8 Related Work

This chapter compares the view–based approach to access control with the most closely related work. Section 8.1 compares the concept of views as defined in this thesis with earlier definitions of the same term. Section 8.2 presents programming language based approaches to protection. Policy languages are discussed in section 8.3; role–based access control is discussed in section 8.4. Section 8.5 compares the view–based approach with other related work.

8.1 Earlier View Concepts

The term *view* has been used for protection concepts before. None of these previous view concepts supports negative authorizations, however, nor can views be systematically restricted or combined in any of these models.

In the context of object–oriented programming languages, [Hailpern and Ossher, 1990] propose views as a mechanism for specifying multiple different interfaces to objects for protection purposes. Access policies are sets of views, which group callers, targets, and operations. This grouping is not restricted in any way, so neither the objects, nor the clients, nor the operations in a single view must be related or structured in any way. Views are not defined in a declarative language, nor can relationships between views be specified.

Views in [Coulouris and Dollimore, 1994a], [Coulouris and Dollimore, 1994b] are defined as "a subset of the operations on an object that are considered together for the purpose of granting access rights" and used as entries in an access matrix. However, even though the benefits of strong typing in a distributed object system are noted, no typed view language is proposed. An issue identified but not addressed in [Coulouris and Dollimore, 1994b] is that "the correspondence between the operations in user–level views and the operations on programming–level objects must be specified in a notation that remains to be defined". Such a notation is defined in this thesis. Another requirement identified by [Coulouris and Dollimore, 1994b] is for a declarative specification of "changes to the permissions for objects as tasks progress through their various stages," which can be achieved using schemas in our model.

In the Guide system [Hagimont, 1994], a view is "a restriction of a class interface which is stored in the class". Views are managed by a secure kernel and are not a specification language construct. In [Hagimont et al., 1996], an extended interface definition language is proposed that expresses how capabilities are exchanged as arguments of remote object invocations. In [Hagimont et al., 1997], views are a language concept used to describe both a list of allowed

operations and the per–operation exchange policy for capabilities. By recursively annotating parameters of view operations with other view names in interface definitions, protection requirements for arguments can be expressed. At run–time, access rights according to these views are passed implicitly. Structural relations between views and negative rights are not addressed.

Views are also used for protection in relational and object–oriented databases [Scholl et al., 1991]. Their use for access control purposes resembles the use of type abstraction as a protection concept. Unlike database views that can span multiple types, a view in our model is restricted to objects of a single IDL type. Joining views on different IDL types $T_1, ..., T_n$ can, however, be modeled by specifying an additional IDL interface T that extends $T_1, ..., T_n$ and by defining a view on T. Another difference is that database views may define content–specific access controls, e.g., by stating that an attribute may only be read if its value is above a certain threshold. While this is a possible extension to our model, it is not possible in its current form.

8.2 Programming Language Approaches to Protection

Programming language approaches to protection relying on abstract data types have been know since the 1970s. An important approach related to ours is [Jones and Liskov, 1978], which aims at increasing software reliability by enabling compilers to statically prove that program texts are *access–correct*. A program is access–correct if the accesses used in the program comply with the access restrictions in program declarations. Restrictions on accesses to instances of an abstract data type that can be declared in [Jones and Liskov, 1978] are based on rights rather than on the coarser–grained notion of package–local accesses or accesses that originate from within the type's implementation or from subtypes, as in Java [Gosling et al., 1996]. Eiffel [Meyer, 1992] supports finer–grained access restrictions than Java but without rights. In Eiffel, each attribute and operation of a class can be exported to individual other classes, not just to packages or descendants as in Java.

Access correctness relies on strong typing and on the notion of a *qualified type*. A qualified type is a data type plus a set of rights. Access restrictions are declared by specifying qualified types for variables and the formal parameters and return types of operations. At runtime, variables hold capabilities, i.e., an object identifier and a set of rights. A qualified type can thus also be regarded as the type of a capability. The compiler can check that assignments are type–correct in the sense that the value of an expression that is assigned to a variable is of the same data type and has at least as many rights as required by the qualified type of the target variable. Thus, it is possible to prove access correctness by checking that any value passed as an operation parameter has at least the set of rights required by the qualified type of the formal parameter. A procedure is access correct if all assignments are legal and every value returned by it is compatible with the qualified return type. Rights are created together with data objects. When a new object is created, the creation expression returns all rights for accesses to that type.

