OMG CORBAmed DTF

Resource Access Decision(RAD)

Revised Submission

2AB, INC.

Baptist Health Systems of South Florida

CareFlow|Net, Inc.

IBM

OMG TC Document corbamed/99-03-01

1 March 1999

© Copyright 1999 by 2AB, Inc.

© Copyright 1999 by Baptist Health Systems of South Florida

© Copyright 1999 by CareFlow|Net, Inc.

© Copyright 1999 by IBM

The submitting companies listed above have all contributed to this "initial" submission. These companies recognize that this initial submission is the joint intellectual property of all the submitters, and may be used by any of them in the future, regardless of whether they ultimately participate in a final joint submission.

The companies listed above hereby grant a royalty-free license to the Object Management Group, Inc. (OMG) for worldwide distribution of this document or any derivative works thereof, so long as the OMG reproduces the copyright notices and the below paragraphs on all distributed copies.

The material in this document is submitted to the OMG for evaluation. Submission of this document does not represent a commitment to implement any portion of this specification in the products of the submitters.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The companies listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material. The information contained in this document is subject to change without notice.

This document contains information which is protected by copyright. All Rights Reserved. Except as otherwise provided herein, no part of this work may be reproduced or used in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems without the permission of one of the copyright owners. All copies of this document must include the copyright and other information contained on this page.

The copyright owners grant member companies of the OMG permission to make a limited number of copies of this document (up to fifty copies) for their internal use as part of the OMG evaluation process.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013.

CORBA, OMG and Object Request Broker are trademarks of Object Management Group.

History

Initial Submission Draft 1	18 October 1998
The Editor of this document is Carol Burt of 2AB. The initial submission to the OMG Resource Access Decision Facility (RAD) is the result of collaboration between the four submitting companies listed on the cover and the list of supporting companies named in the initial submission.	
Revision to Initial Submission	**13 November 1998**
The COAS submitters were invited to a RAD submitters meeting in Burlingame on November 12, 1998. The purpose of this meeting was to discuss the addition of standard ResourceNames in the final specification and the appropriate structural representation of the ResourceName in IDL. As a result, the ResourceName type has been modified to be a sequence of name/value pairs (where the name and value are strings). This document has been updated to reflect that decision.	
Revised Submission	**1 March 1999**
A submitters meeting was held in Austin, TX on February 17-19, 1999 to address outstanding issues. This included consideration of all feedback received on the initial submission from CORBAMed and issues raised during the prototyping effort. A number of changes were made to ensure that implementations could be scalable and be designed for maximum performance. These changes included introducing an Authority Id into the ResourceName as a prefix to the name/value pairs in the structural representation of the name and the concept of a ResourceNamePattern that can be used for grouping. Evaluators were named to allow a DecisionCombinator to apply precedence logic for making decisions. This required that the internal logic flow be modifed such that the ADO calls the DecisionCombinator who is responsible for consulting the appropriate PolicyEvaluator(s).	

Table of Contents

1. Preface ______ *5*

1.1 Submission Contact Points 5
1.2 Supporting Organizations.......... 5
1.3 Conventions.......... 6
1.4 Terminology.......... 7
1.5 Proof of Concept.......... 8
1.6 Changes to Adopted OMG Specifications.......... 8
1.7 Response to RFP Requirements.......... 8

2. Overview of Response ______ *12*

2.1 Introduction.......... 12
2.2 Design Goals.......... 13
2.3 Reference Models.......... 15
2.4 Discussion of Proposal Scope.......... 21

3. DfResourceAccessDecision module ______ *23*

3.1 Types 24
3.2 AccessDecision interface.......... 31
3.3 DynamicAttributeService interface 32
3.4 PolicyEvaluatorLocator interface.......... 33
3.5 DecisionCombinator interface 33
3.6 PolicyEvaluator interface.......... 35
3.7 AccessDecisionAdmin interface 36
3.8 PolicyEvaluatorLocatorAdmin interface 36
3.9 PolicyEvaluatorAdmin interface.......... 42

4. Conformance Classes ______ *45*

5. Appendix - Use Case Example ______ *46*

5.1 Generic RAD Sequence Diagram.......... 46
5.2 Healthcare Scenario: Out-patient Visit to Attending Physician 47

6. Appendix – Resource Names for PIDS ______ *55*

6.1 Changes to Conformance Classes.......... 55

7. Appendix - Complete IDL ______ *57*

1. Preface

This submission is a response to CORBAmed RFP, Healthcare Resource Access Control , Object Management Group (OMG) document number corbamed/98-02-16.

1.1 Submission Contact Points

Carol Burt 2AB 3178-C Highway 31 South Pelham, AL 35124 205 621 7455 cburt@2ab.com	Konstantin Beznosov Baptist Health Systems of South Florida 6855 Red Road Coral Gables, FL 33143 305 596 1960 beznosov@baptisthealth.net
V. "Juggy" Jagannathan CareFlow|Net, Inc. 235 High Street, Suite 225 Morgantown, WV 26505 304 293 7535 juggy@careflow.com	Bob Blakley IBM 11400 Burnet Road, Mail Stop 9134 Austin, TX 78756 512 838 8133 blakley@us.ibm.com

1.2 Supporting Organizations

The following organizations have been involved in the process of developing, prototyping and/or reviewing this submission. The submitters of this response thank them for participating and giving their valuable input. A special thank you goes out to those organizations. The editor would like to extend a special thanks to John Barkley of NIST for providing text and a detailed review of each version of this submission.

