RACCOON — AN INFRASTRUCTURE FOR MANAGING ACCESS CONTROL IN CORBA

Gerald Brose
Institut für Informatik, Freie Universität Berlin, Germany
gerald.brose@acm.org

Abstract Object-level security management of CORBA applications is not sufficiently supported by current management architectures. This paper presents a language-based approach and an infrastructure for managing fine-grained access control policies for CORBA objects at an abstract level.

Keywords: CORBA, Access Control, Security Policies, Management

1. INTRODUCTION

The complexity inherent in distributed systems constitutes a major problem for overall security because of the significant potential for human error. This applies especially in the area of security management. A basic premise of this work is that the correct design, specification, and management of security policies is a central problem in distributed systems because these tasks are both error-prone and by their very nature security–critical. We argue that the potential damage caused by inadequate handling of policies is significant, and that current methods or management tools do not provide adequate support to security administrators.

This paper focuses on one particular aspect of security policy management in CORBA environments, viz. access control. Access control is concerned with preventing unauthorized accesses to shared resources and is used to enforce a given set of security requirements. "Commercial" security policies [Clark and Wilson, 1987] usually concentrate on integrity requirements, but access control can also be used to enforce confidentiality, general resource usage restrictions (availability), or even coordination policies in CSCW systems [Edwards, 1996]. These requirements are addressed in access control policies, which describe which accesses in a system are authorized and which are not. In our use of the term, policies are represented in terms of a formal access model and can be evaluated by an access control mechanism.

To support the management of access policies for CORBA objects, it is necessary to define appropriate management abstractions. These abstractions serve two purposes. First, they allow managers to deal with large–scale systems by abstracting from individual entities such as objects, users, and access rights, i.e., they hide details that are too numerous to handle individually. Examples of abstractions of this kind include domains of objects, user groups, and views. Second, they can be used to hide implementation details that are too involved to be intelligible to security managers, such as an individual operation invocation in a complex object–oriented application protocol. Roles and views are examples of abstractions of this second kind.

An important aspect of access control in object–oriented systems is that it is not sufficient to consider run–time management in isolation. Because of the potentially large number of objects and complex interaction patterns in object–oriented applications, a security administrator cannot be expected to completely understand the internal logic of all applications running under his supervision. The security implications of, e.g., introducing a new right or object type into a running system might be unclear, so managers need security documentation at an adequate level of abstraction. This information can only be provided by application designers, however, so some form of cooperation and communication between developers, deployers and managers is required.

We propose a separate design document that must be delivered with an application in the form of a descriptor file, which can be read by deployment and management tools. It is the responsibility of application developers to prepare this document as an input to the subsequent deployment and management stages in the application life cycle. To do so, policy designers use a specific design language, the *View Policy Language* (VPL).

The remainder of this paper is structured as follows. Section 2 presents a language–based approach to this problem. In section 3, we describe the security architecture required for managing access control in CORBA based on VPL. Section 4 discusses related work. The paper concludes with a summary and an outlook on future work in section 5.

2. VIEW-BASED ACCESS CONTROL POLICIES

The central concept of the access control model proposed in [Brose, 1999, Brose, 2000] and presented in this section is the higher–level modelling concept of a *view* for authorizations. Using views, policies can be written and understood in terms of sets of related authorizations. A view is a named set of access rights, which are both permissions or denials for operations on objects. While access decisions are based on individual rights, views are the units of description and authorization assignment. They are defined statically as part of a policy specification. Figure 1 shows an example of a view definition in VPL.

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

Figure 1. A view definition.

Views are defined as authorizations for objects of a single IDL interface, which is referenced in the `controls`–clause of the view definition. In the example, the view `Reading` controls the IDL type Document, which means that this view can only be assigned on objects of this type or one of its subtypes. Permissions are listed after the keyword `allow`, denials would be introduced by deny. In the example, only operations to read the document and to find words within the document are allowed. A view can be restricted to the roles listed in a `restricted_to` clause so that it is not possible to assign the view to other roles. The `Reading` view can only be assigned to the role `Staff`.

