5 Model Formalization

This chapter gives a more precise definition of the model concepts that were introduced in the previous chapter. These definitions of model properties are necessary to define a view–based access decision algorithm, which is given in section 5.5.

Let $\mathcal{O}$ be the set of CORBA objects, and $type : \mathcal{O} \rightarrow \mathcal{T}$ a function that maps each object to its type, which does not change during the lifetime of an object. The CORBA object model defines a subtyping relation between object types, which has the usual substitution semantics and is denoted here with the symbol $\sqsubseteq$. Let Op be the set of operation names. The function $ops : \mathcal{T} \rightarrow \mathbb{P}(Op)$ maps an object type to the set of operation names defined for this type, $\mathbb{P}$ denoting the power set. Note that in CORBA IDL, $ops(T)$ is indeed a set because all operation names in a type must be unique which means that there is no overloading of operation names.

Let $\mathcal{R}$ be the set of roles and $\mathcal{S}$ the set of subjects, which have no common elements. The active entities whose actions are to be constrained are principals from the set $\mathcal{P} = \mathcal{R} \cup \mathcal{S}$. The sets $\mathcal{O}$ and $\mathcal{P}$ are disjoint, i.e., principals are not objects and cannot be accessed like objects by other principals. $\mathcal{V}$ is the set of views, which will be examined in detail in section 5.3.

5.1 Protection State

An object system's protection state is modeled as a matrix $M : \mathcal{P} \times (\mathcal{O} \cup \mathcal{T}) \rightarrow \mathbb{P}(\mathcal{V})$, i.e., as a function that maps pairs of principals and objects or types to sets of views. The set of matrices is called $\mathcal{M}$. When compared with the original Lampson matrix, principals are the subjects, CORBA objects and types are the objects, and matrix entries are sets of views. This matrix is illustrated in table 5.1.

Subjects and roles have separate rows in the matrix because both discretionary and implicit assignment need to refer to individual subjects, not just to roles. Types have columns in the matrix because this permits assigning a view to a principal (or role) on all objects of a given type t, i.e., on the entire extension of t. The extension of a type includes the extensions of all its subtypes. It would be possible to implement this by simply breaking down the assignment and entering the view in all the corresponding matrix entries, but the implicit assignment itself would not be represented in this case. Moreover, it would then be possible to remove the assignment individually for each object, which is undesirable.

The information about group, role, and view hierarchies is expected to be administered centrally and shared among different policies. Therefore, this information is modeled sepa-

	o_1	...	o_n	t_1	...	t_m
r_1			{Reading}			
...						{Creating}
r_i			{Updating}			
s_1						
...						
s_j		{Managing}				

Table 5.1: An access matrix.

rately and not in additional matrix partitions. IDL type hierarchies are maintained in separate data structures independent of the protection state.

5.2 Groups, Roles, and Accesses

Let $\mathcal{G}$ be the set of groups. Each group in the set $\mathcal{G}$ is associated with a set of member subjects. In our model, the partial order $<$ ("subgroup of") implies a subset relation between group member sets, so a subgroup's members must also be members of the supergroup. Group membership of subjects is modeled using a relation $GA \subseteq \mathcal{G} \times \mathcal{S}$ ("group assignment") for which the subset invariant (5.1) holds.

$$\forall\, g, g' \in \mathcal{G}, s \in \mathcal{S} : ((g, s) \in GA \wedge g < g') \Rightarrow (g', s) \in GA \tag{5.1}$$

Roles

A group g's role assignments are modeled in a relation $RA \subseteq \mathcal{G} \times \mathcal{R}$ ("role assignment"). The function $roles : \mathcal{G} \rightarrow \mathbb{P}(\mathcal{R})$ returns the roles directly assigned to a group, but it also returns any super group's role assignments because a subgroup in our model inherits the role assignments of its super group. A second function $subjectRoles$ returns all roles to which a subject is assigned through its groups.

$$\forall\, g \in \mathcal{G} : roles(g) = \tag{5.2}$$
$$\{r \in \mathcal{R} \mid (g, r) \in RA \vee (\exists\, g' \in \mathcal{G} : g < g' \wedge r \in roles(g'))\}$$
$$\forall\, s \in \mathcal{S} : subjectRoles(s) = \tag{5.3}$$
$$\{r \in \mathcal{R} \mid \exists\, g \in \mathcal{G} : (g, s) \in GA \wedge r \in roles(g)\}$$

