based models, roles form a hierarchical relationship for the sake of efficiency [Ferraiolo et al. 2001]. For example, a project manager has his special permissions as well as all permissions of the project developers reporting to him. Thus, permissions are propagated through the role hierarchy. However in ACAD, propagation is through object and subject hierarchies, and the bank hierarchy instead represents all delegation chains for authorizations. We exploit the bank hierarchy not for propagation of permissions, but instead as a means of retaining control for grantors. This feature allows us to decentralize access control administration. As discussed in Chapter 2, RBAC models have the major innate restriction of a central control over the role hierarchy which consequently impedes the model enhancement towards supporting scalable administrative expansions. If it is necessary to have a role hierarchy in an application, ACAD can simulate it as a part of the subject hierarchy (possibly combined with a group hierarchy) and access banks can remain to handle the decentralized delegation needs. Chapter 7 shows an application of access banks for a system, called user-managed access control, in which both roles and groups exist.

4.8 Related Work

Sections 2.1-2.4 extensively addressed the related work of access control administration models. This section consists of a literature review on storage decentralization mechanisms as well as a comparison between our delegation and revocation mechanisms and that of existing proposals.

The access control matrix is typically large and sparse when subjects or objects can not be classified to a limited number of groups. Decentralization techniques such as access control lists and capabilities improve the storage efficiency. The former technique stores the information from the matrix column-wise while the latter is row-wise. The Compressed Accessibility Map (CAM) is an enhanced technique on capability lists [Zhang 2005]. The CAM algorithms exploit structural locality of subjects' accessibility on a hierarchical data to construct a more efficient tree. Therefore, instead of keeping a list of all accessible nodes, they only keep some crucial nodes and place some additional information on them, so that we can check whether an arbitrary node can be accessed or not by simply looking at relevant crucial nodes. The CAM technique also addresses granularity explained in Section 2.4.

Finally, the *simple delegation* and *delegation by agent*, in ACAD, are similar to delegation mechanisms in several models [Muffett 1990; Bertino et al. 1999; Barka and Sandhu 2000a; Barka and Sandh 2000b; Zhang L. et al. 2002; Zhang L. et al. 2003; Zhang X. et al. 2003; Joshi and Bertino 2006; Ravichandran and Yoon 2006; Wang and Osborn 2006]. Furthermore, simple delegation and cascading revocation have been proposed as part of distributed network security models and trust management systems [Blaze et al. 1996; Rivest and Lampson 1996; Li et al. 2002] However, the

ACAD model is distinguished from other models by its *enhanced delegation* and revocation mechanisms explained in Sections 4.5.3 through 4.5.5. Weak and strong revocations in the work by Bertino et al. are basically based on explicit and implicit authorizations; whereas, ACAD exploits removal from access banks and constraints, respectively, for such purposes. In Section 7.3, ACAD formally proves its mechanism is valid to retain control for data owners in Section 7.3.

Chapter 5

CONSTRAINING DECENTRALIZED ADMINISTRATION

Chapter 4 specifies an access control administration model, called ACAD, to support the spectrum of autocratic to self-governing systems. This chapter specifies the means of constraining decentralized administration by which ACAD can be adjusted anywhere on the spectrum, in order to meet the needs of an enterprise. This means provides the flexibility of defining various levels of decentralization for different types of objects in the system.

As new subjects and new objects are introduced into a running system, and as subjects delegate and revoke access control permissions, the access control state changes, as reflected in the metadata. Thus, depending on the current state of access control, several possible new states can be realized in response to subjects' actions. The space of all possible access control states forms a network, with transitions similar to those in a finite state automaton. An access control policy is a set of rules under which access control states may evolve: in a properly designed system, all reachable states conform to the access control policy chosen by the enterprise.

Recall from Chapter 1 that although access control policies and the generality of their enforcement must be kept in the hands of the enterprise (whether centrally managed or managed in a distributed fashion), there are several reasons to decentralize access control *administration* (the permission to update the access control metadata) either for the whole system or for a subset of object types in the system. Sharing data, including selectively delegating and revoking administrative permissions on the associated metadata, may be supported more reliably and more efficiently by a flexible decentralized access control administration. Consequently, the spectrum of access control administrations raises an important question: to what extent can decentralization be realized? The challenge is to ensure that the enterprise's policies are enforced.

This chapter finds the answer by using *creation time policies* to define the network of permissible access control states: each time an object is created by a subject, an appropriate set of permissions on the object are given to various subjects. This specifies the initial state for each row (object) in the explicit access control matrix, which implies the space of reachable states for that row. At one extreme, if neither delegation nor revocation permissions are ever allowed, the permissions on that object will never change. However, in practice, policies are rather complex and the access control metadata continually evolves over time as the access control states change. This evolution of metadata is difficult for human administrators to manage state by state. By appropriately initializing permissions at object creation time, a set of administrative policies is established, and these constrain subsequent access control state transitions. As an analogy, creation time policies control objects in our system much like genetic codes control cells in biological systems.

The contribution of this chapter is a flexible model for administration by which enterprises are able to adjust the amount of centralized control by defining precisely what permissions are initially to be held by whom at the time of each object's creation. By appropriately configuring these creation time policies, organizations can adjust their access control systems to the desired points along the spectrum.

Creation time policies dictate the states of the explicit access control matrix. So that the power of creation time policies can be presented in a concrete setting, we assume one mechanism for deriving the effective access control matrix. In particular, this chapter assumes a closed system, in which permissions are denied by default, and only positive explicit permissions are provided; the propagation of explicit positive permissions are controlled with "stoppers" (Section 3.8). Section 5.4 argues that the power of creation time policies can be applied equally well to other settings for deriving an effective matrix from an explicit one.

The rest of this chapter is organized as follows. In Section 5.1, the specification of the ACAD constraining mechanism is discussed. Section 5.2 justifies the model by showing how it could be used to control several existing systems. Section 5.3 describes how the current chapter is applied in

ACAD. Section 5.4 provides further discussion on power of creation time policies. Finally, Section 5.5 reviews the literature of related access control models and frameworks.

5.1 The Constraining Mechanism

We first translate access control policies to sets of first order logic rules. Figure 23 depicts the BNF notation which is used for ACAD policy enforcement rules, following the syntax of Active-U-Datalog language [Bertino et al. 1998]. This language can be used to express active rules that are fired when various events occur (e.g. updates). ACAD applies this mechanism to insert into the explicit access control matrix the authorizations that are defined according to creation time policies. Each policy rule consists of a head and a body. The head is one of the reserved predicates, such as *permit*, *deny*, *subscribeTo*, *unsubscribeFrom*, *deleteSubject*, *assignTo*, *removeFrom*, *import*, *addConstraint*, and *deleteBank*. While predicates *permit* and *deny* directly update the explicit access control matrix, other predicates cause an indirect change to access rights. The body consists of both reserved and application-based predicates. The application-based predicates are Boolean functions that typically define the hierarchical relationship within the objects involved in the rule. There may be three types of prefixes (so called *sign*) for each predicate: + indicates that the predicate is an *insert*, - indicates that the predicate is a delete, and ¬ represents a logical negation.

```
Policy_Enforcement_Rule ::=  Body | ε → ( + | - ) Head.
Head ::= Reserved_Predicate
Body ::= Body , Body | Predicate
Predicate ::= Sign Reserved_Predicate | Sign Application_Predicate
Reserved_Predicate ::=  permit | deny | bank | auth | juniorOf
Application_Predicate ::= create | inherit …
Sign ::= + | - | ¬ | ε
```

Figure 23. BNF notation of policy enforcement rules.

Moreover, ACAD groups together sets of rules that are triggered under a specific event, such as object creation. In this work, we define *admin* domains as sets of creation time policies that are invoked when an object is created.

As explained in Section 1.2, the explicit matrix defines access constraints used to derive the effective permissions represented by the effective matrix. As in Section 3.5, in ACAD, the explicit access control matrix is transformed to the corresponding effective matrix by translating each predicate *permit* to one or more predicates *allow*. Each propagation strategy may produce a different effective matrix. If stoppers are in place, a given permission is propagated along the edges of both object and subject hierarchies until meeting a *stopper* or a sink node. Finally, every non-filled cell of the effective matrix will be represented by a corresponding predicate *disallow*.

Section 5.1.1 describes, via the context of the healthcare example, how this model allows us to apply *creation time policies* to decentralize access control administration.

5.1.1 Creation Time Policies

Recalling Example I, introduced in Section 1.3 and illustrated in Figure 13, assume that St. Mary's Hospital is the owner of the medical records. However, also assume that elements of the medical record are created by various subjects, such as the hospital staff or the patient, according to the following scenario:

Receptionists create a blank medical record when a new patient arrives.

Nurses create and append new encounters to an existing medical record.

Doctors create the diagnosis information. They can also create and append therapy sections to an existing record.

A patient's family may create (and thereby "sign") the consent section.

Also assume the set of access control policies described in the XACML use cases [Kudo 2001; Damodaran and Adams 2001], restated as three sets of creation time policies:. Figure 24 presents these policies in the ACAD model. In the remainder of this section, each set of policies is first stated informally and then the corresponding formal presentation is exaplined in detail. For simplicity, we assume that every subject (including St Mary's here) has an associated omni bank, and when a new object *x* is created, all of the corresponding authorizations are automatically added to the omni bank of its owner, identified by the function *omni(x)*. Moreover, in this scenario, all customized banks are created underneath the omni bank.

CTP1 is triggered when a *medical_record* is created, invoking the following rules: St. Mary's hospital, the owner of the *medical_record*, enjoys all permissions. The patient can read *patient_info.* Doctors and nurses can read *medical_record.*

The body of the rule of CTP1 in Figure 24 is a conjunction of five application-based predicates namely *medical_record*, *patient_info*, *patient*, *doctors*, and *nurses*, as well as functions *omni* and *Skole,* that uniquely generates *object ids* using two input parameters (the first parameter represents an object, and the second parameter is used as an index). Predicate *medical_record* determines whether its input parameter is an object of type medical record. The prefix + indicates that the active rule applies when this medical record is just created and its information is being inserted into the system. Predicate *patient_info* determines if its first parameter is the patient information of the medical record determined by the second parameter. Predicate *patient* determines if its first parameter is the subject for this medical record. Similarly, predicate *doctors* (and *nurses*) determines if the parameter is a doctor (nurse). The head of CTP1 includes three reserved predicates namely *auth*, *bank*, and *juniorOf*. Predicates *auth* and *bank* were defined in Section 4.2.3. Predicate *juniorOf* represents the bank hierarchy; its first input denotes a child bank, and the other input is its parent.

Therefore, CTP1 in Figure 24 creates two authorizations a_1 and a_2 to read *patient_info* and *medical_record*, respectively, as well as creating banks b_1 and b_2 (as child banks of *omni*) to hold those authorizations. Moreover, the *patient* is initially assigned to b_1, and *doctors* and *nurses* are assigned to b_2.

CTP2 is triggered when an *encounter* is created, invoking the following rules: Doctors can create *diagnosis_info.* The patient's family can sign the consent.

Similarly, CTP2 in Figure 24 creates a new authorization a_3 (as a permission to create a diagnosis) as well as a bank b_3, which includes a_3, and is a direct child of the appropriate omni bank. In addition, doctors are assigned to this bank.

CTP3 is triggered when the consent is created, invoking the following rules: The patient's family can read *encounter*. The patient's family can also delegate read permission on *diagnosis_info* to others. Nobody, not even the owner of the whole medical record, can change *consent*.

CTP1:

+medical_record(M), patient_info(PI, M), patient(S_1,M) , doctors(S_2), nurses(S_3), omni(M)=b, Skolem(M,1)=b_1, Skolem(M,2)=b_2, Skolem(M,-1) = a_1, Skolem(M,-2)=a_2 $\rightarrow$

+auth(a_1, 'Public', 'permit','read', PI), +bank(b_1,S_1,a_1), +juniorOf(b_1,b), +auth(a_2, 'Public', 'permit','read', M), +bank(b_2,S_2,a_2), +bank(b_2,S_3,a_2), +juniorOf(b_2,b).

CTP2:

+encounter(E, M), omni(M)=b, doctors(S_1) , Skolem(M,3)= b_3, Skolem(M,-3)= a_3 $\rightarrow$

+auth(a_3, 'Public', 'permit','createDiagnosis', E).

+encounter(E, M), consent(C, E), family(F, P), patient(P,M) , omni(M)=b, Skolem(M,4)=b_4, Skolem(M,-4)=a_4 $\rightarrow$

+auth(a_4, 'Public', 'permit','sign', C), +bank(b_4, F, a_4), +juniorOf(b_4,b).

CTP3:

+consent(C, E), encounter(E, M), create(F, C), family(F, P), patient(P, M), omni(M)=b, Skolem(M,5)=b_5, Skolem(M,-5)=a_5 $\rightarrow$

+auth(a_5, 'Public', 'permit','read', E), +bank(b_5, F, a_5), +juniorOf(b_5,b).

+consent(C, E), encounter(E, M), create(F,C), family(F, P), patient(P,M) , diagnosis_info(DI,E), omni(M)=b, Skolem(M,6)=b_6, Skolem(M,-6)=a_6, Skolem(M,7)=b_7, Skolem(M,-7)=a_7 $\rightarrow$

+auth(a_6, 'Public', 'permit','read', DI), +bank(b_6, null, a_6), +juniorOf(b_6,b), +auth(a_7, 'Public', 'permit','assignTo', b_6), +bank(b_7, F, a_7), +juniorOf(b_7,b).

+consent(C,E), omni(M)=b, Skolem(M,8)=b_8, Skolem(M,-8)=a_8 $\rightarrow$

+auth(a_8, 'Public', 'deny','change', C),+bank(b_8, 'Public', a_8), +juniorOf(b_8,b).

Figure 24. Creation time policies for the medical record use case.

The rest of the rules in Figure 24 are similar. Predicate *create* in CTP3 has two parameters to indicate a subject and an object, respectively, and determines whether the subject has created the object. Note that it is assumed the system is defined such that *encounter(x, y)* can be true only if *medical_record(y).* The last rule of CTP3 defines a stopper. It specifies that nobody can inherit permission to execute method *change* on a consent part of any medical record. In particular, this rule stops the propagation of permission *change*, given to the owner of a medical record in the second rule of CTP1, to the element *consent*, even though it will be propagated to the element *encounter*.