The introduction of this concept is motivated by the observation that generic rights such as "read", "write", and "execute" are not adequate for describing authorizations in large–scale systems comprised of typed objects. Generic or uninterpreted rights are ill–suited for application–level access policies in these systems because they provide no means to impose external structure by defining relationships or constraints. Moreover, it is difficult to capture the rich semantics of operations in object–oriented systems with just a small set of generic rights. Generic rights thus lead to modelling problems in the design of access policies and are hard to manage.

To solve the problems outlined above, we propose a new access model that is based on the classical access matrix model by Lampson [Lampson, 1974]. Unlike Lampson's matrix or the standard CORBA access model [OMG, 1998], matrix entries do not contain simple generic rights but named and typed sets of rights, i.e., views. Matrix rows in the model correspond to roles, so a matrix entry represents the authorizations assigned to a role for the particular object denoted by the matrix column. Table 1 illustrates such a matrix.

The notion of roles used here is that of the $RBAC_3$ model [Sandhu et al., 1996]. In $RBAC_3$, roles can be organized into role hierarchies, and role constraints such as mutual exclusion can be expressed. Figure 2 shows role definitions in VPL.

Table 1. Access matrix with view entries.

Role \ Object	f: Folder	chapter: Document	contract: Document
Secretary	Lookup	Reading	Reading
Author	Lookup Appending	Reading Updating	Reading
Editor	Listing Appending Removing	Reading	Reading
Manager	Lookup	Reading	Updating

```
roles
   Staff;
   Author;
   Secretary:  Staff; // Secretary is senior to Staff
   Editor:  Staff {
      maxcard  4 // no more than 4 editors
   };
   Manager:  Staff {
      excludes Author // managers may not be authors
   };
```

Figure 2. Role definitions.

VPL can be used to express implicit authorizations and exceptions, and allows designers to specify priorities that determine how conflicts between permissions and denials are resolved. Moreover, it is possible to statically specify dynamic rights changes using the schema language construct. Figure 3 illustrates how views are assigned and removed in response to the IDL operation create, which acts as a trigger. The schema in the example is defined to react to operations on objects whose type has the same name as the schema, i.e., on DocumentFactory objects. The modifier `with assign option` in the assigns clause is equivalent to SQL's grant option and means that the receiving principal may delegate the view Managing to other principals at his discretion. The identifiers result, caller, and this are dynamically bound to the result of the create operation, the DocumentFactory object itself, and the caller, respectively. A second effect of the create operation is that the `Creating` view is removed from the caller's matrix entry for the DocumentFactory object, assuming that this view was required to call create in the first place.

By fixing the target data model to IDL, it is possible to define a number of static type checks that help to catch policy specification errors early. For example, it can be checked that views are well–formed with respect to the IDL

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

Figure 3. A VPL schema for document creation.

type they control. Additionally, it can be verified that schema clauses comply with role restrictions defined in views, and that conflicts between permissions and denials are resolvable at runtime.

3. AN INFRASTRUCTURE FOR ACCESS CONTROL MANAGEMENT

A number of infrastructure components and tools are required to allow application developers and managers to work with the model abstractions presented in the previous section. 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 compile them into a descriptor file format. Similar to EJB descriptors [Sun Microsystems, 2000], the VPL compiler produces an XML file. The contents of this file is deployed to a role and view repository, respectively, which make role and view constructs accessible at runtime. XML was chosen as an intermediate format so that deployment tools can rely on standard XML parsing libraries and a document type description (DTD) rather than having to implement a full VPL compiler. Deploying new roles potentially involves role renaming, either to prevent name clashes with existing roles, or in order to identify new roles with already existing roles that have the same function but different names. Figure 4 depicts the development and deployment process.