- Concept Five
- Inprise
- Los Alamos National Laboratory
- National Institute of Standards (NIST)
- National Security Agency (NSA)
- Philips Medical Systems

1.3 Conventions

1.3.1 IDL

```
IDL appears using this font and in a border.
```

1.3.2 Precondition/Postcondition semantics

In this specification we use a "design by contract" precondition/postcondition semantics. This means:

- An object which receives a call assumes that its preconditions are true at the time the call is received
- If the preconditions are true, then:
  - When the object returns from the call, the postcondition will be true, OR
  - The object will throw an exception, indicating that it was unable to fulfill its contract
- If ANY of the preconditions is not true, then the object's behavior is undefined, and, in particular, the postconditions are not guaranteed to be true when the object returns from the call

1.4 Terminology

Access Decision Object (ADO) - The RAD object that implements access decision functions. From the perspective of a client requesting an access decision of RAD, this is the only interface that they are required to use. Although similar in function to the CORBAsec object of the same name, the RAD ADO has a different signature and semantics.

ADO Client - the immediate invoker of the RAD Access Decision Object. This could be an integral part of the application that controls the secured resources or it could be an interceptor that decides whether to allow the CORBA Request to reach the application.

Access Policy - the policy or rules that govern access to a secured resource.

AttributeList - a list of security attributes that are used to determine whether access should be allowed. An AttributeList may contain both static and dynamic attributes.

Authorization - The granting of authority, which includes the granting of access based on access rights.[1]

Component - A cohesive set of software services

Credentials - Information describing the security attributes (identity and/or privileges) of a user or other principal. Credentials are claimed through authentication or delegation and used by access control.[1] The attributes contained in an RAD AttributeList are derived from CORBAsec credentials, if possible.

Dynamic attribute - a security attribute that can only be determined at the time an access decision is requested. Dynamic attributes are often based on the relationship between a principal and the secured resource (such as attending physician) and cannot be statically configured. This submission allows dynamic attributes to be resolved and used during the access decision computation.

Identity (attribute) -A security attribute with the property of uniqueness; no two principals' identities may be identical. Principals may have several different kinds of identities, each unique (for example, a principal may have both a unique audit identity and a unique access identity). Other security attributes (e.g. groups, roles, etc...) need not be unique.[1]

Naming Authority - Any organization that assigns names determines the scope of uniqueness of the names and takes the responsibility for making sure the names are unique within its name space. In the same way that ID values are meaningful only within the context of their ID Domains, names are unique only within the context of their naming authority.[2]

Operation - an action which may be performed on a secured resource (such as create, get, set, use..). Operations are represented within the RAD as strings.

Principal - A user or programmatic entity with the ability to use the resources of a system.[1]

Privilege (Attributes) - security attributes which need not have the property of uniqueness, and which thus may be shared by many users and other principals. Examples of privileges include groups, roles, and clearances. [1]

---

[1] This definition is taken from the OMG CORBAsecurity 1.2 specification. OMG document number ptc/98-02-01

[2] This definition is taken from the OMG CORBAmed Person Identification Service (PIDS). OMG document number corbamed/98-02-29

Secured Resource - a “secured resource” is any valuable asset of an application owner, which is accessed by an application on behalf of a principal using it, and access to which is to be controlled according to the owner’s interests.

Security attributes - Characteristics of a principal which form the basis of the system’s policies governing that subject.[3]

Static Attribute - a security attribute that is (typically) statically configured by an administrator. Examples would be access_id:john_doe or role:physician.

System - An application or set of applications that interact with each other, interact with the RAD or implement RAD. System in this context is synonymous with application. Examples of systems might include a hospital or clinical information system, an ancillary system such as a lab or radiology system, or a financial/administrative system such as an ADT.

1.5 Proof of Concept

The revised initial submission has been prototyped and the implementation of the prototype is available to OMG members for non-commercial use. This prototype is on the OMG web server as corbamed/99-01-19.zip. The specification is based on experience gained in implementation of proprietary access control systems by 2AB, Concept Five, and IBM and with requirements input from end user organizations such as Baptist Health Systems of South Florida, and Healthcare vendors such as Phillips Medical Systems and CareFlow|Net. The submission was revised based on issues raised during the prototyping efforts and feedback from the healthcare task force.

1.6 Changes to Adopted OMG Specifications

Changes to the CORBAMed Person Identification Service corbamed/99-02-29 have been made by this submission and are outlined in Appendix 6. These changes specify how an implementation of PIDS must name resources when an RAD is used to provide access decisions.

1.7 Response to RFP Requirements

1.7.1 MANDATORY RFP REQUIREMENTS

Use of the CORBA Security service credentials as the source for identifying caregivers’ privileges

The submission's AccessDecision Interface takes a Security::AttributeList as the source for identifying caregivers' privileges. This attribute list is directly accessible from the Security::Credentials. The Credentials object itself is not passed because it is locality constrained whereas the submission's AccessDecision object is not locality constrained.

Ability to define secured resource categories.

The ResourceName is a sequence of strings where the first string in the sequence is required to be

[3] This definition is taken from the OMG CORBAsecurity 1.2 specification. OMG document number ptc/98-02-01

a NamingAuthority::QualifiedNameStr. This allows an implementation to define categories of secured resources and for a client to determine from the resource name how those categories are arranged into hierarchies. The use of the NamingAuthority module allows these categories to be unique.

