Decoupling Authorization Logic From Application Logic in CORBA Security-Aware Applications

Konstantin Beznosov* BHSSF
Yi Deng† FIU
Bob Blakley‡ IBM
Carol Burt§ 2AB
John Barkley¶ NIST

November 6, 1998

Abstract

We present an approach to decouple security policy decisions logic from application logic for those systems that exercise application level security policies. In particular, we propose a design solution that allows to decouple authorization logic from CORBA-based security-aware applications. Decoupling authorization logic from application logic allows applications with fine-grain access control requirements to be independent from a particular access control policy and from factors that are used in authorization decisions as well as access control models, no matter how dynamic those polices and factors are. It also enables elaborate and consistent access control policies across heterogeneous systems. We show that such decoupling can be done without complicated interfaces and without sending much information between an application and the authorization service, and that factors specific to the application domain can be supported by authorization systems using the traditional access matrix as an underlying implementation. Although the proposed design utilizes CORBA environment, the approach is useful for any distributed computing technology.

1 Introduction

Traditional access control mechanisms [1] provide limited capabilities for authorization decisions to be based on factors that are specific to the application domain. The complexity of access control policies in such application domains as healthcare requires exercising access control policies that are more sophisticated and of finer granularity than the general ones used in security services of such distributed environments as CORBA.[1] This complexity leads application designers to embed domain-specific authorization logic inside their applications. Some even document patterns of designing "application security" [2].

CORBA environment, including the CORBA Security Service, provides a general-purpose infrastructure for developing distributed object systems in a broad range of specialized vertical domains. The CORBA Security service defines the interfaces to a collection of objects that provide a versatile

*6855 Red Road, Coral Gables, FL 33143, beznosov@baptisthealth.net
†University Park, Miami, FL 33199, deng@cs.fiu.edu
‡11400 Burnet Road, Mail Stop 9134, Austin, TX 78756, blakley@us.ibm.com
§3178-C Highway 31 South, Pelham, AL 35124, cburt@2ab.com
¶Gaithersburg, MD 20899-0001, jbarkley@nist.gov
[1]Common Object Request Broker Architecture

set of services for enforcing a range of security policies using diverse security mechanisms. Some of these mechanisms require application systems to be aware of security. Such security models currently require application system designers to implement complex access control decisions based on content and context of interactions between client and target objects.

Security requirements in such domains as healthcare mandate domain-specific factors (e.g. relationship between the user and the patient, emergency context) to be used in access control policies. At the same time, commonality of business domain tasks and security requirements across an enterprise computing infrastructure promotes and requires exercising fine-grained access control policies in a uniform and standard way.

This paper describes a software system design solution that allows fine-grain application-level access control in such a way that application design decisions are isolated from complexity and idiosyncrasies of particular enterprise access control policies. We propose decoupling the authorization logic from application logic by encapsulating the authorization logic into the authorization service external to the application. In addition, our approach permits security administrators and application developers to maintain a clear separation of responsibilities.

Comparatively to other related approaches ([3], [4]), ours requires application-level enforcement of authorization decisions, and assumes agreement on semantics of resource names between the application developer and the owner. On the other hand, the proposed design of the authorization service provides a way to have any level of access control granularity, allows integration with existing authorization models and systems, and supports such dynamic attributes as patient–caregiver relationships using existing authorization models.

This paper shows that decoupling authorization logic from applications can be done without complicated interfaces and without sending much information between an application and the authorization service, and that factors specific to the application domain can be supported by authorization systems using the traditional access matrix as an underlying implementation.

The rest of the paper is organized as follows: the next section provides an overview of CORBA security model and describes its access control model, Section 3 discusses in detail the problem that we address in this paper, the proposed solution is presented in Section 4, pros and cons of the solution are discussed in Section 5, our approach is compared to related work in Section 6, we draw conclusions and discuss future work in Section 7.

2 Overview of CORBA Access Control Model

This section provides background on CORBA security in general and CORBA access control model in particular necessary for understanding some aspects of the problem and the solution discussed in this paper. The material in this section is compiled from [5, Chapter 15], [6] and [7]. We assume that the reader is familiar with basic principles and concepts of security, distributed systems and object-oriented paradigms. We introduce terminology and concepts of CORBA technology as it becomes necessary.

CORBA is an object-based distributed computing technology, which is becoming a worldwide industry standard for constructing distributed systems. The standardization process is led by the Object Management Group (OMG). The computing model CORBA technology adheres to is outlined in [8]. CORBA objects are identified with interfaces defined in the OMG Interface Definition Language (IDL). CORBA interface is a collection of three things: operations, attributes, and exceptions. An implementation of a CORBA object implements an interface identified with that object. In the context of CORBA model and this paper, interfaces and objects mean the same because object implementation is encapsulated in the object, and only its interface is known to other CORBA entities. Objects have object references by which they can be referenced. An object reference is a handle through which one requests operations on the object.

The CORBA Security (CS) model is designed to allow implementations to mitigate the following threats [5]:

- An authorized user of the system gaining access to information that should be hidden from him.
- A user masquerading as someone else.
- Security controls being bypassed.
- Eavesdropping on a communication line, so gaining access to confidential data.
- Tampering with communication between objects - modifying, inserting and deleting items.
- Lack of accountability due, for example, to inadequate identification of users.

The model comprises the following functionalities visible to application developers and security administrators: identification and authentication, authorization and access control, auditing, integrity and confidentiality protection, authentication of clients and target objects, optional non-repudiation, administration of security policies and related information.

One of the objectives of CS is to be totally unobtrusive to application developers. Security-unaware objects should be able to run securely on a secure ORB without any active involvement. On the other hand, it must be possible for security-aware objects to exercise stricter security policies than the ones enforced by CS. In CS model, all object invocations are mediated by the appropriate security functions in order to enforce various security policies such as access control.

Those functions are part of CS and are tightly integrated with the ORB because all messages between CORBA objects and clients are passed through the ORB.

CS uses the notion of principal. "A *principal* is a human user or system entity that is registered in and authentic to the system" [5]. In translation to the traditional security terminology, a principal is a subject. Initiating principals are the ones that initiate activities. CS manages security policies based on the security attributes of principals and attributes of objects as well as operations they implement. Objects which have common security requirements are grouped in security policy domains. "A *security policy domain*[2] is a set of objects to which a security policy applies for a set of security-related activities" [5]. When an object is created, it automatically becomes a member of one or more domains, and therefore is a subject to the security policies of those domains. Security policies are the rules and criteria that constrain the use of objects belonging to the particular security policy domain. Security policies define rules for access control, authentication, secure invocation, privilege delegation, non-repudiation, and audit. Security policies can be enforced either by the ORB or by the application. In the latter case, such an application is called a *security-aware application*. Domains allow application of security policy information to security-unaware objects without requiring changes to their implementations or interfaces.