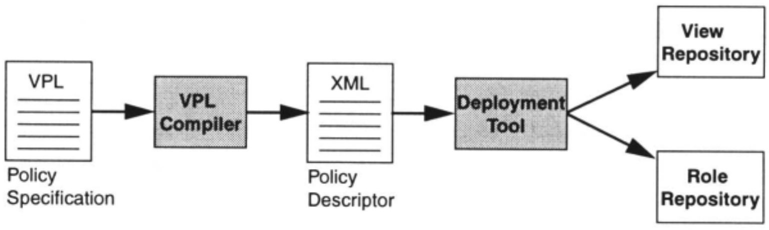


Figure 4. Policy compilation and deployment.

3.1. Operation and Management

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. Our own implementation also uses interceptors.

The UML diagram in Figure 5 illustrates how accesses are transparently intercepted and checked in CORBA. The interceptor calls an AccessDecision object which requires an access policy to determine whether the interceptor can allow the access and let the request pass, or whether it must deny the access by signaling the CORBA system exception NO_PERMISSION to the caller. This exception is not shown in the diagram.

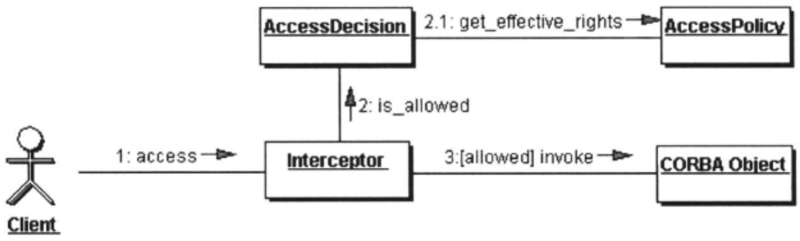


Figure 5. Access Control with Interceptors.

While interceptors are the standard implementation of reference monitor functionality, they are not the only option. It is just as possible to define reference monitors on a per–node or even per–subnet basis rather than only per-process. Moving the reference monitor further away from the protected object has implications for the extent of the object's trusted computing base (TCB), however. In the case of a per–process interceptor implementation, the TCB comprises the target platform's hardware, operating system, middleware, and security service implementation. Delegating this functionality to a component outside the target node, e.g., to an application–level firewall, means placing trust not only in the firewall, but also in any machine within the interior network because operation invocations from these machines do not pass through the firewall and thus remain unchecked.

An interceptor needs access control information to be able to make access decisions, viz. information about the caller and its roles, about the target, and the requested access operation. In addition, it needs access to the access policy for the target object. Figure 6 gives an overview of the components that are involved in an access decision and need to be adapted to the specific access control model used here. The interceptor and the AccessDecision object reside within the server hosting the target object and have been omitted from the diagram. In the remainder of this section, we discuss these infrastructure components, their relationships, and their management interfaces in more detail.

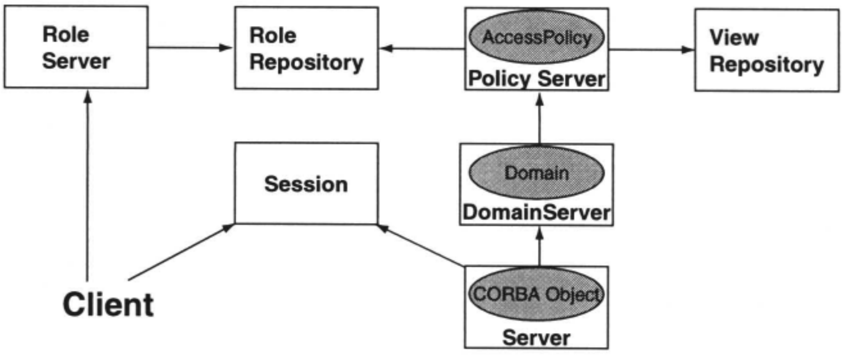


Figure 6. Raccoon architecture.