5.2 Expressivity of the Mechanism

This section validates the constraining mechanism by investigating suitable object creation policies that could be used to achieve the goals of some existing systems, including the UNIX file system, the DB2 database system, and the LiveLink enterprise content management system. Note that throughout the rest of this chapter, we define creation time policies directly on the explicit access control matrix, and not via access banks, to simplify explanations. In UNIX, there are two types of objects, *file* and *directory*, for which two sets of creation time policies are defined in Section 5.2.1. In DB2, there is one object type, namely *schema*, which contains many sub-types such as *base table*, *view*, *index*, etc; Section 5.2.2 defines a set of creation time policies for the schema. In LiveLink, there exist many types of objects having disparate access control requirements; accordingly, several creation time policies are defined in Section 5.2.3 to handle administrative needs in such systems.

5.2.1 Creation Time Policies for the UNIX File System

In UNIX, access control administration is partially decentralized among the objects' owners, but in a very restricted manner. In that system, there are three access permissions: *read*, *write*, and *execute*. When an object (file or directory) is created in UNIX, a predefined list of permissions for the object is automatically created. Then, the object owner can assign or change the permissions mode for three types of subjects: *user*, *group*, and *other*. UNIX does not support hierarchies among the subjects (that is, groups cannot be nested) nor among the objects (permissions cannot be granted on subfiles, and permissions are not inherited from directories to subdirectories or to contained files).

For simplicity, assume a given set of default permissions are based on the following policy: Files are readable and writable by the owner, readable for groups, and not accessible to others (in Unix's notation, file permissions are initialized to 640). Directories are readable, writable, and executable by the owner, and also readable and executable to groups and others (corresponding to Unix's 755). Moreover, subjects can read an object if they have execute permission on all directories in the path from the root to the parent of the object and read permission on the object itself; subjects can delete an object if they have execute permission on all directories in the path from the root to the parent of the object and write permission on that parent (they need no specific permissions on the file itself). System administrators (super-users) have all permissions on all files and directories. These default permissions can be simulated by applying creation time policies as follows, see Figure 25:

U-CTP1:

+file(F), owner(S, F) → *+permit(S, read, F).*

+file(F), owner(S, F) → *+permit(S, write, F).*

+file(F), owner(S, F) → *+deny(S, execute, F).*

+file(F), sameGroup(S,F) → *+permit(S, read, F).*

+file(F), superUser(S) → *+permit(S,*,F).*

U-CTP2:

+directory(D), owner(S, D) → *+permit(S, *, D).*

+directory(D) → *+permit(*, read, D).*

+directory(D) → *+permit(*, execute, D).*

+directory(D), sameGroup(S,D) → *+deny(S, write, D).*

+ directory(D), superuser(S) → *+permit(S, *, D).*

Figure 25. Creation time policies for UNIX file permissions.

U-CTP1 is triggered when a file is created, invoking the following rules: The owner enjoys permissions read and write on the file; execute permissions are not inherited. Group members enjoy permission read on the file. Super-users have all permissions.

U-CTP2 is triggered when a directory is created, invoking the following rules: The owner enjoys all permissions on the directory. Group members and others enjoy read and execution permissions on the directory. Again, super-users have all permissions.

The body of the first rule in U-CTP1 includes two application-based predicates, namely *file* and *owner*. Predicate +*file* determines whether a new file F (not a directory) has been created. Predicate *owner* has two input parameters: a subject and an object, respectively; and, determines whether the subject is the owner of the object. Therefore, the first rule in U-CTP1 states that if the file *F* has been created, and user *S* is its owner, *S* is permitted to execute method *read* on *F*. Similarly, the second rule in U-CTP1 states that if the file *F* has been created, and user *S* is its owner, *S* is permitted to execute method *write* on *F*. The third rule emphasizes that the owner cannot inherit permission *execute* from its group or others. The fourth rule states that users who are in the same group as the file are permitted to execute method *read* on the file, where predicate *sameGroup* determines whether *S* and *F* are in the same group. Finally, the fifth rule in U-CTP1 states that super-users have all permissions on files, where predicate *superUser* determines whether S is a super-user. U-CTP2 is similar to U-CTP1 and expresses policies that are applicable to directories, where predicate *directory* determines whether its input parameter is a directory (not a file).

Note that Figure 23 illustrates creation time policies only, and it does not address other policies in UNIX. For example, in UNIX, users can create files and directories only if they have permissions execute and write on the current path. Such a policy can be expressed in ACAD as: *permit(S, create, *) :- permit(S, execute, currentDirectory()), permit(S, write, currentDirectory())*. However, our focus here is on creation time policies, which are distinguished from other policies by having at least one inserting predicate (i.e. prefixed by +) in their body, and a +permit or +deny in their head. This predicate acts as a trigger that enables various permissions to be defined.

5.2.2 Creation Time Policies for the DB2 Database System

This section considers controlling access to data in IBM's DB2, and investigates how creation time policies can decentralize its administration. There are various objects and therefore various privileges in DB2. Hence, this section focuses on objects of one of the types *schemas*, *tables*, *views*, and *routines* only. However, interested readers can expand our explanation to other types of objects such as triggers, functions, table spaces, etc.

Table 4 illustrates the major DB2 object types and related privileges. A schema is a logical classification of named objects such as tables, views, nicknames, triggers, functions, and packages. When a schema is created, its owner may be granted CREATIN, ALTERIN, and DROPIN privileges to be allowed to create objects, alter objects, and drop objects within the schema, respectively. Moreover the owner is able to grant any of these privileges to other users. D-CTP1 in Figure 26 illustrates this. For instance, the first predicate in D-CTP1 states that when schema S is created by any user, the owner of S is permitted to create objects within S.

Table 4. Objects and privileges in DB2.

Objects	Privileges
Schema	CREATIN, ALTERIN, DROPIN
Table	CONTROL: DELETE, INSERT, SELECT, UPDATE, ALTER, INDEX, REFERENCE; as well as GRANT and REVOKE any of these
View	CONTROL: DELETE, INSERT, SELECT, UPDATE; as well as GRANT and REVOKE any of these
Routine	EXECUTE
other objects	Other privileges

Table 4 also shows that DELETE, INSERT, SELECT, and UPDATE are applicable to both tables and views, allowing authorized users to delete, insert, select, and update rows of tables and views, respectively. ALTER, INDEX, and REFERENCE are applicable to tables only, and let the authorized user add more columns to an existing table, define indices on a table, and create or drop a foreign key that references a table, respectively.

DB2 grants the control of such tables and views to the object creator initially. Users who are authorized to control such objects can grant and revoke such privileges to and from other users. D-CTP2 in Figure 26 illustrates this. For instance, the first predicate in D-CTP2 states that when user U creates table T, U is permitted to control T. Table 4 and D-CTP3 in Figure 26 state that the privilege EXECUTE is initially granted to both PUBLIC and the creator of routines (procedure, function, method).

Furthermore, SYSADM, and DBADM are two DB2 authorities that can execute any privilege at the schema and instance level, respectively. They also can grant and revoke all their privileges.

However, since such rules are defined independently of the creation of specific objects, they are excluded from creation time policies.

D-CTP1:

+schema(S), owner(U) → *+permit(U, createin, S).*
+schema(S), owner(U) → *+permit(U, alterin, s).*
+schema(S), owner(U) → *+permit(U, dropin, S).*
+schema(S), owner(U) → *+permit(U, grantcreatein, S).*
+schema(S), owner(U) → *+permit(U, grantalterin, S).*
+schema(S), owner(U) → *+permit(U, grantdropin, S).*

D-CTP2:

+table(T), create(U, T) → *+permit(U, control, T).*
+view(V), create(U, V) → *+permit(U, control, V).*

D-CTP3:

+routine(R), create(U, R) → *+permit(U, execute, R).*
+routine(R), public(PUBLIC) → *+permit(PUBLIC, execute, R).*

Figure 26. Creation time policies for DB2.

5.2.3 Creation Time Policies for the Livelink ECM System

This section considers Livelink, a product of Open Text Corporation for enterprise content management. The system is built over a complex data repository, in which the data space is divided into one *enterprise workspace* and a collection of *personal workspaces*, one per user. Diverse forms of data objects, including *documents*, *tasks*, *news* items, etc., can be stored in the workspaces. Data objects may be kept in various kinds of container objects, including *folders*, *discussions*, *channels*, and *projects*. Moreover, containers can include other containers, forming a complex network of data objects that is accessible to individual users and groups of users, depending on their permissions. Furthermore, users designated as *Admin* have all permissions on all objects (cf. Unix's super-users, as explained in Section 5.2.1) and individual users have all permissions on all objects stored in their own personal workspaces. All other users may be assigned permissions as individuals or through their membership in a user group, where user groups are structured into an arbitrarily deep, directed acyclic graph. Being based on a hierarchical model, each object resides within some container. Like Unix, Livelink assumes a three-component structure to implement user permissions, including *owner*, *owner group*, and *public group* (which includes all subjects that have "public access" permissions to any object). Figure 27 illustrates how creation time policies model initial permissions of each component when an object is created.

L-CTP1:

+item(O), container(O, C), owner(U,C), permit(U, P,C) $\rightarrow$ *+permit(S, P, O).*

+item(O), container(O, C), ownerGroup(G_1,O), ownerGroup(G_2,C),

permit(G_2, P,C) $\rightarrow$ *+permit(G_1, P, O).*

+item(O), container(O, C), publicGroup(G), permit(G,P,C) $\rightarrow$ *+permit(G, P, O).*

Figure 27. General creation time policies for Livelink's permissions.

In general, the creator of an item is the owner of that item, and that user is initially granted the permissions held on the container by *its* owner. The first rule in L-CTP1 illustrates this policy: the body of the rule is a conjunction of three application-based predicates, namely *item*, *container*, and *owner*; as well as a reserved predicate *permit.* Predicate *item* determines whether a new object has been created. Predicate *container* determines whether *C* is the container of *O*. Predicate *owner* has two parameters to indicate a subject and an object, respectively, and determines whether the subject is the owner of the object. Predicate *permit*, in the body of the rule, determines permissions of the container's owner.

Every item is also associated with an *owner group*, and those group permissions are also initialized to be identical to the owner group permissions on the container. This policy is captured by the second rule in L-CTP1, in which predicate *ownerGroup* determines if the first parameter is the owner group of the second parameter. The other predicates function as in the first rule.

Finally, the public group's permissions are initialized to match its permissions for the container. This policy is captured by the third rule in which predicate *publicGroup* determines if its parameter is the public group.

In addition to the general creation time policies that are applicable to all object types in Livelink, there are several policies applicable to specific object types.

The most common data items are documents, URLs, workflow maps, aliases, releases, generations, and versions. These can be hierarchically organized into folders and compound documents. The creator of any of these objects is initially assigned all permissions held by the owner of the container, as explained in L-CTP1 in Figure 27. All other users are initially assigned the permissions that they have on the container within which the object is created. This policy is captured by L-CTP2, depicted in Figure 28, in which predicate *+docs* determines if the new created object is from one of the above data item types. Recall that L-CTP2 is not a propagation policy as described in Section 5.1: it is applicable only when an object is created, and thereafter permissions on either the object or its container can be updated without affecting permissions on the other.

A second type of data item is a project, which can contain any types of data, including other projects. Projects, like all other items, have permissions assigned for the *owner*, *owner group*, and *public access*, as depicted in Figure 27. Unlike the items described above, however, the assigned access list for a project contains exactly three project-specific groups, namely *coordinator*, *guest*, and *member*, which act as *roles*; users and groups are assigned to these roles by making them members of these project-specific groups. Permissions are assigned to the three roles as follows: a coordinator has all permissions on the project, a guest has permission to see the project, and a member can also see, add, modify, or delete objects within the project. These policies are captured by L-CTP3, depicted in Figure 28, in which predicates *coordinatorRole*, *guestRole*, and *memberRole* indicate that the first parameter is the specific group associated with the object specified as the second parameter to represent coordinator, guest, and member roles, respectively.

When a project is created, the creator is automatically assigned to the coordinator role, and coordinators may assign users and groups to any of the three roles or delete them from those roles. When a project is created as a sub-project of another one, the coordinator, member, and guest lists are copied from the project in which it resides, and the creator is added as an additional coordinator for the sub-project, (see L-CTP4).

L-CTP2:

$+docs(O), container(O, C), permit(S,P,C) \rightarrow +permit(S, P, O).$

L-CTP3:

$+project(O), coordinatorRole(R,O) \rightarrow +permit(R,*, O).$

$+project(O), guestRole(R,O) \rightarrow +permit(R, see, O).$

$+project(O), memberRole(R,O) \rightarrow +permit(R, see, O).$

$+project(O), memberRole(R,O) \rightarrow +permit(R, addTo, O).$

$+project(O), memberRole(R,O), container(X,O) \rightarrow +permit(R, delete, X).$

$+project(O), memberRole(R,O), container(X,O) \rightarrow +permit(R, modify, X).$

L-CTP4:

$+project(O), create(S, O), coordinatorRole(R,O) \rightarrow +assignRole(S,R).$

$+project(O_1), container (O_1,O_2), project(O_2), coordinatorRole(R_2,O_2),$
$assignRole(S_2,R_2), coordinatorRole(R_1,O_1) \rightarrow +assignRole(S_2,R_1).$

$+project(O_1), container (O_1,O_2), project(O_2), memberRole(R_2,O_2), assignRole(S_2,R_2),$
$memberRole(R_1,O_1) \rightarrow +assignRole(S_2,R_1).$

$+project(O_1), container (O_1,O_2), project(O_2), guestRole(R_2,O_2), assignRole(S_2,R_2),$
$guestRole(R_1,O_1) \rightarrow +assignRole(S_2,R_1).$

L-CTP5:

$+project(O_1), container (O_1, O_2), folder(O_2), permit(S_2,allPermissions,O_2),$
$coordinatorRole(R_1,O_1) \rightarrow +assignRole(S_2,R_1).$

$+project(O_1), container (O_1, O_2), folder(O_2), permit(S_2,read,O_2), guestRole(R_1,O_1) \rightarrow$
$+assignRole(S_2,R_1).$

$+project(O_1), container(O_1,O_2), folder(O_2), permit(S_2,addTo,O_2), \neg(permit(S_2,$
$editAttributes, O_2), permit(S_2,modifyContents,O_2)), memberRole(R_1,O_1)$
$\rightarrow +assignRole(S_2,R_1).$

$+project(O_1), container (O_1, O_2), folder(O_2),\neg permit(S_2,addTo,O_2), permit(S_2,$
$editAttributes, O_2), permit(S_2,modifyContents,O_2), memberRole(R_1,O_1)$
$\rightarrow +assignRole(S_2,R_1).$

continue …

Figure 28. Specific creation time policies for Livelink's permissions.

```
L-CTP6:
    +workItem(O), create(S, O) → +permit(S ,administer, O).
    +workItem(O), container(O,C), folder(C), permit(X,allPermissions, C) → +permit(X
                    ,administer, O).
    +workItem(O), container(O,C), folder(C),
                    permit(X,read,C) → +permit(X ,read, O).
    +workItem(O), container(O,C), folder(C), permit(X,addItems, C),
                    ¬ (permit(X, editAttributes, C),
                    permit(X,modifyContents, C)) → +permit(X ,write, O).
    +workItem(O), container(O,C), folder(C), permit(X,editAttributes, C),
                    permit(X, modifyContents, C),
                    ¬permit(X, addItems, C) → +permit(X ,write, O).
    +workItem(O), container(O,C), project(C),
                    coordinatorRole(R,C) → +permit(R, administer, O).
    +workItem(O), container(O,C), project(C),
                    guestRole(R,C) → +permit(R, read, O).
    +workItem(O), container(O,C), project(C),
                    memberRole(R,C) → +permit(R, write, O).
```

Figure 28 (continued). Specific creation time policies for Livelink's permissions.