As can be seen in Figure 1 on page 4, the client-side and target-side invocation access policy governs whether the client can invoke the requested operation on the target object on behalf of the current principal. This policy is enforced by the ORB in cooperation with the security service it uses, for all (security-aware and unaware) applications. A client may invoke an operation on the target object as specified in the request only if this is allowed by the object invocation access policy. The client and the target application access policy is enforced within the client and the target object implementation. This is enforced by security-aware applications.

[2]We will use "domain" to mean "security policy domain" unless otherwise specified.

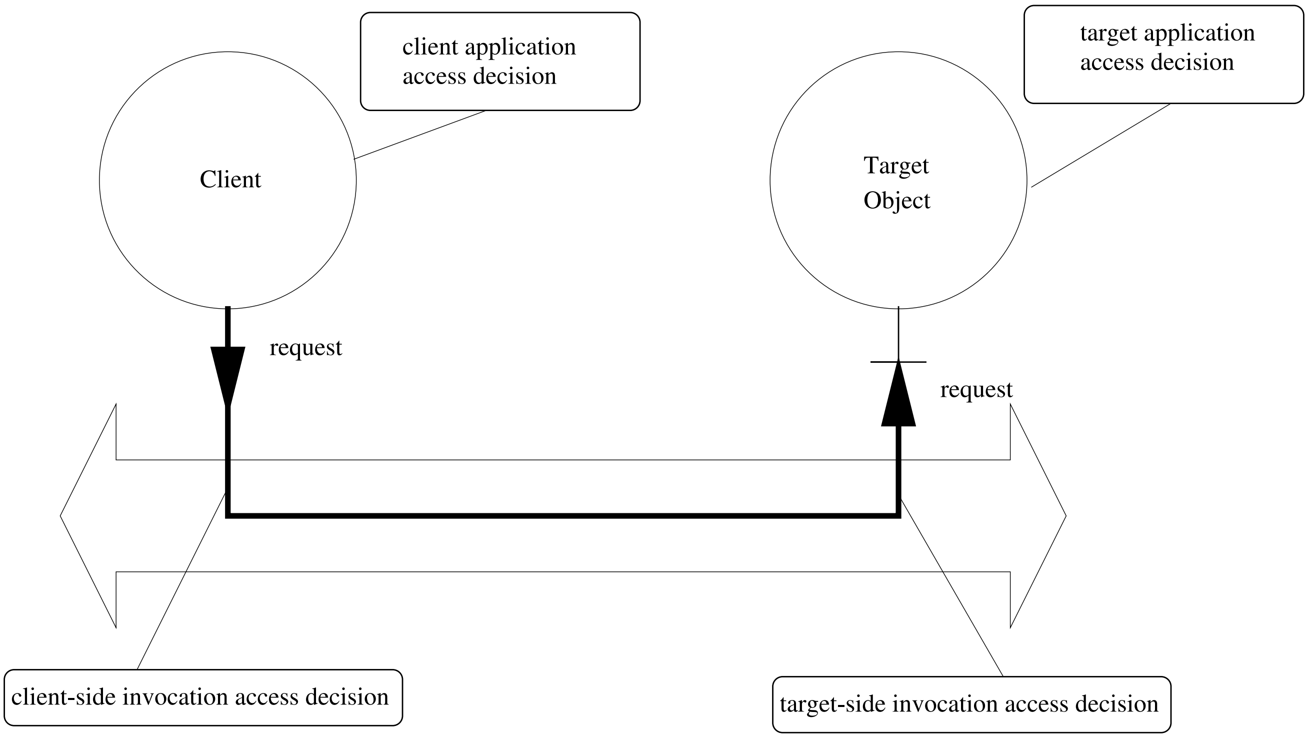

Figure 1: CORBA Security service access control model (from [5])

The access policy for client-side and target-side invocation is built into Access Decision functions. They provide a yes/no answer when asked to check if access is allowed. The most common CS access decision function used in object invocation access control bases its decisions on the following information:

- The security privilege attributes of the principal
- The requested operation
- The control attributes of the target object

A principal can have a variety of privilege attributes used for authorization decisions in Access Decision functions: access identity, roles the user associated with the principal performs, groups the user a is member of, security clearance, capabilities, and other privileges that the user organization defines as being useful for controlling access. CS defines a standard, extensible set of privilege attribute types. For simplicity, we use the name "security attributes" to refer to principal privilege attributes in this paper.

Examples of control attributes of the target object are access control lists and classification of the object. In CS, control attributes of the target object are required rights associated with a set of operations on the object. Therefore, a user with specified privileges may have rights to invoke a specific set of operations. CS rights are qualified into sets of *rights* families. CS defines only one, "corba" family, of rights, containing three rights *get*, *set*, and *manage*. CS access control model groups subjects using security attributes, objects using policy domains, and operations using required rights. A RequiredRights object defines that for the invocation of each operation in the interfaces of a secure object, some set of rights is required. Whereas an AccessPolicy object grants rights to privilege attributes. Required rights are assigned on a "class" basis and not on an "instance" basis. Thus, no change to existing object interfaces is needed for assigning required rights. However, all instances of a class in the same access policy domain will always have the same required rights.

A CS implementation compliant with Security Level 2 specification provides to a security-aware implementation of the target object several interfaces, that allow it to:

- Obtain credentials of the invoking principal,[3] that contain security attributes of the principal besides other information,
- Obtain authorization decision from Access Decision object supplied by CS,
- Obtain required rights for a particular operation,
- Perform other operations related to various CS functions such as auditing, non-repudiation, and quality of protection.

The next section shows why there is need in security-aware implementations of CORBA objects to enforce their own access control policies as well as problems with embedding such control into application systems.

3 Problem Description

3.1 Why application-level access control

There are two main reasons for application-level access control. They are the necessity in fine-grain access control and the need for authorization decisions based on factors that can be "known" only to the application.

Fine-grain access control is necessary because sensitivity of the information accessed via the same *operations*[4] of a CORBA service interface is different sometimes. For instance, in healthcare different parts of the patient medical record have different levels of sensitivity. Trivial examples are patient name and HIV-related test data. However, according to the adopted by the OMG Person Identification Service (PIDS) [9] and to be adopted Clinical Observation Access Service (COAS) [10], various patient traits and clinical observations, which constitute patient records, are supposed to be accessed via the same operations. We illustrate the problem in detail.