3.2. Sessions and Role Server

In the CORBA security framework, information about callers is represented as security attributes and stored in *Credentials* objects. Credentials objects are created on the client side as the result of authentication and stored in the client-side context. Before an operation is invoked, the client runtime system needs to establish a security association or *Session* with the server and transmit its credentials. In our approach, the credentials associated with sessions contain information about the caller's roles. This section describes the necessary role management components, which are not defined by the CORBA Security Service.

Role information is provided in the form of X.509v3 certificates that contain the principal's public key and a single role name in an extension field of the certificate. Upon request, these certificates are created, signed, and returned by a role server component. A *role certificate* is a binding between a principal identifier (a public key) and a role name, and this binding can be verified by checking the role server's signature.

The role server does not need to establish the identity of clients requesting role certificates because a role certificate itself does not convey any authorizations to its holders, and because callers must authenticate separately with the individual servers anyway. However, the role server may still choose to authenticate clients and restrict access to role information if this information is considered sensitive. Because role information about a principal is used to derive authorization information, access control interceptors must verify the authenticity and freshness of the information using the role server's public key. They must also verify that the caller is indeed the subject of the role certificates it presents. Caller authentication is achieved by relying on SSL transport connections.

The process of requesting certificates and storing them locally as credentials is performed by the security service on the client side, which is also responsible for sending these certificates to the server. The basic model of interaction with the role server is thus "client–pull" and allows security–aware clients to 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. The client can perform this role selection using standard CORBA security API calls for credentials management.

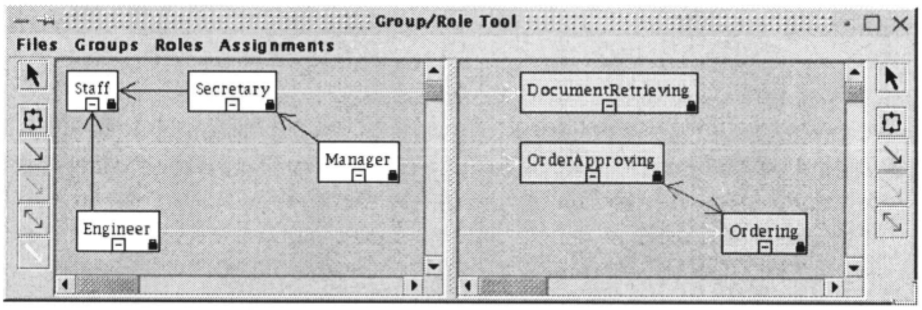


Figure 7. Role Management GUI.

The role server provides administrators with the GUI shown in Figure 7, which supports managing groups, roles, and role constraints. The left half of the window displays information about principals in the form of a group hierarchy. In the right panel, the example shows three roles and a mutual exclusion constraint between the two roles `OrderApproving` and `Ordering`, which is shown as a double–headed arrow. This role information itself is not kept in the role server but stored separately in the role repository. The assignment of groups of principals to roles is performed by drawing an arrow from a group in the left panel to a role in the right panel.

3.3. Domain Server

In large distributed systems, objects are typically not managed individually but grouped into management domains. For this reason, the individual CORBA object in Figure 6 is not directly associated with a policy. This allows managers to structure large systems into manageable subsystems and to model the diverse spheres of management authority and responsibility that are frequently found in large organizations. This section presents a model for and an implementation of a service for CORBA that supports grouping objects into domains, the construction of domain hierarchies, and the assignment of policies to domains. The service itself is policy–neutral. It can also be used as a generic mechanism to attach arbitrary dynamic properties to CORBA objects and is more flexible than the OMG–specified Property Service [OMG, 1997] because it does not require managed objects to implement service–specific interfaces. More information on the domain service can be found in [Brose et al., 2001].

