4 View–Based Access Control

This chapter presents and discusses a declarative access control language designed to meet the manageability requirements introduced in the previous chapters. The language builds on the earlier publications [Brose, 1999], [Brose and Löhr, 1999], [Brose, 2000].

Section 4.1 presents the concepts and syntax of the View Policy Language (VPL), which is used to specify and manage access control in CORBA environments. The main contributions of this language are the introduction of views and schemas as language concepts and the definition of model constraints. These concepts and the access control model underlying the language are discussed and explained in more detail in section 4.2. A formalization of this access control model is given separately in chapter 5.

4.1 The View Policy Language

This section introduces the View Policy Language (VPL) with which policy designers define an application policy. This policy is delivered and deployed together with the application. The subsequent management of the access policy in the target environment uses the same abstractions. This section is intended to give an informal introduction to the language and its syntax, details are discussed in section 4.2. The complete grammar of the language can be found in appendix A.

4.1.1 Policy

VPL policies contain role declarations as well as view definitions and schemas. These concepts are introduced and discussed in subsequent sections. Figure 4.1 shows a simple example policy that gives a first impression of the language. In VPL, access control policies are introduced by the keyword **`policy`**. The policy is defined for a two–dimensional grid object and defines two roles `ValueReader` and `ValueAdmin`. The access operations to determine the size of the grid are width() and height(), the operations set() and get() access grid cells. The example policy defines authorizations in two views, `Getting` and `Setting`, which are initially assigned to the two roles `ValueReader` and `ValueAdmin`.

```
policy Grid
{
   roles
      ValueReader holds Getting on Grid
      ValueAdmin holds Setting on Grid

   view Getting controls Grid
   {
      allow
         height
         width
         get
   }

   view Setting:  Getting
      restricted_to ValueAdmin
   {
      allow
         set
   }
}
```

Figure 4.1: An example policy.

4.1.2 Roles

Roles are the language concept to represent principals and correspond directly to UML actors. The concrete VPL syntax for declaring a role in a policy definition is a **`roles`** clause as illustrated in figure 4.1. Another example is shown in figure 4.2, which defines roles such as lecturer, student, or examiner for a hypothetical policy in a university setting and for a policy in a publishing house.

Roles clauses declare role names and optionally role constraints and initial authorizations. The notation `Head:  Examiner` in figure 4.2 means that the *Head* role is a sub role of the *Examiner* role, i.e., its function is a specialization of the *Examiner* role. This implies that the sub role inherits the super role's authorizations. A role can have more than one super role. A roles clause can also assign views to a role using the keyword **`holds`**. The assignment of this view on the extension of the object type listed after the keyword **`on`** is performed when the policy is initially deployed in a system; during the lifetime of a policy in a running system these assignments are subject to change.

A maximum cardinality constraint is defined with the keyword **`maxcard`**, which is used here to restrict the *President* role to a single member subject. The analog keyword **`mincard`** can be used to specify the minimum number of subjects that must be assigned to a role. The keyword **`excludes`** denotes a mutual exclusion between roles, whereas **`requires`** is used

```
policy University {
    roles
        Examiner holds Examining
        Head:  Examiner // head is a sub role of examiner
        President maxcard 1
        Lecturer
        Candidate excludes Examiner
}

policy Publisher {
    roles
        Staff
        Author
        Secretary:  Staff
        Assistant:  Staff requires Secretary
        Reviewer:  Staff
        Editor:  Staff
        Manager:  Staff
}
```

Figure 4.2: Role declarations.

to define a prerequisite constraint. Mutual exclusion is a symmetric relation between roles, but it is not required that both roles' clauses mention the constraint. In the example, no subject may be assigned to both the *Candidate* and the *Examiner* role, and *Assistants* must have previously been assigned to the *Secretary* role. These constraints are explained in more detail in section 4.2.3.2.

4.1.3 Views

The main contribution of the model presented here is the language concept of a *view* with which policies can be written and understood in terms of coherent sets of related authorizations. A view is a named set of access rights, which are either permissions or denials for operations on objects. While access decisions are made based on individual rights, views are the units of description and authorization assignment. Figure 4.3 shows an example of a view definition in VPL that defines rights for accesses to document objects.

Views are defined as authorizations for objects of a single IDL interface, which is referenced in the **`controls`** clause of the view definition. In the example, the view `Reading` controls the IDL type Document, which is defined in figure 4.4. This means that this view can only be assigned on objects of this type or one of its subtypes. The type in the **`controls`** clause is called the *controlled type*. Permissions are listed after the keyword **`allow`**, denials would be introduced by **`deny`**. In the example, only operations to read the document and to find words within the document are allowed. A view can be restricted to certain roles so

```
view Reading
   controls Document
   restricted_to Staff, Author
{
   allow
      read
      find
}
```

Figure 4.3: A view definition.

```
interface Document {
    readonly attribute string title;
    void read(out string text);
    void write(in string text);
    void append(in string text);
    void annotate(in long where, in string text);
    void insert(in string text, in long where);
    void delete(in long from, in long to);
    void find(in string text);
};
```

Figure 4.4: The Document interface.

that it is not possible to assign the view to any roles except the ones explicitly listed in the **`restricted_to`** clause or their subroles. The `Reading` view can only be assigned to the roles `Staff`, `Author`, and their subroles.

The access model underlying VPL is an access matrix with roles as rows, objects as columns, and views in matrix entries. Because of the emphasis on type–specific rights there is an apparent similarity between views and capabilities [Dennis and Horn, 1966] [Wulf et al., 1974], [Linden, 1976], [Levy, 1984], [Tanenbaum et al., 1986]. However, views are not capabilities. First, a view does not reference an object but only describes access rights; second, views will generally not be implemented in a distributed fashion — an ACL representation or a centralized representation are in fact more likely. Third, capabilities are runtime entities that have no static language definition and do not support the definition of structural relationships between them, such as extension or requirements.

4.1.3.1 View extension

Like object types, views may be related through extension so that definitions can be reused in views intended for more specific access situations. The intuition here is that an extending view allows at least as much as the view that is extended, but that it might be assignable only to a

subset of the objects and roles to which the base view is assignable.

An extending view inherits all its base view's rights — both permissions and denials — and may also add new rights. These added rights can only *increase* the permissions in the view. It is not possible to declare any denials in an extending view. The semantics of view extension is thus one of monotonically adding permissions, just like interface inheritance adds operations to interfaces and never removes them. As a result, whenever a principal holds both the base view and an extending view on a given object, the extending view is substitutable for the base view. For an access decision function, this means that only the most derived views, i.e., the views farthest down the extension hierarchy, need to be checked for access permissions.

```
view Appending                    view Updating:  Appending {
   controls Document                 allow
   restricted_to Staff                  delete
{                                       insert
   allow                          }
      append
}
```

Figure 4.5: View extension.

In VPL, extension is expressed by listing a set of base view names after a colon. In the example in figure 4.5, the view `Updating` extends `Appending`, so it inherits the permission for the operation `append` and additionally permits the operations listed in its own definition. View extension also has an influence on the controlled type and the role restriction of the extending view. Note that `Updating` does not have either an explicit **`controls`** or **`restricted_to`** clause — both are implicitly defined through extension and are simply the same as in the extended view `Appending`. However, an extending view may redefine both clauses by narrowing the controlled object type or the role restriction. Both these kinds of narrowing make the view "less assignable", i.e., they serve to restrict the number of objects and roles that it may be assigned to:

Narrowing the controlled type means defining a controlled type that is a subtype of the controlled type in the extended view. Effectively, this means that the extending view is assignable on a smaller set of objects because the potential target objects must be members of the extension of the controlled type, and this type extension is smaller for subtypes.

Narrowing the role restriction means that each role in the more restricted role set must be a subrole of the roles in the role set of the base view's role restriction or the new role set must be a strict subset of the role set of the base view. A combination is also possible: the narrowed role set can be both smaller than the original one, and some of the roles may be subroles of roles in the original role set. In all these cases, the new role constraint is more restrictive because fewer roles qualify as recipients of this view.

In the case of multiple base views, the controlled object type must be a common subtype of all the controlled types in the base view so that, when compared with all its base views,

the controlled type of the extending view is always more specialized. In the case of extending multiple base views, it is possible to infer the controlled type of the extending view automatically by determining the most general type that is a common subtype of all views' controlled types or rejecting the definition if no such type exists. However, policy specifications would be less intelligible with only implicitly represented controlled types, so we require that they are explicitly listed in the view definition.