The following simple example with PIDS is discussed in mored details in [11]. PIDS service is a set of CORBA objects each providing an interface with various operations. The operations allow to query and manipulate demographic and other information used for person identification. Complete demographic information about a person reveals person identifiers used in a particular clinical system domain. Therefore, information about what type of treatments that person has received and the full identity of the person can be inferred by unrestricted "browsing" of PIDS.

These queries require access control on the level of input argument values.

```
interface DemographicAccess : IdentificationComponent {
        Profile get_profile (
                in PersonId person_id,
                in SpecifiedTraits specified_traits)
            raises ( InvalidID, UknownTraits, DuplicateTraits );
```

[3] More correctly, of the immediate invoker or the originator of the request , or both and credentials of the intermediate objects depending on privilege delegation scheme supported by the CS implementation.

[4] Operation is a synonym to method in OO terminology. We use it according to the object management model [8] from the OMG.

```
    void update_traits(
            in PersonId id,
            in Profile the_profile )
        raises ( InvalidId, UnknownTraits, ReadOnlyTraits, DuplicateTraits );
};
```

The *get_profile()* operation returns the profile or subset of the profile that the service knows about the person. The passed in traits indicate what subset of the profile is being sought by the client. The service is supposed to control what principal can access what traits of what person_id.

The *update_traits()* operation is used to modify the profile of an already existing ID by adding in new traits and over writing any values for existing traits that are also passed in. Any traits already stored for that person but not mentioned in the passed in profile are left intact. The service is supposed to know what principal can update what *the_profile* of what *person_id*.

For the following two operations(*get_all_ids* and *find_candidates*), a reasonable access control policy (such as the one in [12]) would require that these operations return[5] person profiles or sequence of person IDs only of those patients that the requesting principal has a legitimate access to.

```
interface SequentialAccess : IdentificationComponent {
    ProfileList get_all_ids(
            in TraitNameSeq traits_requested,
            in IdStateSeq states_of_interest )
        raises ( TooMany, UnknownTraits, DuplicateTraits, InvalidStates );
};
```

Operation *get_all_ids()* returns profiles for all patients the service knows about that match one of the states passed in. The returned profiles only contain the traits indicated by the passed in parameter. The service is not supposed to return those profiles that the principal does not have access to.

```
interface IdentifyPerson : IdentificationComponent {
    void find_candidates(
            in ProfileSelector profile_selector,
            in Constraint constr,
            in IdStateSeq states_of_interest,
            in unsigned long max_sequence,
            in unsigned long max_iterator,
            out CandidateSeq id_sequence,
            out CandidateIterator id_iterator )
        raises ( TooMany, UnknownTraits, DuplicateTraits,
            InvalidAlgorithms, InvalidStates, InvalidConstraint, InvalidWeight );
};
```

Knowing some identifying information about a person (or group of people with common traits), a client can ask the service by invoking *find_candidates()* to find the candidate persons that the service thinks may match those traits. The service is not supposed to return those IDs that the principal does not have access to.

COAS is facing the same problem – granularity at the level of operations on objects is not sufficient for access control in CORBA healthcare services. Moreover, in the case of COAS, the problem is exacerbated by the fact that potentially the same operation on the same service object can be invoked by a client to access low and highly sensitive information because COAS is a service for accessing all clinical observations.

[5]Note that a method can return results via return value as well as via out and inout arguments.

Even though we use only the healthcare domain as an example throughout the paper, we believe that several other vertical domains, such as electronic commerce and finance, have access control policies of the same or even higher complexity. CORBA object model is used as a real-life example. We believe this problem stands for systems based on any technology.

Another crucial reason for application-level access control is the need in using application domain-specific factors in authorization decisions. Analyses made by one of the authors and discussed elsewhere [13], [11], [14] reveal the necessity of sophisticated access control policies in healthcare systems. They are due to the various legal and liability requirements imposed by state and federal legislation [15]. Ideally, authorization decisions in the healthcare domain should be based on the following factors [16]: subject affiliation, subject role, subject location, access time, and relationship between the subject and the patient whose records are to be accessed.

Relationship is a good example of an authorization decision factor, which is not only specific to the healthcare vertical domain, but also the value of which ideally should be derived from the information scattered across various clinical, billing, and patient registration systems. Some types of relationships that need to be managed in the healthcare context are: patient's primary care provider; admitting, attending, referring, or consulting physician of a particular patient; part of the patient care team; healthcare staff explicitly assigned to take care of the patient; patient's immediate family; patient's legal counsel or guard; personal pastoral care provider. The relationship factor is very dynamic and ideally it should be computed dynamically every time a decision is made. We expect that other vertical domains have similar requirements in access control policies regulated by domain-specific factors that cannot be modeled using groups, roles, or identities.

3.2 Problems with Authorization Logic Embedded in Application Systems

Since the application programmer understands the application functionality most intimately, building authorization logic into the application allows the application to control access at arbitrary granularity level and to use authorization rules of almost unlimited complexity. However, authorization logic coupled with application logic produces serious consequences. Embedding authorization logic into application systems causes problems that can be grouped in the following categories:

Software system engineering

From the point of view of building software systems, mixing authorization logic with application logic:

- complicates system design,
- reduces re-usability of the system and its design,
- affects the functional properties of the system,
- makes the scope of the change to be proportional to the size of the system, and thus increases expenses of doing such changes and of conducting testing and verification of the changes,
- increases maintenance expenses of the system.

Information enterprise security administration

With authorization logic embedded into application systems, enterprise security administrators end up having to configure such access logic on an application-by-application basis, which brings tremendous administrative overhead and highly increases chances of human error. Because each application system has its own access control model, which is administrated via proprietary interfaces, multiple inconsistent security authorization models co-exist in the same information enterprise. It is difficult to ensure consistency of authorization policies across the enterprise. Most of the time, security administrators end up having no guarantee, whatsoever, that access rules and, especially, changes to them are consistent across all application systems as well as with required company policies.

In addition, environment with mixed authorization and application logic merges an administrator's responsibilities with an application developer's responsibilities and visa versa. The approach presented in the next section permits security administrators and application developers to maintain a clear separation of responsibilities.

4 Decoupling Authorization Logic from Security-Aware Applications

In this section, first we describe the interactions between the application system and the authorization service. Then, we describe the proposed design of authorization service.

The primary objective of the proposed solution is to effectively decouple authorization logic from application logic. By this we mean the following:

Simplicity: The proposed design should enable simple, compact and efficient implementations. It should also ease the task of system maintenance and administration.

Generality: The solution should be applicable to a wide range of application domains. Thus, it should:

- allow arbitrary granularity of authorization policies,
- provide capability for the user organization (security and enterprise administrators) to control completely those aspects of the authorization process that depend on the work-flow model and authorization policies of the enterprise where the application system is deployed.