The CORBA specification [OMG, 1999a] defines interfaces for policies and domains but no service API for managing the life cycle of domain objects, the construction of domain hierarchies, or the membership of objects in domains. An object may be a member of multiple domains, but CORBA requires that every object must be a member of at least one domain. Domains may only be associated with one policy of a given type. Individual domains are thus free of conflicts between policies of the same type, such as one access control policy allowing an access and another policy rejecting the same access.

As in [Sloman and Twidle, 1994] and [ISO/IEC, 1996], a policy domain is basically a relation between a set of member objects and a set of policies. Hierarchical relationships between domains are an important means of expressing delegation of responsibility between authorities and can also model refinement between policies. We model domain hierarchies as acyclic directed graphs, with edges representing subdomain relationships. A domain's policies also apply to the members of its subdomain, and changes in parent domain policies may affect objects in subdomains. Policy changes in subdomains, however, only affect the members of that domain and its subdomains.

Assuming that policies of different types are entirely orthogonal, there can be no direct policy conflicts within a single domain. Policy conflicts are possible between policies of different domains, however, because objects can be members of more than one domain, and because a domain's policies also apply in its subdomains in a hierarchy. The domain management service therefore supports the allocation of conflict resolution strategies in the form of *meta policies* [Hosmer, 1993]. A meta policy may, e.g., define a general precedence rule for policies in domains, such as "policies closer to the root of the domain graph take precedence".

A meta policy is basically another policy that is associated with a domain and must be interpreted by an enforcement mechanism. We do not attach any more semantics to a meta policy than that it applies to policies of a given type that in turn apply to the objects in the domain. As with other policies, the actual meaning is determined by the mechanism. Meta policies need not be restricted to conflict resolution as in [Kühnhauser, 1999] and [Lupu and Sloman, 1999], but can be regarded as general policy operators that are implemented by the individual policy enforcement mechanisms.

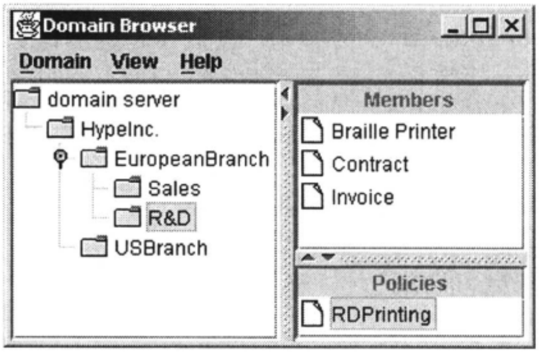


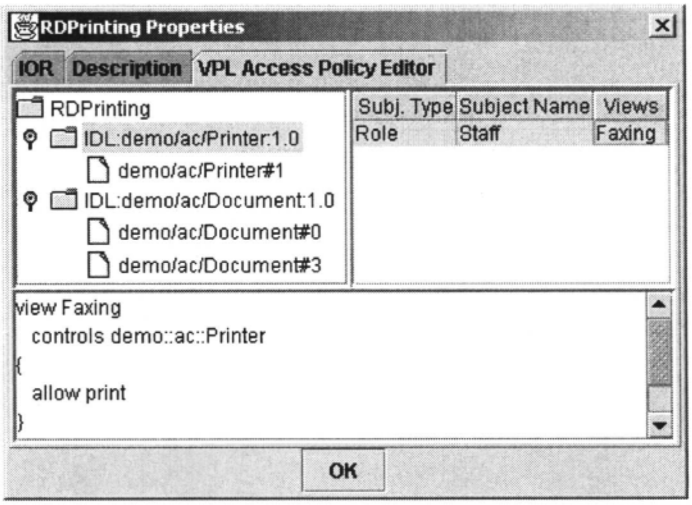
Figure 8. The Domain Browser.

The implementation provides a GUI management tool with which domain services can be used centrally by an administrator. This Domain Browser is shown in Figure 8. It includes an editor for generic policies that can be written as simple lists of name–value pairs and can be conveniently configured to provide different graphical policy editors for different policy types. VPL, e.g., requires a different editor, which is shown in Figure 9.

