7 An Infrastructure for View–Based Access Control

This chapter describes the system architecture and the implementation of an access control infrastructure that supports the abstract concepts provided by VPL. The overall approach is to comply with the OMG Security Service wherever possible and thus to implement standardized IDL interfaces where they are applicable. The degree of freedom for design decisions is thus a priori limited. The goal of this presentation is to show that the proposed access model is both implementable within the CORBA Security Framework and practically feasible, so care was taken to minimize the performance overhead that is introduced by applying fine–grained access control. All implementations were done in Java and run on the Java/CORBA implementation JacORB [JacORB, 2001], of which the author of this thesis is the main developer and maintainer.

A second goal of this chapter is to give an impression of how this infrastructure is managed in practice. Therefore, a number of graphical management tools were designed to demonstrate the usefulness of the general approach to management. These tools are prototypes, however, and have not been designed to meet realistic ergonomic requirements. Also, comprehensive usability studies involving actual users were beyond the scope of this thesis.

The required infrastructure components can roughly be divided into two categories, viz. language and deployment tools versus runtime infrastructure and management components. Section 7.1 describes language and deployment tools. Section 7.2 begins the discussion of runtime components and presents the implementation of the access control mechanism. Section 7.3 presents the design of the role service, section 7.4 focuses on the domain management component. The policy server is described in section 7.5. Some elements of the architecture have been previously described in [Kiefer, 2000], [Brose et al., 2001a], [Noffke, 2001], and [Brose, 2001]. A discussion and evaluation of the architecture presented in this chapter can be found in 7.6.

7.1 Language and Deployment Tools

The first tool that is required in the life cycle of an access policy is a VPL compiler, which is needed to type–check policy descriptions and verify other language constraints. The final product of the policy development stage is a policy design document that is delivered as a

descriptor file with the application and is a suitable input for deployment tools in the target environment. The *Extensible Markup Language* (XML) [W3C, 2000] is an established standard format for descriptors of this kind in environments such as Enterprise JavaBeans (EJB) [Sun Microsystems, 2000] or the CORBA Component Model (CCM) [OMG, 1999c]. The advantage of delivering descriptors in this format rather than in the original VPL syntax is that XML is an open standard for which a variety of tools is publicly available. It is thus simple to develop independent deployment tools. Therefore, the VPL compiler is designed as a combination of a VPL–to–XML precompiler and a backend verifier that accepts XML files as input. Figure 7.1 depicts the components involved in the development process.

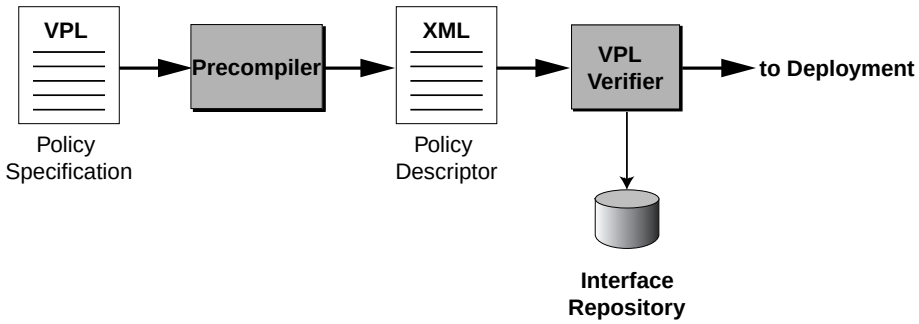

Figure 7.1: Policy development.

The precompiler performs a simple syntactic translation from VPL to XML. It is implemented in Java and based on the VPL grammar in Appendix A. The XML grammar for VPL is defined by a Document Type Definition (DTD) which is also given in Appendix A. Language constraints are checked by the VPL verifier, which reads the XML document created by the precompiler. This verifier needs compile–time access to a CORBA Interface Repository in order to verify that view and schema definitions are compatible with the IDL types to which they refer.

After the development process is completed, the descriptor file is delivered with the application and must be deployed in the target environment. The policy descriptor contains static definitions that are deployed to two different repositories. Role declarations and constraints are stored in a role repository, whereas view and schema definitions are stored in a functionally separate view repository. The contents of these repositories are accessed through appropriate server components, which are presented later in this chapter. This process is depicted in figure 7.2. The interfaces of both repositories share a common ancestor DeploymentTarget, which defines a simple operation to upload XML descriptions. This interface is shown in figure 7.3.

When a policy is deployed, it must be verified again using an Interface Repository in the target environment. The policy may need to be modified to avoid name clashes with existing policies. References to role names in the policy may also be adjusted to make use of existing role definitions in the target environment.

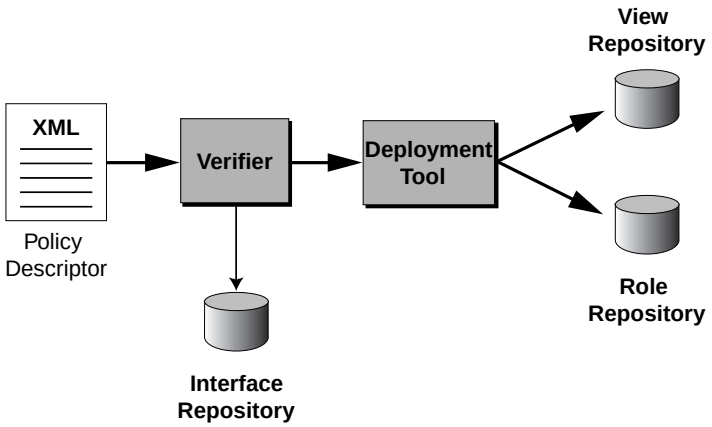

Figure 7.2: Policy deployment.

7.2 Implementation of the Access Control Mechanism

Any mechanism for controlling access to CORBA objects must be able to intercept and check all possible accesses, i.e., the mechanism must be interposed between the object and its callers and not be bypassable. This property is known as *complete mediation* in [Department of Defense, 1985], where the entire mechanism is termed *reference monitor*. The most straightforward allocation of this functionality is to perform access checks in the address space of the process hosting the object implementation. CORBA interceptors are a convenient way to implement this mechanism and the CORBA Security Service specifies an access control interceptor for this purpose. The implementation presented here also uses interceptors. Figure 7.4 illustrates how access requests are intercepted and then checked by an AccessDecision object. If the access is allowed, it is passed on to the target object, otherwise a reply message signaling the CORBA system exception NO_PERMISSION is returned to the caller. This exception is not shown in the diagram.

The figure also illustrates that the AccessDecision object keeps a session context for each

```
typedef sequence<octet> XMLDescriptor;
exception DeploymentException { string reason; };

interface DeploymentTarget
{
    void processDescriptor( in XMLDescriptor desc )
        raises (DeploymentException);
};
```

Figure 7.3: The DeploymentTarget interface.