In those rules the predicate *assignRole* indicates that the first parameter (which is a subject) is made a subgroup of the second parameter (which represents a role). Also as before, predicate *create* has two parameters to indicate a subject and an object, respectively, and determines whether the subject has created the object.

Alternatively, if a project is created in a folder, subject-role associations are initialized based on the permissions that the subjects have on the folder: the project creator and those with all permissions on the folder are added to the coordinator list, those with read permission on the folder are added to the guest list for the project and those with either add items or both edit attributes and modify contents on the folder are added to the member list for the project. As before, the project creator is assigned to the coordinator role by the first rule of L-CTP4. However, the rest of the assignments are captured by

L-CTP5, depicted in Figure 28, in which predicate folder determines if an object is from the type folder, and keyword allPermissions is an abbreviated form of the conjunction of all permissions.

The third class of data item, called work items, includes channels (sequences of news items), discussions (hierarchically nested sequences of topics and replies), and task lists (hierarchically nested sequences of tasks). For this class, the individual elements of the work item do not carry independent permissions, but instead are governed by the permissions assigned to each work item as a whole. For objects in this class, users and groups may be given *read* permission, *write* permission, or *administer* permission (which includes grant and revoke permissions as well as all other permissions). If a work item is created under a folder, the creator and those with all permissions on the folder are assigned *administer* permission, stated by the first and second rules of L-CTP6 in Figure 28. Those with *read* permission on the folder are assigned *read* permission on the work item; stated by the third rule of L-CTP6. Finally, those with either *add items* or both *edit attributes* and *modify contents* on the folder are assigned *write* permission on the work item; stated by fourth and fifth rules of L-CTP6. Likewise, if an item of this class is created under a project, the coordinator group is given *administer* permission, the guest group is given *read* permission, and the member group is given *write* permission; stated by the last three rules of L-CTP6.

5.3 ACAD and the Constraining Mechanism

The constraining mechanism described in Section 5.1 is the means of initializing the access control states in ACAD as well as restricting the possible valid next states. In particular, this mechanism can exploit different classes of ACAD access banks (illustrated in Figure 16) to define creation time policies (e.g. in public banks, as illustrated in Figure 26). Moreover, the initial content of omni banks (illustrated in Figure 17) is in fact determined by this same mechanism. It is creation time polices that define what the initial privileges of owner banks are, whether someone can customize or combine two existing banks, and what privileges are allowed to be imported in such banks.

5.4 Further Discussion

This section describes the power of creation time policies to adjust the level of decentralization. As shown through Section 5.2, it is important to notice that the creation time policy in ACAD is a mechanism to initialize an explicit access control matrix anywhere on the spectrum of access control administration. This section highlights the power of this mechanism towards adjusting the amount of decentralization anywhere from an autocratic setting to an anarchistic state, by manipulating the creation time policies of the motivating example illustrated in Figure 24.

In particular, the second rule of CTP3 in Figure 24 states that only one set of subjects (who are the family member of the patient) can update rows in the explicit access control matrix that correspond to specific patients (but only by setting cells of the corresponding column of diagnosis information to permission read). We illustrate the following cases to demonstrate the effect of creation time policies.

(a) If the second rule is removed from CTP3, no one can update the explicit access control matrix at all. Consequently, the existing cells of the access matrix will not evolve (only new rows or columns may be added to the matrix).

(b) Now, while the second rule does not exist in CTP3, assume adding the following rule

+medical_record(M) ^ secutiy_officer(S) → *+permit(S, *, M).*

to Figure 24. This simulates a centralized (autocratic) administration model since only the security officer is able to update the explicit access control matrix (delegate or revoke rights). All other subjects have to channel their update requests through the security officer.

(c) Alternatively, assume adding the following rule

create(S,M) → *+permit(S, *, M).*

to Figure 24 (while the second rule does not exist in CTP3). This simulates a user managed access control model since whoever creates an object has all of the update rights on it. Unless the creaor chooses to grant delegation permissions to others, all other subjects have to channel their update requests through the creator.

(d) Finally, assume adding the following non-safe rule

+medical_record(M) → *+permit(*, *, M).*

to Figure 24. This rule makes the administration model anarchistic since every subject in the system is able to update the rights on a medical record.

5.5 Related Work

Access control enforcements are traditionally divided into Discretionary Access Control and Mandatory Access Control. The former provides predefined (by users) discretionary rules and access control based on users' identities; the latter controls access based on subjects' and objects' classifications in a system. In mandatory access control environments, access control rules are decided by system policies independently of the owners of objects. Both discretionary and mandatory frameworks have been of interest to researchers, and they are supported by many subsequent models. Role based access control models also have well investigated in the literature as reviewed in Section 2.2. This section reviews another work focused on object creation as follows.

A more specific idea of enforcing security policies at object creation time is shown in the work by Zannone et al.. They propose a mechanism to control information flow with a focus on derived objects that are dynamically created at run time [Zannone et al. 2006]. Essentially, the authors improve the flexible authorization framework [Jajodia et al. 2001] by allowing users to create new objects but avoiding Trojan horses. The idea is that a derived object's authorizations must be a subset of the intersection of its original object's authorizations. The system administrator is warned if a user can access a new object o_1 while he cannot access a subset of objects from which o_1 is derived. It is, however, up to the system administrator to allow or disallow such information flows.

Chapter 6

MODEL SEMANTICS

This chapter represents a formal semantics, defined operationally by a relational model, for the ACAD model. Three data catalogues are defined in ACAD, namely SysObjHier, SysBankObj, and SysBankSubj, to represent the object hierarchies as well as the bank-object and bank-subject assignments, respectively. The assumption is that there are additional relations for all objects representing their various attributes, such as object type, creator, etc. To support these catalogues, three keywords *NoPar*, *Public*, and *Sys* are defined, where *NoPar* represents the virtual parent of all root objects, $Public \subseteq U$ represents all subjects in the access control universe (cf. Definition 1), and $Sys \in Public$ is the *system subject* that is in charge of all automatic operations taken by the system at initialization time.

Table 5(a) illustrates SysObjHier, which represents the information about the object hierarchies including the bank and subject hierarchies. This catalogue consists of four attributes, namely *T#*, *Child*, *Parent*, and *By*. In our illustrations, *T#* i_j indicates that the current tuple results from the *j*th operation of the *i*th transaction. *Child* and *Parent* represent the child-parent relationship within the object hierarchy. Attribute *By*, which is from the *Public* domain, represents which subject has caused the tuple. Note that we assume each root vertex is its own parent; moreover, we do not illustrate cross-references in our catalogues since they have no influence on the algorithms.

Table 5(b) illustrates SysBankSubj, the bank-subject assignments, which include four attributes, namely *T#*, *Bank*, *Subject*, and *Grantor*. Again, *T#* i_j indicates that the current tuple is produced by the *j*th operation of the *i*th transaction. *Subject* and *Bank* represent the subjects which hold all permissions designated by authorizations in the bank. *Grantor* represents who has assigned the subject to the bank.

Table 5(c) illustrates SysBankObj, the bank-object assignments, which include six attributes, namely *T#*, *Bank*, *Object*, *Operation*, *Mode* and *Constraint*. *T#* is as in the other relations. *Bank* represents the bank name. *Constraint*, *Operation*, and *Object*, when *Mode* is set to "permit", represent the domain of subjects who are eligible to execute the operation on the object within this bank. When *Mode* is set to "deny", the tuple represents a negative authorization indicating that the permission cannot be granted to the object. The conflict resolution strategy determines which authorizations can be implicitly obtained by the assigned subjects.

Table 5. ACAD system catalogues.

(a) SysObjHier represents object hierarchies.

T#	Child	Parent	By^c
0_1	b_{00}	b_{00}	Sys

(b) SysBankSubj represents bank-subject assignments.

T#	Bank^c	Subject^c	Grantor^c
0_3	b_{00}	Creators	Sys

(c) SysBankObj represents bank-object assignments.

T#	Bank^c	Object^c	Operation	Mode	Constraint
0_2	b_{00}	NoPar	createChild	permit	Public

Underlined attributes and superscript letters represent primary and foreign keys of these catalogues, respectively. For instance, the primary key of SysBankSubj is the combination of *Bank* and *Subject*; also, attributes *Bank*, *Subject*, and *Grantor* are foreign keys referencing the key of an object table (not illustrated). Moreover, Table 5 depicts the initial state of the catalogues when a new system is initialized. *T#* 0_i indicates that the tuple is produced by the initial transaction in the system. Table 5(a) illustrates that the system subject, *Sys*, initially creates bank b_{00}. Tables 5(b) illustrates that Creators $\subseteq$

Public are subjects authorized to create objects, whereas Table 5(c) illustrates that any subset of Public can be potentially authorized to create objects. If Creators=Public, i.e. every subject has the right to create objects, then there is no constraint at initialization time to be assigned to bank b_{00}; in fact, all subjects (*Public*) are then assigned to b_{00} and thus authorized to create root objects, children of *NoPar*. The rest of this section specifies the semantics of major access control *lookup* and *update* requests in our model.

6.1 The Lookup Requests

There are two major classes of lookup requests in ACAD, *subjectCapabilities* and *objectAccessList*. The former is a class of queries by which one can find the access rights for an arbitrary subject (user, group, application); and the result, called *capabilities*, is a list of pairs of objects with the corresponding operations for which the subject has permission. Similarly, the latter is a class of queries by which one can find the access rights to an arbitrary object (file, directory, group, etc.); and the resulting *access list* is in a list of pairs of subjects with the corresponding operations permitted on the object.

In Chapter 3, we discussed an algorithm to propagate access authorization through hierarchies and resolve possible conflicts based on a variety of combined strategies. The algorithm, called *Resolve()*, exploited the explicit access control matrix (called EACM) as well as subject and object hierarchies (called SDAG and ODAG) as global variables. Here, in Figure 29, Algorithm 0 represents how such variables can be defined using data catalogues depicted in Table 5. In particular, Line 1, in Algorithm 0, defines relation *EACM* to express all explicitly authorized (either permitted or stopped) subject-object-operation triples, by joining catalogues SysBankObj and SysBankSubj where the joint attribute is *bank*. Lines 2 and 3 define relations SDAG and ODAG to represent subjects and objects hierarchies, respectively. (Note that only (active) objects that are members of Public are considered as subjects in ACAD.)

Algorithm II in Figure 29 depicts *subjectCapabilities*. It takes a subject as an input parameter, and computes the union of all pairs <*o*, *p*> for all objects and all permissions if accessible to the subject. Each output pair indicates an operation that the subject is permitted to execute on the corresponding object. Algorithm *Resolve()*, presented in Chapter 3, determines whether or not a given subject can access on object with some permission, based on the current conflict resolution strategy and propagation mode.

Similarly, Algorithm III in Figure 29 depicts *objectAccessList*. It takes an object as an input parameter, and computes the union of all pairs <s, *p*> for all subjects and all permissions for which it is accessible. Each output pair indicates a subject that can execute the corresponding operation on the object.

```
Algorithm 0: PreliminarySteps()
    1. $EACM = \Pi_{subject,object,operation,mode}(SysBankSubj \times_{bank} SysBankObj)$
    2. $SDAG = \Pi_{child,parent}(\sigma_{child \in Public} SysObjHier)$
    3. $ODAG = \Pi_{child,parent} SysObjHier$

Algorithm I: Resolve ($s_1 \in Public, o_1 \in U, p_1 \in Permissions, pMode \in$ {"pass through", block by",
        override"}, $dRule \in$ {"+", "-", "0"}, $lRule \in \{max(), min(), identity()\}$, $mRule \in$ {"before",
        "after", "skip"}, $pRule \in$ {"+", "-"})
    ¤ Cf. Chapter 3, Figure 8

Algorithm II: SubjectCapabilities ($s_1 \in Public$)
    1. $R = \bigcup_{o_1 \in U, p_1 \in Permissions} < o_1, p_1 >$ if $resolve(s_1,o_1,p_1,pMode,dRule,lRule,mRule,pRule) = '+'$
    2. Return $R$

Algorithm III: ObjectAccessList ($o_1 \in U$)
    1. $R = \bigcup_{s_1 \in Public, p_1 \in Permissions} < s_1, p_1 >$ if $resolve(s_1,o_1,p_1,pMode,dRule,lRule,mRule,pRule) = '+'$
```

Figure 29. Lookup algorithms in ACAD.

Note that our focus in this chapter is on the semantics of ACAD, and not on the computational complexities of the algorithms. However, it is obvious that the worst-case time complexity of Algorithm II (and III) is not worse than the one of Algorithm I since we can enhance Algorithm I to propagate all pairs of <object, permission> in the same manner as it currently propagates one pair, with the same worst-case complexity. Furthermore, the conflict resolution algorithm (*Resolve()*) can be replaced with alternative algorithms that are optimized for a particular strategy or subset of strategies to be provided to a customer.

6.2 The Update Requests

Figure 30 illustrates eight update methods, namely *assignTo*, *removeFrom*, *import*, *addConstraint*, *deleteBank*, *subscribeTo*, *unsubscribeTo*, and *deleteSubject* as explained in Section 4.3. To be able to call any of these methods, the caller subject must have been granted the applicable permission within *subjectCapabilites*. For simplicity, we omit code to check that update operations are in accordance with primary and foreign keys constraints, such as ensuring that duplicates are not inserted into the access control tables and references to deleted values are not left dangling.

Algorithm IV represents method *assignTo,* which assigns a given subject $s \in Public$ to a given bank $b \in Banks$ where views $Public \subseteq U$ and $Banks \subseteq U$ represents all subjects and banks in the access control universe, respectively. First, subject *s* must meet all constraints for authorizations in *b*. Then, a corresponding tuple <*T*(), *s*, *b*, *caller*()> is inserted into SysBankSubj, in which function *T*() generates a new transaction number and function *caller*() returns the subject who calls method *assignTo*.