An interface for defining access control rules for secured resources based on credentials.

The submission does not provide interfaces for defining rules, as there are a large number of rule languages supported by existing products, and there does not appear to be a consensus regarding which language or languages should be chosen. The submission does provide interfaces for applying named policies (presumably created and named through proprietary interfaces) to secured resources.

A set of Healthcare specific secured resources.

The submission team decided not to provide generic healthcare resource names. Definition of healthcare generic resource name space did not look a viable way to address this mandatory requirement. Instead, we decided to provide solutions for adopted, in progress, and future Healthcare DTF specifications.

Existing Specifications

The submission team decided to provide rules for creating resource names and operation names for Person Identification Service specification (PIDS) (corbamed/98-02-29) because this is a specification from Healthcare DTF, which is already adopted by the OMG. The submission provides changes to PIDS. The changes define what resource and operation names should be used by PIDS-compliant services if such services choose to use RAD. Section 6 of the submission contains changes to PIDS specification.

Specifications in Progress

The submission team worked with the submitters of the upcoming specification of Clinical Observation Access Service on defining what resource names should be used by COAS-compliant services if such services choose to use RAD.

Future Specifications

We recommend to Healthcare DTF to mandate definition of such rules in all future specifications. The rule should be defined on case by case basis. The submission team believes that all future RFPs recommended by Healthcare DTF should contain a mandatory requirement. The requirement should ask submissions to specify what RAD resource names will be used by compliant implementations if they choose to use RAD-compliant services

An interface to an access control decision facility that may be used to request access control decisions.

The AccessDecision interface provides this capability

1.7.2 OPTIONAL RFP REQUIREMENTS

Provide the ability for secured resources to be grouped for the purpose of defining access control rules

The submission does not constrain an applications' assignment of ResourceNames to application entities; this allows applications to create ResourceNames which refer to groups of application entities if desired.

An interface for defining access control rules based on attributes of the Principal (in addition)

The submission does not constrain the form of access control rules. The AccessDecision interface accepts the list of Principal attributes as an input parameter to the access_allowed() method; this allows AccessDecision objects whose rules are based on Principal attributes to receive the information they need to have in order to evaluate their rules. As noted above, the submission does not provide an interface for defining access control rules.

An interface that extends the definition of access control rules to include context sensitive access control based on a) the day and time when the resource is accessed, b) the location of an invoking principal, c) the values of request parameters.

The AccessDecision interface does not need any explicit parameters to support rules based on day and time; implementations can support such rules using the interfaces defined in the submission.

The submission does not support access control rules based on the location of the invoking Principal, unless information about the invoking Principal's location is provided as a security attribute (it is not clear to the submitters what data could be used to define the invoking Principal's location). The submission does not support access control based on the values of request parameters, unless parameter value information is encoded into the ResourceName by the application.

An interface that extends the definition of the access control rules to include notion of the relationship between a patient and a caregiver

The DynamicAttriubuteService interface was designed specifically as a generic way to support relationship based (and other dynamic-attribute based) decision rules. The DynamicAttributeService interface permits the AccessDecision object to determine at runtime what dynamic attributes (e.g. relationship between caregiver requesting access and patient to whose record access is requested) apply to the requested operation. Each PolicyEvaluator object accepts as input a list of attributes, including dynamic attributes, and uses these to make its access decision.

A reference object model for the healthcare domain that provides a sufficient foundation for access decision logic.

The submission contains a description of the access control model (including object/interface diagrams and object interaction diagrams) and its interaction with invoking healthcare applications.

An interface that permits management of policy, which controls how multiple access control policy decisions governing access to the same resource are reconciled.

The submission defines a DecisionCombinator interface that meets this requirement.

1.7.3 DISCUSSION POINTS REQUESTED

How new CORBAMed specifications will employ the submitted specification.

Due to the generality of the submitted response, the specified service can be used in various ways. The usage patterns will highly depend on a particular enterprise, its workflow and access control policies. Usage of the service even in systems compliant with CORBAmed specifications is expected to vary from company to company. On the other hand, the submission team believes that semantics of the interactions between an RAD service and its clients should be defined completely and precisely. It is the intent of the submission team to provide in further revisions of this response a complete and precise definition of the semantics. Besides defining semantics of interactions between RAD service and its clients, the response provides sample scenarios and use cases. The appendix in Section 6 of the response provides such scenario's and use cases along with discussion

about how the specified service is intended to be used in general cases and specifically in healthcare applications compliant with the OMG standards.

How existing CORBAMed specifications are to be modified.

The submitters do not believe any modification is necessary for existing CORBAMed specifications to use the services of an RAD, however it might be useful for some standard ResourceNames to be defined within the CORBAMed community for common resources within standard services. Such definition would be a compatible extension of existing specifications.

Scalability and Performance of the proposal

The process of making authorization decisions on fine grained resources is an expensive and inefficient action when compared to the course interface/operation control provided by the existing CORBA access control facilities. The RAD service is intended for use when granularity and/or expressiveness of CORBA security access control model is insufficient. Thus, the submission team believes that use of RAD service is a necessary "evil" in terms of overall system performance. Performance impact can be kept to a minimum by clever implementations, but it cannot be eliminated in any security-aware application.