Role hierarchies are defined using a partial order $\leqslant$ on roles. Individual authorizations on objects that are assigned to a role are represented directly in the corresponding matrix entries. In addition to its own, direct assignments, the complete set of a role's authorizations for an object includes those inherited from its super roles and those assigned on the object's type and

its supertypes. This is captured in the definition of the function $roleViews : \mathcal{M} \times \mathcal{R} \times \mathcal{O} \rightarrow \mathbb{P}(\mathcal{V})$, which uses a function $roleViews_{type}$ defined in (5.5) to retrieve views assigned on type extensions:

$$roleViews(M, r, o) = roleViews_{type}(M, r, type(o)) \cup \{v \in \mathcal{V} \mid v \in M[r, o] \vee (\exists\, r' \in \mathcal{R} : r \leqslant r' \wedge v \in roleViews(M, r', o))\} \tag{5.4}$$

$$roleViews_{type}(M, r, t) = \{v \in \mathcal{V} \mid v \in M[r, t] \vee (\exists\, t' \in \mathcal{T} : t \sqsubseteq t' \wedge v \in roleViews_{type}(M, r, t'))\} \tag{5.5}$$

Role Constraints

The configuration constraints on roles that are directly expressable in VPL are *cardinality, prerequisite,* and *exclusion* constraints. The predicate $excludes$ is introduced to model mutual exclusion between roles and the predicate $requires$ models prerequisite roles. The role exclusion and prerequisite constraints are written as:

$$\forall\, r_1, r_2 \in \mathcal{R} : excludes(r_1, r_2) : (\neg \exists\, s \in \mathcal{S} : r_1 \in subjectRoles(s) \wedge r_2 \in subjectRoles(s)) \wedge \tag{5.6}$$

$$requires(r_1, r_2) : (\forall\, s \in \mathcal{S} : r_1 \in subjectRoles(s) \Rightarrow r_2 \in subjectRoles(s)) \tag{5.7}$$

The two cardinality constraints $mincard$ and $maxcard$ define a lower or an upper bound on the number of subjects that can be assigned to a role via groups. This set of subjects is retrieved using the function $subjects : \mathcal{R} \rightarrow \mathbb{P}(\mathcal{S})$. This function and the cardinality constraints are defined as follows:

$$\forall\, r \in \mathcal{R} : subjects(r) = \{s \in \mathcal{S} \mid \exists g \in \mathcal{G} : (g, s) \in GA \wedge r \in roles(g)\} \tag{5.8}$$

$$\forall\, r \in \mathcal{R}, n \in \mathbb{N}^+ : mincard(n, r) : |subjects(r)| \geqslant n \wedge \tag{5.9}$$

$$maxcard(n, r) : |subjects(r)| \leqslant n \tag{5.10}$$

Accesses

When a principal accesses an object, it may be either a single subject or a compound principal, i.e., a subject in one or more roles. Accesses are modeled as tuples from the relation $Access \subset Op \times \mathcal{O} \times \mathcal{S} \times \mathbb{P}(\mathcal{R})$, i.e., an individual access contains an operation name, a target object, a subject, and a set of active roles. The set of roles that a subject may play for an access is

constrained to a subset of the roles to which it is assigned. This is expressed in (5.11), using $groups(s) = \{g \in \mathcal{G} \mid (g, s) \in GA\}$ as a shorthand for a subject's groups.

$$\begin{aligned} &\forall\, (op, o, s, active\_roles) \in Access : \\ &\quad \forall\, r \in active\_roles :\ \exists\, g \in groups(s) : r \in roles(g) \end{aligned} \tag{5.11}$$

To determine whether an access is allowed, the access decision function needs to retrieve the views that are available for the principal and interpret these with respect to the operation op. Views are formalized in the following section and the access decision function is described in section 5.5.

5.3 Views

Views have names and optionally modifiers, a controlled object type, and optionally a role restriction. Moreover, views may depend on other, required views. Finally, views define a set of access rights for operations. Let ID be the set of view identifiers, $Mode = \{allow, deny\}$ the set of rights modes, $Prio = \{strong, weak\}$ the set of priorities. The set of rights is defined as $Rights = Op \times Mode \times Prio$, so rights are tuples (op, m, p) that comprise an operation name, a mode, and a priority.

Let $Modifier = \{assignable, virtual, static\}$ be the set of view modifiers. The set of views can now be defined as $\mathcal{V} \subset ID \times \mathbb{P}(Modifier) \times \mathcal{T} \times \mathbb{P}(\mathcal{R}) \times \mathbb{P}(Rights)$. A view is thus a tuple $(name, modifiers, controlledType, roleRestriction, rights)$, i.e., a name, an optional set of modifiers, a controlled object type, a role restriction set, and a set of rights. Views required by other, conditional views are recorded in the relation $RV \subset \mathcal{V} \times \mathcal{V}$, which is a partial order.