Flexibility: The solution should support a wide variety of policies and their changes. The proposal should be implementable using a variety of policy management and enforcement engines.

Scalability: The solution should scale well, both in terms of runtime performance and in terms of management interface simplicity and management data size.

Conservatism: The solution should take advantage of the CORBA Security service mechanisms rather than replacing them with a different model. The proposed design should be implementable using CORBA security environment as a base. Specifically, the proposed authorization service should use the CORBA security attribute structure to identify authenticated subjects to the access control mechanism.

4.1 Interaction Between Application Service and Authorization Service

The objective of the proposed approach is to decouple application-level authorization logic from application logic. In the proposed design, we encapsulate authorization logic into an authorization service external to the application. A simplified schema of interactions among application client, application service and an instance of authorization service is depicted in Figure 2. In order to perform an application-level access control, an application requires an authorization decision from such a service and enforces that decision. Simple interfaces between the application and the authorization service are used. An application programmer is required to make a single invocation on the authorization service in order to obtain a decision.

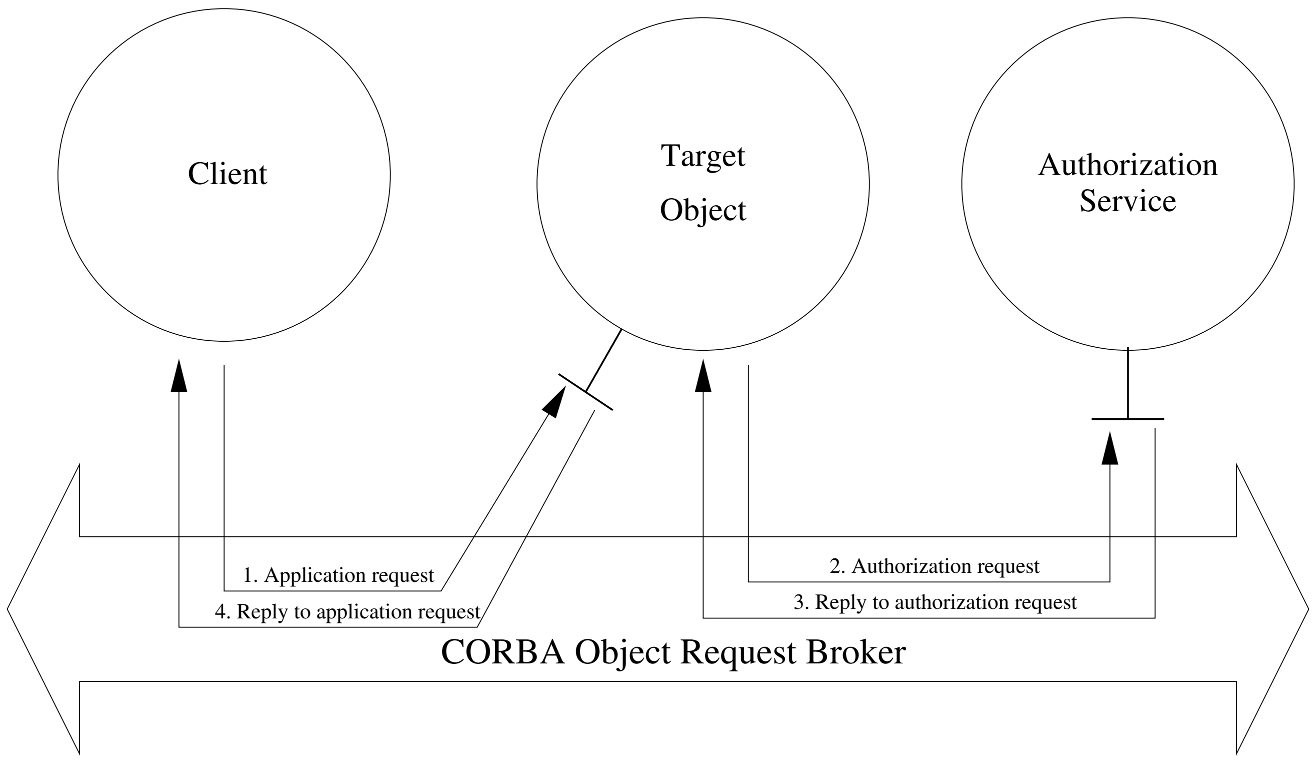

Figure 2: Interactions between client, application system, and authorization service.

In the presented schema, we depict entities *Client*, *Target Object* and *Authorization Service* (AS), as "tenants" of an instance of CORBA-compliant object request broker (ORB). They all perform in ORB environment and use CORBA services to interact with each other via operation *request* and *reply* primitives going through the ORB. The design of Authorization Service exemplifies a concrete implementation of the approach proposed in this paper. In a less abstract representation than Figure 2, each of these entities functions in a separate instance of ORB, potentially, remotely from any other entity. For example, a client is usually located on a host different than target object, and, consequently, different ORB. Since the abstraction of CORBA ORB provides us with sufficient level of detail, we will describe the proposed solution at this level of abstraction.

The sequence of the interaction, illustrated by Figure 2, is as follows:

1. An application client invokes an operation on the interface provided by the target object. The invocation is translated by the ORB into a request message. This corresponds to step 1 in the figure.

2. The target object ORB receives the request and translates it into invocation of the appropriate operation on the target object.

3. While processing the invocation, the target object requires an authorization decision from the AS by invoking a corresponding operation on the object. The invocation, as in the previous case, is translated to a request by the target object ORB. Step 2 in the figure.

4. The request is translated back into invocation of the appropriate operation on the AS.

5. The AS completes the invocation and the reply is translated by the ORB into return from the invocation performed in 3. This corresponds to step 3 in the figure.

6. The target object, after receiving an authorization decision, enforces it. If access was granted by the AS, the target object returns expected results of the invocation. Otherwise, it either returns partial results or raises an exception. Step 4 in the figure.

7. The client receives either results of the invocation or an exception.

Various issues, like discovering the location of authorization object by target object, are resolved via standard CORBA services [5] such as Naming or Trader. In the proposed design, each target object obtains an authorization decision only from one AS. It is the contract between the target object and its enterprise environment to request an authorization decision and to enforce it. Before we proceed with greater details on the design of an authorization decision object, we will describe syntaxes and semantics of a request for authorization decision.

From AS perspective, any target object requesting an authorization decision is an AS client. From now on, we will use the term "AS client" to refer to any entity of the distributed system that requested an authorization decision from an AS.

A nominal amount of data is passed between the application and the authorization service in order to make authorization decisions. When making a request for an authorization decision, an AS client passes the following three parameters:

- a sequence of strings representing a name of the resource to be accessed on behalf of the client,
- name of access operation[6] (e.g. "create", "read", "write", "use", "delete"),
- authenticated security attributes of the principal on behalf of which the client is requesting access to the named resource.