With multiple base views, the role restriction of an extending view must also be at least as restrictive as each of the base views' role restrictions. With the same justification as for controlled types, the role restriction must be defined explicitly.

4.1.4 Implicit Authorizations, Denials, and Conflict Resolution

An *implicit authorization* [Rabitti et al., 1991] is one that is implied by other authorizations whenever a grouping construct is used for assignment. For example, if a view v on an object is assigned to a role r, this implies assigning v to all subroles of r. While it is more convenient to specify general access rules implicitly than to assign each of the implied authorizations individually, it must also be possible to express exceptions to general assignments, e.g., that one particular role is denied one particular operation on an object.

4.1.4.1 Denials

Because the absence of a permission cannot be used to override an existing permission, it is necessary to define a means by which *negative authorizations* [Stiegler, 1979] or denials can be specified. Denials are not only required to express exceptions, they can also be used to explicitly specify constraints in an access policy. Consider a high–level policy that states that "students must not have access to color laser printers". If an access policy does not contain any permissions that would allow students such an access, it complies with the high–level policy.[1] An explicit representation of this statement in the access policy in the form of a denial is thus redundant, but it has two positive effects. First, it is easier to reconstruct the overall intention of the higher–level policy from a more explicit lower–level representation. Second, the denial can be interpreted as an additional constraint on access model configurations and thus help detecting inadvertent violations of the original policy, which could happen through the addition of views by uninformed administrators. This is not compatible with a policy that generally permits conflicts, however, but a variation of this model could require that policies are entirely conflict–free and do not even contain exceptions. Such a policy would reject the assignment of a view to a principal if the assignment contains a conflict with views that are already present, which would be detected by checking the additional constraint. In the model presented here such an inadvertent policy violation would only be detected if the resulting conflict is not statically guaranteed to be resolvable according to the rules in the next subsection.

[1] Assuming it is a *closed* policy, where accesses that are not allowed are forbidden.

4.1.4.2 Conflicts and Priorities

If it is possible to define both permissions and denials, conflicts can arise. Some of these conflicts, however, arise because of incomplete or contradictory policy designs rather than as a consequence of deliberately specified exceptions. Because exceptions are legitimate and add to the expressiveness of the language, it is necessary to define which cases are regarded as inconsistencies and how they are detected and treated. For this purpose, we describe a strategy that determines whether, in a given conflict situation, the denial or the permission takes precedence. This resolution strategy should be sufficiently intuitive for policy designers to be practically useful. Situations that cannot be resolved by this strategy are regarded as inconsistent and must be statically detected and rejected.

The conflict resolution strategy relies on the extension relation between views and on explicit priorities in view definitions. Priorities in our model can only take one of two values: strong or weak. As in [Rabitti et al., 1991], the intention of marking a right as "strong" is that it should never be possible for another right to override the strong right in case of conflicts. Marking an explicit denial for student laser printer access as strong guarantees that no permission can override the denial, so the corresponding policy statement cannot be violated, regardless of other views that might need to be evaluated for runtime conflict resolution.

```
view BaseView controls T
{
  allow
     op_1
     op_2
  deny
     strong op_3
     op_4
}
```

```
view DerivedView:  BaseView
{
   allow
      op_4
      strong op_3 // incorrect
}
```

Figure 4.6: Denials and explicit priorities.

As an example for explicit priorities, consider figure 4.6. In the definition of the view `BaseView`, the keyword **`strong`** marks the denial for operation `op_3` and the permission for the same operation in `DerivedView` as "strong"; all remaining rights in both views are weak. To control how extending views may redefine inherited rights, we add a restriction to the definition of extending views: An extending view may only redefine weak rights. Strong rights may not be redefined, so the definition of `DerivedView` in figure 4.6 is found to be incorrect because the strong denial of `op_3` in `BaseView` cannot be redefined.

Conflicts between a permission and a denial for an operation `op` can arise if both apply to the same object. Because a single view is free of conflicts, this situation requires that a principal holds two views `A` and `B` on the same object, as illustrated in figure 4.7. This is possible only in two cases, both of which require that the object types `S` and `T` are either equal or related.

```
view A: C controls T {          view B: D controls S {
   allow                           // inherits a denial for
      op                           // op from its ancestors
}                               }
```

Figure 4.7: Potentially conflicting views.

Conflict resolution for related views

The first case is that the views A and B are related by extension. Two views are related by extension only if one view is directly on the path along the extension hierarchy from the other view to the root of the hierarchy. Otherwise, these views are unrelated even if they have a common ancestor. In figure 4.7, A is related to B if A's ancestor C is a subview of B or if B's ancestor D is a subview of A, but it is not related to B if C extends D.

In the case of related views, the more derived view takes precedence, so if A is more derived, the conflict would be resolved in favor of the permission defined in A, which simply overrides the denial in its indirect base view B. Because derived views may not contain denials, an overriding right can only be a permission.

Conflict resolution for unrelated views

In the second case the conflicting views are not related and the conflict is introduced through the polymorphism supported by the object–oriented data model: a view defined to control a type can also be assigned on objects of a subtype, for which further views may exist that need not be related to the views for the supertype. Here, conflict resolution can rely on priorities only: the priorities of the permission and the denial are evaluated and the stronger right takes precedence. To guarantee that such a resolution is always possible at runtime, situations where both rights' priorities are equal must be either be excluded or require special treatment. Figure 4.8 illustrates a potential conflict between two strong rights in unrelated views.

```
    view A controls T {          view B controls T {
       allow                        deny
          strong op                    strong op
    }                            }
```

Figure 4.8: Conflict between strong rights.

It can be argued that both cases, conflicts between two weak or two strong rights, should be excluded by a language rule. However, a static type checker can only detect *potentially* conflicting definitions and will thus exclude more cases than actually necessary. In order to be less restrictive, we allow conflicts between weak rights in unrelated views and simply let the

denial take precedence, which will err on the conservative side if the conflict actually arises at runtime. Such an approach is not possible for conflicts between two strong rights because it would violate the semantics of **`strong`**. Consequently, potential conflicts between strong rights must be detected and rejected by static analysis of view definitions.

For the data model defined by OMG IDL, it is possible to statically detect view definitions with potentially inconsistent rights definitions. A type checker can reject specifications or emit warnings if two potentially conflicting definitions are both strong. For two unrelated views, runtime conflicts between rights definitions are only possible if their controlled object types are either equal, as in figure 4.8, or related by inheritance. This is the only situation in which two identical operation names can refer to the same operation in an IDL interface. A type checker can detect such a situation and reject the policy specification, thus guaranteeing the semantics of strong priorities.

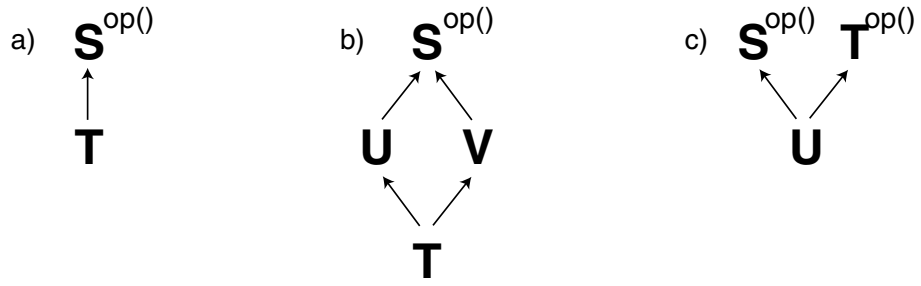

Figure 4.9: Types and the inheritance of operations.

Note that this situation can be detected because of a restriction inherent in the data model, which prevents interfaces inheriting operations with the same name from different, unrelated supertypes: case c) in figure 4.9 is illegal in IDL. This case is problematic because two unrelated views for the two types S and T that define a strong permission and a strong denial for op() could be assigned on the same object of type U because the two types' extensions overlap in U. This would result in a violation of the semantics of `strong` that cannot be caught statically — unless the IDL type hierarchy is checked for occurrences of c). Relying on such a check is also problematic because it means that even simple extensions of the type hierarchy that do not modify any existing type definitions can break the VPL policy and would necessitate recompilation of all policies. Since this is not practical, the exclusion of case c) is necessary to statically guarantee that a strong right cannot be overridden.