Scalability and performance of systems implementing the proposed specification and usage of it by other CORBA-compliant systems are highly dependent on the following factors:

- Internal design and implementation of the RAD-compliant service, including ability to cache policies.
- Co-location of RAD services and RAD clients.
- Distribution of load over multiple instances of the RAD services.
- Organization of the resource space.
- Complexity of access control policies.

Taking the above discussion into account, the submission team does not believe there is anything in the proposed design model of RAD service that would preclude implementation and deployment of scalable, high performance RAD services.

1.7.4 Mechanisms provided for extensibility

This discussion will be provided in the final submission after outstanding issues are resolved.

2. Overview of Response

2.1 Introduction

This document is a response to the Healthcare Resource Access Control RFP. The response describes a specification of Resource Access Decision(RAD) Service. RAD service is a mechanism for obtaining authorization decisions and administrating access decision policies. It enables a common way for an application to request and receive an authorization decision. The service is intended to be used by security-aware applications.

This submission provides access decision functionality not supported by CORBAsecurity which is required in healthcare and other application environments. It is intended to be implementable using CORBAsecurity as a base; it is also intended to be implementable in ORB environments which do not provide CORBAsecurity. For detailed information about the healthcare environment's access control requirements, refer to the RAD RFP (OMG document number corbamed/98-02-23).

In the proposed design, authorization logic is encapsulated within an authorization service that is external to the application. In order to perform an application-level access control, an application requests an authorization decision from such a service and enforces that decision. A simplified schema of application flow is depicted in Figure 1.

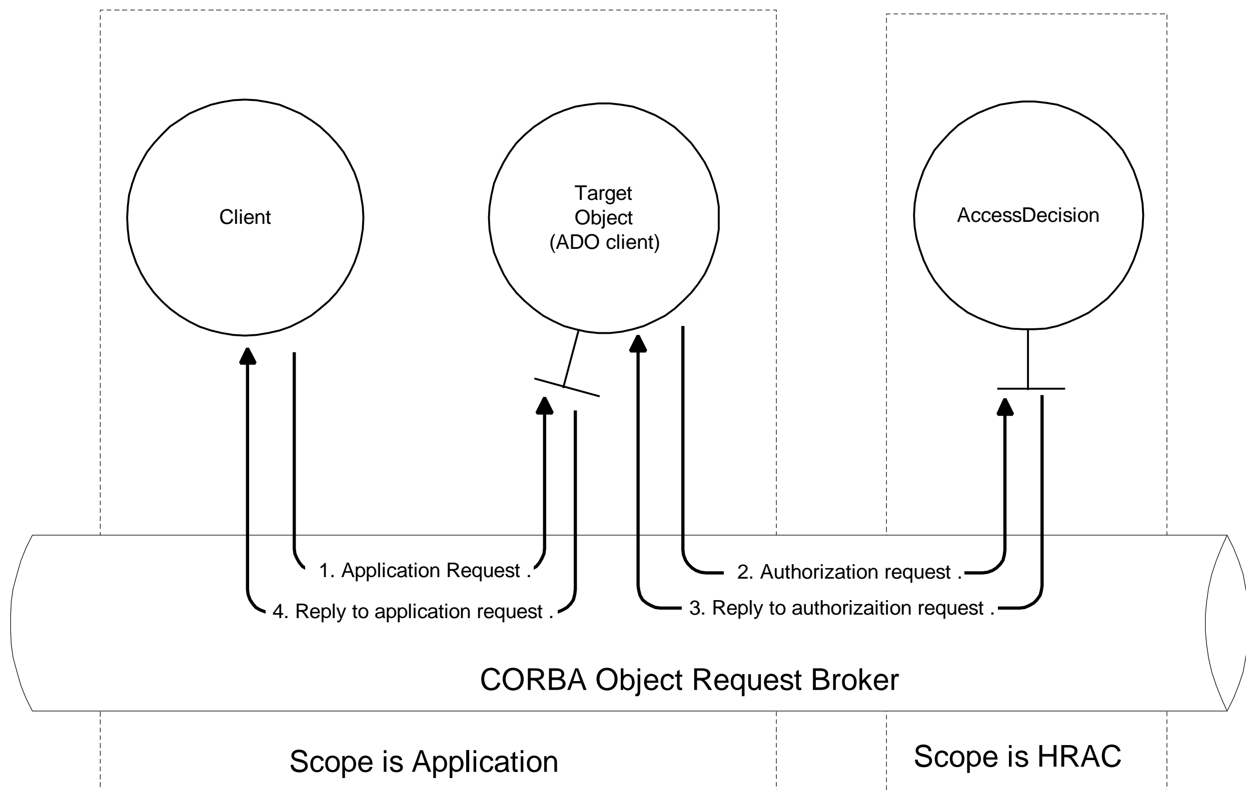


Figure 1

The sequence of the interaction, illustrated by Figure 1, is as follows:

1. An application client invokes an operation of the interface provided by the target object. The object request broker transfers this request to the target object and causes invocation of the appropriate method in the target object.
2. While processing the request, the target object requests authorization decision(s) from the Access Decision object by invoking the access_allowed() method of the ADO.
3. The Access Decision object consults other objects that are internal to the RAD (described in this submission) to make an access decision. The access decision is returned to the Target Object (ADO client) as a boolean.
4. The target object, after receiving an authorization decision, is responsible for enforcing the decision. If access was granted by the ADO, the target object performs the requested operation and returns the results. If access to secured resources was denied, the target object may return partial results or raise an exception to the Client.