Security attributes here are exactly the same attributes as those that can be obtained by a *security-aware* application in the case of CORBA Security level 2 environment. They are described in details in Section 2 on page 2. The interesting parameters are the first two: resource name and access type. They are described below.

We introduce an abstraction called "protected resource name" or just "resource name." Resource name is used to abstract application-dependent semantics and syntaxes of entities under application-level access control. A resource name can be associated with any valuable asset of an application owner, which is accessed by a client on behalf of a principal using it, and access to which is to be controlled according to the owner's interests. For example, electronic patient medical and billing records in a hospital are usually its valuable assets. The hospital is interested in controlling access to the records due to various legal, financial and other reasons. Therefore, the hospital considers such records as protected resources. Moreover, different information in those records count as different resources. Examples of different resources can be records from different visits or episodes for one patient patients. At the same time, a resource name can be associated with less tangible assets, such as computer system resources, including CPU time, file descriptors, sockets, etc. The AS does not attempt to interpret semantics of the resource name. We will show in the discussion of the AS design that it uses the resource name only to obtain additional security attributes and to look up a set of policies that govern access to the resource associated by an application system with the resource name.

Access operation abstracts semantics of access to resources associated with resource names. An application may manipulate with patient records on behalf of different care-givers, or may provide different hierarchies of menus to different technicians of the hospital lab. In either case, it is up to

[6] When we discuss access on an entity associated with a resource name, we deviate from OMA's definition of the term "operation" [8] and use it in a more traditional meaning.

the application system vendor and the enterprise security administrators to agree on semantics of the operation name used for each access. The AS does not interpret semantics of access operation as it is shown in the description of the AS design.

Before a target object invokes the AS, it is supposed to identify what the resource name and the access operation name are associated with servicing the client request. There is not any particular algorithm defined for performing such an association. For every application, or at least for every application domain, the way of associating protected entities with abstract resource names can be different. Although we assume that such an association is possible, it remains to be shown that it is possible to abstract all protected resources into resource names.

4.2 Design of Authorization Service

Authorization service is composed of the following objects[7]:

- *Access Decision Object* (ADO) receives requests on authorization decisions from AS clients.
- Zero or more *Policy Evaluators* provide evaluation decisions for those policies that govern access to the given resource. If a policy evaluator does not have any policy associated with the given resource name, the evaluator returns a result meaning "don't know," therefore delegating the evaluation combinator to apply either a closed-world or open-world policy while combining results from potentially several evaluators, depending on the combinator configuration.
- *Policy Evaluator Locator* keeps track of and provides references to potentially several policy evaluators.
- *Dynamic Attribute Service* provides dynamic attributes of the principal in the context of the intended access operation on the given resource associated with the provided resource name.
- *Evaluation Combinator* object combines results of the evaluations made by policy evaluators into a final decision by resolving evaluation conflicts and applying, if necessary, closed and open policies.

Figure 3 presents an object interaction diagram of the authorization service.

Once the authorization service received a request via an ADO interface, it makes an authorization decision according to the following algorithm:

Algorithm 1

1. Obtain object references to those *PolicyEvaluators* that are associated with the resource name in question,
2. Obtain dynamic attributes of the principal in the context of the resource name and the intended access operation on it,
3. Obtain results from zero or more *PolicyEvaluators*, and
4. Combine evaluation results from step 3 into an authorization decision.

[7] Since in OMA a service entity can implement multiple interfaces, and objects are nothing else but implementations of interfaces, we refer here to an object to signify a particular interface implementation. An implementation of the authorization service described here can implement any number of the specific interfaces in one entity of the CORBA environment.

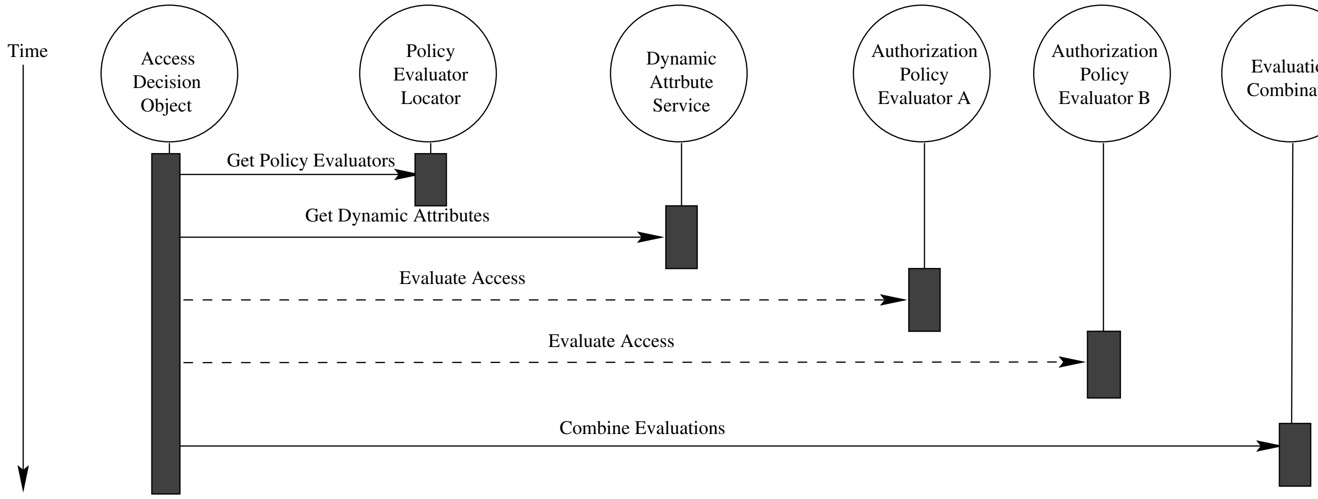

Figure 3: Authorization Service: Object Interaction

As it can be easily seen from the figure and its description, ADO performs the role of Mediator [17].

One of the significant points of the proposed approach is handling the factors specific to the application domain in the manner neutral to their semantics. All such factors are handled as dynamic attributes. They are obtained from the enterprise environment via specialized dynamic attribute services. An authorization service does not interact with such services directly. It delegates the generic dynamic attribute service to collect all dynamic attributes from specialized services. The semantic of a particular application domain (patient–care-giver relationship) can be expressed in the form of dynamic attributes. This allows utilization of already existing authorization mechanisms such as the traditional access matrix.