Algorithm V represents method *removeFrom*, which removes a given subject $s \in Public$ from a given bank $b \in Banks$. The method removes the tuple corresponding to bank *b* and subject *s* from SysBankSubj.

Algorithm VI represents method *import*, which allows the caller to import any subset of authorizations $omniRights \subseteq Authorizations$ from a given bank $source \in Banks$ to a new bank $newBank \in Banks$. Hence, *newBank* becomes a child of *source* in the bank hierarchy by inserting tuple <*T()*, *newBank*, *source*, *caller()*> into SysObjHier where function *caller()* determines the importer subject. Moreover, recall from Section 4.5, the exporter (owner of *source*) retains control over his bank hierarchy by specifying (using parameter *omniRights*) which authorizations are inserted into the customized omnibank of the importer.

Algorithm VII represents method *addConstraint*, which excludes subjects from the existing constraint of a given permission $p \in Permissions$ of a given bank $b \in Banks$ on a given set of objects where view *Permissions* represents all permissions of the system and views *Banks*, *U*, and *Public* are

```
Algorithm IV: assignTo(s∈Public; b∈Banks)
    if $s \in constraint(b)$
    then insert <T(), s, b, caller()> into SysBankSubj;

Algorithm V: removeFrom (s∈Public; b∈Banks)
    delete from SysBankSubj where subject=s and bank=b;

Algorithm VI: import(newBank∈Banks; source∈Banks; omniRights⊆Authorizations)
    insert <T(), newBank, source, caller()> into SysObjHier
    update omni(caller()) with omniRights;

Algorithm VII: addConstraint (b∈Banks; o ∈ U; p ∈ Permissions; s ∈ Public)
    if $s \in \Pi_{\text{constraint}} \sigma_{\substack{bank=b, \\ object=o \\ operation=p \\ mode=\text{permit}}} (SysBankObj)$
    then
        update SysBankObj set constraint=constraint-{s} and t# = T()
            where bank=b and object=o and operation=p and mode=permit;
        delete from SysBankSubj where subject = s and bank=b;
        addConstraint(children(b), o, p, s);

Algorithm VIII: deleteBank (b∈Banks)
Pre: l has no child
    delete from SysBankSubj where bank=b;
    delete from SysBankObj where bank=b or object=b;
    delete from SysObjHier  where child=b;

Algorithm IX: subscribeTo (member∈Public; group∈Public)
    insert <T(), member, group, caller()> into SysObjHier;

Algorithm X: unsubscribeFrom(member∈Public; group∈Public)
    delete from SysObjHier where child=member and parent=group;

Algorithm XI: deleteSubject (s∈Public)
Pre: s has no child
    delete from SysBankSubj where subject=s;
    delete from SysBankObj where object=s;
    delete from SysObjHier  where child=s;
```

Figure 30. Update algorithms in ACAD.

as explained in Algorithm IV. First, if *s* does not intersect the existing constraint, the algorithm terminates; otherwise, the corresponding tuples <_, *b*, *o*, *p*, *permit*, *Constraint*> in SysBankObj are replaced by new ones <*T*(), *b*, *o*, *p*, *permit*, *Constraint*-{*s*}>, in which function *T*() generates a new transaction number. Calling this method may revoke bank *b* and its descendants from some subjects.

Therefore, the corresponding tuples for subject s in *s* are removed from SysBankSubj. Finally, the algorithm is cascaded to the descendants in order that their constraints remain contained within the revised constraints in the ancestor (as required by Definition 11). Note that the method is applicable to permission authorizations only, not to stoppers.

Algorithm VIII represents method *deleteBank*, which deletes a leaf bank $b \in Banks$ from the bank hierarchy. Calling this method removes all corresponding bank-subject assignments from SysBankSubj, all corresponding bank-object assignments from SysBankObj, all tuples in which *b* is treated as an object from SysBankObj, and all incident edges for vertex *b* from SysObjHier.

Algorithm IX represents method *subscribeTo*, which includes a given subject $member \in Public$ in a given subject $group \in Public$. The corresponding tuple <*T*(), *member*, *group*, *caller*()> is inserted into SysObjHier in which functions *T*() and *caller*() are as in Algorithm IV.

Algorithm X represents method *unsubscribeFrom*, which removes a given subject $member \in Public$ from a given subject $group \in Public$. The method removes the tuple corresponding to *member* and *group* from SysObjHier.

Algorithm XI represents method *deleteSubject*, which deletes a leaf subject $s \in Public$ from the subject hierarchy. Calling this method removes all corresponding bank-subject assignments from SysBankSubj, all tuples in which *s* is treated as an object from SysBankObj, and all incident edges for vertex *s* from SysObjHier.

6.3 Object Creation

When a new object is created, one or more corresponding tuples are inserted into the catalogue SysObjHier to represent the object and its parent(s). (Recall, in ACAD, each object is its own parent too.) Furthermore, zero or more update requests (depicted in Figure 30) are automatically triggered based on the creation time policies (discussed in Chapter 5). As depicted in Figure 23, any of the ten types of reserved predicates explained in Section 4.3 may be triggered in a creation time policy. The explanation of each case is as follows:

Predicate *permit* (or deny) causes a root bank to be created by Sys; consequently, one tuple is inserted into catalogue SysObjHier and several tuples are inserted into catalogues SysBankObj and SysBankSubj (at least one each), to represent the bank, its authorization(s), as well as the corresponding object and subject assignments.

Predicates *subscribeTo* and *unsubscribeFrom* cause an update to the subject hierarchy, which either insert tuples to (or delete from) catalogue SysObjHier, to represent the group subscription and unsubscription, respectively. Similarly, predicate *deleteSubject* causes one or more deletions from catalogue SysObjHier, to represent deletion of a leaf subject and removing corresponding edges from its parents.

Predicates *assignTo* and *removeFrom* cause an update to the subject-bank assignments, which either insert or delete tuple(s) from catalogue SysBankSubj to represent the simple delegation and revocation, respectively. Note that *assignTo* must first ensure that the subject meets the constraint.

Predicate *import* causes a child bank of an existing bank to be created by Sys; several tuples are inserted into catalogues SysObjHier, SysBankObj and SysBankSubj, to represent the bank, its parents, its authorization(s), as well as the corresponding object and subject assignments. Recall that *import* is the means of enhanced delegation in ACAD, explained in Section 4.5.3.

Recall also that *addConstraint* is the means of strong revocation, explained in Section 4.5.5. Predicate *addConstraint* causes an update to Catalogue SysBankObj; the update is limited to further constrain the constraints attribute of an existing tuple. This may also cause several other updates for the tuples associated with the descendants of the constraining bank as well as deleting some tuples form SysBankSubj to remove subjects who no longer meet the new constraint.

Finally, predicate *deleteBank* causes one or more deletion from catalogue SysObjHier, to represent deletion of a leaf bank and removing corresponding edges from its parents.

6.4 Related Work

The idea of defining the ACAD semantics operationally by a relational model has been inspired by the paper by Bertino et al. describing authorization model for relational databases [Bertino et al.

1999]. They have defined four relational catalogues to represent privilege-table (or view) assignments, subject-privilege assignments, subjects' ancestors, and subjects' parents relationships. It is clear that the ancestor relationship can be derived from the parent relationship; however the authors decided to materialize it for implementation efficiency. Note that they do not have to worry about the object hierarchy since DB2 tables are independent from each other (they use views mostly for content-dependent authorizations). However, in ACAD, relying on the fact that everything is essentially an object, all subject, object, and bank hierarchies are represented in one catalogue (SysObjHier). Moreover, the implementation choices of the ancestor relationship, which is computed by deriving reachable nodes for each hierarchy, remains intentionally open to users for flexibility in storage and time efficiencies.

Chapter 7

USER MANAGED ACCESS CONTROL

Chapters 3 through 5 describe the mechanism to specify access control administration, which is called ACAD. This chapter provides an illustrative example in which ACAD is applied to define a User Managed Access Control (UMAC) system (through the motivating example of healthcare applications, introduced in Section 1.3). In particular, in UMAC, subjects fully manage the formation of groups, the structure of objects, and the expansion of access banks as well as the assignment of authorizations, without interfering with one another and without requiring a centralized administration to update the access control structure. Section 7.1 introduces the UMAC specification. Section 7.2 illustrates UMAC by a use case of healthcare systems. Section 7.3 proves how users can retain control in UMAC. Finally, Section 7.4 reviews related literature.

7.1 Specification

Recall from Chapter 1, access control models form a spectrum of *autocratic* to *self-governing* administrations. UMAC fits at the latter end. There are various types of self-governing systems, such as systems in which every object creator is the object owner (for example, web-based file sharing systems), systems in which owners are defined at configuration time (for example, corporate applications), etc.

UMAC is a self-governing system, in which some subjects who create objects are considered as owners whereas some other creators may act as agents or employees of the object owner. At creation time, some subject is designated as owner and initially receives all permissions on the created object.

Creation time policies may additionally grant initial permissions to other users as well. Propagation of permissions uses the stopper form of denial. Therefore, in UMAC, each subject may possess two faces: the administrator of its own objects, and at the same time, a user of others' objects. With their administration face, subjects require full access control over their own objects, but as users they are typically licensed to more limited levels of access.

User managed access control becomes complicated when the hierarchy of subjects is not consistent with the hierarchy of objects. In other words, subjects (e.g. technicians in the healthcare application) may have access to many small parts of objects (their patients' relevant personal data). Moreover, accessible domains of various subjects form diverse structures that should be recognized for optimization purposes. For instance, a physician's accessible data mainly consists of a collection of disconnected nodes each of which corresponds to a particular patient treated in various clinics; while, a technician's accessible data is basically medical data of patients treated in a particular laboratory.

Definition 12 (Principle of *Non-interference*). "One group of users, using a certain set of commands, is not interfering with another group of users if what the first group does with those commands has no effect on what the second group of users can see" [Goguen and Meseguer 1983].

Goguen and Meseguer defined the principle of non-interference by which subjects are prevented from certain interference activities that violate some security policies. Notice that this approach avoids a transitive-closure computation which generally exists in verifying security by determining which subjects potentially can interfere with others.

ACAD's features provide the flexibility of defining a UMAC model. For instance, the generic and customized banks (introduced in Section 4.2.) can be applied towards the self-governance property explained in Chapter 1. In particular, there is no administrative superiority in Figures 16 (a) or (b) since neither S1 nor S2 can disrupt the other's actions, and there is no centralized control.

7.2 Use Case: Healthcare Systems

This section illustrates the UMAC model through a more concrete instance of Example I. This application has been initially inspired from the XACML use cases [Kudo 2001]; however, we have adapted it to reflect a more decentralized application environment [Chinaei and Tompa 2005]. Figure 31 illustrates the schema of a typical *medical record,* which consists of one element of *patient* *information* and zero or more elements of *encounter*. An *encounter* consists of elements *hospitalization information*, *diagnosis information*, and possibly completed *consent* form. Each element of *diagnosis information* contains zero or more elements of *therapy information*. A medical record may be accessed by different subjects such as doctors, patients, receptionists, accounting system, etc. Each subject should be authorized to have only the minimal privileges it needs. Here, the functions of UMAC are briefly described based on the following scenario that focuses on both efficiency and self-governance.

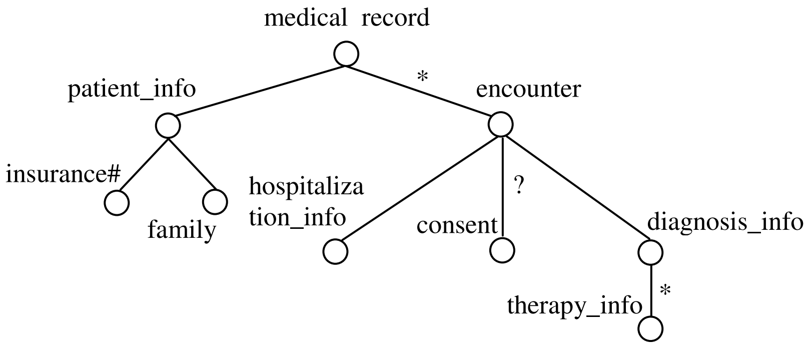


Figure 31. An example of medical records schema.

The scenario assumes an instance of the system is installed for St. Mary’s hospital. The hospital is the owner of medical records; however, various users of the system, such as patients or caregivers, may create elements of a medical record (such as personal data, hospitalization information, and diagnoses). Moreover, groups of users and their job functions may be defined when the system is initialized. For instance, assume group *Patients* is created and assigned to bank $predefined_1$ which includes an import authorization that allows importers to create the element *patient_info* within medical records. Similarly, assume groups *Receptionists* and *Doctors* are created, and their members are able to create sub-elements *hospitalization_info* and *diagnosis_info*, respectively, by exploiting corresponding banks $predefined_2$ and $predefined_3$. Furthermore, all users are assigned to a predefined bank which includes authorization <*C*, *permit, subscribeTo, Patients*> by which they can subscribe themselves to group *Patients* when they need to see a doctor.

This use case represents the application of different classes of access banks, explained in Section 4.2.4. For readability of Figures 32 to 35, the subject hierarchy is omitted from the illustration.

Assume Patricia becomes a member of *Patients*. Therefore, through *predefined*$_1$, she can create an element in the St. Mary's hierarchy to contain her personal information. When she does so, as part of the hospital's check-in procedure, she is directed to create banks b_1 and b_2, and to assign receptionists of St. Mary's Hospital to b_2. Figure 32, which combines aspects of Figures 20 and 22, illustrates this. Note the use of *delegation by agents* and *enhanced delegation*. Moreover, as a part of the check-in procedure, a family member of the patient is assigned to a generic bank *genBank*$_4$, by which they are able to create the consent.