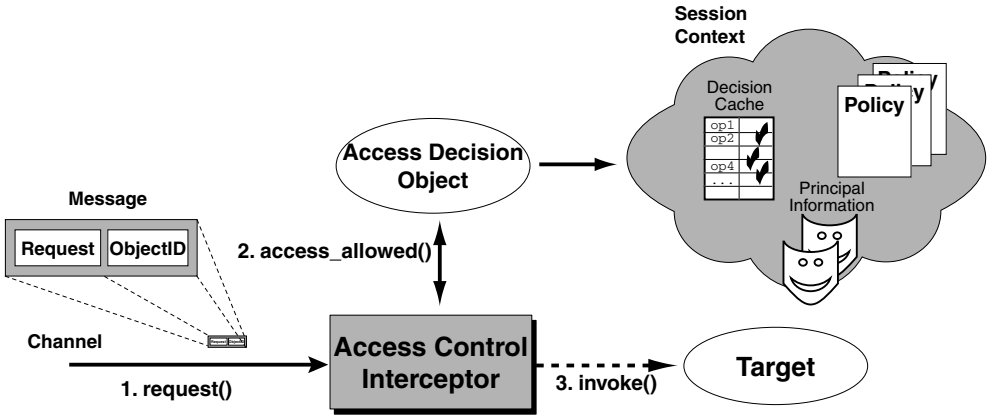

Figure 7.4: Access Control with Interceptors.

principal–target pair. This session context contains the access policy information that applies to the target, the principal information about the caller, and a cache of previously made access decisions. The interceptor establishes this context upon the first request to a given target and the AccessDecision object retrieves and updates it, if necessary.

This approach was chosen mainly out of performance considerations. In the general case, the access control information required by the AccessDecision object must be assembled from multiple, potentially remote sources and thus requires additional remote invocations. To ensure an acceptable overall system performance, this overhead must not be incurred on each access. It is thus an explicit design decision to assemble the entire set of required access control information initially and then reuse it for subsequent accesses without any further remote invocations in the infrastructure. While this requires considerable effort at the time a client accesses a target object for the first time, the cost of the initial binding between client and target is dominated by the cost of establishing a protected transport connection. The initial establishment of plain TCP connections alone is comparatively costly, but further cost is incurred during connection authentication using SSL, which involves a number of relatively expensive cryptographic operations for key exchanges and certificate verifications. Thus, the additional communication cost for retrieving access control information is negligible at this time.

7.2.1 Authentication

The access control interceptor must rely on authentic access control information about the caller and its roles. For modularity, the authentication mechanism that authenticates the calling principals is not implemented within either the ORB itself or the access control interceptor but in a separate interceptor, which is responsible for obtaining role information of the principal. This interceptor and the management of roles in general are described in more detail in section 7.3. The subject element of the calling principal is taken from the access ID that the ORB security service delivers in the ReceivedCredentials object. In JacORB, this identity is the subject

name taken from the X.509 certificate which is delivered by the underlying SSL transport layer. Technically, this is accomplished using yet another interceptor, but conceptually it is part of the ORB security layer.

Figure 7.5 illustrates the runtime objects involved in authenticating a client as a principal. The client–side ORB security service uses a PrincipalAuthenticator object to create a Credentials object that contains a chain of X.509 certificates, which are used by the ORB to initiate an SSL connection to the server. The server–side ORB extracts the client subject's public key certificate and stores it, as mentioned above, in the ReceivedCredentials object. The figure also shows a server–side interceptor that is only called on returning from the processing of a request. This schema interceptor is responsible for realizing the semantics of schemas and is explained below in section 7.2.2.

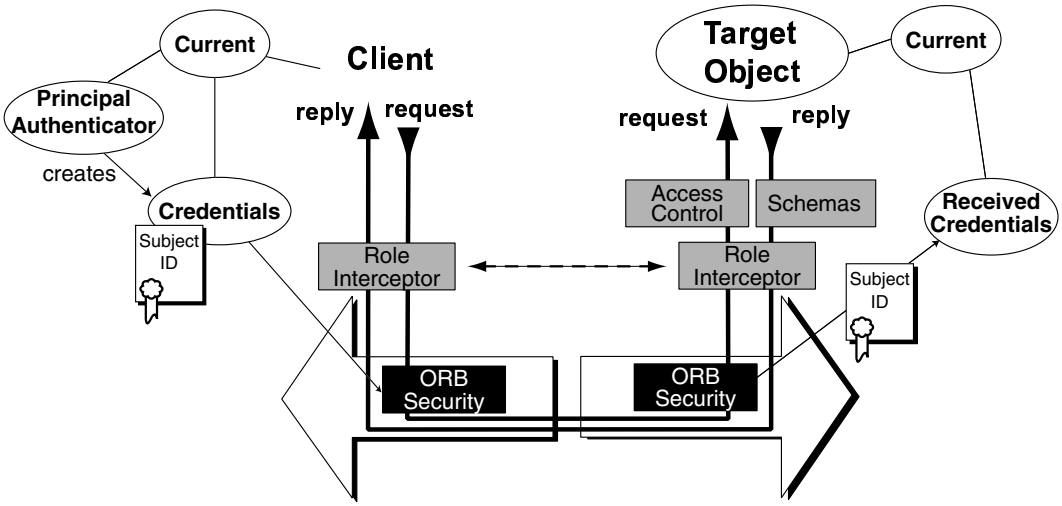

Figure 7.5: Authentication and Interceptors.

A session context contains policy information in addition to the principal information that is provided by the role layer. When the AccessDecision object accesses the session context for the first time, this information is obtained by determining all access policies that apply to the target object and by retrieving all views for the currently speaking principal, i.e., for the subject and all its active roles. Since a target object itself does not know its policies, it is necessary to first find the security policy domains of which the target is a member. These domains are then queried for access policies. To perform these retrieval operations the runtime infrastructure needs access to a domain service and a policy service. Both services are presented in more detail later in this chapter.

At this stage, the session context acts as an authorization cache and contains the complete access control information for the target object, so the AccessDecision object can decide all possible accesses without further remote invocations. However, both the principal information and the authorization information may change dynamically and thus need to be updated. As will be described in more detail in the section on roles, principal information is represented in

certificates with a bounded lifetime. If one of the role certificates in the session context expires, the session context is invalidated and a new context must be established before further accesses are possible.

The general design approach is that authorization information is not updated dynamically during the lifetime of a session. Thus, dynamic changes of authorizations in the policy are only visible after sessions as a whole are destroyed or the authorization data times out. This happens automatically after a configurable timeout has expired. In addition to timeout expiration, there are two other events that cause a session's authorization data to become invalid. The first of these events is that a local schema has triggered an authorization change. In this case, the session's authorization information is invalidated and refetched on the next access attempt so that application level policies that rely on these changes are not forced to use otherwise unreasonably short timeout values — or need to handle NO_PERMISSION exceptions caused by local, out–dated cache entries.

The second event that invalidates cached authorization data is that a policy becomes unreachable. This condition is detected using a simple heartbeat mechanism: a separate thread "pings" all policies on which the session depends in regular intervals. If any of these policies cannot be reached, all dependent access information is invalidated so that no accesses are possible until the policy can be contacted again. This behavior effectively switches off session–based caching immediately when loss of communication is detected and was introduced to make the infrastructure "fail safe", i.e., to stop operations immediately when potentially unsafe conditions are detected, e.g., when an attacker tries to hide authorization changes by blocking access to a policy. A downside of this approach is that a policy server becomes a single point of failure so that performing a successful denial of service attack on a policy server can bring an entire system or large portions of it to a halt. To prevent attacks of this kind, it is necessary to replicate policies.

7.2.2 Schemas

As noted above, the assignments and removals specified by a schema definition are carried out by a dedicated interceptor which observes successful returns from operations that were processed locally, i.e., on an object implementation that is currently incarnating a specific CORBA object. To do so, the interceptor needs to know about all schema definitions that apply to the object from which processing returns. This information is retrieved during the initial session setup and stored in the session context.