A detailed description of the object model and design of the ADO (and its interaction with other RAD objects) can be found in Section 2.3 of this submission.

2.2 Design Goals

The submitters had the following goals in mind during the design of this submission:

2.2.1 Conservatism:

The proposal should extend the CORBAsecurity mechanisms rather than replacing them with a different model. The proposal should be implementable using CORBAsecurity as a base. Specifically, the proposal should use the CORBA security attribute structure to identify authenticated subjects to the access control mechanism.

2.2.2 Minimality:

The proposal should define the smallest number of interfaces and methods possible. The proposal should be easy to implement, and implementations should be small.

2.2.3 Simplicity:

The proposal should have a simple administrative model and a trivial runtime-programming model.

2.2.4 Generality:

The proposal should be applicable to and useful in domains other than healthcare.

2.2.5 Relevance:

The proposal should satisfy the healthcare access control requirements set forth in the Healthcare Resource Access Control RFP. Specifically, the proposal should:

(a) Define a notion of controlled resource, which allows extension of CORBA security protection to system entities other than CORBA objects.

(b) Support enforcement of policies which take the following factors into account when making access control decisions:

- relationship between the requester and the accessed resource or its owner, subject, or referent
- value or sensitivity of information contained within resource
- time (e.g. time of day, day of week)

(c) Support management of access control policy in a policy-language-independent way

(d) Support OMG PIDS and COAS access scenarios

2.2.6 Flexibility:

The proposal should support a wide variety of policies (especially healthcare-appropriate policies). The proposal should be implementable using a variety of policy management and enforcement engines (including existing healthcare security packages).

2.2.7 Scalability:

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

2.3 Reference Models

Two views of the RAD are presented in the following models. The first is the access decision model. This represents the relationship of objects involved in making an access decision. The second view is the Administrative view and represents how an RAD is configured. Administration of Access Policy is beyond the scope of the RAD and is clearly indicated as such on this model diagram.

The Resource Access Decision facility reference model defines a framework within which a wide variety of access control polices may be supported. The reference models below clearly indicate the scope of this submission response by heavy dotted lines. In some cases there are types that occur within the scope of this response that represents concepts and/or services that lie beyond the scope of the RAD. An example of this is the concept of a "secured resource" which is only represented within the scope of the RAD by a ResourceName. Where this occurs these external concepts appear in the model, but outside the dotted line to aid the reader in an understanding of the relationship between the RAD and the external concepts and/or services. The appearance of objects outside the scope of the submission is conceptual and is presented only to aid in understanding the types that occur within the RAD.

RAD types that represent or encapsulate external concepts and/or services:

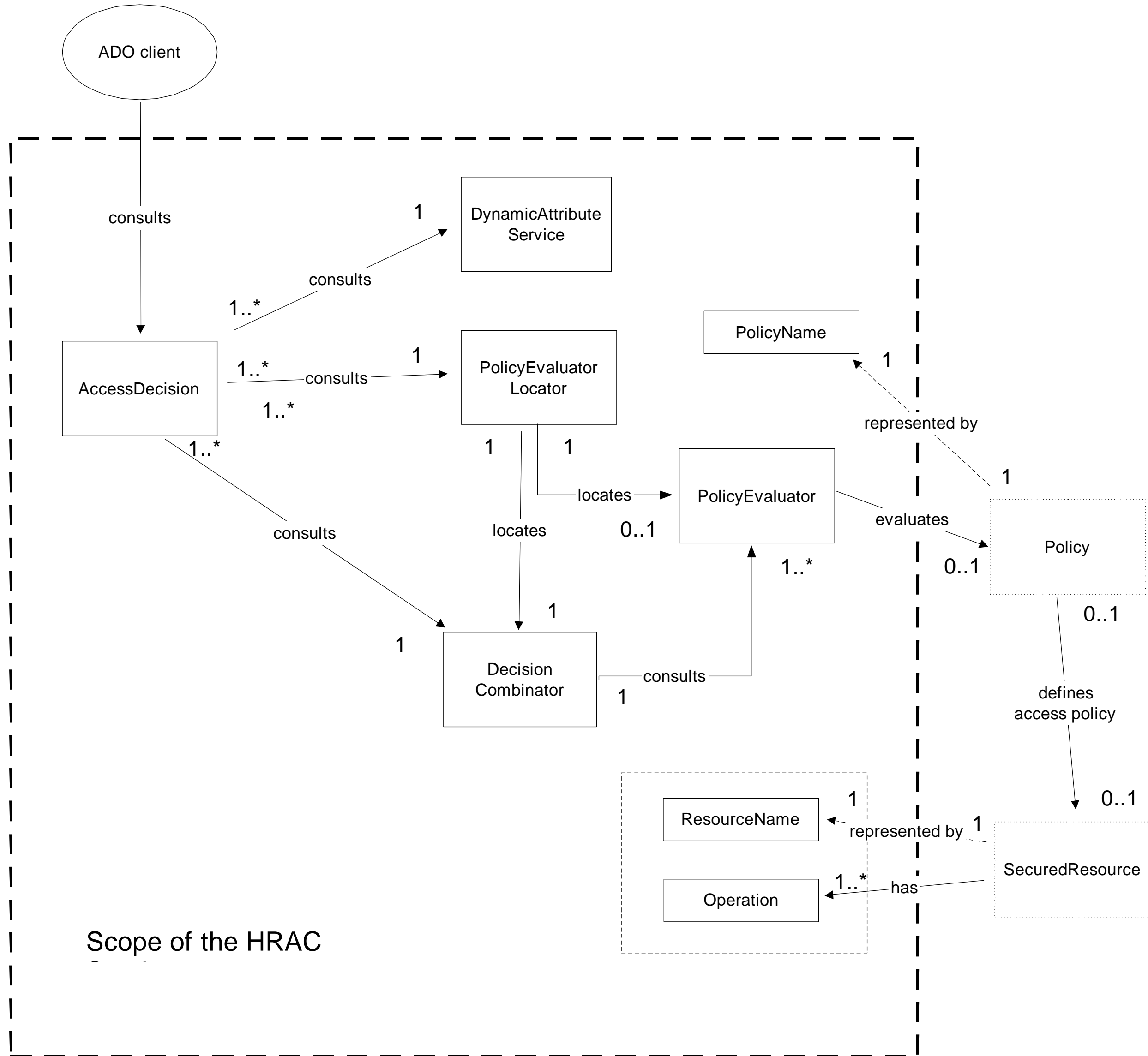
ResourceName: A "secured resource" is represented within the RAD by a ResourceName that is a structure containing an AuthorityId for the namespace and a sequence of name/value pairs..