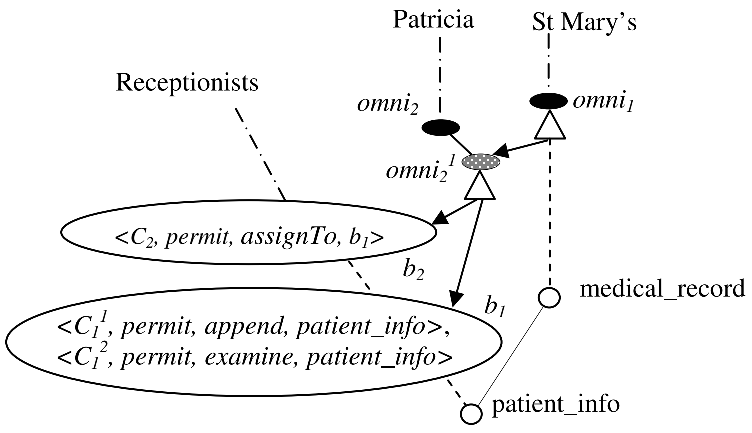

Patricia
St Mary's
Receptionists
$omni_2$
$omni_1$
$omni_2^1$
$<C_2, permit, assignTo, b_1>$
b_2
b_1
medical_record
$<C_1^1, permit, append, patient\_info>$,
$<C_1^2, permit, examine, patient\_info>$
patient_info

Figure 32. A new patient arrives in a healthcare system.

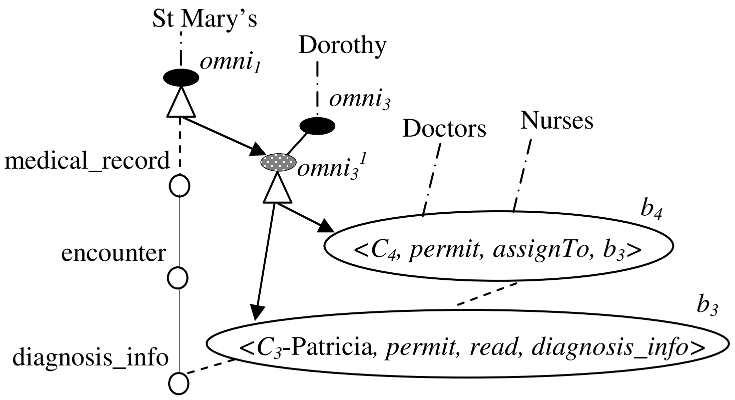
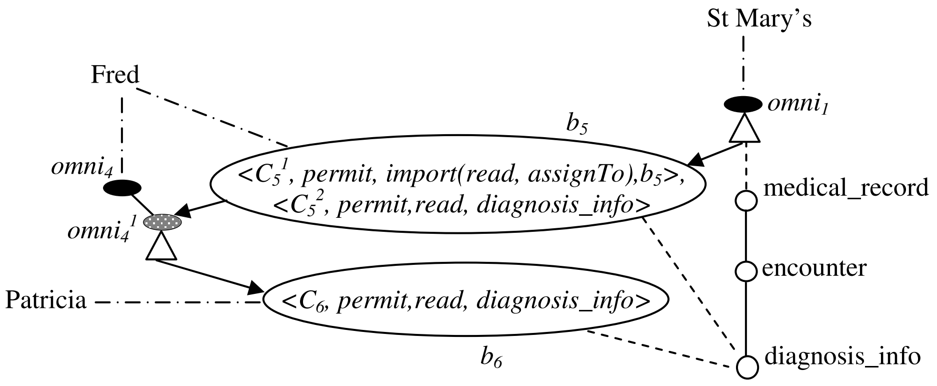


Figure 33. A doctor attends the patient.

Then, assume Robert, who is a member of the *Receptionists* group, and thus inherits the permissions for *Receptionists*, exploits bank *predefined*$_2$ to serve Patricia's request by creating her hospitalization information (such as room number and arrival date), and also uses the grant permission in b_2 to assign the group *Doctors* to b_1 (not diagrammed). Then, as illustrated in Figure 33, doctor Dorothy attends to Patricia, and creates her diagnosis information by exploiting bank *predefined*$_3$. Dorothy also creates banks b_3 and b_4, and as the doctor in charge, assigns *Doctors* and *Nurses* to b_4 so that they can delegate the permission to themselves or to others to read Patricia's diagnosis information. However, she excludes Patricia using the constraint of b_3 to prevent her from reading her own diagnosis information even if she is a doctor or nurse (unless she is granted permission through another route).



St Mary's
Fred
$omni_1$
b_5
$omni_4$
$<C_5^1$, permit, import(read, assignTo),$b_5>$,
$<C_5^2$, permit,read, diagnosis_info>
medical_record
$omni_4^1$
encounter
Patricia
$<C_6$, permit,read, diagnosis_info>
b_6
diagnosis_info

Figure 34. The patient is permitted to see her diagnosis information.

Figure 34 illustrates that St. Mary's can create a new bank b_5 that imports from $omni_1$ and include a read authorization with grant option, and assign Fred, a member of Patricia's family, to it upon creating the informed consent by him. Therefore, Fred can use the enhanced delegation mechanism to create bank b_6 and assigns Patricia to it in order to permit her to read her diagnosis information. Figure 35 combines Figures 32 to 34 and depicts the complete medical record for Patricia. For readability, details of Figures 32 to 34 are not depicted in Figure 35; for instance, Figure 35 does not illustrate access banks' constraints. Moreover, authorizations are shown by abbreviated symbols R, R+, and R*, which represent permission read, simple delegation of permission read, and enhanced delegation of permission read, respectively. For instance, R+ in bank b_2 means subjects assigned to b_2 (e.g. receptionists) are allowed to assign other subjects (e.g. doctors) to bank b_1, which permits them to read Patricia's personal information. Dotted boundaries highlight the domains within which each subject creates and controls a part of the health record database. As explained in Section 5.1, the creators' capabilities are defined by creation time policies. For instance, if the policy states that doctors have all capabilities on the diagnosis information that they create, then Dorothy's connections to banks b_3 and b_4 will be from her omni bank instead of from the customized omnibank.

This example shows how corporate policy can allow various subjects such as patients, receptionists, doctors, and patient family to administer access control over different parts of a medical record. Every time an item of data is created, a corresponding owner bank with all methods is automatically created or updated. As an example of a combined role, Fred can create one combined bank on several diagnoses and assign Patricia to see them all. As an example of selective revocation, Dorothy prevents Patricia from seeing her own record by excluding her in the constraint component of bank b_3. St Mary's can prevent Fred and others assigned to role b_5 or its descendents (b_6) from reading the diagnosis information by removing the connection between the omnibank and b_5; the read permission will no longer be inherited.

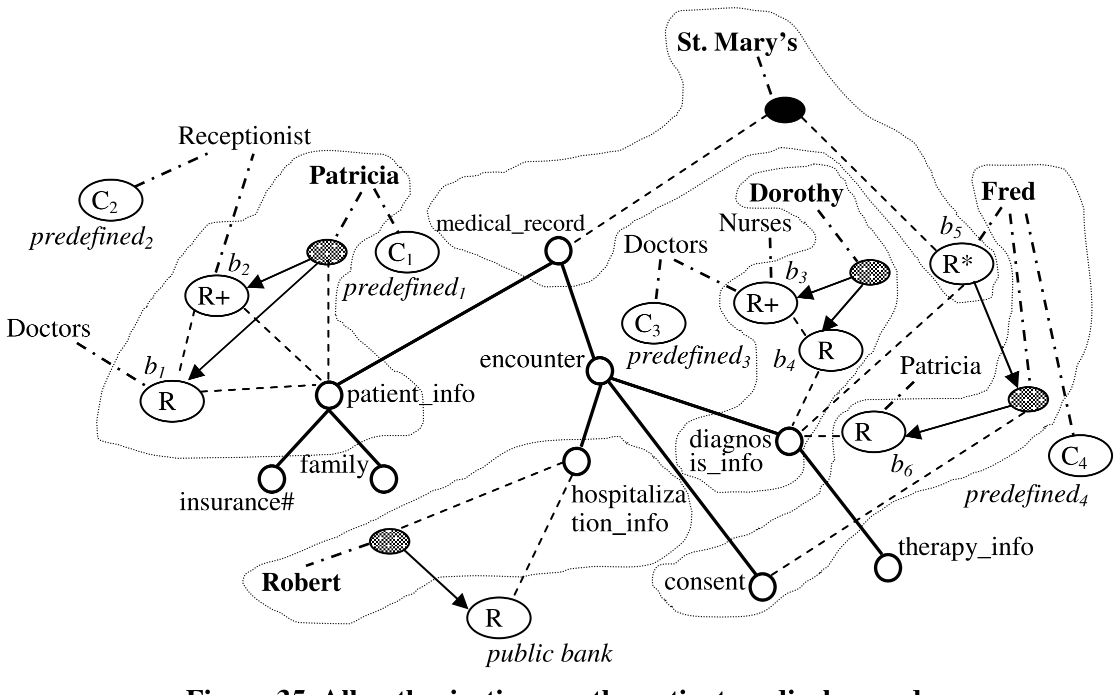


Figure 35. All authorizations on the patient medical record.

One may extend this example in many directions. For instance, since St. Mary's has control over a sub-graph of Dorothy's bank hierarchy rooted at the customized omnibank *omni*$_3^1$, the hospital authority can selectively revoke bank b_4 from some or all nurses. In a worldwide application including many hospitals in the system, a patient may create sub-elements and assign permissions for various hospitals and a doctor may cooperate with several clinics. Thus, Patricia (and Dorothy) may have several customized omnibanks, each controlled by a different hospital. However, St Mary's has control over the corresponding subgraphs of Patricia's (Dorothy's) bank hierarchy rooted at $omni_2^1$ ($omni_3^1$) only, and cannot interfere with the banks of other hospitals. With such an approach, hospitals, and their users, can share their information based on corporate policy agreements.

7.3 Proof of Retaining Control

Theorem. Assume subject S_1 owns object d and delegates any subset of its authorizations with respect to d to any other subjects through a customized bank of its bank hierarchy G. S_1 is able to strongly revoke any given authorization a operating on object d from any given subject S_2 by removing S_2 from the corresponding constraint of omnibank b_0.

PROOF (by contradiction). Assume S_1 is not able to revoke authorization a from S_2. Because S_1 owns object d, all authorizations for d are initially placed in S_1's omnibank or in banks in the hierarchy rooted at that omnibank and nowhere else. Any of those authorizations can appear in other banks only through the import operation, which always creates the new bank as a descendent of the exporting bank. Therefore, there is a customized bank b_1 in G (rooted at S_1's omnibank) by which S_2 enjoys authorization a. There are only two cases then: either b_1 is not a descendent of b_0 which contradicts the definition of an omnibank; or, b_1 is a descendent of b_0 but the constraint of authorization a in b_1 is not a subset of the corresponding constraint in b_0, which contradicts the definition of an access bank. Therefore, S_1 is able to revoke any given authorization a from any given subject S_2. □

Corollary 1. Any subject assigned to an omnibank o is able to strongly revoke any authorization operating on any object associated with o from any given subject S by removing S from the corresponding constraint of o.

Corollary 2. Any subject S_1 who is authorized to add more constraints to a fertile bank f is able to revoke any authorization a delegated to S_2 via any descendant of f by removing S_2 from the corresponding constraint of f.

7.4 Related Work

The user managed access control model explained in this chapter is distinguished from other proposals by its unique property of self-governance (no designated administrator in the system). In fact, from UMAC's point of view all users are in a flat administrative level with no superiority over one another, whereas other proposals often assume a hierarchical administration in the system in which some subjects have administrative superiority with respect to other subjects [Moffett 1990; Firozabadi et al. 2001]. Moffett proposes the idea of administrative domains, each of which has an administrator to update the metadata. The whole system is considered as a major domain with a super administrator who controls other administrators. This idea has been exploited by other researchers to propose a variety of hierarchical administrations; see Sections 2.2 and 2.3. Firozabadi et al. improve the idea of administrative domain with two major contributions: adding constrained delegation and basing the model on the cryptography approaches rather than identity-based access control lists. The former enriched the model by controlling the domain of grantees. The latter modified the model to be based on public keys rather than identities. UMAC is very different from such models since it does not necessarily require domains.

UMAC is also different from the proposals that enhance the RBAC model for non-centralized environments [Wedde and Lischka 2003; Park and Hwang 2003]. As opposed to UMAC, none of these propose a flat administration. In Cooperative Role-Based Administration [Wedde and Lischka 2003], the authors propose local *Authorization Teams* who exercise access control on a set of disjoint organizational units; called an *Authorization Sphere.* They adapt Petri Nets to implement quorum and veto features for granting rights from owners to users. Members of an Authorization Team may jointly modify the set of rules of their authorization sphere. An inheritance principle is applied based on hierarchical relationships between Authorization Spheres. Park and Hwang introduce a three level access policy for a peer-to-peer architecture [Park and Hwang 2003]. Each peer makes the access control decision based on the enterprise, the community, and the peer policies locally. Since different roles may have the same privileges, or conversely, roles with a common name but in different communities may define different privileges, the authors assume a function called *Role Ontology*, which determines similar roles in different communities. In this way, the authors propose a centralized administration for User-Role assignments, but support decentralized Permission-Role assignments in different communities and also within peers.

Chapter 8

COMPARISON OF MODELS

This chapter highlights our contributions by comparing ACAD to four other noteworthy models, so called AFS, FARDMS, FAF, and Ponder, all of which have been well cited in the literature of access control. AFS is the security model for the Andrew File System proposed by Howard et al. [Howard et. 1988]. FARDMS, Flexible Authorization Model for Relational Databases Management Systems, proposed by Bertino et al. [Bertino et al. 1999] extends the System R model by supporting access control exceptions and strong enforcement. FAF [Jajodia et al 2001] is a specification language to support various access control policies in a system. Ponder is a declarative policy specification language for management and security of distributed network systems proposed by Damianou et al. [Damianou et al. 2001]. Further information about each model is given in Sections 8.1 and 8.2.

8.1 Desirable Features

This section undertakes an overview of various access control features. These features have been partially inspired from existing models [Bertino et al. 1999; Tolone et al. 2005]. We also define several complementary features by studying new enterprises. In particular, we review the *overarching* feature (defined in Section 4.1) that dominates the entire model; and, we divide other features into four categories: *functional*, *administration*, *security*, and *performance* requirements, based on which ACAD is comparrd to AFS, FARDMS, FAF, and Ponder. The comparison is illustrated in Table 6.

8.1.1 Overarching Requirement

ACAD is based on one single model in which subjects, banks and objects are treated uniformly, that is, data and metadata are treated with the same security model. However, none of AFS, FARDMS, FAF, or Ponder support the overarching requirement. AFS provides two hierarchies: one for users and groups and another for files and directories, which are controlled by different security models. FARDMS provides three hierarchies for subjects, roles, and objects, and each is controlled differently. Moreover, the models for administrative privileges and ordinary ones differ. For instance, FARDMS concept of strong or weak privileges does not apply to its administrative privileges. Therefore, FARDMS fails in providing a uniform model. FAF defines three, similar yet differently administered, hierarchies for users, objects, and sets of privileges (called roles in the current version). In particular, FAF defines several predicates to represent authorizations, but it does not specify how subjects or objects are represented in the authorization system. As an immediate shortcoming, the materialized views suggested to represent the authorizations, based on the assumption of more frequent access requests with respect to update requests, becomes inefficient in the case of frequent updates in users or objects hierarchies. Ponder does not address the data representation of its authorizations, subjects, or targets, because it focuses on the policy specification language.