The current implementation does not support a rollback and thus cannot ensure that the authorization changes effected in the course of a delegated invocation are atomic, as it was argued in section 4.1.5.2. In the case that an invocation is aborted that has already passed through a number of objects, its partial effects on the protection state cannot be undone. As explained in section 7.3.1, the effects of schema activities are generally only visible within the same access session and in those new sessions that are created after schema changes have become effective.

7.3 Role Management

As presented in chapter 4, the access model relies on a role–based notion of principals. The infrastructure contains the following components to support roles and role management:

1. Role interceptors as part of the access control mechanism,
2. a role server that issues role membership certificates on request,
3. a role repository that stores role definitions and constraints from policies, and
4. a graphical role management tool for managers.

The presentation in this section takes a bottom–up approach and first discusses the interceptor mechanism by which role information is exchanged between clients and servers. The role server and the repository that provide this information are described in section 7.3.2, and finally a management tool is presented.

7.3.1 Role Interceptors

The client–side role interceptor in figure 7.5 is responsible for extracting information about the principal's roles from its credentials and transmitting it. Figure 7.6 illustrates the task of the client–side role interceptor in more detail.

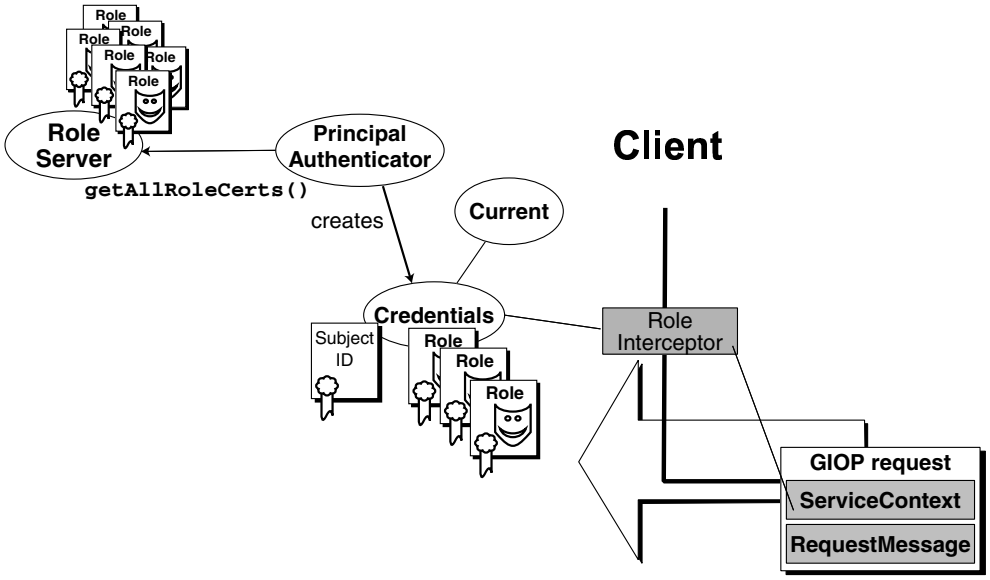

Figure 7.6: A client–side role interceptor.

The client–side PrincipalAuthenticator object does not only store a public key certificate containing the subject's name in the Credentials object, it also contacts the role server and

retrieves all role certificates for the subject. These certificates are also stored in the Credentials object. Role certificates are signed by the role server's public key and contain the subject's public key and a single role name in an extension field of the certificate.

The basic model of interaction with the role server, as just described, is "client–pull" rather than "server–pull", where the server–side AccessDecision object would contact the role server to find out for which roles a subject principal may speak. The motivation for this approach is that security–aware clients can select a subset of roles for subsequent accesses rather than presenting the full set of certificates. This can be desirable for two reasons. First, a client might wish to use only the minimal set of roles required to carry out a certain task in order to restrict the potential damage of Trojan horses or of its own mistakes in using the application. Second, our access model supports assigning explicit denials to certain roles so that a client might in fact be more restricted in his actions when using the union of all his roles than when using only a specific subset of roles. A client can manipulate the set of role certificates stored in its Credentials object using operations from the standard CORBA Security API.

When the client invokes an operation on the target, the client–side role interceptor checks whether there is an existing security session between the client and the server, in which case it simply reuses the current session. Otherwise, it establishes a new session and transmits the role certificates that it retrieves from the Credentials object. The session setup protocol piggybacks on the standard CORBA GIOP request protocol by using the opaque ServiceContext field of the GIOP request for the transmission of role certificates and the exchange of session identifiers.

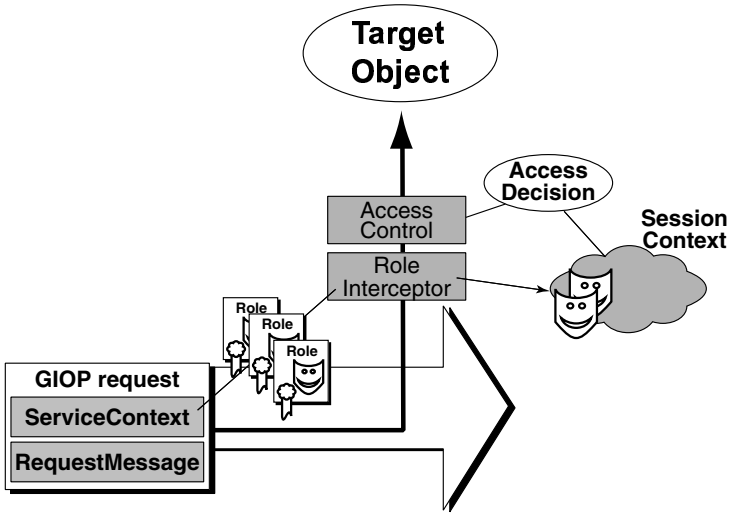

Figure 7.7: A server–side role interceptor.

If a new session is established, the server–side role interceptor extracts these role certificates from the GIOP service context, verifies their validity and the role server's signature, and stores the role names in a newly created session object. This is illustrated in figure 7.7. The overall life time of a new session is equal to the shortest certificate validity period, so the ses-

sion as a whole will expire as soon as the first role certificate becomes invalid. The access control interceptor delegates the access decision to the AccessDecision object, which consults the session context to find out which roles belong to the calling principal.

7.3.2 The Role Server and Repository

The role server object has a simple interface to retrieve role information in certificates, which is shown in figure 7.8. The main operation is getAllRoleCerts(), which returns certificates for all roles of which a subject is a member.

```
interface RoleServer
{
    RoleIdSeq getRoleNames()
        raises( UnknownPrincipal );

    EncodedCertSeq getAllRoleCerts( in EncodedCert subject )
        raises( UnknownPrincipal );

    EncodedCertSeq getRoleCerts( in RoleIdSeq roles )
        raises( UnknownRole, UnknownPrincipal );

    EncodedCert getRoleCert( in RoleId name )
        raises( UnknownRole, UnknownPrincipal );

    EncodedCert getCACert();
};
```

Figure 7.8: The RoleServer interface.

The role server as such is stateless and acts only as a factory for role membership certificates. Generally, it does not need to establish the identity of clients requesting role certificates because a role certificate itself does not convey any authorizations and because principals must authenticate separately anyway. However, the role server may still choose to authenticate clients and restrict access to role information if this information is considered sensitive and should not be passed around freely.

The role server does not store any role information or certificates itself but relies on the role repository for persistent storage. The role repository contains role definitions that were deployed from policy descriptors. As shown in figure 7.9, the individual roles are scoped by the defining policy to avoid name clashes. The repository interface supports a number of operations to traverse the role hierarchy which are needed by policy objects, as explained in section 7.5.