The objective of the dynamic attribute service (DAS) is to proxy several specialized DASs. We applied Proxy and chain of Responsibility design patterns [17] for achieving it. Our design allows more than one specialized DAS object to resolve dynamic attributes of the principal. It provides the capability to issue a request on obtaining dynamic attributes without specifying the receiver of the request explicitly. The set of specialized DAS objects that can handle a request are specified dynamically via registering them with the generic DAS using administrative interface to it.

Figure 4 on page 13 illustrates the intent of DAS services.

Dynamic attributes are those attributes that express properties of a principal but are not administrated by security administrators. A user usually has dynamic attributes due to the various activities the user performs in the enterprise work-flow. Dynamic attributes are so called because their values usually change more frequently than traditional user (static) security attributes. Traditional static security attributes are used for describing relatively fixed properties of users and/or resources. The values of static attributes are typically set by security administrators and are obtained by an application in an environment specific manner, e.g., from a principal's credentials in case of CORBA environment. While the use of a dynamic attribute in an access decision is determined by a security administrator, the values of dynamic attributes are usually set as part of normal processing, i.e., dynamic attribute values are usually part of information content not separately maintained security metadata. Consequently, dynamic attribute values must be obtained at the time an access decision is required. This is in contrast to traditional static security attributes whose values are usually obtained when a session is established. The values of dynamic attributes may change during a session as a result of normal workflow processing.

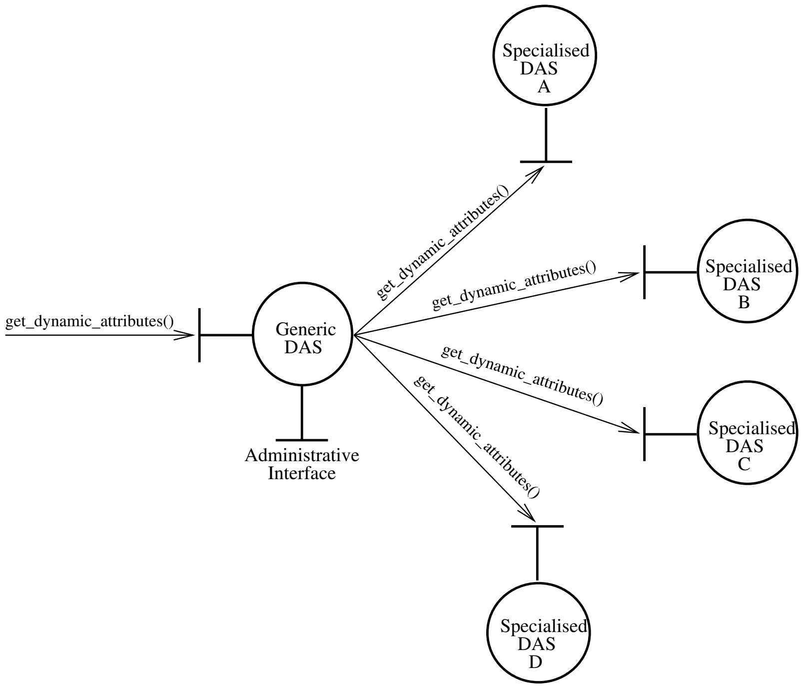

Figure 4: Provision of dynamic attributes by generic DAS via several specialized DASs.

Consider the following example of a dynamic security attribute. Physician A attends patient B. The physician has an attribute specifying such a relationship when principal A (representing physician A) is accessing resources associated with medical records of patient B. This relationship attribute is an example of a dynamic attribute in our model. It has the value "attending physician" returned by a generic DAS only when A accesses B's records. The generic DAS obtains the value of this relationship attribute by consulting a specialized DAS, which has capabilities to compute the value of relationship attribute (probably by looking at the corresponding fields of B's patient record which contains a list of B's attending physicians). When A is accessing resources not associated with any patient, this dynamic attribute of type relationship is not returned by the corresponding specialized DAS and consequently it is not returned by generic DAS.

There are hidden semantical dependencies between specialized DASs and application systems via resource names. Specialized DASs use resource names in order to obtain work-flow information or information specific to application domain in order to determine the corresponding dynamic attributes of the principal in the context of the current access request. For example, consider a specialized DAS, which provides relationship attributes specifying what the relationship is between the accessing principal and the patient associated with the named resource being accessed. In the healthcare environment, a relationship service is a good example of providing access to information specific to the application domain. In order for a dynamic attribute of type relationship to become available, a relationship service with the support of specialized DAS should be available and registered with generic DAS. The DAS needs to understand semantics of the resource name. Understanding of the resource name semantics by specialized DASs and by application system developers requires either a common framework for mapping protected resources into resource names or use of application-specific DASs for each application. The later option is least useful because specialized DASs are intended to be enterprise-dependent and not application dependent for those enterprises where COTS[8] applications are deployed. Thus, a common

[8]COTS stands for Commercial **O**ff **T**he **S**helf

framework for naming a protected resource in some application domains is needed.

There are might be hidden semantical dependencies between *PolicyEvaluators* and specialized DASs. PolicyEvaluators make authorization decisions by evaluating access control policies. Those policies potentially use dynamic attributes provided by specialized DASs. The user organization has to make sure that it exercises authorization policies that use only available dynamic attributes as decision factors and semantics of those attributes matches the ones used in the policies.

Another significant design element is encapsulation of authorization policies and their evaluators into first-class CORBA objects – *PolicyEvaluators*. Policy evaluators can be considered either as distinct authorities each representing a different set of authorization policies, or they can be considered as policy evaluation machines each supporting a particular policy language. Such design insulates representation and interpretation of policies from the authorization service. It also allows adding and removing policy evaluators dynamically. By encapsulating the evaluation of those policies in *PolicyEvaluator* objects, the design supports implementation of arbitrary authorization policies.

Each policy evaluator comes with its own interface for expressing and administrating authorization policies, if any. Since *PolicyEvaluators* are CORBA objects, they can be arbitrarily complex or simple. The enterprise security designers, depending on many factors including disclosure policies, available resource and accessible technologies, define how complex policy evaluators they want to use.

Separation of concerns among various stakeholders involved in the authorization process (application developers, enterprise security administrators, authorization service developers, "work-flow administrators"[9]) enables control of different factors in the authorization process by appropriate parties:

- Application developers decide what functions of their application map into what access operations.
- User administrators control what users (or roles) are assigned what static security attributes.
- Implementors of the authorization services and other third party vendors control quality, performance and other properties of the authorization service implementation.
- "Work-flow administrators" indirectly control what dynamic attributes are assigned to what users in the context of what resources.
- Security administrators administrate what access control policies govern what access to what named resources.

Generality of the design allows use of the authorization service in any application domain. It can even be utilized for implementation of CORBA Security service authorization mechanisms.