8.1.2 Functionality

Functional features express operational expectations that are desired from access control models in practical applications. There are four functional features such as support for both closed and open policies, granularity, support for exceptions within hierarchies, and support for contextual information.

—*Support for both Closed and Open Policies:* A closed access control system exploits the principle that subjects have no access to an object unless corresponding positive access authorizations exist. Similarly, an open access control system states that subjects have access to objects in the absence of negative access authorizations. Thus, closed systems minimize authorization while open systems maximize it. Access control models should be able to implement either assumption properly since both are common in practical applications.

AFS implements a closed policy only, and FARDMS can simulate a restricted open policy that does not support exceptions, whereas FAF and Ponder can easily support an open policy with exceptions by interpreting permissions negatively. Since the conflict resolution component is not hardwired to the rest of the model, applying various policies (including the open policy) is straightforward in ACAD.

—*Granularity:* Data often form hierarchical structures, e.g. user groups and their members or relational tables and their tuples; hence, defining access controls on higher levels of hierarchies and propagating them down to other levels is a space-saving technique as well as providing convenience to users who can simply specify many related authorizations. In contrast, finer grained access control is an important functional characteristic even though it is space consuming. There are many applications in which access control rules are to be defined for a specific individual rather than for a group of users and/or on parts of objects rather than on the whole object. Access control models should support various levels of granularity based on the application needs.

The granularity of AFS is very coarse. Access permissions are defined on a whole directory rather than a specific file in order to retain conceptual simplicity and storage efficiency. FARDMS provides a finer granularity, at the relation level, as well as supporting views. However, the view access level is restricted to positive privileges only. ACAD, similar to Ponder, provides a fine-grained access control at any level of objects. Both FARDMS and Ponder are restricted to coarse granularity with respect to administrative privileges that are inseparable. However, ACAD also unbundles authorizations, since no permission is dependent on another. For instance, a revocation right (such as *removeFrom*) does not require holding any delegation ability (such as *assignTo*). FAF is as fine-grained as ACAD.

—*Support for Exceptions within Hierarchies:* This feature is important for applications in which not all access control policies are defined by general rules and exceptions are inevitable. Although often an authorization should be propagated down to the leaves in the hierarchy, there are situations in which propagation should be stopped somewhere before reaching the leaves. Access control models that do not support exceptions are both inconvenient and more space consuming since more fine-grained authorizations must be explicitly defined in such cases.

All models include both negative and positive privileges in order to support exceptions. Yet, this is limited in AFS and FARDMS. AFS supports one level of exception only, in which a subdirectory may be accessible to some users as opposed to its inaccessible parent; however, it is impossible in

AFS to permit a subfolder to be accessible to any user when its parent is inaccessible to the same user. Similarly, in FARDMS, it is meaningless for a view to be inaccessible to a subject while its base table is accessible to the same subject. There are no such limitations in FAF, Ponder, or ACAD, where exceptions are possible anywhere in the hierarchies.

8.1.3 Administration

This section compares the models based on the features that pertain to administering access control. Enterprises require sophisticated administration features such as support for manipulating hierarchies, the effective timing of access control, delegation and revocation, and implementing various policies. Flexible administration features allow access control to be adjustable to various degrees of (de)centralization. In contrast, models that do not provide such features are only applicable to limited situations e.g. to mandatory access control policies or to applications that have a fixed body of administration (no delegation or revocation in the system).

—*Support for Updating Data Hierarchies:* Access control models should provide update authorizations for manipulating hierarchies, e.g. defining new user groups or adding new attributes to XML elements. Models that do not support such update authorizations separate data administration from data usage; consequently, these models cover limited applications.

AFS does not allow users to create their own groups. FARDMS only allows privileged subjects to update the subject or object hierarchy. Neither FAF nor Ponder discuss updates of the subject hierarchy. In the FAF formalism, the authors explicitly state that the subjects, objects, and roles are disjoint sets. Moreover, no subject can be treated like an object in the formalism of FAF authorizations; the object component is from the object domain only. Therefore, manipulating the object hierarchy differs from manipulating the subject hierarchy. Ponder states that the *target* of an authorization can be network resources or service providers. Ponder allows the definition of role hierarchy and *management structures* at configuration time, however, it does not specify whether or not they are updatable. In general, these proposals assume that access requests are far more frequent than update requests, while ACAD does not rely on that assumption. ACAD allows subjects to update hierarchies. Moreover, updating hierarchies can be centralized or decentralized. Decentralized hierarchy update provided by ACAD has at least two advantages: simpler administration and self-governance.

—*Active/Passive Mechanism:* Users, objects, and authorizations change within the system lifetime. Changes can take effect either immediately or at a suitable breakpoint depending on the application. For instance, if a user subscribes to a group, all group access authorizations can be granted to him either immediately or at the next sign in time. The former is called active while the latter is passive. Access control models should be able to control the timing of these effects.

Changes in AFS take effect immediately because they are hard wired to the state of the operating system. Similarly, changes in Ponder take effect immediately because Ponder is an event-driven language in which authorizations are triggered as soon as certain events happen. In contrast, FAF has to postpone changes to take effect at a suitable breakpoint since authorizations are represented by materialized views in which update is an expensive operation. FARDMS and ACAD can support both active and passive mechanisms since these models are not tied to the implementation.

—*Delegation and Revocation:* Users often need to transfer or extend their responsibilities and authorizations to other users. In contrast, sometimes they need to revoke a specific authorization from other subjects. Models with no flexible delegation or selective revocation features are not suitable for most discretionary access control systems.

AFS provides a limited level of delegation in which only owners can grant privileges to others. Similarly, Ponder supports one level of delegation, in which the network administrator can delegate actions to domain administrators. FAF also allows only a single central administrator to delegate or revoke privileges. Consequently, cascading revocation is not meaningful in the AFS, Ponder, or FAF systems. FARDMS provides a richer delegation mechanism by which further delegation is allowed; however, it requires that a subject must hold a permission in order to delegate it. Moreover, administrative privileges are an atomic package that the owner delegates either as a whole or not at all. Selectively revoking permissions from a grantee in the middle of a delegation path is not supported in FARDMS. On the other hand, the delegation and revocation mechanisms are at the core of ACAD: a single authorization can be delegated to others with or without further delegation option, revocation authorization is independent from delegation, and selective revocation is well supported.

—*Policy Neutral:* Access control models should not impose any particular access control policy; otherwise, their applicability is limited. For example, the Bell-LaPadula model imposes a mandatory access control policy, which is appropriate for specific applications. Other models might depend on the existence of "super-users". Expressive models are policy neutral, leaving it to enterprise administrators to specify their own policy at configuration time.

In AFS and FARDMS, the propagation and conflict resolution polices are hard wired to the rest of system, whereas in FAF, Ponder and ACAD, it can be replaced with any arbitrary policy.

—*Flexibility:* The flexibility of access control models is the characteristic of supporting various degrees of restriction. The setup of an access control system could be different even in two instances of the same application. For instance, one healthcare system may authorize doctors to read patients medical histories by default and another system may not. Access control models that do not cover a wide variety of needs cannot be widely applicable.

In contrast to all other models, ACAD is novel in support of various levels of administration to support systems from the *autocratic* end to *self-governed* (and even *anarchistic*) end of the spectrum of access control policies. This flexibility is elaborated in Section 8.2.

8.1.4 Performance

This section concludes the comparison based on scalability.

—Scalability: Many access control applications deal with many objects. Furthermore, applications usually grow over time. Hence, the scalability of access control component of such applications is extremely important.

AFS and Ponder target large distributed systems. Therefore, the systems chose to be restricted for the sake of efficiency. On the other hand, FARDMS is restricted with respect to scalability in practice due to providing different security models for different hierarchies as well as separating administrative privileges from ordinary ones. We believe FAF scalability is fairly limited due to a possible bottleneck for the single central administrator as well as not being tied to implementation of its authorization predicates by materialized view. These limitations have been avoided in ACAD by meeting the overarching requirement, providing adjustable level of decentralization, and being independent from implementation choices.

Table 6 summarizes the result of this comparison. In summary, ACAD is distinguished from all other four models in five features namely, support of overarching requirement, support for updating data hierarchies, expressive delegation mechanism, support for selective revocation, and administrative flexibilities.

Table 6. Comparison of existing models w.r.t. major requirements

	AFS	FARDMS	FAF	Ponder	ACAD
Overarching Req.	*unsupported*	*unsupported*	*unsupported*	*not addressed*	***supported***
Closed/Open Policy	*only closed*	*possible/ restricted*	*possible*	*possible*	*possible*
Granularity	*very coarse*	*coarse*	*very fine*	*fine*	*very fine*
Exceptions	*limited*	*limited*	*no-limit*	*no-limit*	*no-limit*
Update Hierarchies	*centralized*	*centralized*	*limited*	*not addressed*	***centralized or decentralized***
Active/Passive	*active*	*both possible*	*passive*	*active*	*both possible*
Delegation	*very limited*	*limited*	*very limited*	*very limited*	***flexible***
Selective Revocation	*not applicable*	*unsupported*	*not applicable*	*not applicable*	***supported***
Cascade Revocation	*not applicable*	*supported*	*not applicable*	*not applicable*	***supported***
Policy Neutral	*no*	*no*	*yes*	*yes*	*yes*
Administrative Flexibility	*very limited*	*limited*	*very limited*	*limited*	***flexible***
Scalability	*good*	*limited*	*limited*	*good*	*good*

There are other features that can be considered for the evaluation of access control models, such as *supporting contextual information*, *understandability*, *usability*, *complexity*, and *security properties*. However, we do not discuss such features here due to their lack of influence on the result of the comparison. In particular, all the above systems can be extended to support contextual information; also, discussion on understandability and usability imposes subjective opinions; and, theoretically provable criteria, such as complexity and security properties have not been extensively measured in any of the systems.

8.2 Access Control Space

This section brings together creation time policies and conflict resolution policies (introduced in Chapters 5 and 3, respectively) to introduce the access control space. Creation time policies are our means of initializing the *explicit* access control matrix to support a comprehensive spectrum of administration models, from the autocratic end to the self-governed end. Conflict resolution policies are essential to models that propagate authorizations within hierarchies or support both negative and positive authorizations. As an analogy, creation time policies represent the access control initializations for a network of access control states, similar to a finite state automaton, whereas conflict resolution policies provide different *interpretations* for each state. In ACAD, all reachable states as well as their interpretation conform to the access control policy chosen by the enterprise.

Figure 36 represents such a space, in which one axis maps the creation time policies (Administration) and the other maps the conflict resolution policies (Interpretation). For the sake of comparison, we partition the Administration axis to represent five *classes* of models with respect to the amount of administrative decentralization, namely *no admin*, *single admin*, *group admin*, *hierarchical admin*, and *user admin*. Systems in which there are no metadata updates and each component authority is fixed in the life cycle are from the no admin class. The 4D1-IRIX operating system, in which only one user is allowed to update some data, is an example of the single admin class. Similarly, UNIX, in which a group of users may take the role of super-user, is from the group admin class. Role-based access control models, in which roles often map the organizational hierarchy, are from the hierarchical admin class. Finally, user-managed access control models (introduced in Chapter 7), in which each user potentially can administer various parts of the system, are from the user admin class. For simplicity of discussion, we identify the administration classes with numbers 0, 1, 2, 3, and 4, respectively. It is important to notice the higher the number is, the more flexible class it represents. Hence, models in class 0 (no admin) can be described by models in class 1 (single admin), and so on.

Similarly, we partition the Interpretation axis to represent four *levels* of models with respect to the flexibility of the conflict resolution component, namely *0 rule*, *1 rule*, *2 rules*, and *2+ rules*. Level *0 rule* represents access control models in which conflicts are not possible or allowed; an error is raised in the latter case. Level *1 rule* represents access control models in which conflicts are resolved by one rule only for instance *negative-takes-precedence*. Level *2 rules* represents models in which conflicts are resolved by two rules, for instance *the-most-specific-takes-precedence* and if there is still a conflict then *positive-takes-precedence*. Level *2+ rules* represents models in which the conflict resolution component is not hard wired to the system and can be replaced by any conflict resolution strategy. Similar to the administration classes, we identify the interpretation levels with numbers 0, 1, 2, and 3, respectively; and, the higher the level is, the more variety of interpretations it represents. Hence, models with rank 0 (no conflict resolution) can be described by models with rank 1 (resolving conflicts by 1 rule), and so on.

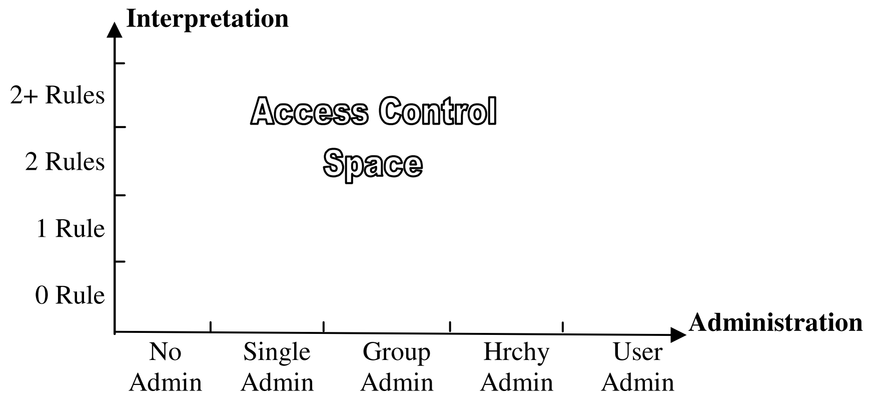


Figure 36. Space of access control administration models.

We suggest to represent the expressivity of each model by a pair of <class, level>, where the class identifies the model administrative capabilities and the level identifies its support of variety of conflict resolution strategies. Therefore, the space of access control administration models provides a visualized mechanism to compare existing models, and leads to a better understanding of their functionalities as well as highlighting their overlaps and differences.