A number of simple projections on views are declared here for convenience. Let $rights : \mathcal{V} \rightarrow \mathbb{P}(Rights)$ return a view's rights and $modifiers : \mathcal{V} \rightarrow \mathbb{P}(Modifier)$ return the modifiers. The function $controlledType : \mathcal{V} \rightarrow \mathcal{T}$ maps a view to its controlled object type, $roleRestriction : \mathcal{V} \rightarrow \mathbb{P}(\mathcal{R})$ returns a view's role restriction set. The function $requiredViews : \mathcal{V} \rightarrow \mathbb{P}(\mathcal{V})$ returns the views required by a conditional view. It is defined as $requiredViews(cv) = \{v \in \mathcal{V} |\ (cv, v) \in RV\}$.

5.3.1 Configuration Constraints

Using this definition of views, it is possible to express the following configuration constraints on matrix entries. These constraints are checked whenever an attempt is made to enter a view into the matrix.

The *typing constraint* in (5.12) ensures that matrix entries are well–typed, i.e., that the object (or type) in the column of the matrix has the same type or a subtype of the view's controlled type. This constraint guarantees that the views entered into the access matrix are

always applicable to the object or object type:

$$\begin{aligned}&\forall\, M \in \mathcal{M}, v \in \mathcal{V}, p \in \mathcal{P}, o \in \mathcal{O}, t \in \mathcal{T} : \\ &\quad (v \in M[p,o] \Rightarrow \mathit{type}(o) \sqsubseteq \mathit{controlledType}(v)) \wedge \\ &\quad (v \in M[p,t] \Rightarrow t \sqsubseteq \mathit{controlledType}(v))\end{aligned} \tag{5.12}$$

In addition to the typing constraint, matrix entries must respect any *role restrictions*. If the set of roles to which a view v is restricted is non–empty, then v may only be entered into matrix rows for roles that are included in the role restriction set or in rows for subroles of these:

$$\begin{aligned}&\forall\, M \in \mathcal{M}, r \in \mathcal{R}, o \in \mathcal{O} \cup \mathcal{T}, v \in \mathcal{V} : \\ &\quad (v \in M[r,o] \wedge roleRestriction(v) \neq \emptyset) \Rightarrow \\ &\quad (r \in roleRestriction(v) \vee \exists\, r' \in roleRestriction(v) : r \leqslant r')\end{aligned} \tag{5.13}$$

Note that (5.13) does not say anything about the assignment of views with role restrictions to individual subjects. It is thus necessary to dynamically check that only those views from a principal's matrix entry are considered in access decisions that do not violate the role restriction. This check will be defined below in the definition of the access decision algorithm.

Finally, views marked as `static` are restricted such that they may only be assigned in matrix rows for roles, but not for subjects. (5.14) requires that a static view may only exist in role entries.

$$\begin{aligned}&\forall\, M \in \mathcal{M}, s \in \mathcal{S}, o \in \mathcal{O} \cup \mathcal{T} : \\ &\quad \neg\exists\, v \in \mathcal{V} : v \in M[s,o] \wedge \mathit{static} \in \mathit{modifiers}(v)\end{aligned} \tag{5.14}$$

5.3.2 Definition Constraints

We define a number of syntactic restrictions that ensure the well–formedness of view definitions. First, the operations for all rights in a view must be supported by the object type that is controlled by the view:

$$\forall\, v \in \mathcal{V} \ : \forall\, (\mathit{op}, m, p) \in \mathit{rights}(v) : \mathit{op} \in \mathit{ops}(\mathit{controlledType}(v)) \tag{5.15}$$

Second, views may only contain a single right for any given operation, so that a single view has no conflicting rights:

$$\forall\, v \in \mathcal{V} \ : \forall\, (\mathit{op}_i, m_i, p_i), (\mathit{op}_j, m_j, p_j) \in \mathit{rights}(v) : \mathit{op}_i = \mathit{op}_j \Rightarrow i = j \tag{5.16}$$