The role repository is also the component that stores information about the assignments of subjects to roles. For management purposes, these assignments are always performed indirectly

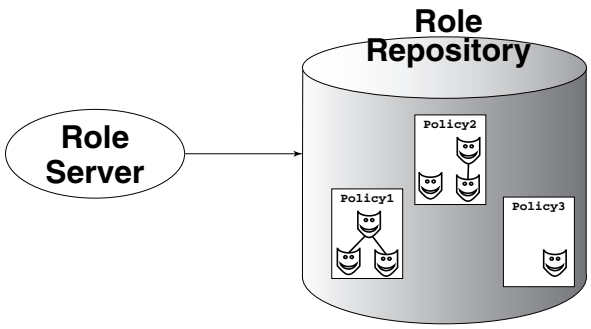

Figure 7.9: Role server and repository.

through groups, which is explained in the next section. Groups as a concept exist in the role management infrastructure only and are transparent to clients and to access policies.

7.3.3 A Role Management Tool

When a policy descriptor is deployed, a number of role descriptions are stored in the role repository together with information about any role constraints or hierarchical relationships between roles. No subjects are assigned to roles at this time. Managers can modify the contents of the role repository through a graphical user interface, which provides an editable graphical representation of the contents of the role repository. Figure 7.10 shows the roles for the conference application example in the panel on the right side. The management tool offers different kinds of edges between graph nodes representing roles: inheritance, mutual exclusion and prerequisite edges.

The left half of the window displays information about subject groups in the form of a *group graph*. The assignment of groups to roles is performed by drawing an arrow from a group in the left panel to a role in the right panel.

7.3.3.1 Groups

Managing individual subjects and their assignments to roles is cumbersome and may even become altogether infeasible if the number of subjects exceeds a certain limit. To cope with scalability, subjects are grouped together for management purposes in virtually all existing access control systems. Groups are a simple management abstraction that help to reduce the number of entities a manager must deal with when assigning roles. There is no textual syntax to define groups and group hierarchies here because groups are not designed as part of the application–oriented part of the policy. They are only used by user managers with the support of graphical tools.

The simple group model used here is similar to [Osborn and Guo, 2000]. It allows managers to create groups as named sets of principals and to set up hierarchical relationships be-

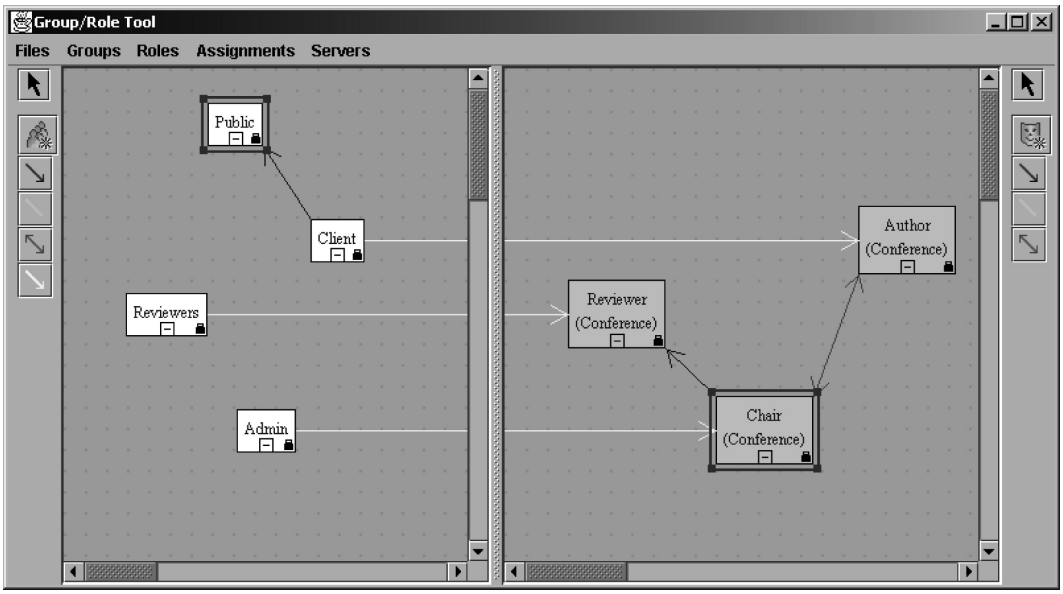

Figure 7.10: Role Management GUI.

tween these groups. The semantics of the subgroup relation is that the set of members of a subgroup is a subset of the supergroup's members and that the supergroup's role assignments are inherited by subgroups.

As an aggregational construct, groups complement roles. Roles are used to model actors and are thus a *task–oriented* abstraction, which is defined by an individual policy. Groups, in contrast, are not policy–specific but should be used to express *principal–oriented* characteristics, such as a user's position in an organizational structure. This distinction is similar to the one between *structural roles* and *functional roles* as expressed in [Dobson and McDermid, 1989]. An example group graph with role assignments is shown in figure 7.11.

The figure shows how a number of subjects in a publishing company are grouped. All subjects working in that department are members of the group `CSeriesDepartment` to which a *DocumentReader* role is assigned. Generally, the groups of subjects to which different tasks and the respective roles are assigned will be in similar positions within an organization. For example, the role *Employer* that allows hiring new employees will be assigned to only a few people in specific positions. Thus, groups are used to model organizational hierarchies. Another example is the role *Annotator* that is assigned to subjects in the `CopyEditors` group, which are people employed as copy editors in the department and thus modeled as a subgroup of `CSeriesDepartment`. Other subgroups are `Secretaries` and `SeriesEditors`. Authors are assigned to an `Authors` group that is not a subgroup of `CSeriesDepartment`.

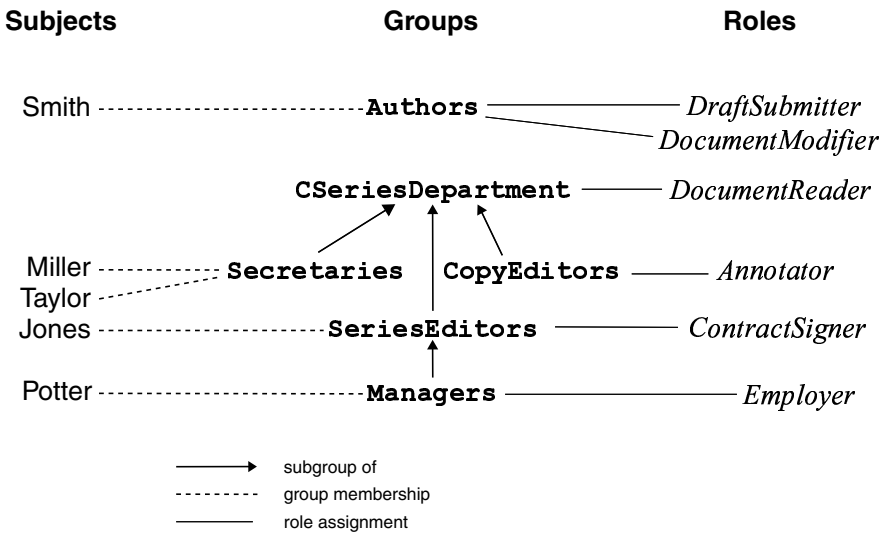

Figure 7.11: A group graph with role assignments.