Figure 37 illustrates the position of AFS, FARDMS, FAF, Ponder, and ACAD within the access control space. In terms of administrative capabilities, AFS and FAF are in the administrative Class 1 since they allow only one user to be the security administrator. However, it is clear that both models can be extended to support a group of users, with equal capabilities, and therefore be in Class 2. FARDMS is in the administrative class 2 since it currently supports a group of privileged subjects to take administrative capabilities. Ponder, in Class 3, provides a more general administrative model, with respect to previous models, since it supports a hierarchical administration, for instance appropriate for distribute computer networks. However, it is important to notice that hierarchical administration models cannot express graph-based administration models such as the user managed access control model supported by ACAD in Chapter 7. Therefore, ACAD, in Class 4, is the most flexible model with respect to other existing models. In terms of interpretation variety, AFS is in level 1 since it supports a single rule of *negative-takes-precedence*. FARDMS is richer than AFS since it supports the combination of two rules, the *most-specific-takes-precedenc*e and *negative-takes-precedence*, and is in level 2 then. Moreover, the conflict resolution component in both AFS and FARDMS is hard wired to the rest of model, which cause support of different strategies difficult. However, FAF, Ponder, and ACAD are all in level 3, which reflects that they are independent from the conflict resolution component. FAF, and Ponder explicitly support any combination of three rules of *the-most-specific-takes-precedenc*e, *negative-takes-precedence*, and *positive-takes-precedence*, which is equivalent to two strategy instances. ACAD explicitly supports four rules of *locality*, *majority*, *default* and *preferred* authorizations, which covers 48 conflict resolution strategies including the one supported by other existing models (discussed in Chapter 3).

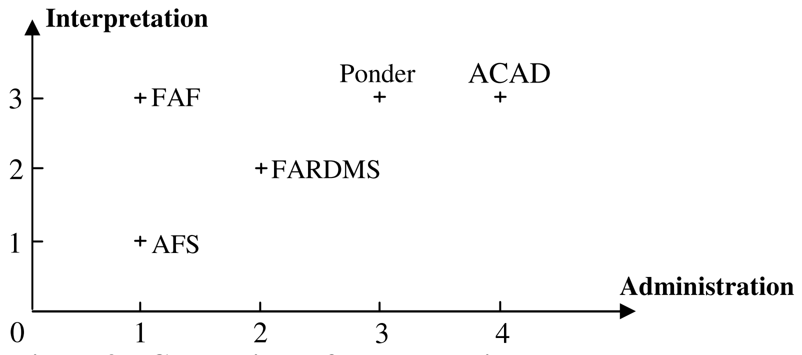


Figure 37. Comparison of the models in access control space.

In summary, the expressivity of AFS, FARDMS, FAF, Ponder, and ACAD can be represented by <1,1>, <2,2>, <1,3>, <3,3>, and <4,3>, respectively. Considering the fact that AFS and FAF are simply extensible to <2,1> and <2,3>, respectively, one can easily conclude that, in terms of administrative capabilities, these models obey the following rule

AFS < FARDM <FAF < Ponder < ACAD

in which < means “can be captured by”. This is illustrated by Venn diagram in Figure 38. Moreover, ACAD is novel in providing mechanisms to set up the system with the desired level of decentralization for each object type and suitable conflict resolution strategy, at configuration time.

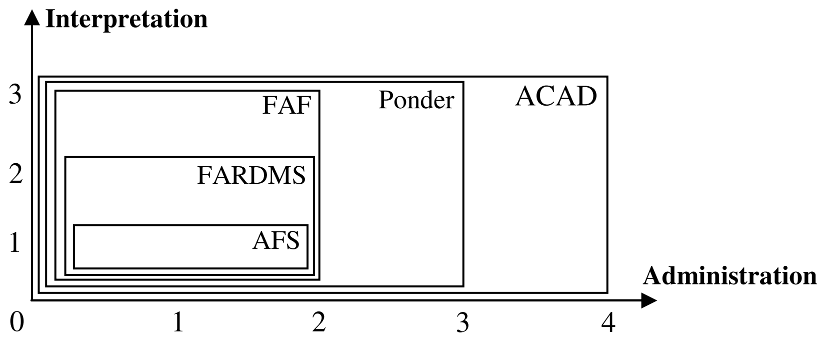


Figure 38. Expressivity of the models in access control space.

8.3 Related Work

There are two recent papers addressing the requirements of access control models [Bertino et al. 1999; Tolone et al. 2005]. Bertino et al., in their authorization proposal for relational databases systems, discuss several protection requirements among which the following influenced this thesis: support for exceptions and strong enforcement, possibility of delegation and retaining control, and support for grouping subjects. Tolone et al. summarize several access control requirements, addressed by different groups in earlier works [Edwards 1996; Jaeger and Prakash 1996; Ferraiolo and Barkley 1997; Bullock 1998], in their proposal for collaboration systems, among which the followings influenced this thesis: scalability, access control granularity, and active/passive mechanism.

Chapter 9

CONCLUSIONS AND FUTURE WORK

This chapter first summarizes the contributions of ACAD in Section 9.1. Then, several related research directions are discussed in Section 9.2.

9.1 Summary of Contributions

The major contribution of this thesis is an **ac**cess control **ad**ministration model with adjustable decentralization (called ACAD), to protect both data and metadata. ACAD is *uniform*: all types of data and metadata are protected with the same mechanism. Moreover, ACAD is *administratively flexible*: it is *adjustable* to set any degree of decentralization of administration for each object, it allows users to *update all hierarchies*, and it provides very *rich mechanisms for delegation and revocation*. Hence, ACAD covers the spectrum of access control administration from the autocratic end to the self-governed end; its mechanism of updating hierarchies is the same as the one for updating access control data; and, it holds the right to delegate a specific authorization and the right to revoke it independently of each other and independently of the authorization itself. It is important to notice that ACAD is unique in terms of these characteristics with respect to other noteworthy models, discussed in Chapter 8. Details of our contributions are as follows.

ACAD is *policy-neutral*, and therefore independent from the conflict resolution component. As a part of this thesis, we have implemented a unified algorithm to support several conflict resolution strategies simultaneously in the presence of sophisticated data hierarchies, which can be used as the conflict resolution component of any system. ACAD is *fine-grained,* and therefore flexible to define authorizations on any level of objects and to administer authorizations and the right to delegate them independently. It is controlled by flexible creation time policies that allow a single system to be deployed in a wide range of application environments.

Furthermore, ACAD supports a wide-ranging set of access control features and is supported by a formal semantics defined operationally by a relational model. Within this framework, we also introduce the *self-governance* property in the context of access control, and show how the model facilitates it. We have shown how ACAD can set up a conflict-free and decentralized access control administration model, called UMAC, in which all users are able to retain complete control over their own data while they are also able to delegate any subset of their rights to other users or user groups. We have also characterized a novel mechanism to constrain access control administration for each object type at object creation time, as a means of adjusting the degree of decentralization when the system is configured.

Finally, we have compared ACAD and several other significant models, namely AFS, FARDMS, FAF, and Ponder, to highlight its important features as well as its expressivity in the space of access control administration models.

9.2 Future Work

There are several directions to extend this work. The details are as follow:

– Define a metric to measure the decentralization degree. There is no formal technique to verify decentralized access control administration models in terms of "the degree of decentralization." Decentralization may increase anarchy, and centralization may cause an administration bottleneck. In other words, decentralization, e.g. in information sharing systems, is a special type of optimization problem in which the degree of decentralization needs to be maximized while keeping the anarchy below a specific amount. Similarly, centralization, e.g. in governments, is an optimization problem in which the degree of centralization needs to be maximized while keeping the administration load below a specific level. Nevertheless, each optimization problem requires a well defined metric. It is important to develop such metrics from which both system buyers and system developers can benefit.

– Define a metric to measure the restrictedness degree. Conflict resolution policies together with propagation policies raise an interesting question: how restricted is the combined system overall? Intuitively, this question addresses the ratio of positive and negative authorizations in the effective access control matrix. We believe that developing such a metric to measure the degree of restrictedness of a system will result in two immediate profits: first, understanding if a given system is closer to closed policy systems or to open policy ones, which consequently has several advantages, including choosing the right data structure for efficient representation; second, we believe such metrics can help in verifying the data availability and safety properties of access control models.

– Develop a data structure that is flexibly adjustable by the system to match the specific configuration of banks, subjects, and objects of ACAD present in a given enterprise. Designing corresponding efficient access control algorithms will be the major contribution of this direction.

– Develop a more flexible delegation mechanism for ACAD. In the current work, grantors delegate their access *privileges* at their wish. However, there are applications in which grantors may delegate an *obligation* or a *responsibility* only if the potential grantee agrees too. Moreover, in the current work, a privilege can be delegated as soon as a grantor delegate it (*1-delegation*) whereas there are applications in which the privilege is granted if a minimum number of grantors, delegate it (*k-delegation*).

– Develop bag semantics for ACAD. There are applications in which the collection of authorizations for a given subject should semantically be a bag rather than a set; in this case, the revocation mechanism must keep track of the path of delegated rights in order to properly cascade the operation.

This thesis provide an adjustable access control administration model, which is distinguished from other noteworthy existing model in terms of comprehensiveness and expressivity (as justified in Table 6, as well as Figures 37 and 38); yet, the formal proofs of security properties (such as safety, accountability, and protection against attacks) as well as practical directions to measure performance criteria (such as complexity) remain open to follow as future work.

Bibliography

AHN, G.J., ZHANG, L., SHIN, D., AND CHU, B. 2003. Authorization management for role-based collaboration. In Proceedings of the 2003 IEEE International Conference on Systems, Man, and Cybernetics, Washington, D.C, USA, October, 2003. 5, (October), 4128-4134.

AL-KAHTANI, M.A., AND SANDHU, R. 2003. Induced role hierarchies with attribute-based RBAC. In Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003), Como, Italy, June 2003. ACM Press, New York, NY, 142-148.

BARKA, E., AND SANDHU, R. 2000a. Framework for role-based delegation model. In Proceedings of the 23rd National Information Systems Security Conference, Baltimore, October 2000, 101–114.

BARKA, E., AND SANDHU, R. 2000b. A role-based delegation model and some extensions. In Proceedings of the 16th Annual Computer Security Application Conference, New Orleans, Louisiana, December 2000. 168-176.

BELL, D., AND LAPADULA, L. 1976. Secure computer system: unified exposition and Multics interpretation. Tech. Rep., ESD-TR-75-306, The MITRE Corp, March 1976.

BERTINO, E., CATANIA, B., GERVASI, V., RAFFAETA, A. 1998. Active-U-Datalog: Integrating active rules in a logical update language. Transactions and Change in Logic Databases. H. Decker, B. Freitag, M. Kifer, A. Voronkov, Eds. Lecture Notes in Computer Science, No.1472, Springer-Verlag, 107-133.

BERTINO, E., JAJODIA, S., AND SAMARATI, P. 1999. A flexible authorization mechanism for relational data management systems. ACM Transactions on Information Systems, 17, 2 (April), 101-140.

BERTINO, E., CATANIA, B., FERRARI, E., AND PERLASCA, P. 2001a. A logical framework for reasoning about access control models. ACM Transactions on Information and System Security (TISSEC), 6, 1 (February), 71-127.

BERTINO, E., CATANIA, B., FERRARI, E., AND PERLASCA, P. 2001b. A logical framework for reasoning about access control models. In Proceedings of the 6th ACM Symposium on Access Control Models and Technologies (SACMAT 2001), Chantilly, Virginia, USA, May 2001. ACM Press, New York, NY, 41-52.

BERTINO, E., BONATTI, P.A., AND FERRARI, E. 2001c. TRBAC: A temporal role-based access control model. ACM Transactions on Information and System Security (TISSEC), 4, 3 (August), 191-223.

BHATTI, R., JOSHI, J.B.D., BERTINO, E., GHAFOOR, A. 2004. X-GTRBAC admin: a decentralized administration model for enterprise wide access control. In Proceedings of the 9th ACM Symposium on Access Control Models and Technologies (SACMAT 2004), Yorktown Heights, New York, USA, June 2004. ACM Press, New York, USA, 78-86.

BHATTI, R., GHAFOOR, A., BERTINO, E., JOSHI, J.B.D. 2005a. X-GTRBAC: an XML-based policy specification framework and architecture for enterprise-wide access control. ACM Transactions on Information and System Security (TISSEC), 8, 2 (May), 187-227.

BHATTI, R., SHAFIQ, B., BERTINO, E., GHAFOOR, A., JOSHI, J.B.D. 2005b. X-GTRBAC admin: A decentralized administration model for enterprise-wide access control **.** ACM Transactions on Information and System Security (TISSEC), 8, 4 (Nov), 388-423.

BIBA, K.J. 1977. Integrity considerations for secure computer systems. Technical Report, ESD-TR-76-372, USAF Electronic Systems Division, April 1977.

BLAZE, M., FEIGENBAUM, J., AND LACY, J. 1996. Decentralized trust management. In Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, USA, May 1996, 164-173.

BOTHA, R.A., AND ELOFF, J.H.P. 2001. Separation of duties for access control enforcement in workflow environments. IBM Systems Journal, 40, 3, 666-682.

BULLOCK, A. 1998. SPACE: Spatial access control for collaborative virtual environments. PhD. Thesis, University of Nottingham.

BURROWS, M., ABADI, M., AND NEEDHAM, R. 1990. A logic of authentication. ACM Transactions on Computer Systems, 8,1 (Feb), 18-36.

CHANDER, A., DEAN, D., AND MITCHELL J.C. 2001. A state-transition model of trust management and access control. In Proceedings of the 14th IEEE Computer Security Foundations Workshop, June 2001. IEEE Computer Society Press, pages 27–43.

CHINAEI, A.H., AND TOMPA, F.Wm. 2005. User-managed access control for health care systems. In Proceedings of the 2nd VLDB Workshop on Secure Data Management, Trondheim, Norway, September 2005. W. JONKER AND M. PETKOVIC, Eds. Springer-Verlag Berlin Heidelberg, 63-72.

CHINAEI, A.H., AND ZHANG, H. 2006. Hybrid authorizations and conflict resolution. In Proceedings of the 2nd VLDB Workshop on Secure Data Management, Seoul, Korea, September 2006. W. JONKER AND M. PETKOVIC, Eds. Springer-Verlag Berlin Heidelberg, 131-145.

CHINAEI, A.H., CHINAEI, H.R., AND TOMPA, F. Wm. 2007. A unified conflict resolution algorithm. In Proceedings of the 2nd VLDB Workshop on Secure Data Management, Seoul, Korea, September 2007. W. JONKER AND M. PETKOVIC, Eds. Springer-Verlag Berlin Heidelberg, 1-17.

CRAMPTON, J. 2002. Administrative scope and role hierarchy operations. In Proceedings of the 7th ACM Symposium on Access Control Models and Technologies (SACMAT 2002), Monterey, California, USA, June 2002. ACM Press, New York, USA, 145-154

CRAMPTON, J., LOIZOU, G. 2003. Administrative scope: A foundation for role-based administrative models. ACM Transactions on Information and System Security (TISSEC), 6, 2(May), 201-231.