Third, as defined in section 4.2.5, an assignable view may not contain denials:

$$\begin{aligned}&\forall\, v \in \mathcal{V} \ : \mathit{assignable} \in \mathit{modifiers}(v) \Rightarrow \\ &\quad \neg\exists\, \mathit{op} \in Op : \exists\, \mathit{prio} \in Prio : (\mathit{deny}, \mathit{op}, \mathit{prio}) \in \mathit{rights}(v)\end{aligned} \tag{5.17}$$

Fourth, as defined in section 4.1.6, a virtual view defines no rights:

$$\forall\, v \in \mathcal{V}\ : virtual \in modifiers(v) \Rightarrow rights(v) = \emptyset \tag{5.18}$$

5.3.3 View extension

View hierarchies are represented by the extension relation $\leqslant$ ("extends") between views. This relation is a partial order and has the following additional properties:

$$\begin{aligned}
&\forall\, v, w \in \mathcal{V} : v \leqslant w \Rightarrow \\
&\quad controlledType(v) \sqsubseteq controlledType(w) \wedge && (5.19)\\
&\quad (roleRestriction(w) \neq \emptyset \Rightarrow roleRestriction(v) \neq \emptyset \wedge && (5.20)\\
&\qquad \forall\, r \in roleRestriction(v) : \exists\, r' \in roleRestriction(w) : r \leqslant r') \wedge \\
&\quad requiredViews(w) \subseteq requiredViews(v) \wedge && (5.21)\\
&\quad static \in modifiers(w) \Rightarrow static \in modifiers(v) \wedge && (5.22)\\
&\quad rights(w) \subseteq rights(v) \wedge && (5.23)\\
&\qquad \forall\, (op, m, p) \in (rights(v) \setminus rights(w)) : m = allow
\end{aligned}$$

Property (5.19) requires an extending view to control the same or a more derived object type than its base views. Property (5.20) means that an extending view must be role restricted if the base view is role restricted. Also, all roles in an extending view's role restriction must be subroles of the roles in the base view's role restriction, but there may be fewer roles than in the base view's role restriction. All required views of a base view are also required by the extending view (5.21). An extending view inherits the `static` modifier from its base view (5.22), which means that it also inherits the restriction that it cannot be entered into a matrix row for subjects.

Property (5.23) requires all rights introduced in the extending view to be permissions, and that the extending view has at least as many rights as the base view. Note that we do not exclude extension of multiple base views here so that view hierarchies can be designed along object type inheritance hierarchies, which permit multiple inheritance of object types.

5.3.4 Priorities and conflict resolution

Conflict resolution is based on priorities, in particular on the semantics of `strong`, which was presented in section 4.1.4. To guarantee this semantics, it is necessary to restrict redefinitions of rights such that strong rights may not be redefined in extending views. This latter restriction is expressed by the following additional property of view extension:

$$\forall\, v, w \in \mathcal{V} :\ v \leqslant w \Rightarrow \tag{5.24}$$
$$\forall\, (op, m', p') \in rights(v), (op, m, p) \in rights(w) :$$
$$(p = weak \wedge (m' = m \Rightarrow p' = strong))$$

Property (5.24) allows redefinitions that change an inherited right's mode, but because of property (5.23) only denials can be redefined as permissions and not vice versa. A redefinition may also be used to make a right `strong` without changing the right mode, so that it cannot be further redefined in more derived views.

Given the definition of `strong`, it is possible to dynamically resolve all potential conflicts between positive and negative rights except for one case. Section 4.1.4 explained that conflicts between permissions and denials can always be resolved if the views that contain these rights are related by view extension. The more derived view takes precedence in this case. Because of property (5.24), this view's rights are also guaranteed to be at least as strong as the rights in the base view.

If the two views are unrelated, conflicts can still be resolved if the two rights have different priorities. If rights priorities are equal and both rights are strong, the corresponding view definitions must be rejected because they could potentially violate the semantics of `strong`. This case can be excluded by verifying the following definition constraint. If this verification fails, the definitions must be rejected. Note that checking this condition requires global analysis, i.e., all existing views must be checked:

$$\forall\, v, w \in \mathcal{V}, t_v, t_w \in \mathcal{T} : \tag{5.25}$$
$$(t_v = controlledType(v) \wedge t_w = controlledType(w) \wedge$$
$$t_v \not\sqsubseteq t_w \wedge t_v \not\sqsupset t_w \wedge v \not\leqslant w \wedge v \not> w) \Rightarrow$$
$$\forall\, (op, m1, p1) \in rights(v), (op, m2, p2) \in rights(w) :$$
$$(m1 \neq m2 \wedge p1 = p2) \Rightarrow p1 \neq strong$$

5.4 Authorization Scheme

In addition to the protection state represented by an access matrix, an access model describes an *authorization scheme*, which defines a set of rules that controls how the protection state may evolve. Traditionally, the authorization scheme in matrix–based models is expressed in terms of *commands*. Commands are parameterized and conditional applications of the six primitive matrix operations which create or delete rows (principals), create or delete and columns (objects), and enter or delete authorizations in matrix entries [Harrison et al., 1976].