Operation: Secured resources have one or more operations which may be performed on them (such as create, get, set, use..). These operations are represented within the RAD as strings.

PolicyName: "Policy" (the rules used for controlling access to secured resources and their operations) is beyond the scope of the RAD, but when referenced within the RAD, is identified by a PolicyName that is a string.

DynamicAttributeService: The DynamicAttributeService may consult an external AttributeEvaluator. The submitters plan to include standard administrative interfaces for this facility in the revised submission.

2.3.1 Access Decision Model



An Access Decision is requested by a client by invoking the access_allowed() method of the AccessDecision object (ADO) passing a ResourceName, Operation, and SecAttributes. The ADO consults a DynamicAttributeService to obtain an updated list of SecAttributes that include any dynamic attributes currently applicable for this access decision. The DynamicAttributeService may consult externally provided dynamic attribute evaluators as part of it's implementation. The AccessDecision object also consults the PolicyEvaluatorLocator to obtain object references for the PolicyEvaluator(s) and the DecisionCombinator that are required for an access decision. The AccessDecision object consults the DecisionCombinator that consults with any PolicyEvaluators responsible for interpreting access policy that controls access to the ResourceName/operation. The DecisionCombinator encapsulates policy combination logic and is responsible for understanding the policy that controls how a series of results from PolicyEvaluators are combined including any precedence rules that may apply. It is the response from the DecisionCombinator that is returned to the client.

2.3.2 Administrative Model

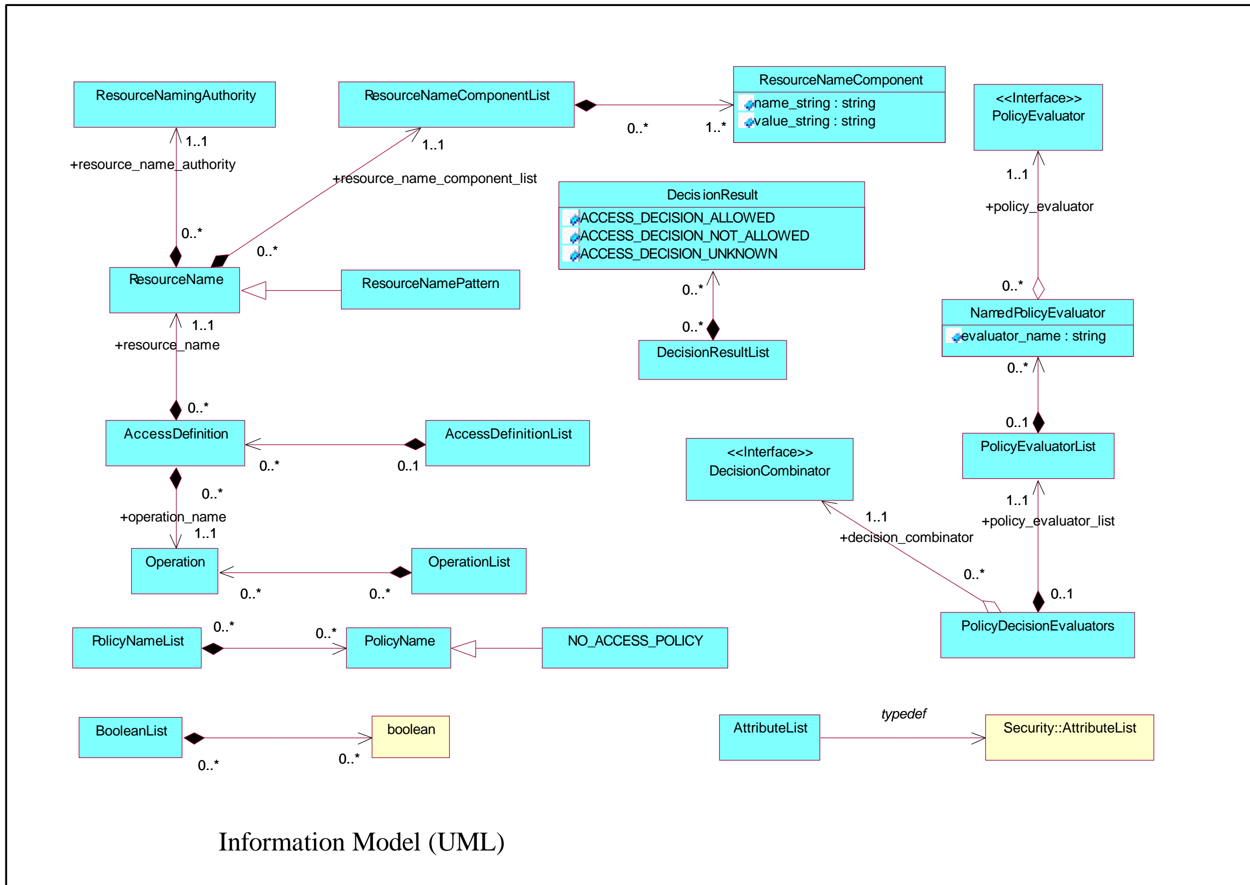
The administrative model of RAD is designed to allow replaceable RAD objects within an implementation and to allow RAD clients to apply previously defined policy to resources.