3.4. Policy Server

The editor tool shown in Figure 9 combines information from different sources. The upper left panel shows information about the objects in the policy domain, which is part of the domain server. The upper right panel shows the access policy itself, i.e., assignments of views to roles on the objects selected in the left panel. In this case, the `Staff` role holds a `Faxing` view on the printer object in the domain. The complete access policy is represented by a separate CORBA object of type AccessPolicy, which implements the view–based access matrix model.

The policy editor supports adding new views on the objects and types in the domain. In the lower half of the window, the policy editor displays the definition



Subj. Type	Subject Name	Views
Role	Staff	Faxing

```
view Faxing
  controls demo::ac::Printer
{
  allow print
}
```

Figure 9. The VPL Policy Editor.

of the view `Faxing` that it retrieved from the View Repository in response to the selection of this view in the upper right panel. Finally, the editor can also display IDL type information in the lower panel if an IDL type name is selected. For this purpose, it requires access to a CORBA Interface Repository.

4. RELATED WORK

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

Our basic approach of relying on a separate, abstract language for the specification of non–functional aspects of applications can be compared to *Aspect–Oriented Programming* (AOP) [Kiczales et al., 1997]. AOP also relies on separate *aspect languages* that are processed by a tool called *aspect weaver*. This tool generates code for aspects such as concurrency and distribution, and performs the integration of functional code with aspect code. There is currently no aspect language in AOP that addresses access control. The use of descriptor files that are processed by deployment tools is common in environments such as EJB [Sun Microsystems, 2000] or the CORBA Component Model (CCM) [OMG, 1999b], both of which also support the expression of simple access policies in descriptors. Both descriptor languages do not provide adequate management abstractions, however, and only support access control decisions at the granularity of types, not individual objects.

In the context of policy–based management, a number of general–purpose policy languages have been proposed, e.g. [Sloman and Twidle, 1994, Koch et al., 1996, Tu et al., 1997]. The information models underlying these languages are more general than the CORBA object model, and they do not offer higher–level authorization concepts like views. These languages are thus not suitable for collaborative use by CORBA application developers, who have to provide an initial policy design, and security managers, who subsequently deploy, refine and manage the policy. We argue that for such an approach, it is necessary to sacrifice some generality for better integration with design abstractions, i.e., IDL interfaces.

A general framework for defining arbitrary access control policies is proposed in [Jajodia et al., 1997], where policies are formulated as a set of rules in a logic–based language. This model leaves open all design decisions about how implicit authorizations are derived, how rights propagate in groups, which conflict resolution strategies are used, and how priorities are employed. Rules embodying these design decisions have to be defined first as part of a policy library. The data model for protected objects is also left open and has to be described separately.

5. SUMMARY AND FUTURE WORK

This paper presented a language–based approach to the problem of managing application–level access control policies in CORBA. We discussed a view and role–based access model and presented the runtime infrastructure required to manage policies according to our access model. A prototypical Java implementation of the Raccoon architecture was done using the CORBA implementation [JacORB, 2001]. The prototype implements the subset of the CORBA Security Service that is relevant for access control and replaces its standard access model. Moreover, it extends the security service with role and domain management components.

To demonstrate the feasibility of the approach, we plan to evaluate the performance implications of the proposed architecture, and to conduct a more complex application case study. The prototypical implementation has not been specifically designed to withstand denial of service attacks, which would require appropriate replication of services. Another direction for future work is to use VPL for managing application–level firewalls for CORBA.

Acknowledgments

This work is funded by the German Research Council (DFG). The author would like to thank Peter Löhr and Nicolas Noffke for fruitful discussions.

References

[Adiron, 2001] Adiron (2001). http://www.adiron.com/.

[Brose, 1999] Brose, G. (1999). A view–based access model for CORBA. In Vitek, J. and Jensen, C., editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, LNCS 1603, pages 237–252. Springer.