In the following description, creating and deleting principals or objects is understood as making principals or objects known (or unknown) to the policy. These operations do not directly affect the life cycles of principals or objects, which we assume are not under the control of a policy administrator. This section does not add new concepts to the ones already discussed

but rather rephrases and formalizes discretionary assignments and schemas in terms of enter and delete commands on the access matrix.

5.4.1 Creating and Deleting Principals

It is important to distinguish between the lifetime of principals and the matrix rows that hold policy–specific authorization information. These lifetimes are controlled by potentially separate management authorities. Moreover, there is a varying degree of control that managers can exercise over these lifetimes: the lifetime of roles is completely controlled by the management authority responsible for credentials management, because they can create and delete roles in their administrative domains at will. Policy managers do not generally create or delete subjects, which are potentially remote processes running "somewhere" in the distributed system. Because policies refer to subjects using locally known names, managers can effectively delete subjects by invalidating the name or the binding between the subject and its name. The name and its binding are assumed to be under the control of a trusted certification authority for names, which is part of the public key infrastructure.

Principals cannot independently create or delete other principals at all. Even though the creation of processes is not controlled, it is not possible for an arbitrary subject to furnish a newly created process with credentials that would turn it into a principal. It can only share its credentials and secrets with such a process, and thus effectively create a copy of itself. Creating new principals is only possible with the cooperation of trusted authorities that sign credentials.

Effects on the Protection State

Creating new and empty matrix rows has no immediate consequences for the overall protection state. In contrast, deleting a row might affect the protection state if the matrix row still contains views, which are deleted with the principal. This operation might have unexpected and undesirable effects, e.g., if no other principals hold the same views and the views are the last ones on the respective objects. If positive authorizations are deleted, existing applications might not be able to continue operation because functionality is denied for lack of permissions, so liveness is not guaranteed. In the case of application protocols involving discretionary assignments, the deleted principal might have been the last source of a particular view or it might have been the last principal allowed to revoke a particular view from another principal, which can now only be done by an administrator. If the principal that is removed from the matrix is a role that has subroles, the operation also effectively removes views from these subroles. If views with denials are removed from subroles, these roles' permissions are effectively *increased*, which may be unsafe. For these reasons, deleting principals which still hold views and, in the case of role principals, have subroles is thus a very sensitive operation and must be performed with great care. Deleting principals that do not hold any views and have no subroles is not a critical operation, however.

5.4.2 Creating and Deleting Objects

The operations to add or remove matrix columns for objects and object types should never be explicitly performed by policy managers. Rather, these operations are invoked implicitly as the consequences of operations by the manager of the policy domain. Since domains are not explicitly modeled here, there is no difference between a domain operation and the matrix operation, but it is important to point out the separation of concerns between these two management roles.

In our model, an object is protected by a policy after the object becomes a member of the policy domain. From this point on, every access to the object is checked for compliance with the domain's access policy. Thus, assigning an object to a policy domain means adding a matrix column for that object. The inverse operation, removing an object from the domain and thus from the matrix, has an analogue semantics and simply means that accesses are no longer controlled by the policy. Any views on this object are no longer meaningful and are therefore deleted with the object.

Note that both operations are not tied to the object's actual life cycle. An object may be added to a domain at creation time, but it is also possible to create initially unprotected objects that only later enter a domain. Moreover, objects may be moved between different domains. The only real lifecycle dependencies are obviously that an object cannot be protected before it is created and after it is removed from the system. It is important, however, to define means that ensure that an object can be protected by an access policy from its creation onwards if necessary. These issues are discussed in more detail in chapter 7.

Columns in the access matrix are also used to hold authorizations on object type extensions, but the life cycles of types are not defined in CORBA at all and there is no management role directly responsible for object types. For the purposes of this model, we assume that a matrix column for a type is created implicitly on two occasions. First, a type is added to the matrix when the first object of that type is added. Second, for practical reasons it is desirable that views can be assigned on types even before the matrix knows about objects of that type. Thus, a type is added implicitly whenever a view assignment is performed on a type that is not already known. Type columns are never removed from the matrix.