The administrative model is **not** intended to provide the Administrative interfaces necessary to define access policy. The definition of access policy (the rules that govern access to secured resources/operations) is outside the scope of this submission. This Administrative model clearly indicates this by placing Policy administration outside the dotted line that delineates the scope of the RAD submission.

The PolicyEvaluatorLocatorAdmin interface is used to associate PolicyEvaluators and DecisionCombinators with a ResourceName. Multiple PolicyEvaluators may be associated with a single ResourceName. These evaluators will all be consulted during access decisions. There is only one DecisionCombinator provided for a ResourceName. This combinator is responsible for taking the results of the PolicyEvaluators evaluate() method and making a final access decision. PolicyEvaluators have an endless series of options for implementation. For this reason, the interface is public and evaluators may be "plugged-in" to an RAD framework by vendors and/or users. In the same sense, there are many possible policies for combining policy decisions. Some secured resources should not be accessible unless all the PolicyEvaluators return ACCESS_DECISION_ALLOWED. Other secured resources may be accessible if any one of the PolicyEvaluators allow access. Defining an interface for the DecisionCombinators allows custom combinators to be configured for a secured resource. It is possible to assign a default DecisionCombinator.

The PolicyEvaluatorAdmin interface is used to apply an existing named access policy to a secured resource. An application that wished to dynamically apply policy to newly created resources would be required to specify the names of those policies. The policy would be configured by an administrator using the administrative interfaces of the underlying access policy system and the required name associated with it (this is outside the scope of the RAD admin interfaces). Once this had been accomplished, an RAD client could apply this named policy using the PolicyName to a ResourceName. The PolicyEvaluatorAdmin also allows default policy to be assigned "by name" and a list of existing PolicyNames can be retrieved via the interface.