This restriction is not present in other languages, however. In Java, e.g., a class can inherit the definition of an operation `op()` from two unrelated interfaces. This is legal because multiple inheritance is allowed for interfaces in Java, so there can only be one implementation of that operation to which calls are dispatched. If Java was used as the target object model for VPL, case c) in figure 4.9 would be legal and a type checker would not be able to guarantee that a strong right cannot be overridden.

4.1.5 Dynamic rights changes

A system's protection state is usually not constant. Objects, roles, and groups are added to or deleted from a system and views may be assigned or removed for administrative reasons, for the purposes of delegation of responsibility, or as part of an application–specific security policy. A general distinction can be made between *explicit* operations on the protection state, which only specifically authorized callers may perform, and *implicit* changes of the protection state as reactions to specific system events. Both kinds of changes are discussed in this section. The discussion focuses on rights changes here, adding or removing objects or principals is described in section 5.4 of the next chapter.

A related topic that is not discussed again here is *delegation*. As explained in section 3.1.3 of chapter 3, delegation occurs during the course of an operation invocation when a target object becomes an intermediate and delegates the call to another object. This kind of delegation does not change the actual access rights in the matrix, however, but defines who may use whose authorizations.

4.1.5.1 Explicit Assignment: Assignable Views

Explicitly entering or deleting views from the access matrix may have far–reaching consequences and therefore requires special authority. Such authority is usually given to policy managers, who are responsible for monitoring the evolving policy and may need to modify it in response to specific conditions, such as lack of authorizations for a role. If the role needs these authorizations to perform its tasks and should therefore be authorized, managers need to add the necessary views.

A second kind of explicit assignment or removal of views that is not restricted to administrators is related to policies that are based on Discretionary Access Control (DAC). It occurs when a principal explicitly calls an assignment or removal operation on the access matrix in order to pass on one of its views to another principal. Until now, principals were only roles, but for dynamic discretionary assignment it is necessary to also support individual subjects as principals, not only entire roles. A subject is an individual principal entity rather than an aggregational concept. This distinction will be discussed in more detail in section 4.2.1.

Views can only be passed on if their definition was marked as *assignable*. To allow for explicit, discretionary assignment the view `PublicReviewing` in the example in figure 4.10 is marked with the modifier **`assignable`**. In the example, the `PublicReviewing` view is intended to be initially assigned to the *Reviewer* role, but subjects in this role may freely pass on this view, e.g., to get outside input to help them with reviewing. The operations allowed by this view are annotate(), read() and find(). The last two permissions are inherited from the `Reading` view. Also inherited is the controlled type Document. The role restriction is narrowed to `Staff` only and prevents a principal from assigning the `PublicReviewing` view to roles other than `Staff` or one of its subroles, i.e., it excludes `Author` which was listed in the role restriction of the base view `Reading`.

```
assignable view PublicReviewing:  Reading // on Documents
   restricted_to Staff
{
   allow
      annotate
}
```

Figure 4.10: An assignable view.

By default, views are not assignable, and without explicitly adding this modifier to a view definition no discretionary assignment is possible. Unlike role restrictions or controlled types, the **assignable** modifier expresses no constraint or restriction on the matrix entries that contain the view. There is no relationship between assignability and view inheritance, which was stated informally as "adding permissions for more special (and thus more restricted) situations". Consequently, assignability is not inherited by an extending view. Each view that is to take part in discretionary assignment must be marked individually — there is no implicit assignability.

Discretionary assignment of views requires that a principal who wants to pass on a view must have received this view with the *assign option* set, i.e., he must have been allowed to further assign the view. This boolean option is the exact equivalent of SQL's grant option and also found in systems like SPKI [Ellison et al., 1999b]. A principal can prevent further propagation of his views by assigning them without the assign option. A final constraint is due to the fact that views can contain both permissions and denials, so assigning a view with denials could effectively *reduce* the recipient's permissions if the recipient cannot refuse to accept the view. To avoid such discretionary "denial of service" assignments, views assigned in this way may only contain permissions. This constraint can be statically verified by a VPL compiler which checks that the definitions of assignable views do not contain denials.

4.1.5.2 Implicit Assignment: Schemas

The term *implicit assignment* (or *implicit removal*) of views refers to changes in the protection state that are not caused by an explicit change operation on the matrix but happen as a reaction to an application event. This kind of rights change is particularly important in order to express state–based policies like Chinese Walls [Brewer and Nash, 1989]. The kinds of events that are of interest here are operation invocations, so implicit assignments are automated assignment actions that occur as a reaction to an operation invocation and can be compared to triggered SQL statements. More precisely, implicit assignments are triggered by a successful return from an operation.

To describe the conditions for implicit assignments and removals of views VPL offers the *schema* language construct, which is the second major contribution of this work. Schemas define a set of assignment and removal statements for views on objects. These assignments or

removals are triggered by operations on *all* objects of a given type in the policy domain if a schema is defined on that type. There is no provision to attach schemas only to some objects, so schemas are generally coarser–grained than views. As an example, we describe how an owner status is assigned to the subject calling a factory object's create() operation. For this example, the IDL interface DocumentFactory in figure 4.11 is used.

```
interface DocumentFactory {
    Document create();
};
```

Figure 4.11: Interface DocumentFactory.

Figure 4.12 lists the view definition `Managing` that extends the `Updating` view on Document objects to add permissions to copy or to destroy the document, and to make it assignable to other principals. Because this view transitively extends `Appending`, potential receivers of this view are limited to the role `Staff` and its subroles. The view `Creating` allows the create() operation on DocumentFactory objects.

```
assignable view Managing:  Updating {
   allow
      copy
      destroy
}

view Creating controls DocumentFactory {
   allow
      create
}

schema DocumentManaging
   observes DocumentFactory
{
   create
      assigns
         PublicReviewing on result to caller
            with assign option
      assigns
         Managing on result to caller
            with assign option
      removes
         Creating on this from caller
}
```

Figure 4.12: Views and schema for document creation.

To describe the dynamic rights changes upon returning from the create() operation on DocumentFactory objects, a schema `DocumentManaging` observing operations on the DocumentFactory type extension is defined. To give owner–like status for an object to the principal invoking create(), the schema has an **`assigns`** clause for the create() operation. This clause specifies that the views `PublicReviewing` and `Managing` on the **`result`** object are to be assigned to **`caller`**. Both views are assignable and passed with the assign option so that the recipient may pass them on using discretionary assignment. It is not required that a view be marked as **`assignable`** to be allowed to appear in a schema clause. If the view that is to be assigned is already present in the recipient's matrix entry for the target object, the assignment operation has no effect. The effects of schema operations are discussed in more detail in section 4.2.6.

At runtime, the reserved identifiers **`result`** and **`caller`** are bound to the result of the operation and the calling subject. Views can alternatively be assigned to role principals instead of the calling subject by listing one or more role names. Schema rules can also refer to out or inout parameters of the operation in case it is necessary to modify views on objects passed to the caller this way. In these cases, the schema can use the name of the formal parameter, which is unique in the operation context. To illustrate removals the schema also contains a **`removes`** clause, which specifies that a `Creating` view is to be removed from the caller, thereby removing his right to call the create() operation again. The remove operation has no effect if the view that is to be removed is not present in the matrix entry.

The syntax in the **`assigns`** and **`removes`** clauses requires as arguments a list of view names, a list of role names or the keyword **`caller`**, and an object identifier (such as a parameter name, or the keywords **`result`** or **`this`**) or a type name. The example illustrates assignment and removal of a view to and from a subject principal — the caller — and on specific objects — the target and the result of the call, respectively. Instead of referring to the calling subject, it would have been possible to give a role name. Also, instead of referencing objects the schema clause could have specified a type name, so the assignment or removal would have been applied to all objects of that type in the domain.

To allow for additional constraints that help to control the effect of schema definitions more closely, the view modifier **`static`** is introduced. This modifier expresses the constraint that a view so marked may only be assigned to roles, and not to subject principals. Effectively, such a view is "more static" because the assignments of views to roles are expected to change much less frequently than those of views to individual subjects. A static view is thus easier to track and to control. The **`static`** modifier blends well with inheritance and is inherited by view extension: if this keyword is present in a base view, it cannot be undefined again in extending views, so every more derived view is also implicitly static. A type checker can statically verify the definition constraint that schemas do not define implicit assignments of static views to individual subjects, i.e., to **`caller`**. The **`static`** modifier also combines with the modifier **`assignable`**. In this combination it disallows discretionary assignments to individual subject principals.