5.4.3 Creating and Deleting Matrix Entries

Administrators may assign or remove views by directly operating on the access matrix. In addition to these administrator actions, the access control model supports two ways of statically defining commands that are either invoked by principals, viz. *discretionary assignments* (and removals), or automatically by the protection system itself, viz. *implicit assignments* (and removals).

Discretionary Assignments

Discretionary assignments and removals of views are possible with assignable views. A principal may assign views on an object that he holds himself if these views are assignable and the assign option was set when the principal received these views. He may then choose whether this option should also be set when the view is passed on to the recipient principal, which may be both a role or a subject. If the recipient does not already possess the view, it is assigned and the assigning principal receives the right to revoke the view again. If a view is removed that was assigned with the assign option set, the view must be recursively revoked from all principals to which the recipient principal assigned the view. Cascading revocations are not modeled here for brevity.

The ASSIGN command is defined to model discretionary assignment.

command ASSIGN($M : \mathcal{M}, src : \mathcal{P}, v : \mathcal{V}, target : \mathcal{O} \cup \mathcal{T}, dest : \mathcal{P}, ass\_opt : bool$)
if
$v \notin M[dest, target] \wedge (M, v, src, target) \in may\_assign$
then
$v \in M'[dest, target] \wedge$
$(src, M', v, dest, target) \in may\_remove' \wedge$
$(ass\_opt = true \Rightarrow (M', v, dest, target) \in assign\_options')$
end

The ASSIGN command relies on a relation $may\_assign \subseteq \mathcal{M} \times \mathcal{V} \times \mathcal{P} \times (\mathcal{O} \cup \mathcal{T})$ that determines whether a principal src that is the source of a view assignment is an element of the calling, potentially compound principal. This relation is defined in (5.26). The principal src receives the remove right for a view in a matrix entry as a consequence of assigning the view. The remove right for the assigning principal src is recorded in the new state of the relation may_remove[1], which is referred to as may_remove'. Note that src must be a simple principal, i.e., either a subject or a single role because a remove right cannot be recorded for compound principals.

$$\begin{aligned}
&\forall\, M \in \mathcal{M}, v \in \mathcal{V}, src \in \mathcal{P}, target \in \mathcal{O} \cup \mathcal{T} : \\
&\quad (M, v, src, target) \in may_assign \Leftrightarrow \\
&\qquad src \in calling_principal \wedge\ v \in M[src, target] \wedge \\
&\qquad assignable \in modifiers(v)\ \wedge \\
&\qquad (M, v, src, target) \in assign_options
\end{aligned} \tag{5.26}$$

The may_assign relation relies on another relation $assign\_options \subseteq \mathcal{M} \times \mathcal{V} \times \mathcal{P} \times (\mathcal{O} \cup \mathcal{T})$ that records whether the assign option is set for a given view in a given matrix entry.

[1] The contents of the may_assign and may_remove relations could have been stored in the access matrix. The matrix was deliberately kept simpler in the previous sections for presentation purposes, however.

When a view is entered with the ASSIGN command, a new tuple is added to this relation if the assign option is set. The new state is referred to as $assign\_options'$. The identifier $calling\_principal$ denotes the set of principals that constitute the calling principal.

The REMOVE command refers to the may_remove relation to determine whether the principal is allowed to remove the view. The remove right itself is removed after successful removal of the view. Unlike for the ASSIGN command it is not necessary to explicitly name a simple principal as the remover in the arguments to the REMOVE command because for any given view there can only be a single principal with a remove right. The condition of the REMOVE command is only true if this principal is an element of the calling principal.

command REMOVE($M : \mathcal{M}, v : \mathcal{V}, target : \mathcal{O} \cup \mathcal{T}, recp : \mathcal{P}$)
if
$v \in M[recp, target] \wedge$
$\exists\, p \in calling\_principal : (p, M, v, recp, target) \in may\_remove$
then
$v \notin M'[recp, target] \wedge$
$(p, M', v, recp, target) \notin may\_remove'$
end

Implicit Assignments

Implicit assignment and removal operations are specified using schemas. This section formalizes schemas and their definition constraints.

$$Clauses \subset MOp \times \mathcal{V} \times \mathbb{P}(\mathcal{R} \cup \{\texttt{caller}\}) \times (\mathcal{T} \cup \{\texttt{this}, \texttt{result}\}) \times bool \quad (5.27)$$

$$OC \subset Op \times Clauses \quad (5.28)$$

$$Schemas \subset \mathcal{T} \times OC \quad (5.29)$$

A schema is modeled as a tuple (t, OC), where $t \in \mathcal{T}$ and OC is a relation between operations and clauses. Schema names are not explicitly modeled because schemas are never referenced by name. The characteristic property of a schema is the IDL type which it observes. Multiple schemas with different names observing the same type are treated as a single schema definition for that type.