5 Discussion

Our solution has the following advantages:

Simplicity: At the time of writing this paper, we do not have an implementation of the authorization service. However, as Figure 3 shows, the proposed design allows straight-forward implementation of the service due to its decomposition into objects with simple interfaces. Simple interfaces between the application and the authorization service are used. An application programmer is required to make a single invocation on the authorization service in order to obtain a decision. All required information is represented by such simple structures as resource names, operation names,

[9]There is no such term as "work-flow administrators." We created it to refer to those who affect the enterprise work-flow process.

and principal security attributes. A nominal amount of data is passed between the application and the Authorization Service in order to make authorization decisions.

The programming model of the Authorization Service described by Algorithm 1 is simple. The programming complexity of making authorization decisions for an individual policy is encapsulated in *PolicyEvaluatorLocator, DynamicAttributeService,* and *PolicyEvaluator* objects. Thus, simple policies allow overall simplicity of the model. The complexity increases only by introducing complex types of authorization policies and sophisticated *specialized DynamicAttributeServices. PolicyEvaluatorLocator* can be as simple as an implementation of relational table indexed by resource name. For a simple implementation, when no invocations to the external services are made by the authorization service, the communication complexity of the authorization service is $O(n+k)$, where n is the number of *PolicyEvaluators* that evaluate authorization policies for a given access request, and k is number of specialized DASs.

Generality: Due to the design, the proposed solution can be applied to various application domains. It introduces the notion of resource name, which in its turn allows arbitrary granularity of protected resources. The application system decides, depending on the application domain, how small the unit of access control is. The resource name, principal security attributes as well as request dynamic attributes, and the intended operation name should communicate any semantic information that can be used for applying reasonable[10] authorization policies. The design supports arbitrary authorization policies by encapsulating the evaluation of those policies in *PolicyEvaluator* objects.

Flexibility: Due to the use of CORBA infrastructure with object implementation location transparency and its services such as Naming and Trader, the proposed design enables implementations adaptable to changes in authorization policies and their types as well as in the work-flow of the user organization via replacement of *PolicyEvaluators* and *specialized DynamicAttributeServices.* New *PolicyEvaluators* can be registered with the *PolicyEvaluatorLocator* and new *specialized DynamicAttributeServices* can be registered with the *DynamicAttributeService* object or obtained via CORBA Naming or Trader services. The semantic of a particular application domain (patient–care-giver relationship) can be expressed in the form of dynamic attributes. This allows utilization of already existing authorization mechanisms such as the traditional access matrix. Separation of concerns among various stakeholders involved in the authorization process enables control of different factors in the authorization process by appropriate parties.

Scalability: Computational complexity of the authorization service implementation is bound by communication complexity, which is $O(n+k)$, as we showed above. Thus, the performance is proportional, in a general case, to the number of policies that govern particular access and to the number of specialized dynamic attribute services registered with the generic dynamic attribute service. Since communication complexity is the dominant factor in the computational complexity of distributed systems, we do not expect significant performance degradations with an increase in the number of resource names per authorization service. Also, we expect that in an average case only a few PolicyEvaluators will be consulted per each access request. We expect the number of specialized *DynamicAttributeServices* to depend on the particular application domain and user organization, and be in the order of 10. If our expectations are correct, then computational complexity of the authorization service should be $O(1)$ and should not depend on the number of users or resources per application system.

Administration interfaces are required to manage correlation between access requests and *PolicyEvaluators* as well as to register specialized *DynamicAttributeServices.* Since we expect only a few specialized *DynamicAttributeServices* to be registered with particular authorization service, no scalability concerns are expected with managing *DynamicAttributeServices*. On the other hand, due to the arbitrary granularity of the protected resources, the number of resource names can be large enough in some application domains to cause high administration overhead. Current design of the administration interfaces [18] allows only explicit administration of the correlation between a resource name and *PolicyEvaluator*s, which makes the management data size to be $O(r+p)$ where r is the number of resource names and p is the average number of *PolicyEvaluator*s per resource name. Some application domains might require better scalability of the management data. However, such administration interfaces enable simplicity of their implementation.

Conservatism: The proposed solution takes advantage of the CORBA security environment. It uses principal security attributes, in addition to the resource and the access operation name, to obtain dynamic attributes inherent in the access request context.

We can see the following outstanding issues with the proposed approach:

- It is not clear whether it is possible to abstract all protected resources into resource names. The proposed solution requires such abstraction.
- Matching in dynamic attribute semantics between policy evaluators and specialized dynamic attribute services has to be maintained.
- One of the ways to reduce performance penalties of obtaining a decision from an authorization service is to co-locate an application system and an authorization service. Simple co-location increases the number of authorization service instances to administrate. On the other hand, an optimum administration solution would be such that it requires to administrate only one instance of administration interfaces. Current design of the authorization service does not provide ways to have a single set of administration objects and multiple instances of authorization services.

6 Related Work

In this section, we discuss other approaches for decoupling authorization logic from application logic and compare them with the proposed one. The systems described here have other features and some of them are not directly addressing the issue of separating authorization logic from an application. We review and discuss only their aspects related to the topic of this paper.

6.1 PCASSO

The primary purpose of the PCASSO[11] system is to provide "secure access to highly sensitive patient information over the Internet" [3]. A PCASSO server stores all application data in an RDBMS. The data is received for RDBMS storing in the form of HL7[12] messages. Every access request is identified with an accessing user, a patient identity (if the performed role is patient-specific), relationship[13] of the user with the patient or user role if the access is not patient-specific, as well as arguments and a reference to a predefined RDBMS procedure. All data in the RDBMS[14] instance is associated with a

[10] We do not define here what policies fall in the scope of reasonable ones. We think it is the subject of separate research, which we describe in Section 7.

[11] Patient Centered Access to Secure Systems Online

[12] Health Level 7

[13] [3] does not use the term "relationship" although it distinguishes between patient-specific roles (what we call "relationships") and patient-independent (what we call "roles").

[14] Relational Data Base Management System

sensitivity label. Sensitivity of data is by default determined using HL7 type of data because certain HL7 types have clear sensitivity and could be mapped into the five PCASSO sensitivity levels. Also, a sensitivity level can be set or altered by an authorized user. The system uses label-based access control. Authorization decisions are made and enforced by the RDBMS security mechanisms. Privileges to access data of particular sensitivity are granted to various patient-centric relationships. For each patient, a relationship has an access control list that determines what users can assume the relationship with the patient. The mechanisms insure that no data with a sensitivity level higher than the one permitted to the user, having a particular relationship with the patient, is accessed by the user.