7.4 Domain Management

As explained in section 7.2, the runtime infrastructure needs access to an object's security policy domains and managers need tools to manage these domains. This section presents a domain management service for the grouping of managed objects into domains and the association of policies with these domains. Although security policy domains are its prime application area, the domain management service is not limited to security policies but can handle arbitrary policy types. It supports domain hierarchies and meta–policies for the resolution of conflicts between policies in overlapping domains and refines the policy and domain models outlined by the CORBA Specification and the OMG Security Service Specification [OMG, 2001b]. Moreover, it defines a framework for the integration of new policy types and type–specific graphical policy editors.

7.4.1 A model of policy domains

This section refines the Security Service's domain model that was outlined in section 3.1.5 into a practically useful domain graph model and sketches how this model addresses the issues that were left undefined by the specification. As in [Sloman and Twidle, 1994] and [ISO/IEC, 1996c], the underlying model of a policy domain is a relation between a set of member objects and a set of policies. As noted above, there can be no direct conflicts between policies of the same type within a single domain, such as one access control policy allowing an access and another policy rejecting the same access. Policy conflicts are possible, however, between policies of different domains because objects can be members of more than one domain and because a domain's policies also apply in its subdomains in a hierarchy.

7.4.1.1 Domain Hierarchies

Hierarchical relationships between domains are an important means of expressing delegation of responsibility between authorities. Domain hierarchies can also model refinement between policies, so that general policies are assigned to domains higher up in a hierarchy, e.g., a whole organization. More specific policies are then assigned to subdomains, e.g., individual departments. It is also possible to define general rules in the policies of higher–level domains that are not refined but rather overwritten in subdomains, so that policy conflicts represent exceptions to these general rules.

Domain hierarchies are modeled as directed acyclic graphs where an edge between two domains means that a domain is a child of another domain. The semantics of this relationship is that the parent domain's policies also apply to the members of the child domain. Changes in parent domain policies may affect governed objects in subdomains whereas policy changes in the child domain only affect the members of that domain and its subdomains, but not the members in parent domains. A domain may have multiple child and parent domains, so the domain graph is not a tree. Allowing multiple parent nodes is useful because it allows to create a union of the parent nodes' policies for the child. Note that a subdomain's DomainManager object need not be a member of the parent domain. Domain membership and the subdomain relationship are kept separate so that different policies can be applied to domain members and subdomains domains.

Figure 7.12 illustrates a domain graph with six domains **A** to **F**. Domain **A** has a policy `p1` of type T1 that also applies to its subdomains **B** and **C**. Domain **C** has its own policy `p2` of type T1, so **C**'s member objects are governed by the policies `p1` and `p2`. **C**'s subdomains **D** and **E** also both apply their own T1 policies, `p3` and `p4`. The two domains **D** and **E** overlap in a single member object `o1`, which means that all policies of both domains and their parent domains, i.e. `p1` to `p4` apply to `o1`. The domain **F**, finally, is a subdomain of both **D** and **E**, so the policies of both **D** and **E** apply to **F**'s member objects. Since **F** does not associate any new policies with its members, these are the same policies that apply to `o1`.

Domain hierarchies also provide a means of naming domains. Just as in naming contexts or file systems, a domain **Y** can be identified relative to another domain **X** if a path from **X** to **Y** exists. The relative name of **Y** is formed by simply providing the names of the domains along the path. It must be ensured, however, that the names of all subdomains in a domain are unique. Note that a domain can have more than one name in a hierarchy because there are potentially multiple paths to it. In Figure 7.12, domain **F** can be identified relative to domain A using the names **A/C/E/F** or **A/C/D/F**.

7.4.1.2 Meta–Policies

Figure 7.12 shows that multiple policies of the same type can apply to an object, so policy conflicts may arise. Examples of conflicting security policies are auditing policies that define different events that should be audited and different data that should be recorded in the audit log. Another example is access control policies that both allow and deny an access. These

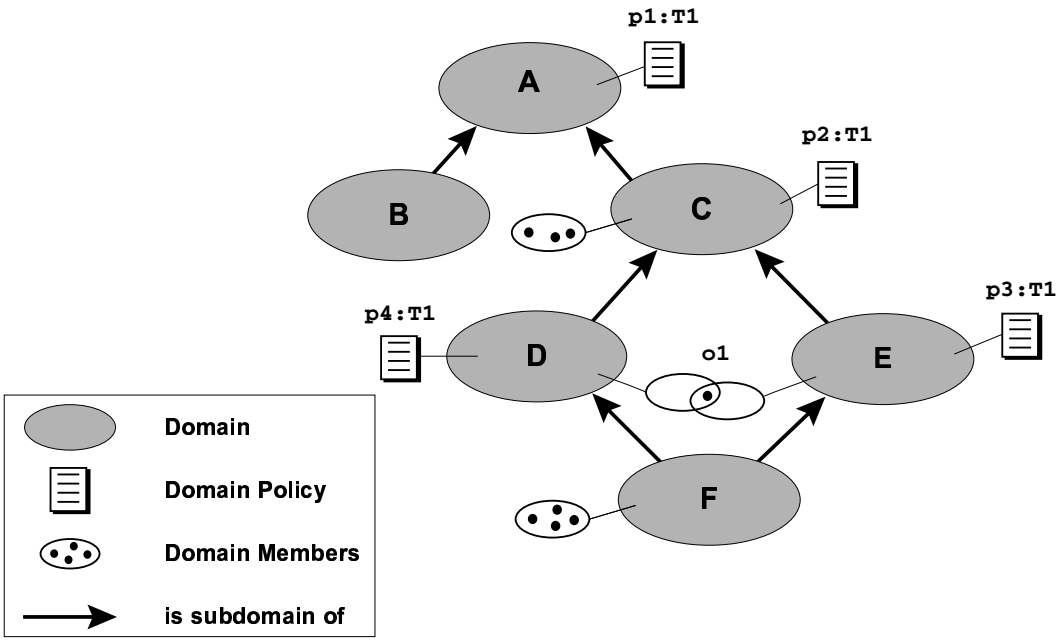

Figure 7.12: A domain graph.

conflicts between policies could be resolved by defining a general precedence rule for policies in domains, such as “policies closer to the root of the domain graph take precedence,” or “the policy closest to the object’s domain takes precedence.” However, defining a single precedence rule will only be appropriate for some policies and applications, but not for others. Strategies not easily expressible using precedence rules are a majority vote between applicable policies or a composition of policies rather than a selection.

To allow for a more flexible way of handling policy conflicts, the domain model supports the definition of *meta–policies* [Kühnhauser, 1999],[Lupu and Sloman, 1999], i.e., policies about policies. A meta–policy is basically another policy that must be interpreted by a mechanism enforcing domain policies. As with the domain policies, the actual meaning is determined by the interpreting mechanism. Meta–policies need not be restricted to conflict resolution as in [Kühnhauser, 1999] and [Lupu and Sloman, 1999], but could also be used to perform arbitrary computations, as long as the enforcement mechanism mandates this.

For this reason, meta–policies are not defined as functions that take a set of policies and return a resulting policy. While this would be a straightforward approach, both the semantics and the implementation of policies and their interpreting mechanisms can vary to an extent that would render such a definition of meta–policies too narrow to be useful. Also, modeling each computation as such a function would lead to inefficient implementations. For example, the different conflict resolution strategies listed above show that different meta–policies may require very different graph traversals to determine the applicable policies that must be consulted to resolve a conflict. A meta–policy can thus be seen as an additional, dynamically

adjustable parameter that determines how base–level policies of a given type are enforced by the mechanism. An example of using meta–policies is presented in section 7.4.3.