A clause is a tuple (m_op, v, R, t, ass_opt), i.e., a matrix operation name from $MOp = \{assign, remove\}$, a view, a set of recipients, a target object or type, and a boolean value that specifies whether the recipient should be allowed to assign an assignable view to other principals. The set of recipients to which schemas can refer is the set of roles extended with the identifier `caller` that is bound at to the calling subject runtime. Similarly, the target is either a type or one of the two object identifiers `this` and `result`.

A number of restrictions are verified statically to catch definition errors, as explained in section 4.1.5.2. A schema's clauses may only refer to operations in the interface type for which

the schema is defined (5.30). If a view occurring in a schema clause is marked as static, the recipient must not be the calling subject (5.31). If the assign option is set, the matrix operation must be an assignment and the view must be assignable (5.32). If two clauses for the same operation both assign and remove the same view and the intersection of the recipients sets is non–empty, then the two targets must not both be objects. Otherwise, it cannot be guaranteed that the assign and the removal operation do not operate on the same matrix entry, i.e., that the clause is free of conflicts (5.33).

$$
\begin{aligned}
&\forall\,(t, oc) \in Schemas:\ \forall\,(op, (m\_op, v, R, o, ass\_opt)) \in oc: \\
&\qquad op \in ops(t) \wedge && (5.30)\\
&\qquad (static \in modifiers(v) \Rightarrow \texttt{caller} \notin R) \wedge && (5.31)\\
&\qquad (ass\_opt = true \Rightarrow (m\_op = assign \wedge assignable \in modifiers(v))) && (5.32)\\
&\qquad \forall\,(op', (m\_op', v', R', o', ass\_opt')) \in oc: && (5.33)\\
&\qquad\qquad (op' = op \wedge m\_op' \neq m\_op \wedge v' = v \wedge R' \cap R \neq \emptyset) \Rightarrow \\
&\qquad\qquad\qquad (o \in \{\texttt{this}, \texttt{result}\} \Rightarrow o' \in \mathcal{T})
\end{aligned}
$$

Similar to condition (5.33), it must be verified that there can be no conflicts between clauses in different schemas that observe operations on objects of related types:

$$
\begin{aligned}
&\forall\,(t, oc), (t', oc') \in Schemas: (t \sqsubseteq t' \vee t \sqsupset t') \Rightarrow && (5.34)\\
&\qquad \forall\,(op, (assign, v, R, o, ass\_opt)) \in oc, (op', (remove, v', R', o', ass\_opt')) \in oc': \\
&\qquad\qquad (op' = op \wedge v' = v \wedge R' \cap R \neq \emptyset) \Rightarrow \\
&\qquad\qquad\qquad (o \in \{\texttt{this}, \texttt{result}\} \Rightarrow o' \in \mathcal{T})
\end{aligned}
$$

To model the fact that schemas listen to operations, we introduce a matrix command NOTIFY that is called automatically by the system whenever an operation returns.

command NOTIFY($op : Op, target : \mathcal{O}, schemas : \mathbb{P}(Schemas)$)

$$
\begin{aligned}
&\forall\,(t, oc) \in schemas: \\
&\qquad type(target) \sqsubseteq t \Rightarrow \\
&\qquad\qquad \forall\,(op', (m\_op, v, R, o, ass\_opt)) \in oc: \\
&\qquad\qquad\qquad op = op' \Rightarrow \\
&\qquad\qquad\qquad\qquad \forall\, rcp \in R: \\
&\qquad\qquad\qquad\qquad\qquad ((m\_op = assign \wedge v \notin M[rcp, o]) \Rightarrow \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\, v \in M'[rcp, o] \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad opt = true \Rightarrow \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (M', v, rcp, o) \in assign\_options')) \wedge \\
&\qquad\qquad\qquad\qquad\qquad ((m\_op = remove \wedge v \in M[rcp, o]) \Rightarrow \\
&\qquad\qquad\qquad\qquad\qquad\qquad (v \notin M'[rcp, o] \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad \neg\exists\, p \in \mathcal{P}: (rcp, M', v, p, o) \in may\_remove'))
\end{aligned}
$$

end

The command evaluates all applicable schemas and performs the necessary matrix enter and remove operations. If an assignment is made with the assign option set the command records the permission to pass on the view. If a removal is performed, the command ensures that no remove rights remain for the view that was removed.