Like our own approach, [Jones and Liskov, 1978] propose to integrate access rights with type–specific accesses and strong typing. Also, it is possible to statically describe dynamic rights changes. Rights can be gained when objects are created and, through rights amplification, when procedures are entered. Rights are lost when a capability is assigned to a variable which is declared to hold fewer rights than the capability's type. The access policies that can be expressed in this model cannot refer to roles as principals, however. Access rights are granted to individual variables. Also, access policies are part of the application and not described in a separate declarative language. Another difference is that there are no other structural relations between qualified types than assignment compatibility, i.e., it is not possible to combine types or extend them. Moreover, it is not possible to explicitly describe denials or to dynamically extend or restrict the rights of a running program through administrator activities.

Aspect–Oriented Programming

In contrast to [Jones and Liskov, 1978], *Aspect–Oriented Programming* (AOP) [Kiczales et al., 1997] relies on separate *aspect languages* for the specification of non–functional aspects of applications. These languages are processed by a tool called *aspect weaver* which generates code for aspects such as concurrency and distribution, and performs the integration of the functional code with the generated aspect code. The approach taken by AOP focuses on software development and emphasizes reuse and modularity by separating functional from non-functional code. Languages describing non–functional aspects can also be regarded as describing policies that are later to be interpreted by aspect mechanisms, but there is currently no aspect language in AOP that addresses access control. VPL can be seen as such a language.

CACL

[Richardson et al., 1992] propose an ACL–based protection scheme called CACL for object–oriented programming environments or database systems. CACL supports discretionary access control where owners define ACLs for objects. Unlike in [Jones and Liskov, 1978], access rights are not statically defined so accesses are checked dynamically. To control discretionary transfer of ownership, the former owner of an object remains the object's *method principal* until the new owner accepts ownership and complete responsibility for the object by becoming the new method principal, preferably after inspecting the object's implementation and verifying that the object is not a Trojan Horse. Objects always execute on behalf of their designated method principal, so until the new owner accepts to also become the new method principal, the object's methods will still execute on behalf of the former owner.

The CACL model relies on a trusted runtime system and a trusted compiler and is not immediately applicable to distributed systems. Moreover, access policies are not explicitly represented but embedded within the application code. There are no language means to specify denials, dynamic rights changes, and structural relations between access rights.

The Java 2 Access Model

A different approach to securing a program running in a single address space is taken by the Java 2 security model [Gong et al., 1997]. This approach relies on dynamic access checks rather than static verification and focuses on providing flexible protection domains for application code assembled from multiple, potentially untrusted sources. To ensure that untrusted code can be restricted, the application code is partitioned into separate protection domains based on the code source. Policies grant access permissions to domains and thus to code sources, which are the principals in this model. Java supports the checking of digital signatures on code so that principals can be authenticated. Policies can be specified either within a program by using a special API, or using external textual descriptions.

Accesses to resources are controlled by inserting statements into the code that check for permissions. The current principal and the protection domain is determined by inspecting the call stack of the current thread of execution. Permissions are generic and need not correspond to the attempted access in any way, but it is possible to structure permissions by defining subtypes of the `java.security.Permission` class which is the base class for all permissions. Still, there is no static typing of access rights that would ensure that only applicable permissions are checked for a given access. Dynamic rights changes are based on the concept of rights amplification [Wulf et al., 1974] and can occur when the execution passes the boundary to a more privileged protection domain. Denials cannot be explicitly expressed.

This code–based access model is extended to also support principal–based authorizations with the Java Authentication and Authorization Standard (JAAS) [Lai et al., 1999], [Sun Microsystems, 2001]. JAAS provides a pluggable architecture for authentication modules and extends the policy language such that the permissions described above can also be granted to principals. Principals are abstract entities that are represented by names and are associated with the running program as a result of authentication. The JAAS framework supports hierarchies of principals, so it is possible to use JAAS for role–based access control. The JAAS file–based policy implementation can be replaced, so it is possible to integrate a VPL–based policy module, for example. However, permissions must be represented by objects which are checked by explicitly invoking a Java SecurityManager object. Thus, the application code is still mixed with enforcement code. JAAS can be used by remotely accessible services to authenticate a caller using a suitable authentication module. The service can then act on the caller's behalf when accessing protected local resources. The JAAS framework could thus be used for an implementation of the CORBA security mechanisms.