7.4.1.3 Initial Membership

As noted above, it is necessary to define a way of automatically mapping objects to initial domains at creation time. If this were not possible, an object might remain unprotected forever simply because its creation went unnoticed by an administrator. Object reference creation is performed by CORBA's Portable Object Adapter (POA). A POA can be configured with a set of POA policies, which are not domain–based but simply determine how the POA manages its objects. To be able to specify the initial domain for newly created objects, the most appropriate approach is to define a new type of POA policy that tells the POA in which domain it should initially place new objects. In our implementation, it is possible to associate a different *InitialMapPolicy* with each POA in a process, either by explicitly creating the POA with this policy in the application code or by externally specifying a system property to this effect. If no such policy has been set, the POA uses a default policy and places all objects in the *ORB domain*, which is an implicit domain for the local process. This implicitly created domain and its rationale are explained in section 7.4.2.

The *InitialMapPolicy* can also insert object groups of different granularities into domains. The default policy initially inserts all objects in the process into the ORB domain, which constitutes the first level of granularity. Other granularities are possible by creating POAs with different policies. First, it is possible to parameterize the default *InitialMapPolicy* with different target domain names, so all objects created by a POA can be inserted into the same set of domains. Second, it is possible to configure the POA with user–defined implementations of this policy. These implementations can choose arbitrary domains for objects. It is thus possible to sort all objects into domains based on their types, for example.

7.4.2 Implementation Aspects

The domain management service is a remotely accessible CORBA service, i.e., it provides a CORBA interface to its clients, which extends the standard CORBA interface DomainManager. The implementation of this and additional interfaces is mostly straightforward and not discussed here. The domain management service includes a graphical tool for centralized management called Domain Browser. A screen shot is shown in figure 7.13 in the next subsection. The Domain Browser includes an editor for *property policies*, with which simple policies can be written. Property policies are generic lists of name–value pairs and a flexible — if low–level — way to express policies. Enforcement mechanisms must know how to interpret these name–value pairs. We consider this only appropriate for very simple policies that do not require dedicated language support, however. The Domain Browser can be configured to provide different graphical policy editors for different policy types. For example, the graphical interface to the policy service presented in section 7.5 can be configured as a "policy editor" and thus be used from within the Domain Browser.

The most important aspect of the implementation is the requirement to minimize the performance overhead that is associated with domain–based policy management because this extra cost may render the entire approach infeasible. To enforce access control policies, every single object access must be checked. It is imperative to avoid remote accesses originating from the enforcement mechanism wherever possible. Two standard strategies to avoid remote accesses are employed. The first is to use caching of domain state and invalidation messages from domains to their subdomains whenever state changes affect subdomains, i.e., when policies are edited, added, or removed. The second approach to avoid remote calls is to provide and exploit locality of domain operations. As a first step, all domain management functions are also available locally to applications that do not need to be part of a global domain graph but still want to manage domain–based policies. In this case, no remote operations should be necessary at all. For this purpose, a purely local domain graph with its own root is created. Since this root domain's scope is actually equivalent to the process or ORB boundary, it is called *ORB domain*. Every process implicitly creates such a domain.

In the general case, local objects will need to be part of a larger domain graph, so the local ORB domain can be attached to the global domain graph. In this case, the ORB domain supports locality because it adds local paths between local graph nodes. Without these, even visiting local domains during graph traversal would require remote accesses. Finally, the ORB domain also becomes the demarcation line beyond which caching is required. To ensure that all accesses to all process–local domains are indeed local, the creation of paths to and from local domains that cross process boundaries and do not pass through the local ORB domain are prevented. In graph–theoretic terms, the ORB domain thus becomes an articulation point for local domains. Obviously, this poses a restriction on the free construction of domain graphs from process–local domains, but we argue that this is actually a helpful restriction as arbitrary interconnections between local domains are undesirable. Moreover, complex domain relations should be modeled in those parts of the domain graph that are managed by the domain management service and not in local processes, which are potentially shorter–lived than the domain service. Preventing these interconnections thus reduces coupling between processes.

7.4.3 An Access Control Example

As a simple application example, consider an international company *Hype Inc.* with different branches in the US and Europe. In its top–level policy, the company allows remote access to all its networked printers, which permits sending memos by directly printing them at the receiver's printer rather than sending them as a facsimile. The European branch of the company has different departments at different locations, amongst others a sales department and an R&D department. The R&D department has specialized printing equipment that should only be used for material to be worked on by R&D staff. Other accesses are explicitly disallowed to avoid the high printing costs on these devices.

Figure 7.13 shows a screen shot of the Domain Browser with a tree view of the appropriate domain graph and a printer object in the R&D domain. Also shown are the access control policy and a meta–policy in this domain. When an R&D printer object is accessed, the access

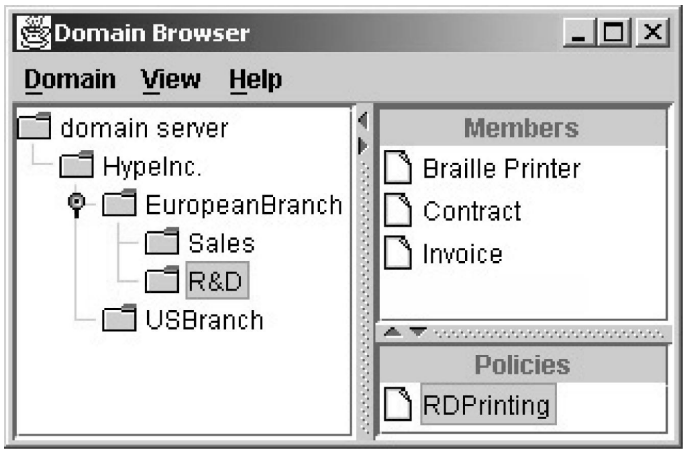

Figure 7.13: A domain graph in the Domain Browser.

control enforcement mechanism at the target side has to make an access decision and contacts its local AccessDecision object. The AccessDecision object retrieves the access control policies that apply to the target. When it consults these policies, it detects a conflict between a permission in the company–wide access policy and a denial in the R&D policy if the access is initiated from within the company but from a principal not authorized by the R&D department. Intra–policy conflicts are statically guaranteed to be resolvable in VPL according to the resolution strategy described in chapter 4. Conflicts between different access policies cannot generally be detected before policies are deployed because policy designers do not have advance knowledge of all policies deployed in the target environment. To resolve this conflict, the decision object now looks for a meta–policy that applies to access control policies. A simple conflict resolution policy that gives precedence to denials is defined as a property policy with two values in figure 7.14.

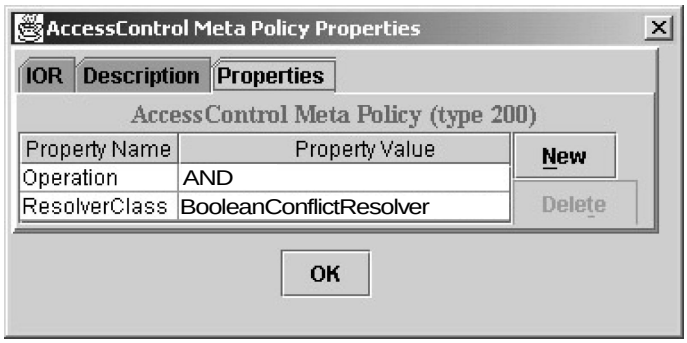

Figure 7.14: A meta–policy.