4.1.6 Conditional and Virtual Views

To provide a flexible way of expressing access rights that depend on a combination of views, a new **`requires`** clause for views is introduced. In the example in figure 4.13, holding a `SafeOpening` view on an object is not sufficient to be allowed to open the safe. Principals that want to do so need to hold other views, viz. `FirstKey`, `SecondKey`, and `ThirdKey`. These views are all declared as **`virtual`** and have no bodies. Virtual views are the only views that permit empty bodies. Note that a required view does not have to be virtual.

The views in the example are also declared assignable so that key holders can cooperate by discretionary assignment. When a principal tries to open a Safe object, the access decision function checks that the principal holds a view that lists open() as allowed. It also verifies the constraint that the principal possesses all views that are listed as required by the view that permits the access. If it does not possess all these views, the principal's access will be denied and the principal will have to obtain the missing views, e.g., by having other principals assign them.

```
view SafeOpening
   controls Safe
   requires FirstKey, SecondKey, ThirdKey
{
   allow
      open
}

virtual assignable view FirstKey
   controls Safe
   restricted_to KeyHolder

virtual assignable view SecondKey
   restricted_to KeyHolder

virtual assignable view ThirdKey
   controls Safe
   restricted_to KeyHolder
```

Figure 4.13: Conditional access rights.

By declaring a view as virtual, actual operations in interfaces are decoupled from access rights: a virtual view does not refer to operations, its only function is to enable an authorization in combination with other views. Because a virtual view defines no rights for operations, it need not be typed at all. In figure 4.13, the view `SecondKey` does not have a **`controls`** clause and can thus be assigned on any object. However, it can still be useful to define a controlled type for documentation purposes, so virtual views may have a **`controls`** clause. If they have no such clause, the controlled type is implicitly defined to be CORBA::Object. Virtual

views are similar to generic rights in that they do not have any inherent meaning. Rather, that meaning depends on their interpretation in a concrete context. Unlike generic rights, however, virtual views can be restricted to certain roles and may even require other views themselves. A virtual view can be extended like any other view but it can only extend other virtual base views. Otherwise, it would actually inherit rights and thus no longer be virtual.

4.2 A Discussion of the View–Based Access Model

This section revisits the concepts introduced in section 4.1 and discusses them in more detail. In particular, it discusses the notion of principals, explains the matrix model underlying VPL, and introduces constraints. The section also examines how the language concepts meet the requirements identified in the previous chapters of this dissertation.

4.2.1 Principals and Roles

As argued in chapter 2, principals are used to describe a caller's *role* — its function in the interaction with the target — because this interpretation integrates well with the notion of an actor as it is used in requirements analysis. From the perspective of a developer, roles aggregate and abstract from individual callers, which are generally not known at development time. By statically assigning authorizations to a role, a developer groups these for a specific use case or task. From the perspective of an administrator, roles are job functions and support abstracting from concrete applications and their required authorizations.

Restricting the assignment of authorizations to role principals is not practical for discretionary access control, as already mentioned in section 4.1.5.1. For these cases, it is necessary to refer to individual entities, not only to abstract, aggregational constructs like roles. While it is possible to integrate individual subjects into a role–only model by creating a specific, exclusive role for each subject, this does not combine with the interpretation of roles as actors: Role authorizations are based on *function*, whereas subject authorizations are based on *identity*. Rather than sacrificing the actor interpretation of roles, a second, non–role principal notion is added, viz. that of a subject.

A subject is an active entity that can make requests to objects and has its own set of credentials, including a unique identity. Within a system, the only entities that can initiate requests are processes, so subjects are either identifiable processes, such as named services, or represented by processes which are acting on their behalf. Using the terminology of [Lampson et al., 1992] again, the general form of request statements in this model is thus:

C **says** S **says** $request$

where C is a channel and S is a subject. If the receiver of the request can verify that the channel speaks for the subject, i.e., $C \Rightarrow S$, it can accept the request as originating from S. Subjects can directly make such requests on channels because a subject is assumed to be able to establish a channel and make it speak for himself, which roles alone cannot do. Roles

can thus only make requests in conjunction with a subject that is authorized to speak for the roles, i.e., as a *compound* principal. In this model, roles allow subjects to accumulate task–specific authorizations. A compound principal's statement will be checked using the union of the combined principals' authorizations. The form of request statements made by a channel for a compound principal is:

C **says** $(S$ **as** $r_1 \wedge ... \wedge r_n)$ **says** $request$

where S is the subject and the r_i are roles. The receiver of the request must verify that S is a member of all these roles by checking that $S \Rightarrow r_1 \wedge ... \wedge S \Rightarrow r_n$. Note that **as** in this interpretation is only syntactically different from $\wedge$, the meaning is that S and all its roles make a joint request statement. For each individual request, a subject may choose if and for which roles it wants to speak, so it can select a subset of the roles for which it is entitled to speak. These roles are the subject's *active roles* for the access. Alternatively, the subject that is the immediate source of a request may quote other compound principals as making the request. Quoting other subjects means that the request is a delegated call. Such a request has the form

C **says** $(S \mid B$ **as** $r_1 \wedge ...r_n)$ **says** $request$

where S quotes ("|") B in a number of roles as making the request statement. Note that S does not appear in a role himself but simply forwards a request it received on behalf of B **as** $r_1 \wedge ...r_n$. A receiver of such a request needs to determine whether it trusts that S is acting on B's behalf and accepts the request as if made by B **as** $r_1 \wedge ...r_n$. Without an infrastructure for controlled delegation that lets B restrict S's authority to act on its behalf to specific requests, this will only be the case if $S \Rightarrow B$, i.e., if B has previously delegated authority to S to speak for him regarding *any* statement with the "handoff" statement B **says** $S \Rightarrow B$. The verifier must be able to retrieve evidence for this statement by B before it can accept the delegated request. The implementation in chapter 7 does not support delegation because the required CORBA protocol CSIv2 is not yet implemented.

4.2.1.1 Roles

Roles are visible to both application policy designers and managers, so the textual VPL syntax for roles is complemented with a graphical syntax in a management tool. Management tools are presented in chapter 7.

The role model defined for the purposes of this thesis does not introduce original features. Rather, the novel point here is the application of the actor interpretation to the concepts of role–based access control (RBAC) [Sandhu et al., 1996] and the replacement of the generic privileges of these models with views. To emphasize the task–centric approach, roles are given the following definition: A role is a logical function of an initiator in the interaction with one or more targets. For further integration with use case modeling, role inheritance is supported as defined by UML, i.e., as a behavioral rather than a structural notion. The semantics of role inheritance is that an inheriting role can participate in the same interactions as its base roles.

For the purposes of credentials management based on organizational structure, roles are

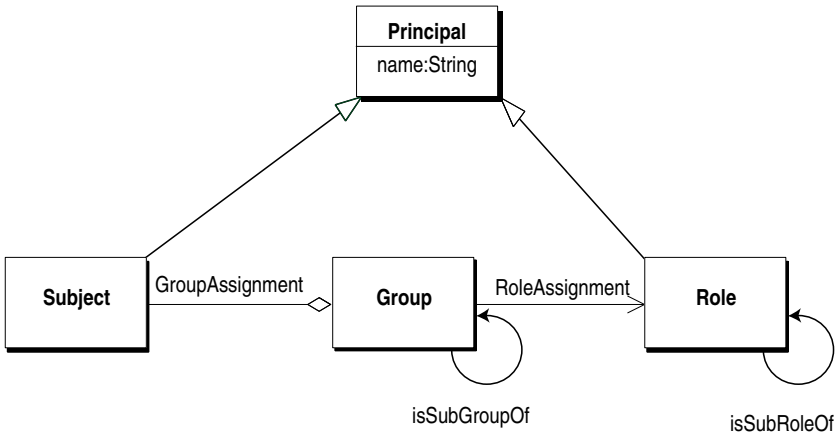

Figure 4.14: Groups and Roles.

complemented with a group concept. Groups are not principals, rather they are used to facilitate the collective assignment of subjects to roles and can be hierarchically organized using a subgroup relation. This indirection was introduced for management purposes and does not affect the design of policies, which refer to principals (subjects and roles) only. Groups are discussed together with the implementation of principals through credentials, both of which are part of the role management infrastructure discussed in chapter 7. Figure 4.14 illustrates the user and credentials management concepts described above and their relationships as a UML class diagram.