CUPPENS, F., CHOLVY, L., SAUREL, C., and CARRERE, J. 1998. Merging security policies: analysis of a practical example. In Proceedings of the 11th Computer Security Foundations Workshop, 1998. 123–136.

DAMIANI, E., VIMERCATI, S.D.C.D., PARABOSCHI, S., and SAMARATI, P. 2002. A fine-grained access control system for XML documents. ACM Transaction on Information and System Security (TISSEC), 5, 2 (May), 169–202.

DAMIANOU, N., DULAY, N., LUPU, E., AND SLOMAN, M. 2001. The Ponder policy specification language. In Proceedings of the Policy Workshop on Distributed Systems and Networks. Bristol, UK, January 2001. Springer-Verlag LNCS 1995, 18-39.

DAMODARAN, S., AND ADAMS, C. 2001. XACML – Summary of use cases. www.oasis-open.org/committees/xacml/repository/draft-xacml-requirements-01.doc.

DENNIS, J., and, VAN, H. 1966. Programming semantics for multiprogrammed computations. Communications of ACM, 9, 3 (March), 143-155.

EDWARDS, W.K. 1996. Policies and roles in collaborative applications. In Proceedings of the 10th ACM Conference on Computer-Supported Cooperative Work. Boston, MA, November 1996. 11-20.

ELLISON, C. 1996. Establishing identity without certification authorities. In Proceedings of the 6th USENIX Security Symposium, San Jose, CA, July 1996. 67-76.

FERRAIOLO, D.F., AND KUHN, D.R. 1992. Role based access control. In Proceedings of the 15th NIST-NCST National Computer Security Conference, Baltimore, MD, October 1992. 554-563.

FERRAIOLO, D.F., BARKLEY, J.F., AND KUHN, D.R. 1999. A role based access control model and reference implementation within a corporate intranet. ACM Transactions on Information and System Security (TISSEC), 2, 1 (February), 34-64.

FERRAIOLO, D.F., SANDHU, R., GAVRILA, S.I., KUHN, D.R., AND CHANDRAMOULI, R. 2001. Proposed NIST standard for role-based access control. ACM Transactions on Information and System Security (TISSEC). 4, 224-274.

FERRAIOLO, D.F., AHN, G.J., CHANDRAMOULI, R., AND GAVRILA, S.I. 2003. The role control center: features and case studies. In Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003), Como, Italy, June 2003. ACM Press, New York, USA, 12-20.

FIROZABADI, B.S., SERGOT M., AND BANDMANN, O. 2001. Using authority certificates to create management structures. In Proceeding of the 9th International Workshop on Security Protocols, Cambridge, UK, April 2001. Springer-Verlag Berlin Heidelberg. 134-145.

FOCARDI, R., AND GORRIERI, R. 1997. Non interference: past, present and future. In Proceedings of DARPA Workshop on Foundations for Secure Mobile Code, California, USA, March 1997.

GARCIA-MOLINA, H., ULLMAN, J.D., WIDOM, J. 2002. Database systems, the complete book. Prentice Hall, Chapter 10.

GOGUEN, J., AND MESEGUER, J. 1983. Security policy and security models. In Proceedings of the 1982 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 1982, IEEE Computer Society Press, 11-20.

GRAHAM, G.S., AND DENNING, P.J. 1972. Protection - principles and practice. In Proceedings of the AFIPS Spring Joint Computer Conference, Montvale, New Jersey, 1972, AFIPS Press, 40, 417-429.

GRIFFITH, P.P., AND WADE, B.W. 1976. An authorization mechanism for a relational database system, ACM Transactions on Database Systems, 1, 3, 242-255, 1976.

HARRISON, M.A., RUZZO, W.L., AND ULLMAN, J.D. 1976. Protection in operating systems. Communications of ACM, 19, 8 (August), 461-471.

HOWARD, J.H., KAZAR, M.L., MENESS, S.G., NICHOLAS, D.A., SATYANARAYANAN, M., SIDEBOTHAM, R.N., AND WEST, M.J. 1988. Scale and performance in a distributed file system. ACM Transactions on Computer Systems, 6,1 (February), 51–81.

JAEGER, T., AND PRAKASH, A. 1996. Requirements of role-based access control for collaborative systems. In Proceedings of the 1st ACM Workshop on Role-based Access Control. Gaithersburg, MD. November 1995. 53–64.

JAEGER, T., AND TIDSWELL, J.E. 2001. Practical safety in flexible access control models. ACM Transactions of Information and System Security (TISSEC), 4, 2 (May), 158 – 190.

JAEGER, T., ZHANG, X., AND CACHEDA, F. 2003. Policy management using access control spaces. ACM Transactions of Information and System Security (TISSEC), 6, 3 (August), 327-364.

JAJODIA, S., SAMARATI, P., SUBRAHMANIAN, V.S., AND BERTINO, E. 1997. A unified framework for enforcing multiple access control policies. In Proceedings of ACM SIGMOD International Conference on Management of Data, Tucson, Arizona, USA, June 1997, 474-485.

JAJODIA, S., SAMARATI, P., SAPINO, M.L., AND SUBRAHMANIAN, V.S. 2001. Flexible support for multiple access control policies. ACM Transactions on Database Systems (TODS), 26, 2 (June), 214 – 260.

JANSEN, W.A. 1998. Inheritance properties of role hierarchies. In Proceedings of the 21st NIST-NCSC National Information Systems Security Conference, Arlington, Virginia, USA, October 1998, 476-485.

JONES, A.K., LIPTON, R.J., AND SNYDER, L. 1976. A linear time algorithm for deciding security. In Proceedings of the 17th IEEE Symposium on Foundations of Computer Science, Houston, Texas, October 1976. IEEE Computer Society Press, 128, 33-41.

JONES, V.E. 1997. Access Control for Client Server Object Databases. Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.

JOSHI, J.B.D., BERTINO, E., LATIF, U., AND GHAFOOR, A. 2001. Generalized Temporal Role Based Access Control Model (GTRBAC) (Part I)– Specification and Modeling. CERIAS TR 2001-47, Purdue University, USA.

JOSHI, J.B.D., BERTINO, E., AND GHAFOOR, A. 2002a. Temporal hierarchies and inheritance semantics for GTRBAC. In Proceedings of the 7th ACM Symposium on Access Control Models and Technologies (SACMAT 2002), Monterey, California, USA, June 2002. ACM Press, New York, NY, 74-83.

JOSHI, J.B.D., BERTINO, E., AND GHAFOOR, A. 2002b. Hybrid role hierarchy for generalized temporal role based access control model. In Proceedings of the 26th International Computer

Software and Applications Conference (COMPSAC 2002), Oxford, England, August 2002. IEEE Computer Society, 951-956.

JOSHI, J.B.D., BERTINO E., SHAFIGH, B., AND GHAFOOR, A. 2003. Dependencies and separation of duty constraints in GTRBAC. In Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003), Como, Italy, June 2003. ACM Press, New York, NY, 51-64.

JOSHI, J.B.D., BERTINO, E., LATIF, U., GHAFOOR, A. 2005. A generalized temporal role-based access control model. IEEE Transactions on Knowledge and Data Engineering, 17, 1 (January), 4-23.

JOSHI, J.B.D., AND BERTINO E., 2006. Fine-grained role-based delegation in presence of the hybrid role hierarchy. In Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT 2006), Tahoe, California, USA , June 2006. ACM Press, New York, NY, 81-90.

KERN, A., SCHAAD, A., AND MOFFETT, J. 2003. An administration concept for the enterprise role-based access control model. In Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003), Como, Italy, June 2003. ACM Press, New York, NY, 3-11.

KOCH, M., MANCINI, L.V., and PARISI-PRESICCE, F. 2002. Conflict detection and resolution in access control specifications. In Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures, Grenoble, France, April 2002. 223–237.

KUDO, M. 2001. Use cases for access control on XML resources. http://www.oasis-open.org/committees/xacml/docs /UseCase.doc.

KUHN, D.R. 1997. Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems. In proceedings of the 2nd ACM Workshop on Role-Based Access Control, New York, USA, November 1997. ACM Press, 6, 7, 23-30.

LAMPSON, B.W. 1971. Protection. In Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems, Princeton, New Jersey, USA, March 1971, 437-443.

LI, N., MITCHELL, J.C., AND WINSBOROUGH, W.H. 2002. Design of a role-based trust-management framework. In Proceedings of the 2002 IEEE Symposium on Security and Privacy.Washington, DC, USA.IEEE Computer Society, 114-130.

LI, N., GROSOF, B.N., AND FEIGENBAUM, J. 2003. Delegation logic: A logic-based Approach to Distributed Authorization. New York University. ACM Transactions of Information and System Security (TISSEC), 6, 1 (February), 128-171.

MAUGHAN, D., SCHERTLER, M., SCHNEIDER, M., AND TURNER, J. 1998. Internet security association and key management protocol (ISAKMP), RFC 2408, November 1998.

MOFFETT, J.D. 1990. Delegation of authority using domain based access rules. PhD Thesis, Department of Computing, Imperial College, University of London.

MIGNET, L., BARBOSA, D., and VELTRI, P. 2003. The XML web: a first study. In Proceedings of the International World Wide Web Conference. Budapest, Hungary, May 2003. 500–510.

MOSES, T., eXtensible Access Control Markup Language Version 2.0, Technical Report, OASIS, February 2005.

OH, S., AND SANDHU, R. 2002. A model for role administration using organization structure. In Proceedings of the 7th ACM Symposium on Access Control Models and Technologies, Monterey, California, USA, June 2002. ACM Press, New York, NY, 155-168.

OH, S., SANDHU, R., AND ZHANG, X. 2006. An effective role administration model using organization structure. ACM Transactions of Information and System Security (TISSEC), 9, 2(May), 113-137.

OSBORN, S., AND GUO, Y. 2000. Modeling users in role-based access control. In Proceedings of the 5th ACM Workshop on Role-Based Access Control, 2000.

PARK, J.S., AND HWANG, J. 2003. Role-based access control for collaborative enterprise in peer-to-peer computing environments. In Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003), Como, Italy, June 2003. ACM Press, New York, NY, 93-99.

RAVICHANDRAN, A., AND YOON, J. 2006. Trust management with delegation in grouped peer-to-peer communities. In Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT 2006), Lake Tahoe, California, USA , June 2006. ACM Press, New York, NY, 71-80.

RIVEST, R.L., AND LAMPSON, B. 1996. A simple distributed security infrastructure. Presented at CRYPTO'96 Rumpsession.

ROSENTHAL, A., AND SCIORE, E. 2001. Administering permissions for distributed data: factoring and automated inference. In *Proceedings of IFIP 11.3 Working Conference on Data and Application Security*, Niagara on the Lake, Ontario, Canada, August 2001. B. Thuraisingham, R. van de Riet, K. Dittrich and Z. Tari, Eds. 215 Kluwer 2002, 91-104.

ROSENTHAL, A., AND SCIORE, E. 2004. Enabling collaborative administration and safety fences: factored privileges in SQL databases. IEEE Data Engineering Bulletin, 27, 1 (March), 42-47.

SALTZER, J. 1974. Protection and the control of information sharing in Multics. Communications of ACM, 17, 7 (July), 388-402.

SANDHU, R. 1993. Lattice-based access control models. IEEE Computer. 26, 11 (November) , 9-19.

SANDHU, R., BHAMIDIPATI, V., AND MUNAWER, Q. 1999. The ARBAC97 model for role-based administration of roles. ACM Transactions of Information and System Security (TISSEC), 2,1(February), 105-135.

SANDHU, R., AND MUNAWER, Q. 1999. The ARBAC99 model for role-based administration of roles. In Proceedings of the 18th Annual Computer Security Applications Conference, Phoenix, Arizona, USA, December 1999. 229-238.

TOLONE, W., AHN, G.J., PAI, T., AND HONG, S.P. 2005. Access control in collaborative systems. ACM Computing Surveys, 37, 1 (March), 29-41.

TRIPUNITARA, M.V., AND LI. N. 2004. Comparing the expressive power of access control models. In Proceedings of the 9th ACM Symposium on Access Control Models and Technologies (SACMAT 2004), Yorktown Heights, New York, USA, June 2004. ACM Press, New York, USA, 62-71.

WANG, H., AND OSBORN, S. 2003. An administrative model for role graphs. In Proceedings of the 17th Annual IFIP WG11.3 Working Conference on Database and Applications Security, Estes Park, Colorado, USA. August 2003. 302-315.

WANG, H., AND OSBORN, S. 2006. Delegation in the role graph model. In Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT 2006), Lake Tahoe, California, USA , June 2006. ACM Press, New York, NY, 91-100.

WANG, L., WIJESEKERA, D., AND JAJODIA, S. 2004. A logic-based framework for attribute based access control. In Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering, Washington DC, USA, October 2004. 45-55.

WEDDE, H.F., AND LISCHKA, M. 2003. Cooperative role-based administration. In Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003), Como, Italy, June 2003. ACM Press, New York, NY, 21-32.

YU, T., SRIVASTAVA, D., LAKSHMANAN, L.V.S., and JAGADISH, H.V. 2002. Compressed accessibility map: efficient access control for XML. In Proceeding of the 28th International Conference on Very Large Data Bases (VLDB '02), Hong Kong, China, August 2002. 478–489.

ZANNONE, N., JAJODIA, S., AND WIJESEKERA, D. 2006. Creating objects in the flexible authorization framework. In Proceedings of the 20th Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec 2006), Sophia Antipolis, France, July 2006. LNCS 4127, Springer-Verlag GmbH, 1-14.

ZHANG, H., ZHANG, N., SALEM, K., AND ZHUO, D. 2005. Compact access control labeling for efficient secure XML query evaluation. In Proceedings of the 2nd International Workshop on XML Schema and Data Management. Tokyo, Japan, April 2005.

ZHANG, L., AHN, G., AND CHU, B. 2002. A role-based delegation framework for healthcare information systems. In Proceedings of the 7th ACM Symposium on Access Control Models and Technologies (SACMAT 2002). Monterey, California, USA, June 2002, 125–134.

ZHANG, L., AHN, G., AND CHU, B. 2003. A rule-based framework for role-based delegation and revocation. ACM Transactions of Information and System Security (TISSEC), 6,3(August), 404-441.

ZHANG, X., OH, S., AND SANDHU, R. 2003. PBDM: A flexible delegation model in RBAC. In Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003), Como, Italy, June 2003. ACM Press, New York, NY, 149-158.