The first property tells the AccessDecision object which conflict resolution operation it should perform. In this case, this operation is a boolean AND of the access decisions of the

individual policies considered, which means that a single denial by any policy suffices to reject the access. The second property determines the actual conflict resolution strategy by giving the Java class name of a resolver, which is then dynamically loaded by the implementation. This implementation is not predefined but can be changed if desired, e.g., if new resolution strategies are implemented that cannot be exploited by the original implementation. Since the resolution operation is actually performed by the resolver, the "Operation" property is a parameter for the resolver and not for the AccessDecision object.

7.5 Policy Management

Three infrastructure components are provided to support view–based access policies and their management:

1. Policy servers, which host access policies as remotely accessible CORBA objects,
2. a view repository that stores view and schema definitions and initial view assignments, and
3. a graphical management tool with which managers can monitor and modify access policies.

7.5.1 Policy Server and View Repository

In CORBA, access policies are represented by objects of type SecurityAdmin::AccessPolicy. The standard access policy interface is restricted to two operations that set and retrieve the effective rights in the standard CORBA access model. These operations are unsuitable for the view–based model implemented here, so a different interface raccoon::policy::AccessPolicy was defined to support operations for setting and retrieving views.

The implementation of this interface implements the access matrix model presented in this thesis. The AccessPolicy interface does not extend but rather replaces the standard CORBA interface because a view policy cannot be regarded a specialization of a standard CORBA access policy. The policy server hosts AccessPolicy objects and thus makes them available to AccessDecision objects, which rely on them to retrieve authorizations. AccessPolicy objects depend on a number of other infrastructure objects, which are shown in figure 7.15. These dependencies are explained below.

When a new policy object is instantiated in a server process it first needs to determine to which objects it applies, i.e., it must be attached to a policy domain. The policy server accepts a domain name as a configuration parameter, which is used to look up the policy domain in the domain service. The policy object then queries the domain for its members and records these objects and their types in internal tables.

The policy object further needs access to the role repository to implement operations to return authorizations implied by the role hierarchies, i.e., to return the views assigned to a

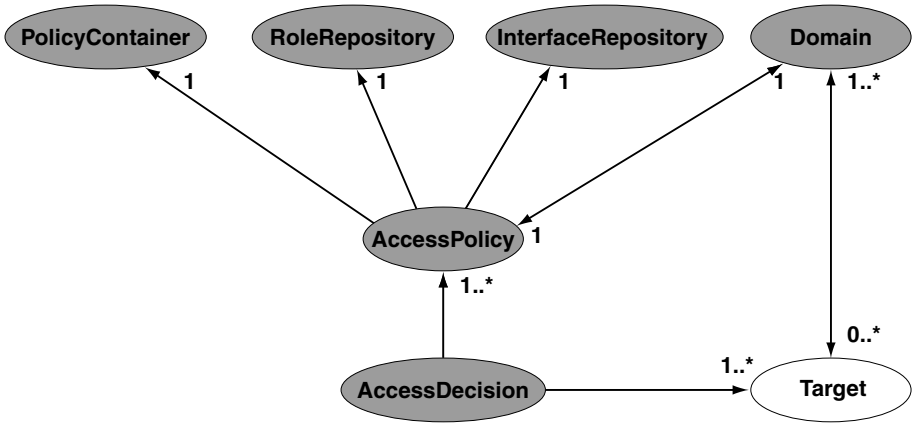

Figure 7.15: Object associations.

role's super roles together with those assigned to the role itself. For the same reason the policy object also depends on a CORBA Interface Repository, viz. to return authorizations implied by the type hierarchy.

The View Repository

Finally, a policy object needs to load view and schema definitions for the objects and types that it protects. Also, it needs to retrieve authorization information to perform initial view assignments. Both view and schema definitions and the specification of initial view assignments are stored in the view repository during policy deployment, so the policy object needs access to the view repository to retrieve this information.

The contents of the view repository is organized as a set of named containers where each container holds the static policy information for a single policy. The view repository interface supports operations to retrieve the container for a given policy, which in turn supports a number of storage and retrieval operations for VPL definitions. The main operations from the PolicyContainer interface are shown in figure 7.16.

A policy object loads the information from its container *incrementally* whenever a matrix entry becomes available to hold it, i.e., whenever objects become members of the domain. To find out about objects entering or leaving the policy domain, the policy object registers a callback interface as an event listener at the domain service, which then notifies the policy object of changes in the domain membership.

7.5.2 A Policy Management Tool

Policy objects can be accessed by managers through a graphical management interface, which is shown in figure 7.17. The screen shot in this figure shows one configuration of the policy for

```
interface PolicyContainer
{
    readonly attribute string containerName;
    void defineView( in ViewDescription v ) raises (AlreadyDefined, IllegalDefinition);
    ViewDescription lookupView( in string name ) raises (UnknownView);
    void defineSchema( in SchemaDescription v )
        raises( AlreadyDefined, IllegalDefinition );

    SchemaDescription lookupSchema( in string name ) raises (UnknownSchema);

    void defineInitialViews( in RoleId roleName, in ViewTypePairSeq initial )
        raises( UnknownView, UnknownType);
    // ...
};
```

Figure 7.16: The PolicyContainer interface.

the conference system in chapter 6. The upper left panel contains a tree view of the objects, types, and attached schemas in the domain. This tree view is structured like the IDL type hierarchy, with types as its inner nodes and objects at the leafs. Schemas are also represented as leafs attached to type nodes. The upper right panel shows the principals and views for an object or type that is selected in the upper left panel.

The panel in the bottom half of the window displays textual information about IDL interfaces or VPL view or schema definitions. Which description is displayed depends on the selection in either of the upper panels. In the figure, an interface is selected, so the interface definition is retrieved from the Interface repository and displayed. Clicking on a view name in the upper right panel would retrieve and display a view definition in VPL. Similarly, clicking on a schema in the tree view would display the schema definition in the bottom panel.

In its current implementation, the policy GUI does not support dynamically editing the policy by modifying view or schema descriptions or by adding new views.

7.6 Architecture Evaluation

The main goal of the development of the security infrastructure was to demonstrate the practical feasibility of a view–based approach to access control. This section evaluates the architecture with respect to this goal, in particular the integration into the standard CORBA Security Framework, performance implications, and the resulting architectural complexity.