Confined Types

The Java 2 access model controls access to Java objects using runtime checks placed at the location in the code where an access is made. This technique does not protect against the leaking of object references to untrusted code where no such access checks are placed. This is essentially a problem of controlling aliasing because aliases can provide alternative and uncontrolled access paths to objects. [Vitek and Bokowski, 2001] propose a language extension

called *confined types* that statically controls that references to sensitive objects are not leaked to untrusted code outside the protection domain.

Here, protection domains are defined to be packages. The approach defines a new Java keyword with which classes can be declared as confined. Additionally, a number of syntactic restrictions are defined which must be enforced by an extended compiler and ensure that no references to objects of a confined type can leave the package. No additional dynamic checks are necessary. This approach does not define language means to declaratively express access policies on a per method basis but rather complements the Java 2 access model.

8.3 Policy Languages

In the context of policy–based management, a number of general–purpose policy languages have been proposed, e.g. [Sloman and Twidle, 1994], [Koch et al., 1996], [Tu et al., 1997]. These languages generally focus on the definition of management tasks and authorization policies. They are not, however, suitable for use by application developers, who have to provide an initial policy design. The management information model underlying these languages typically does not map easily to the application data model.

[Koch et al., 1996] propose the refinement of policies from general requirements down to an operational level. The policy definition language can express obligations for managers as well as permission or denials for operations. Because policy rules are triggered by events, it is possible to describe dynamic rights changes. Policy rules cannot be combined or aggregated and do not make use of object typing.

Ponder

Ponder [Damianou et al., 2001] is a declarative language designed for both management and security policies. Ponder is much more general in scope than VPL and incorporates its own event language. Since it is independent of any particular enforcement mechanisms, Ponder policies are not operational policies in the sense used here or in [Koch et al., 1996] but must eventually be translated into mechanism–specific formats.

A policy in Ponder is a single rule rather than a set of related rules. Ponder can express both positive and negative authorization rules and can statically detect conflicts between these modalities. The language supports policy types, which are parameterized templates for the creation of policy instances that support factoring out common policy elements. To support the combination of individual rules into larger sets, Ponder defines group and role concepts. A group is a set of policies that has some inherent semantic relationship while a role is a set of policies that are related to a position within an organisation. Ponder also supports inheritance between policy types, but this notion is defined purely syntactically. Other concepts supported by Ponder include meta–policies, delegation policies, constraints, content–specific filters, and scripting.

Because of the lack of a formal meta–model the semantics of most concepts is not precisely defined and it is not clear how these concepts map to the different enforcement mechanisms. While Ponder claims strong typing and static detection of specification errors, it is also unclear which classes of errors can actually be detected. Ponder is therefore not suitable for developers who need to define fine–grained access restrictions on application–level objects.

ASL

A general framework for defining arbitrary access control policies is proposed in [Jajodia et al., 1997] where policies are formulated as a set of rules in a logic–based language called *Authorization Specification Language* (ASL). This model leaves open design decisions about how implicit authorizations are derived, how rights propagate in groups, which conflict resolution strategies are used and how priorities are employed. Rules for these questions have to be defined first as part of a policy library. The data model for protected objects is also left open and has to be described separately. The protection state is extended with a history component that logs all accesses as facts in a database in order to enable state–based policies like *Chinese Wall*. This model exhibits a more complex concrete syntax than ours, policy specifications are less structured.

Trust Management

In [Blaze et al., 1996], a trust management system called PolicyMaker is presented that takes a unified approach to the problem of describing policies, credentials, and trust relationships in large–scale networked systems that cross organizational boundaries. The main proposition of this approach is that authorization should not be based on access identities but on more general trust relationships expressed through certificate chains. Moreover, decision rules should be fully programmable and evaluated at runtime rather than predefined in ACLs. For this purpose, PolicyMaker contains a generic skeleton language for *policy assertions*. Policy assertions describe what a given public key is trusted to do, which is expressed in a *filter* that is part of the assertion. In principle, filters can be programmed in any interpreted language. PolicyMaker directly supports a regular expression language, a safe version of the AWK pattern matching language, and a special macro language. Applications perform access checks themselves by querying the policy database which then interprets filter programs to make access decisions.