4.2.2 A typed matrix model

To explain the details of the VPL concepts introduced above, it is necessary to refer to the underlying access model. The access model proposed here is based on the classical access matrix model of [Lampson, 1974]. Unlike Lampson's matrix or the standard CORBA access model [OMG, 1999a], matrix entries in the new model do not contain simple generic rights but named sets of rights, i.e., views. Relying on typed views rather than generic and unstructured rights allows for a number of useful model constraints, which are examined in section 4.2.3. Matrix rows in the model correspond to principals, so a matrix entry represents the authorizations assigned to a principal for the particular object denoted by the matrix column. This is illustrated in table 4.1. Role principals are represented in the upper part of the matrix, subjects in the lower.

The matrix in table 4.1 represents a system's *protection state* as a set of authorizations, and thus describes which accesses are currently allowed and forbidden. The set of transitions that control how the state evolves — operations on the matrix that insert or delete view, principals, and objects or types — is called *scheme*. An *access policy* is a description of a protection state (a matrix) and a scheme. The authorization schemes that can be defined for this model are described in section 5.4 of the next chapter.

Object Principal	f: Folder	chapter: Document	contract: Document
Secretary	`Lookup`	`Reading`	`Reading`
Author	`Lookup` `Appending`	`Reading` `Updating`	`Reading`

Editor	`Listing`
	`Appending`
	`Removing`
		`Reading`	`Reading`

Manager	`Lookup`	`Reading`	`Updating`
Reviewer	`Lookup`	`Reading` `PublicReviewing`	
Paul		`Reading`	
Ringo		`Reading` `PublicReviewing`	

Table 4.1: An access matrix with views.

To determine the effective views of a compound principal for an access, the views for all its constituent principals must be combined, i.e., the views for the initiating subject and one or more active roles. In the situation in table 4.1, the views on the Folder object f for the compound principal Paul **as** (*Secretary* $\wedge$ *Editor*) are `Lookup, Listing, Appending` and `Removing`. These views are consulted to determine whether the access is allowed or not. This approach contrasts with systems were decisions are based on a *single* entry that is selected from a set, e.g., as in the UNIX file system. In UNIX, file permissions are checked in a specific order (owner, groups, other), and the first matching set of permissions is chosen — but always only a single set.

The central contribution of this thesis is this model and the corresponding policy language VPL, which is sufficiently descriptive to be manageable at a high level of abstraction. However, even the most manageable model is still useless in practice if it is not also possible to implement it efficiently and if it does not support expressive and flexible policies. The central task is thus to find a compromise that delivers good results in all of these areas, manageability, expressiveness, and efficiency.

4.2.3 Constraints

> *There is an essential conflict between the expressive power of an access control model and tractability of safety analysis. [...] Safety is undecidable for most policies of practical interest.* [Sandhu, 1992]

The general meaning of *safety* in the context of protection is that no access rights can be leaked to an unauthorized principal. As shown in [Harrison et al., 1976], safety is only decidable in very restricted cases that generally do not allow the definition of realistic security policies, even considering recent advances such as [Soshi, 2000]. While this has not been explicitly proven, role–based models such as the one presented in this thesis are expected to fall outside the set of decidable cases.

Rather than restricting the expressiveness of a model such that in can be verified to be safe in general, such as lattice–based models like [Bell and LaPadula, 1973], an alternative approach is to extend the model with a language that supports the specification of *constraints*. Constraints can be used in the definition of a policy to restrict the potential configurations in such a way that breaches of the policy are either impossible or at least constrained to specific cases. Constraints directly express safety requirements and thus support reasoning about the safety of a policy even though the underlying security model does not a priori preclude unsafe configurations. Enforcing separation of duty between individuals is a typical example of such a constraint. As argued in [Tidswell and Jaeger, 2000], this approach shifts the burden of safety analysis to the evaluation of constraints and implies that constraint languages must be carefully designed such as to a) support efficient evaluation, and b) allow policy designers to directly express their security requirements. From a manageability perspective, the use of a constraint language has the additional advantage of making security–relevant assumptions explicit [Anderson, 1994], which would otherwise remain implicit in the underlying security model.

In principle, constraints can apply to all components of a model and to arbitrary relationships between these, so constraint evaluation can become arbitrarily complex. Obviously, it is important to provide suitable language concepts for constraints, which is not only a question of efficient implementations but also a language design responsibility.

4.2.3.1 Classifying Constraints

Typically, constraints are classified as either *static* or *dynamic* [Kuhn, 1997], [Simon and Zurko, 1997], based on the time at which these constraints are checked. Static constraints are checked when authorizations are assigned, dynamic constraints apply at runtime when accesses are made. If constraints can be checked statically and the system guarantees that they cannot be violated dynamically then no further run time checks are necessary. Examples for static constraints include static mutual exclusion of two roles, which means that no principal may be assigned to both of these roles at the same time. This constraint can be used to express and enforce static separation of duty between these roles. Alternatively, dynamic separation of duty would allow a principal to be assigned to both roles, but would dynamically check that no principal may activate both roles at the same time. While enforcing the same policy, this dynamic constraint is less restrictive than its static counterpart but requires more verification effort. Some practically useful dynamic constraints apply not just to model configurations, but to *histories* of configurations. As an example, consider policies like *Chinese Walls* [Brewer and Nash, 1989] that prevent principals from accessing

one set of objects after they accessed objects from a conflicting set.

The distinction between static and dynamic constraints is not suitable for the constraints of this model because the overall life cycle of model elements has more than just two stages. Moreover, the term “static” suggests that constraints are not checked at runtime of the system, which is misleading. Static constraints are indeed *independent* of the runtime of applications, but they may very well be performed at runtime. Therefore, the following categories are proposed and used for a general classification of constraints:

1. *Definition constraints* are language rules that restrict the set of valid concept definitions that can be used for the specification of individual policies, such as roles, views, and object types. Constraints in this class include both syntactic and semantic rules and are checked by compilers or other language tools. An example of a definition constraint is that an extending view may not define denials.

2. *Configuration constraints* are checked on operations that modify the configuration of the access model, i.e., a policy. These operations may either be explicit administrative activities or implicitly triggered changes that happen during the course of an application’s lifetime. These kinds of constraints are defined both through predefined, general model properties and through constraint definitions supported by the policy language. As an example, configuration constraints can prevent entering a role–restricted view in a row of the access matrix for an incompatible role.

3. *Operational constraints* apply at the time access decisions are made. Examples are checks for the presence of required views before using a conditional view in an access decision. Strictly speaking, these constraints do not determine whether a given model configuration is valid but rather serve to dynamically mask out those parts of the configuration that are not applicable for the current access. For example, conditional views are only usable in combination with certain other views. Since these views might be available in other contexts, masking out unusable views does not imply that these parts of the configuration are permanently invalid and should be removed.

4.2.3.2 Principal Constraints

The principal concepts to which a policy in our model can refer are roles and subjects. Since groups are not principals, policies cannot make assumptions about them and can also not define constraints on groups and their relationships with roles and subjects. This is not to say that group constraints may not be useful in the context of principal management. For example, a principal manager might want to ensure that an individual removed from a group can never become a member of that group again, so an exclusion constraint could be defined to prevent reassignment of the subject to the group. However, this aspect of principal management is considered to be part of modeling the organizational environment of a policy rather than part of the definition of application access policies. Group constraints are thus not discussed here.

Potential constraints on subjects are subject–subject exclusion to prevent two subjects from ever being assigned to the same role or subject–role exclusions that prevent a specific subject from being assigned to a specific role. However, the general approach for policy modeling here is abstracting from individual subjects because these are not yet known during policy development and before deployment. Policies should therefore not rely on rules that refer to specific subjects. For similar reasons as for groups, it may be useful to express subject constraints for principal management purposes, but these are not regarded as part of the access policy.

The role model in [Sandhu et al., 1996] defines three types of constraints that restrict the assignment of subjects to roles. First, *cardinality constraints* are introduced as predicates on the number of subjects that can be assigned to a role. For example, there can be a maximum cardinality defined for a particular role to prevent the assignment of more than one subject to the role. While arbitrary predicates on cardinality are possible, we limit this discussion to *minimum cardinality constraints* and *maximum cardinality constraints* because these are regarded as the most useful. Minimum cardinality constraints can be used, e.g., to express that a given role must always have at least one subject assigned to it — or at least ten, as in figure 4.15.

```
roles
    ProjectMember
    Developer:  ProjectMember mincard 10
    TestEngineer requires ProjectMember excludes Developer
```

Figure 4.15: Role declarations.