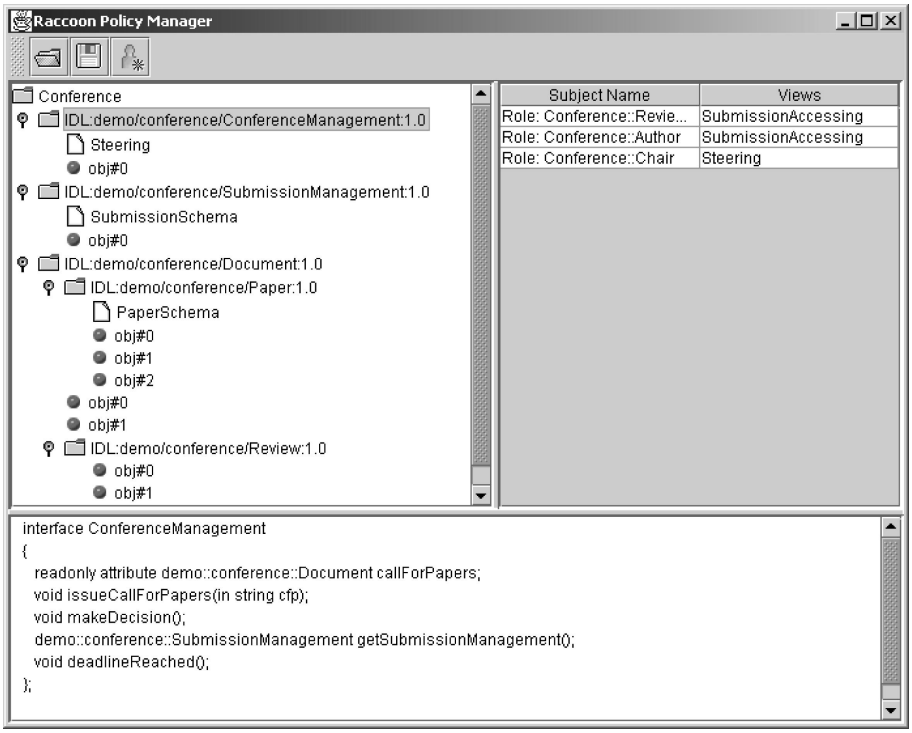

Figure 7.17: The Policy Management GUI.

7.6.1 CORBA Integration and Portability

In general, the access control infrastructure presented in this chapter is fully integrated with the standard CORBA Security Framework. With the exception of the non–standard AccessPolicy interface for view–related policy operations, all components presented here actually comply to standardized CORBA interfaces, even if they extend them to provide additional functionality. Because the integration with the ORB was achieved using the Portable Interceptors API, porting implementations to other ORBs than JacORB is straightforward, provided that SSL is supported as a transport security mechanism.

7.6.2 Performance Implications

Any access control mechanism introduces performance penalties for applications when performance is compared with unchecked accesses. Therefore, it is important to be able to estimate whether the additional overhead connected with flexible and fine–grained access control is acceptable for an application or not. Since the implementations are unoptimized prototypes, assessing performance in absolute terms is not meaningful. Rather, the goal is to show that the use of a flexible and manageable access control infrastructure does not necessarily incur an unreasonable performance penalty over using access control based on the standard CORBA

access model. Therefore, the following paragraphs compare the overhead with the one that would be introduced by the standard CORBA access control model.

Since no implementation of standard CORBA access control was available for testing, a dummy AccessDecision implementation was set up for the comparison. In this test, the access_allowed() operation that is called from an access control interceptor does not perform any authorization evaluations but simply returns TRUE so that only successful attempts are compared and the overhead for raising exceptions is avoided. This dummy implementation thus delivers a constant response time that is necessarily better than with any realistic, even very simple access decision algorithm. The comparison implementation also relies on the same SSL setup for authentication as the view–based model, but without the additional role layer. SSL was configured to send unencrypted data for both access models. The domain service was also used to retrieve the applicable access policy for both cases, but this does not have consequences for the measured access times.

All measurements were taken on a single machine that runs client, server, and all required infrastructure components. In the view–based case, the overall machine load was higher because more server processes are required, e.g., the standard access model needs no role server and no repositories running. Since the view–based model requires more access control information from diverse sources, the initial context setup is necessarily more expensive than in the standard case. Therefore, the cost of an initial connection including authentication and context setup is not considered here.

The tested access operation has no arguments and returns a four–byte integer. Averaged over 5000 accesses, the cost of single checked access using a standard CORBA access setup and measured at the client varies between 1.1 and 1.6 msecs. These figures should be taken with a grain of salt and used only for comparisons because measurements were not isolated from factors such as activities of background processes and asynchronous Java garbage collection. This observed access time is about twice as long as the round–trip time of the same access without any protection, which takes around 0.5–0.6 msecs. on the target platform.

Using the access mechanism presented in this chapter, an additional overhead of around 50% on average is observed, which is mostly due to the effort of managing access sessions and transmitting GIOP service contexts, which are not needed in the standard case. The actual VPL access decision algorithm is also more complex, but this difference is marginal compared to the other factors just mentioned. As a positive consequence of this approach, the implementation of the access mechanisms does not cause any remote invocations to retrieve authorization information after the initial session setup. In a more realistic implementation of the standard access model, a similar approach to sessions would be expected for the same reasons. That no remote invocations are needed after session establishment holds only for accesses that do not cause schema activity, however. If a schema effects authorization changes and then invalidates and reestablishes the current session context, the overall round–trip takes between 35 and 60 msecs.

In summary, the added flexibility of role and view–based access control does incur a certain performance overhead. Considering that the tested implementation is largely unoptimized and

that the compared test implementation of the standard CORBA access model is overly simplistic, this overhead is comparatively small. Schemas have no counterpart in the standard access model, so accesses involving schemas cannot be fairly compared at all. Thus, the implementations demonstrate that the access model presented in this thesis is practically implementable and can deliver a performance that should be acceptable wherever the standard CORBA access model is acceptable. This conclusion must be qualified for cases where objects are only accessed once, so that all accesses are burdened with the full cost of session establishment. If an application exhibits such an access pattern and at the same time also requires high performance and very fine–grained accesses, then the implementation of the view–based access model is unlikely to provide satisfactory performance.

7.6.3 Architectural Complexity and Protection of the Infrastructure

An AccessDecision object requires a number of different infrastructure objects in its environment to make view–based access decisions. As shown above in figure 7.15, these objects are:

1. Domain objects,
2. policy objects,
3. a view repository object,
4. a role repository object, and
5. an Interface Repository object.

Because these objects are functionally independent, they can be hosted by separate servers. Additionally, one or more role server objects are required to provide clients with role membership certificates. Each of these servers requires careful configuration and management so it is desirable to reduce the number of servers in the system and thus the management complexity of the infrastructure itself.

Since it is expected that the repositories will rely on similar persistent storage techniques, e.g., LDAP, the integration of the three repositories into a single server is a useful first step. Collocating and integrating servers is not only useful to reduce complexity, it also helps to avoid remote communication between components that interact frequently, such as the role server and the role repository. Thus, a second step toward reducing the number of servers is to integrate the role server with the repository server.

7.6.3.1 Protecting the Infrastructure

The components of the access control infrastructure itself are targets for attacks because successful attacks on the infrastructure can lead to either a complete denial of service or to unauthorized access to protected objects. On the one hand, the infrastructure could be tricked into

making only negative access decisions and rejecting all accesses, which constitutes a successful denial of service attack. On the other hand, the access decision function might also make incorrect positive decisions if it sees incorrect or incomplete authorization information.

Denial of service attacks are simpler because it suffices to prevent the access decision function from accessing *all* authorization descriptions, e.g., by blocking or bringing down the policy server. In such a situation, all accesses are rejected because no permissions are found. To restrict the damage caused by such attacks and ensure higher availability of authorization information, policy servers should be replicated, which obviously conflicts with the goal of reducing the number of components in the system.

Another critical component is the domain server, which manages the mapping from objects to policies. If an object is a member of more than one domain and these domains are hosted by different servers, a successful strategy is to attack only specific policy domains. If an access is allowed by a given policy but denied through the combination with another, blocking the denying policy can result in the attacker gaining unauthorized access. Other strategies for gaining unauthorized accesses can target the type, view, and role repository. These attacks would manipulate implicit authorizations by modifying the type, view, or role hierarchies or simply the set of rights in a view. Thus, all components in the access control system must be protected against modification. The prototype implementations presented in this chapter do not provide adequate protection for the protection system itself and would need to be extended for use in potentially hostile environments.