An important aspect of the PolicyMaker approach is that filters assign authorizations directly to keys, which must be known at the time filters are created. The only notion of a principal is thus that of a key holder. This supports the flexible definition of interesting relationships between keys, but it does not support imposing structure on policy specifications. Semantic notions such as roles would need to be modeled separately. Aggregational concepts that cater for scalability in large systems are not provided. Moreover, operation requests are uninterpreted strings, and type checking of policy statements against targets is not supported.

The Simple Public Key Infrastructure (SPKI) [Ellison et al., 1999a] takes a similar ap-

proach. However, access decision rules are not fully programmable. Rather, certificates directly contain authorization tokens. SPKI adds a local name space concept to the notion of keys as global identifiers. While this certificate–based approach supports delegation in a manner similar to capabilities, it results in a fully distributed representation of access policies, which is undesirable as it does not allow centralized management of policies.

While trust management and SPKI aim to integrate authentication and access control, [Herzberg et al., 2000] propose a separate *trust establishment* component as a complement to a role–based access control component which is eventually responsible for making access decisions. The central idea of trust establishment is to "assign roles to strangers", i.e., to use a specific policy that describes how potentially unknown and untrusted subjects can be mapped to roles that are accepted by the RBAC component. For this purpose, [Herzberg et al., 2000] define a Trust Policy Language that expresses rules how the subjects of X.503 certificates can be mapped to roles based on other certificates, e.g., recommendations from trusted sources. This approach is not directly related to the view–based access model but could be used as an extension to the role server component in the VPL runtime infrastructure that was presented in chapter 7.

Adage

The Adage toolkit [Zurko et al., 1999], [Simon and Zurko, 1997] is related to our approach in a number of respects. First, usability is one of its main design goals, which corresponds to our aim to enhance access control manageability. Special emphasis is placed on the design of the human–computer interface. Second, Adage builds on a role–based access model that supports RBAC_3. As in our approach, users are represented as actors, although these actors do not directly map to roles in Adage. Third, it supports a dedicated authorization language, and fourth, it is designed as an authorization service for distributed applications and actually uses CORBA as the internal communication mechanism between Adage clients and servers.

The differences between our approach and Adage are both conceptual and architectural. The Adage authorization language (AL) is generic in the sense that it supports arbitrary applications and data models. AL supports a number of different separation of duty constraints and is generally more expressive than VPL. Because there is no strongly typed data model the language interpreter can only perform superficial consistency checks and cannot determine whether the authorization descriptions actually match the protected resources. The authorization language is derived from the Tcl scripting language and has a number of commands to modify an authorization database, so it is not a declarative language. The main architectural difference between our implementation and Adage is that access decisions in Adage are made by a centralized authorization decision service. This contrasts with our approach of central administration but decentralized, i.e., local enforcement, which generally delivers better performance.

8.4 Role–Based and Task–Based Models

View–based access control as introduced in this thesis relies on roles. Roles were defined in chapter 4 as "a logical function of an initiator in the interaction with one or more targets." This behavioral definition contrasts with, e.g., the general reference role models in [Sandhu et al., 1996] and [Nyanchama and Osborn, 1999], which provide more abstract, structural definitions. In [Sandhu et al., 1996], roles are defined as "a collection of users on one side and a collection of permissions on the other. The role serves as an intermediary to bring these two collections together." These intermediaries themselves are abstract entities so roles are modeled as a given set and interpreted only through their relationships to users and permissions. The definition of roles in [Nyanchama and Osborn, 1999] is slightly less abstract as roles are defined directly in terms of privileges, but it also does not assign an inherent semantics to roles.

As in the $RBAC_3$ role model in [Sandhu et al., 1996], the role model used here relies on role hierarchies and constraints. In both models, a role hierarchy is a partial order on roles, and roles inherit their super roles' authorizations. In [Sandhu et al., 1996], authorizations are always permissions, so substitutability of a role for its super roles is implicit with regard to allowed accesses. In our model, behavioral substitutability is explicitly defined as the semantics of role inheritance. In [Nyanchama and Osborn, 1999], role inheritance is directly expressed in terms of permissions.