A second constraint is the *prerequisite constraint* between roles which prevents the assignment of a subject to a role if that subject is not also assigned to another role, e.g., role *TestEngineer* might require that any subjects are first assigned to a *ProjectMember* role. While the effect of this constraint with respect to the tasks and authorizations of the subject is the same as if *TestEngineer* was a subrole of *ProjectMember*, the constraint enforces that subjects are assigned to both roles individually and in a predefined order.

The final constraint mentioned in [Sandhu et al., 1996] is *mutual exclusion*, which means that a subject can never be assigned to two mutually exclusive roles at the same time. This constraint can be used to express *separation of duty* requirements, e.g., that a *TestEngineer* may not also be a *Developer*. These three simple constraints are adopted using the syntax introduced in section 4.1.2. All three kinds of constraints, if used in a policy definition, are enforced by the role management component of an overall infrastructure, not by an access control mechanism. Enforcement is done at the time managers assign subjects to roles.

It should be noted again that principals are generally under the control of a management role other than policy management. Thus, an access policy must either rely on principal managers to correctly enforce constraints on principals or repeatedly re–verify its constraints. The

role constraints examined here are generally enforced by checking every operation that changes any of the involved data structures, i.e., assigns subjects to roles (via groups) or adds a new constraint to exiting subjects and roles. These constraints are thus *configuration* constraints.

Both the exclusion constraint and the prerequisite constraint could alternatively be checked when a principal tries to access an object, i.e., as operational constraints. An operational exclusion constraint could then be used to enforce dynamic separation of duty. An operational prerequisite constraint would enforce that a given role is only used as an element of a compound principal and together with certain other, required roles. For language design reasons, these more fine–grained operational constraints are not included in this model. Defining syntax for both configuration and operational constraints necessitates additional keywords and also requires designers to understand the distinctions between the two kinds of exclusions and prerequisites. To keep the language simple, only configuration constraints are adopted.

4.2.4 Views and Matrix Constraints

The introduction of views as a language concept is motivated by the observation that generic rights such as “read”, “write”, and “execute” are not adequate for describing authorizations in large–scale systems comprised of typed objects. Generic or uninterpreted rights are simple and flexible, but they are inadequate for application–level access policies in large–scale object systems because they exhibit neither external structure nor inherent semantics. Generic rights are hard to manage because they provide no ways to impose external structure by defining relationships or constraints. The rights families introduced by the CORBA Security Service are an attempt at providing structure by grouping rights, but this structure does not actually provide a useful abstraction mechanism that would support rights management, e.g., relations between rights families. Managing large numbers of rights without any abstractions soon becomes unwieldy.

This is in fact the reason why the CORBA Security Service discourages the definition of new rights families and recommends using the standard family. Effectively, the number of different rights in the system is restricted in order to avoid these management problems. This approach leads to modeling problems in the design of access policies because it is very difficult to capture the rich semantics of operations in object–oriented systems with just a very small set of generic rights. The root of this problem is simply that such a restricted set of rights exhibits only very limited expressiveness.

Generic rights are by their very nature *untyped*, i.e., there are no constraints that would restrict the relationship between a right and the actual access modes (operations) provided by the target object. The mapping between an operation and the authorization required to invoke it is arbitrary — there is no inherent semantics. This can be problematic because the designers of a policy may decide to require a right, such as “write”, for an operation in a way that is not compatible with the way this right is used in other cases. An example for such a situation was given in chapter 3.

4.2.4.1 Constraints on Matrix Entries

An important property of views is that they are typed by the object interface they control. It is therefore possible to impose a type constraint on the access matrix, which ensures that each view entered into the matrix is actually applicable to the type of the target object in the matrix column. By detecting type errors at view assignment time, this configuration constraint helps to catch a class of simple administrative errors that could severely compromise security. Typing views not only prevents assignment errors, it also plays an important role in a definition constraint that helps to statically detect errors in view definitions because it is now possible to check that all operations listed in a view do in fact belong to the same interface. Ill–formed views can thus be detected at definition time by the type checker. As a simple example, consider the two object types Folder and Document, which both provide an append() operation.

Because appending to a directory–like folder is different than appending to a document, we assume an example policy that wants to express that a particular role may append to the folder but not the document, such as the *Editor* role in table 4.1. The matrix typing constraint guarantees that no operation on the matrix can enter a Folder–typed view with an append right in a Document object's column, which would allow append access to the document and violate the intended policy.

The type constraint is based on the types of the objects in the matrix column. An analog configuration constraint can be applied to matrix rows: Using role restrictions, views can be defined such that they can only be assigned to certain principals. Like the object type constraint, this constraint limits the potential for administrative error and helps enforcing the *need–to–know* principle. It also documents the intended policy semantics more closely. This constraint can only restrict view assignments to specific roles and not to individual subjects because subjects cannot be referenced statically in view definitions.

Unlike the type constraint, this role restriction does not help to catch general errors in the definitions of views. However, a role–restricted view that is entered in a subject's matrix entry entails an *operational* constraint: Whenever a subject's permission for an access depends on a role–restricted view, it can be checked whether the subject has currently activated the role to which the view is restricted. If the role is not active the view is ignored in the access decision, so even if the view cannot be prevented from being entered in that matrix entry, it can be prevented from being *used* without the appropriate active role.

4.2.5 Explicit Discretionary Assignment and Removal

An important general concept in many systems is delegation of authority. Delegation of the authority to assign authorizations means that someone in charge of assigning authorizations, e.g., a system administrator, allows someone else to assign a subset of these authorizations. Usually, this is done for decentralization and division of labour purposes. If this authority leaves the small circle of appointed administrators and is passed on to regular users, it is called *discretionary access control* — the assignment of some authorizations is now left to the discretion of privileged users rather than kept under system control.

In many systems, the concept of resource ownership is used for this privileged user. In the UNIX file system, the creator of a file also becomes its owner and may grant or revoke rights on this file to other users. While the owner may give up ownership of a file to other users, there is always just a single owner for any file. Consequently, there is only a single potential grantor or source of granted rights at any time for any given file — if we disregard the omnipotent UNIX super user for the moment.[2] Grantees can only further grant their rights to other users if they have also been given ownership; revocation of granted rights from specific users is straightforward.

Similarly, the creator of a database table in the SQL security model [ISO/IEC, 1992] becomes the owner of the table and can now grant rights on this resource to other principals. In addition to simply granting the right, the owner may choose to grant rights with the grant option. A grantee who receives a right with grant option may further grant this right to other principals. In this case, there are potentially multiple sources for granted rights, although there is a single root source, viz. the owner. Since granted rights may have propagated through any number of users, revocation of a single right might lead to a cascading revocation [Griffiths and Wade, 1976], [Fagin, 1978] of this right from all users that were transitively granted this right through the first grantee. The model presented here does not have a built–in ownership concept, but this concept can be modeled by assigning a special set of views to the principal who creates an object and by allowing this principal to pass on these views as he deems appropriate.

4.2.5.1 Assignable Views

Assignable views are similar to capabilities in that holding a view on a particular object includes the right to pass this view on this object on to others, provided that the view was received with the assign option set. However, as pointed out above, views are not capabilities. Unlike in typical capability schemes, assignable views can be directly removed again, and their propagation can be statically restricted to specific roles. When a role–restricted view is assigned to an individual subject rather than a role, it would also be possible to check that the subject is a member of that role or one of its subroles. However, this restriction is stricter than necessary and would also require that role–restricted views are removed again when the subject is removed from the role. A simpler but sufficient restriction is to guarantee that role–restricted views that are assigned to subjects are not *usable* when the roles to which they are restricted are not activated by the principal. Thus, as explained in section 4.2.4.1, role restrictions also constitute operational constraints.

The right to assign a view is in fact a meta–right and not modeled as a view itself or as a permission or denial in the assignable view. Assigning a view corresponds to the `enter` command in matrix models such as [Harrison et al., 1976], [Sandhu, 1992]. A principal holding an assignable view that he received with the assign option may further assign this view to other principals, with or without the assign option.

[2] Files may have different owners at different times because file ownership can be transferred in UNIX.

Explicitly Removing Assignable Views

By assigning a view to a role or a subject, an assigner acquires the right to remove that view again at his discretion, which potentially leads to further, cascading revocations if the recipient has passed on the view. Cascading revocations are necessary to prevent a principal from immediately obtaining a removed view again by having another recipient assign the view back to him. If the recipient already holds this view, the attempted assignment has no effect. Otherwise, the assigner would receive a remove right and could thus remove all those views that both the assigner and the recipient possess: the assigner could simply assign views that the recipient already holds, thus acquiring the right to remove them again without actually becoming the source of these views.