5.5 Access Decisions

Decisions about attempted accesses are made by an access decision function $adf : Access \times \mathcal{M} \rightarrow \{allow, deny\}$. For an access request $(op, o, s, active\_roles)$, the function first needs to determine which policy must be applied to the target object, i.e., find the access matrix M. In a distributed system, this may involve finding out about the target's domains and looking up the policy in the domain scope. This process is not described here.

To look up a compound principal's views on object o, the views on o that were assigned to the principal's subject s and kept in the matrix entry $M[s, o]$ have to be combined with the views assigned to s on the object's type and its supertypes. This set of views then has to be combined with the views assigned to the principal's active roles. The function $validSubjectViews : \mathcal{M} \times \mathcal{S} \times \mathcal{O} \times \mathbb{P}(\mathcal{R}) \rightarrow \mathbb{P}(\mathcal{V})$ retrieves views from the principal's subject entries and returns all views that either have no role restriction or whose role restrictions match the currently active roles for the access. To retrieve the subject's views the function relies on the function $subjectViews$ in (5.36) which is defined similar to $roleViews$ in (5.4).

$$\begin{aligned} &validSubjectViews(M, s, o, active\_roles) = \\ &\quad \{\, v \in \mathcal{V} \mid v \in subjectViews(M, s, o) \wedge (roleRestriction(v) \neq \emptyset \Rightarrow \\ &\qquad \exists\, r \in active\_roles, r' \in roleRestriction(v) : r \leqslant r')\} \end{aligned} \tag{5.35}$$

$$subjectViews(M, s, o) = M[s, o] \cup subjectViews_{type}(M, s, type(o)) \tag{5.36}$$

$$\begin{aligned} &subjectViews_{type}(M, s, t) = \\ &\quad \{v \in \mathcal{V} \mid v \in M[s, t] \vee \exists\, t' \in \mathcal{T} : t \sqsubseteq t' \wedge v \in subjectViews_{type}(M, s, t')\} \end{aligned} \tag{5.37}$$

The complete set of views available to a principal for an access can be retrieved using the function $principalViews : \mathcal{M} \times \mathcal{S} \times \mathcal{O} \times \mathbb{P}(\mathcal{R}) \rightarrow \mathbb{P}(\mathcal{V})$, which combines the valid subject views with the views assigned to the principal's roles, which are retrieved using the function $roleViews$ that was defined in 5.4:

$$\begin{aligned} &principalViews(M, s, o, active\_roles) = \\ &\quad validSubjectViews(M, s, o, active\_roles) \cup \\ &\quad \{\, v \in \mathcal{V} \mid \exists\, r \in active\_roles : v \in roleViews(M, r, o)\} \end{aligned} \tag{5.38}$$

Before this set of views can be consulted for permissions or denials, however, those condi-

tional views must be removed from the set that have unresolved dependencies on other views, i.e., that require views not in the set. The function $resolvedViews : \mathbb{P}(\mathcal{V}) \rightarrow \mathbb{P}(\mathcal{V})$ filters out views with unresolved dependencies:

$$resolvedViews(V) = \{\, v \in V \mid \forall\, v' \in requiredViews(v) : v' \in V \,\} \tag{5.39}$$

The set of applicable views for a principal's access $(op, o, s, active\_roles)$ is thus

$$applicableViews(M, s, o, active\_roles) = resolvedViews(validSubjectViews(M, s, o, active\_roles)) \tag{5.40}$$

To determine whether the access is allowed, a permission for the operation op must be found in the views returned by $applicableViews$. If no permission is found, the access is denied. Because (5.25) statically excludes conflicts between strong rights, the access is allowed if a permission is found that has a strong priority. No further checks for denials are necessary in this case. If a permission with weak priority is found, however, it is necessary to check for conflicting denials. If no denial is found, the access is granted. If a strong denial is found, it is denied. A conflict between two (or more) weak rights, however, needs to be resolved.

Let $V_{conflict}(op) = \{v_1, .., v_n\} \subseteq applicableViews$ be the set of views with conflicting rights regarding operation op. All subsets of $V_{conflict}$ that are partially ordered by the relation $\leqslant$ on views are replaced by the single "most derived" view, so the conflicting set now contains only unrelated views. Denials take precedence in the case of conflicts between unrelated views, so if any denials remain, the access is denied. Otherwise, it is allowed.