[Brose, 2000] Brose, G. (2000). A typed access control model for CORBA. In Cuppens, F., Deswarte, Y, Gollmann, D., and Weidner, M., editors, *Proc. ESORICS 2000*, LNCS 1895, pages 88–105. Springer.

[Brose et al., 2001] Brose, G., Kiefer, H., and Noffke, N. (2001). A CORBA domain management service. In *Proc. KiVS 2001.*

[Clark and Wilson, 1987] Clark, D. D. and Wilson, D. R. (1987). A comparison of commercial and military computer security policies. In *Procs. IEEE Symposium on Security and Privacy,* pages 184–194.

[Department of Defense, 1985] Department of Defense (1985). Department of Defense Trusted Computer System Evaluation Criteria. DoD 5200.28-STD.

[Edwards, 1996] Edwards, W. K. (1996). Policies and roles in collaborative applications. In *Proc. Computer Supported Cooperative Work (CSCW),* pages 11–20.

[Hosmer, 1993] Hosmer, H. H. (1993). The multipolicy paradigm for trusted systems. In *Procs. ACM New Security Paradigms Workshop*, pages 19–32.

[ISO/IEC, 1996] ISO/IEC (1996). *Information Technology — Open Systems Interconnection — Security Frameworksfor Open Systems: Overview.* International Standard, ISO/IEC 10181–1:1996(E).

[JacORB,2001] JacORB (2001). http://www.jacorb.org.

[Jajodia et al., 1997] Jajodia, S., Samarati, P., Subrahmanian, V. S., and Bertino, E. (1997). A unified framework for enforcing multiple access control policies. In *Proc. International Conference on Management of Data,* pages 474–485.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingiter, J.-M., and Irwin, J. (1997). Aspect–oriented programming. In *Proc. ECOOP*, LNCS. Springer.

[Koch et al., 1996] Koch, T, Krell, C., and Krämer, B. (1996). Policy definition language for automated management of distributed systems. 2nd International Workshop on Systems Management, IEEE Computer Society.

[Kühnhauser, 1999] Kühnhauser, W. (1999). *Metapolitiken.* GMD Research Series. GMD.

[Lampson, 1974] Lampson, B. W. (1974). Protection. *ACM Operating Systems Reviews*, 8(1): 18–24.

[Lupu and Sloman, 1999] Lupu, E. C. and Sloman, M. (1999). Conflicts in policy–based distributed systems management. *IEEE Transactions on Software Engineering,* 25(6):852–896.

[OMG, 1997] OMG (1997). *CORBAservices: Common Object Services Specification.*

[OMG, 1998] OMG (1998). *Security Service, Revision 1.5.*

[OMG, 1999a] OMG (1999a). *The Common Object Request Broker: Architecture and Specification, Revision 2.3.*

[OMG, 1999b] OMG (1999b). *CORBA 3.0 New Components Chapters.* OMG, TC Document ptc/99-10-04 edition.

[Sandhu et al., 1996] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. (1996). Role-based access control models. *IEEE Computer,* 29(2): 38–47.

[Sloman and Twidle, 1994] Sloman, M. and Twidle, K. (1994). Domains: A framework for structuring management policy. In Sloman, M., editor, *Network and Distributed Systems Management*, chapter 16. Addison–Wesley.

[Sun Microsystems, 2000] Sun Microsystems (2000). *Enterprise JavaBeans Specification, Version 2.0, Final Draft.*

[Tivoli, 2001] Tivoli(2001). http://www.tivoli.com.

[Tu et al., 1997] Tu, M., Griffel, F., Merz, M., and Lamersdorf, W. (1997). Generic policy management for open service markets. In *Proc. International Conference on Distributed Applications and Interoperable Systems (DAIS'97),* pages 212–222, Cottbus, Germany. Chapman & Hall.