If a principal removes a view for which it has a remove right, the principal loses his remove right for that view. Otherwise, the principal could always remove that same view if it is later reassigned from a different source, e.g., directly by an administrator. For the same reason, the remove right is lost when a view that was assigned using discretionary assignment is removed by other means, e.g., implicitly as explained in the next section, or explicitly by an administrator.[3]

4.2.6 Implicit Assignment and Removal

The concept of implicit assignments (and removals) of views is essential for expressing application–oriented access policies where rights changes occur regularly as part of the application logic. As an example, consider workflow or CSCW–style applications where objects are shared between principals and undergo different processing stages, each associated with different use cases and access modes. An application example illustrating such a situation is presented in chapter 6. If changing access rights between processing stages regularly required administrator actions, these applications would not be practical. Implicit assignments automate these changes and support expressive policy specifications that capture these application features in static descriptions.

The remainder of this subsection discusses VPL schemas in more details, in particular the consistency checks based on definition constraints that are necessary to prevent ill–formed schemas with conflicting clauses. Note that schemas are isomorphic to IDL interfaces, so the state changes that are described by schemas could in principle also be described in an extended interface notation, or as IDL annotations. A separate language construct was chosen because this permits the representation of policy concepts in a single language and also does not rely on an extended IDL compiler.

[3] Note that a cascading revocation must also be performed in these cases of removals. Otherwise, any views that were assigned further by the original grantee could not be removed.

Effects of Schema operations

The **assigns** and **removes** clauses in schemas describe an operation's effect on the protection state of the system, so they can be regarded as operation postconditions with respect to the protection state. An implicit precondition is that a principal must have the authorization to invoke the operations that triggers the schema activity, i.e., that it has a view allowing the operation and no denial in another view overrules this permission. This precondition is not introduced by schemas, however, and if it is false no schema activity for that operation is triggered.

When a schema tries to assign a view, it is possible that the view is already present in the matrix entry, either because of previous schema activities or because of discretionary assignments. If the schema were to perform the assignment again, a number of subtle questions arise as to what the effect of this assignment would be. In particular, the possible interactions between assign options and remove rights would lead to complex rules. For example, if the assignments differ with respect to the assign option, it is not clear what the combined effect should be. If the view that is already present has the assign option set and was assigned to further principals by its holder, overwriting it without this option could interfere with a cooperation protocol between principals. Also, if a schema were to overwrite a view that was assigned by another principal, it would become the new source of the view and would thus have to remove that principal's remove right. Otherwise, it would be unclear which views should be removed in case of a later, recursive removal of the view from the original recipient.

Because these interactions lead to complex and unintuitive behavior, multiple assignment and removal attempts should not have a combined effect; all interferences between multiple sources of assignments should be prevented. In section 4.2.5.1, an explicit, discretionary assignment was defined to have no effect to prevent the assigning principal from obtaining the remove right. Consequently, a view assigned by a schema blocks any subsequent attempts of discretionary assignment of the same view. Likewise, schema assignments may only be performed if the view is not already present. Otherwise, the assignment has no effect. Whichever source of views performs the assignment first thus blocks assignments of the views from other sources.

For removals, the situation is similar — the first removal attempt simply removes the view. Removals do not interfere with each other because the effect is always the same: the view and all associated information such as the assign option or the remove right is no longer present; if the view was assigned further through discretionary assignments, it must be recursively removed from these recipients. The only potential interference of different removals is temporal, i.e., with multiple schemas — and potentially principals — attempting removals, a view might be removed earlier than expected.

If a compound principal holds a view both in his subject matrix entries *and* by virtue of being assigned to one or more roles, then the effect of a **removes** clause that only removes that view from one matrix entry is not sufficient to prevent that principal from accessing the target object. Because the absence of a view in one entry — one of the role entries and the

subject entry — cannot override permissions in another, the only way of reliably blocking an access for a principal is by assigning denials for that access. If policy designers are careful to define views with strong denials for this access, assigning such a view to a subject prevents the access. Assigning the denying view to the role might result in blocking too many other principals and is not guaranteed to achieve the desired effect since the principal might have activated a different role set which does not contain such a denial. Views in the subject matrix entries, however, will always be checked, so the denial is guaranteed to be effective.

Conflicts in Schema clauses

The effect of schema clauses for an operation potentially consists of multiple view assignments or removals. To make these effects predictable, it is important to guarantee that effects are deterministic, which means that, given the same access matrix, the invocation of an application operation always produces the same effect on the protection state. To be able to make effects deterministic, it is necessary to define rules for dealing with conflicts in schema clauses.

A schema clause has conflicts if it specifies that the reaction to an operation comprises both the assignment of view V to role R on an object or a type T and the removal of the same view on the same object or type from the same role. The effect of executing the removal first and the assignment second would be that view V is present in the matrix entry. Executing the assignment first and then the removal would have no observable effect on the matrix entry, however. Generally, conflicts can be managed by either excluding conflicting clauses a priori or by defining conflict resolution rules that determine the order of operations, e.g., that assignments are always executed before any removals.

Schemas are a powerful and flexible concept, but they do increase the complexity of policies, so it is desirable not to increase this complexity further by defining elaborate conflict resolution rules. In designing language rules, the general approach taken in this thesis is to facilitate management as much as possible and shift complexity from the manager towards the designer of a policy wherever possible. Therefore, static analysis should verify that schemas are free of conflicts. If this verification is not possible or if it fails, schema definitions contain potentially conflicting clauses and must be rejected. Naturally, the goal is to reject only the smallest possible set of schema definitions and to accept all cases that can be verified as conflict–free.

An actual conflict occurs if the same view on the same object is to be both entered and removed from the same matrix entry. Schema clauses that affect different views or operate on different matrix entries, or that only assign or only remove, are free of conflict. Figure 4.16 shows an example schema that can be statically verified as conflict–free.

The schema in figure 4.16 defines two potentially conflicting clauses. The first potential conflict is between the assignment and removal of V_2. However, the assignment applies to role R_1 and the removal to role R_2, so the two changes affect different matrix entries and thus do not conflict. Hierarchical relationships between roles do not lead to conflicts because the assignment of a view to a role is only recorded in a single matrix entry and not directly assigned

```
schema ConflictFree observes S
{
   op1
      assigns
         V1, V2 on T to R1
      assigns
         V3 on this to caller
      removes
         V2 on this from R2
      removes
         V3 on T from R1
}
```

Figure 4.16: A conflict–free schema.

to all its subroles. In the example, the two changes also operate on different matrix columns because the assignment affects the matrix column for a type T and the removal the column for an individual object, viz. **this**. These columns are distinct regardless of the IDL type hierarchy and whether the type S of **this** is related to T or not. The second potential conflict in figure 4.16 is between the assignment and removal of V_3, but again the target principals — **caller** and R_1 — are not identical.

In the general case, static analysis can verify freedom from conflicts in all cases where either the views, the principals, or the target types in the clauses differ. Moreover, all cases where one target of an assignment or removal is an object and the target in another clause is a type are guaranteed to be conflict–free. As explained above, inheritance relations between views, types, or roles do not complicate matters here. Conflicts are obvious if two clauses refer to the same view, to the same role (or both refer to **caller**), and to the same type extension or same object, i.e., both refer to **this** or **result**.

```
schema Conflicts observes R
{
   op1
      assigns
         V2 on result to R1
      assigns
         V3 on S to caller
      removes
         V2 on this from R1
      removes
         V3 on T from caller
}
```

Figure 4.17: Potential conflicts in a schema.

One case is statically undecidable, however. Because of aliasing and polymorphism, it cannot be guaranteed that two clauses do not refer to the same object if the target objects are **this** in one clause and **result** in the other. The types of these references are statically known, but even in the case of two unrelated types S and T is it possible that a reference of a common subtype U is bound to both **this** and **result**. Therefore, those assignment and removal clauses must be rejected that only differ in these object identifiers. Figure 4.17 shows a schema definition with assignments and removals that apply to the same views and principals. The schema must be rejected by the verifier because the assignment of V_2 and the removal of the same view cannot be guaranteed to affect different matrix columns.