A role–based approach very similar to ours is Napoleon [Thomsen et al., 1998], which also recognizes the need for collaboration between application developers and security administrators. [Thomsen et al., 1998] propose a framework that provides separate layers for the specification of access policies. At the lowest level, objects need to be grouped for management purposes, but no domain concept is defined. At the next higher level, *object handles* are defined as a way of grouping access rights for operations. Object handles are created with the assistance of graphical tools and correspond to views in VPL, but these handles do not support the definition of constraints. Denials cannot be expressed; delegation, discretionary, and implicit assignments of views are not addressed. Roles are called *application keys* for application–specific roles and *enterprise keys* for application–independent roles and need to be mapped to each other at deployment time. The layered approach suggests a temporal order and functional dependencies between the tasks involved, but this is actually not the case since object management does not predate the definition of object handles, nor does the definition of enterprise "keys" happen only after applications are installed.

The general RBAC role models do not provide specific support for workflows that are defined in terms of ordered, well–defined tasks. While roles in RBAC are sufficiently general to model task–specific aspects of principals, there is no mechanism to automatically assign and remove task–specific authorizations to and from principals participating in a workflow, i.e., for "synchronizing the authorization flow with the workflow" [Atluri and Huang, 1996]. For this purpose, [Atluri and Huang, 1996] introduce authorization templates with which task–specific permissions can be specified that are only assigned to a subject during a time interval that is

defined in the workflow definition. An authorization template is a triplet (s, T, p), where s is a subject, T is an object type, and p is a permission. The authorization template is attached to a task and used to dynamically grant the permission p to s on an object of type T during the time the task is active. Authorization templates are similar to views in that they are statically defined and contain type constraints. However, they do not group rights or permit the specification of denials. Also, there are no relationships between them, such as extension or dependency relationships. The mechanism for specifying the period during which authorizations can be used is less general than schemas because these periods are bounded. Thus, for every implicit assignment there will always be a matching removal. Discretionary access control is not part of this model.

[Thomas and Sandhu, 1997] state a similar motivation as in [Atluri and Huang, 1996] and propose the concept of an *authorization step* for defining rules for discretionary assignments of authorizations. These assignments have to be performed explicitly, however. [Coulouris et al., 1998] propose the concept of *task templates* to statically assemble roles, access rights, and object categories. Object categories are groupings of objects that are supposed to be "user–level" rather than programming–level objects. When a task is instantiated, users must be assigned to roles and objects to categories. Access rights are untyped, however, and there is no mechanism to automatically instantiate or terminate tasks, so dynamic access rights changes are still managed explicitly. Neither of the approaches in this section defines a declarative policy language for design and management purposes.

8.5 Other Related Work

A general work on concepts for object–oriented specifications of access rights is [Brüggemann, 1997], which explicitly introduces operation classes as a category in addition to the usual subject and object classes. Views can be regarded as an extension of operation classes. The model can express both permissions and denials and uses a more general and more complex system of priorities, which can be any integer value. The model does not provide means to describe dynamic rights changes, conditional rights cannot depend on the presence of other rights. A concrete language syntax is not defined.

[Baldwin, 1990] argues that grouping privileges increases manageability of access rights and proposes *Named protection domains* (NPDs) to enhance support for security management in relational databases. NPDs are management abstractions that group privileges on objects and form a privilege graph that contains paths from privileges to users. Named protection domains inherently group not only privileges but also users and are thus very similar to the RBAC notion of roles. Similar to the notion of protection domains in operating systems, however, only one NPD can be active at any time for a given user. As with roles, it is possible to define hierarchies of NPDs. Our approach is similar, but more fine–grained and more modular: Views describe authorizations on individual objects and combine with more appropriate management concepts for users and objects, viz. roles and domains. We believe that protection domains are not applicable to the richer data models of distributed object systems.

Products and Standards

An existing management product that supports security management in CORBA environments is [Tivoli, 2001], which comprises a comprehensive suite of tools. Tivoli does not specifically address access control management at the level of application objects, and provides no separate specification language. The CORBA security service product by [Adiron, 2000] does provide an access control language, but this language is not object–oriented and limited to the restricted standard model of access control in CORBA.

The use of descriptor files that are processed by deployment tools is common in environments such as EJB [Sun Microsystems, 2000] or the CORBA Component Model (CCM) [OMG, 1999c], both of which also support the expression of simple access policies in descriptors. Both descriptor languages do not provide adequate management abstractions, however, and only support access control decisions at the granularity of types, not individual objects.