Business logic in applications developed using PCASSO system is expressed in the form of queries on relational database tables. Authorization logic in PCASSO systems is separated from application logic by assigning (explicitly or by default) sensitivity labels to all data to be accessed. Such an approach is straight-forward in implementing authorization mechanisms.[15] However it has several disadvantages comparatively to the approach proposed here. It restricts the access control model to the one based on sensitivity levels combined with patient-centric relationships, which are protected by access control lists. All protected resources have to be associated with data elements stored in a database. Relationships with patients are managed explicitly by primary care providers (usually attending physicians) instead of deriving such relationships from already available data in other healthcare services available in the enterprise environment. The design does not allow to use other, besides patient–care-giver relationship, domain-specific factors in authorization decisions. The design explicitly requires an application to assume the identity of the accessing user, which allows only one type of credentials delegation – impersonation.

6.2 Argos

Argos [4] primary purpose is to serve as an access control system at the global layer of heterogeneous database federations. The primary contribution of Argos, which we discuss here, is a configurable access control model which is able to enforce a variety of access control policies in the area of identity-base access control. In Argos, a federated DBMS (FDBMS) receives a request on invocation of a method on an object and invokes the responsible instance of Argos reference monitor. The monitor checks if the request is permitted and sends the results of the authorization decision to the FDBMS, which enforces it. It is possible to have several instances of the reference monitor, but only one can be active for a system. A reference monitor is configured during its initialization. Several parameters of initialization define corresponding aspects of the policy that the monitor is to apply for authorization decisions. The variety of configuring parameters allow definition of a broad range of access control policies. The granularity of available policies is at the level of methods on object types (classes in C++ and Java). Besides groups, Argos supports implicit access rights based on role hierarchies, (nested) subject domains, object type hierarchies, (nested) protection object domains, and method classes.

In Argos, the finest level of access control decisions is an object method. In our approach, the authorization decision is obtained after the method on the object is invoked. Hence, an application can exercise access control of any granularity level by associating a resource name with protected elements of any size and semantics. One reference monitor (supporting a particular policy) at a time is used in Argos to evaluate requested access. Due to introduction of multiple evaluators and a combinator, we provide ways for more than one policy (of different types) to govern authorization decisions. The proposed concept of dynamic attribute service gives enough flexibility in using enterprise-specific factors to support all implicit access rights that Argos does as well as other types of access rights. Our approach allows Argos reference monitor to be used as one of the policy evaluators in the authorization service

[15] Class B2 COTS RDBMS is used in PCASSO implementation.

described here. This would be similar, although not exactly the same, to what Johnscher and Ditrich suggest in [4] when they write that "Argos can be used as an access control service for any application that is connected to the corresponding object request broker."

7 Conclusions

In this paper we presented an approach in decoupling authorization logic from application logic for those CORBA-based application systems, which resort to application-level access control in order to achieve fine granularity of protection or to use factors specific to the application domain in authorization decisions, or both. We described the design of an authorization service that allows any level of access control granularity, applying authorization policies of different types and from different authorities, as well as providing application domain-specific factors for evaluating such policies.

The following two results are the main contribution of the paper:

- Decoupling access control from applications can be done without really complicated interfaces and without sending much information between an application and the authorization service.
- Dynamic attributes, such as the patient–caregiver relationship, can be supported using a traditional access matrix as an underlying implementation.

The body of the work described in this paper has been served as a foundation of the initial submission [18] for the Object Management Group specification on Healthcare Resource Access Control. At the time of writing this paper,[16] a complete set of run-time interfaces is defined in OMG interface definition language (IDL). Initial prototyping is underway.

We plan to complete design of administration interfaces, to show what types of policies can be supported by the proposed design effectively, to develop a more precise specification of the authorization service, and to obtain experimental data on performance and scalability of the described solution.

References

[1] Ravi Sandhu and Pierangela Samarati. Access control: Principles and practice. *IEEE Communications*, 32(9), September 1994.

[2] Joseph Yoder and Jeff Barcalow. Application security. In *Proceedings of The 4th Pattern Languages of Programming Conference*, 1997.

[3] Dixie B. Baker, Robert M. Barnhart, and Teresa T. Buss. PCASSO: Applying and extending state-of-the-art security in the healthcare domain. In *Annual Computer Security Applicatications Conference*, 1997.

[4] Dirk Jonscher and Klause R. Dittrich. Argos – a configurable access control system for interoperable environments. In *Proceedings of the IFIP WG11.3 Ninth Annual Working Conference on Database Security*, pages 39–66, Rensselaerville, NY, 1995.

[5] Object Managment Group. *CORBAservices: Common Object Services*, November 1997. OMG document number: formal/98-07-05 http://www.omg.org/corba/csindx.htm.

[16] November, 1998

[6] Ulrich Lang. Security aspects of the common object request broker architecture. Master's thesis, Royal Holloway University of London, 1997. http://www.cl.cam.ac.uk/~ul201/mscdissertation.pdf.

[7] Günter Karjoth. Analysis of authorization in CORBA security. Technical report, IBM Research Division, Zurich Research Laboratory, December 1996.

[8] Richard Mark Soley and Christopher M. Stone. *Object Management Architecture Guide.* John Wiley & Sons, 3 edition, June 1995.

[9] Object Management Group. *Person Identification Service*, February 1998. OMG document number: corbamed/98-02-29.

[10] Object Managment Group. Clinical Observations Access Service RFP. OMG document number: corbamed/97-12-28 http://www.omg.org/library/schedule/Clinical_Observations_RFP.htm, December 1997.

[11] Wayne Wilson and Konstantin Beznosov. *CORBAmed Security White Paper.* Object Management Group, November 1997. OMG document number: corbamed/97-11-03.

[12] Ross Anderson. A security policy model for clinical information systems. In *Proceedings of IEEE Symposium on Security and Privacy*, 1996.

[13] Konstantin Beznosov. Applicability of corba security to the healthcare problem domain. OMG document number: corbamed/97-09-11, September 1997.

[14] Konstantin Beznosov. Issues in the security architecture of the computerized patient record enterprirse. In *Proceedings of Second Workshop on Distributed Object Computing Security*, Baltimore, Maryland, USA, May 1998. The Object Management Group and the United States National Security Agency. http://www.omg.org/docsec/1998/.

[15] Konstantin Beznosov. Taxonomy of CPR enterprise security concerns at Baptist Health Systems of South Florida. http://www.bhssf.org/IT/Projects/cpr/security/progress-reports/categorize-requirements.html, December 1997.

[16] Konstanantin Beznosov. Requirements for access control: US healthcare domain. In *Proceedings of the Third ACM Workshop on Role-Based Access Control*, page 43. Fairfax, Virginia, USA, October 1998.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[18] Object Management Group. *Healthcare Resource Access Control (Initial Submission)*, October 1998. OMG document number: corbamed/98-10-02.