Multiple Schemas

It is possible that more than one schema reacts to an operation. This can happen if a schema is assigned to a type extension T and another schema to the extension of type S, which is a subtype of T. Operations on objects of type S are now observed by both schemas. This is illustrated in figure 4.18. Also, multiple schemas can be defined to observe the same type. However, this is only a syntactical convenience and is equivalent to splitting up a single schema definition into smaller, more manageable pieces. Conceptually, multiple schemas observing the same type form a single schema.

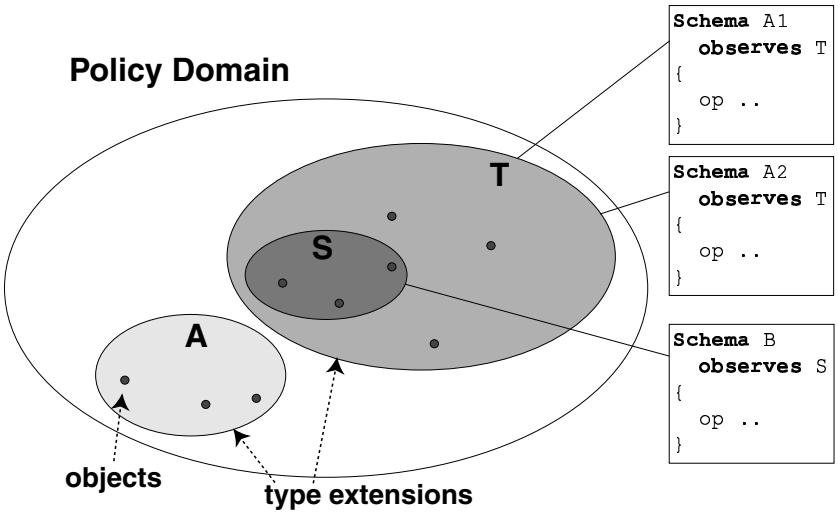

Figure 4.18: Multiple Schemas.

If multiple schemas are assigned to the extensions of types that are in a subtyping relation and if they define clauses for overlapping sets of operations, they can introduce the same kinds of conflicts that were excluded for a single schema above. To prevent conflicts, any schema definition for a type S must be statically compared with all other schema definitions for S's supertypes. If a schema observing S defines a clause for an operation that is also listed in another schema, the union of the two schemas' clauses is checked for conflicts as described

above. If a schema for S conflicts with any other schema applicable to objects of type S, it is rejected by the verifier.

A Discussion of Schemas

A number of points require discussion here. The first issue is the increase in complexity introduced by schemas. At first sight, the expressiveness of the schema concept seems to make the evolution of the protection state much less predictable and intuitive. However, schemas should simply be seen as listeners that react to event notifications; the assignments or removals of views that are performed by schemas must be regarded as administrator responses that would be performed anyway. Another point that needs to be stressed is that schemas do not introduce either new views or new authorization conflicts. Thus, resolvability of all potential conflicts between rights is still preserved. Moreover, schema definitions can be statically checked for conflicts and for violations of the type and role constraints on views. This is possible because the types of all objects that can be referenced in a schema definition are statically known, as is the controlled type of any view that can be referenced in a schema.

The role restrictions of all views referenced in the schema are also known statically. However, these restrictions can only be checked statically in schema assignments if the **`assigns`** clause refers to a role name. If a view is to be assigned to an individual subject, then the set of roles that the caller may activate is not known at definition time. In some cases, a role inference algorithm could prove statically that the role restriction of the view that is to be assigned to the caller is not violated. However, the algorithm is not practically useful because it requires that all views that allow the triggering operation are actually role–restricted, which is too strong an assumption in the general case. Moreover, view–role assignments can change after the schema was checked, so additional dynamic checks are still required.

Relying on runtime checks is unproblematic because, as argued in the previous section, role restrictions can be enforced at runtime if the security service simply disallows the use of role–restricted views if none of the roles to which a view is restricted is activated. Rather than defining definition constraints that statically reject schemas that cannot be proven to comply to role restrictions, all schema definitions with assignments to subject principals are simply accepted and the assignments to subjects are performed. This liberal approach is feasible because views that are assigned to subjects that are not members of the roles to which the view is restricted are unusable because of the operational constraints for role–restricted views. Thus, they do not permit breaches of the policy. The only drawback of not being able to statically detect these cases is that designers are not warned if schema definitions contain assignments that may have no effect, but this does not appear to be a serious problem.

A final point in this discussion of schemas is that an implementation of this concept must be able to undo the effects of the **`assigns`** and **`removes`** clauses if they occur in an invocation context that is later aborted, e.g., because of lack of permissions for a subsequent operation invocation or because of an exception. Consider the following scenario: A principal invokes an operation op on an object which delegates the invocation to n other objects, which are called

sequentially. The implementation of op has successfully made the first i invocations, and these have triggered a number of assignments and removals of views because of the presence of schema definitions. The protection state might be inconsistent if the original invocation of op is aborted at this stage and does not return successfully but the postconditions of the first i delegations were to persist. Effectively, an operation must be atomic with respect to its effects on the protection state and thus contained within a transaction.

4.2.7 Conditional and Virtual Views

The access control model presented here closely relates the concepts of *operation* and *access right* because access rights defined in views must have the same name as the operations that they allow or deny. Since operations are atomic units, so are access rights. While this model is more appropriate for object–oriented systems than approaches which separate access rights from operations, it lacks the flexibility offered by the option of *combining* more abstract access rights to obtain a certain privilege. As an example, consider the access control policy for the safe with multiple locks, again. The safe can only be opened when a certain number of key holders cooperate. Accessing the safe should require *multiple* views which correspond to the key holders' keys. A single key view by itself does not authorize anything. Without the additional concept of conditional views, however, we cannot express conditions on operations that require the caller to possess further authorizations than just the one that permits the operation.

The **`requires`** clause used in VPL to define conditional views actually represents yet another constraint on the access matrix: it is now possible to express a requirement on the state of the matrix entry itself, not just on the object in the matrix column or the subject or role in the matrix row. Unlike the role restriction or the type constraint, the **`requires`** clause is not a configuration constraint that would prevent assignments of views to principals, but an operational constraint that must be checked before a view can be considered in an access decision. Figure 4.19 illustrates all three types of matrix constraints.

Like other constraints, the **`requires`** clause is inherited by extending views. The extending view can define its own **`requires`** clause that is implicitly appended to any inherited restriction. Again, this serves to make an extending view "less applicable" by defining stricter context requirements. If a view inherits multiple **`requires`** clauses, the resulting clause is the conjunction of the individual view requirements in the base views and potentially the extending view itself. A conditional view must not depend on itself, so cycles in the dependency graph must be prevented.

The policy for opening the safe is similar to general threshold or quorum policies, where the presence of at least n out of k privileges is required for an access. Cooperative access policies like these can also be described in terms of threshold subjects [Ellison et al., 1999a] or compound principals, neither of which is supported by this model, however. In such a setting, the individual key holders would cooperate by authorizing one of them to speak for the others or for a compound principal, all of which would require the creation and management of appropriate credentials. The preferred approach here is to rely on discretionary granting and

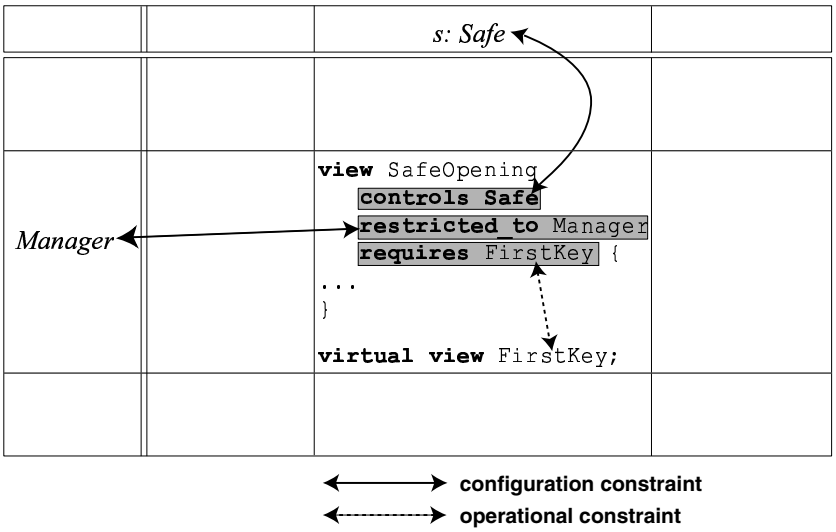

Figure 4.19: Matrix restrictions.

conditional views for cooperation policies as described above.