2.3.3 Information Model

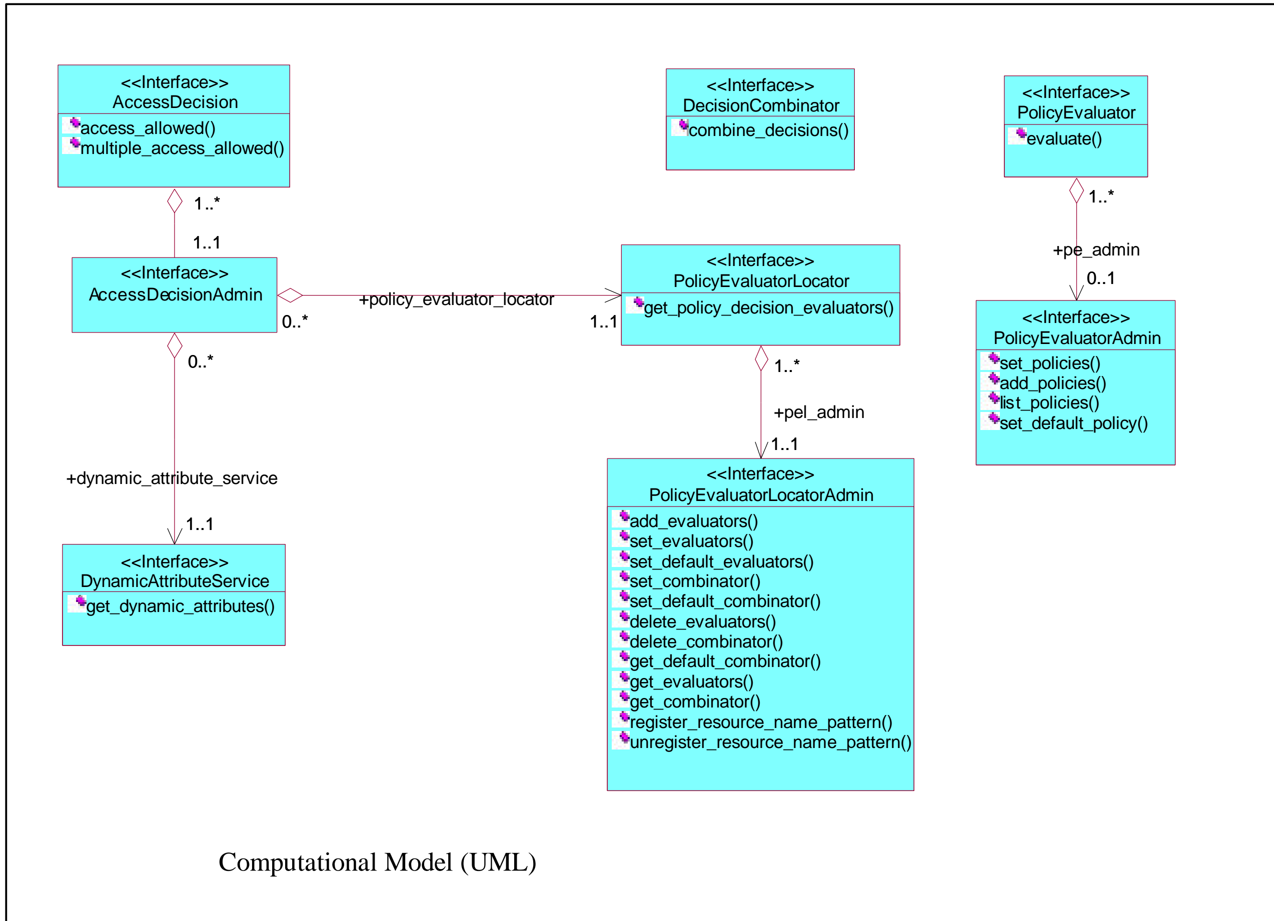


Information Model (UML)

The information model of RAD is designed to be simple to implement and to use. [TODO: add a brief description of the information model]

The information model will be discussed in detail in Section 3.1.

2.3.4 Computational Model



Computational Model (UML)

The computational model of RAD consists of two interface groups:

- Runtime interfaces: AccessDecision, DynamicAttributeService, PolicyEvaluator, PolicyEvaluatorLocator, and DecisionCombinator.
- Administrative interfaces: AccessDecisionAdmin, PolicyEvaluatorAdmin, and PolicyEvaluatorLocatorAdmin.

Among runtime interfaces, AccessDecision, PolicyEvaluatorLocator, and DynamicAttributeService are singletons, i.e. one instance of each interface is available in every implementation of RAD. On the other hand more than one instance of DecisionCombinator and PolicyEvaluator may be available.

The computational model will be discussed in detail in Section 3 .

2.4 Discussion of Proposal Scope

2.4.1 Scope as defined by the RFP

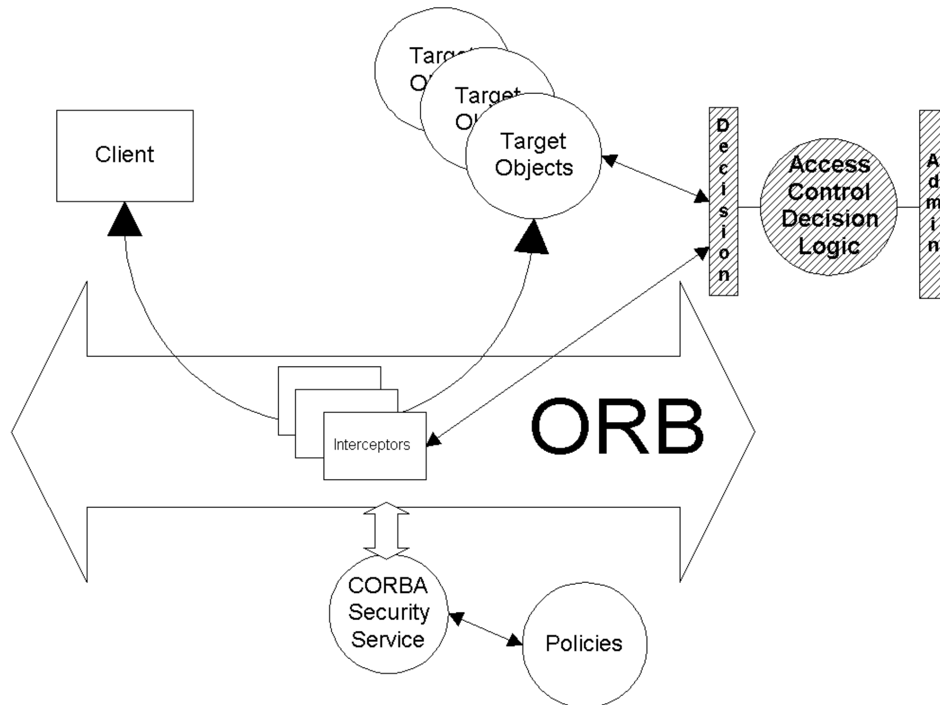
The CORBAmed Healthcare Resource Access Control RFP defined the scope of proposals sought as follows:

"Mechanisms this RFP is asking for are sought to allow application systems to be unaware of advanced security policies existing in healthcare enterprises where those systems are deployed.

This RFP scope is threefold:

1. *to de-couple access control decision logic from application logic,*
2. *to provide a standard interface for the definition of access control rules,*
3. *to provide a standard interface for requesting access control decisions.*

An illustration of the RFP scope is provided in Figure 1. The RFP scope is limited to the additional security decision logic shown in the figure with striped background. It has a "Decision" interface to an interceptor(s) performing access control functions and an application itself to consult such Security Decision Logic for access control decisions. The "Admin" interface allows defining access control rules.



A Possible Solution

The RFP is looking for a solution where individual applications, or target objects, play the most minimal role possible in the realization of an enterprise security policy. Optimally, each application's, or object's, contribution to security will be limited to requesting and enforcing access control decisions - without knowing or caring about how the decisions are made. This then

allows the definition, implementation and management of each application, or object, and the enterprise-specific security policy to be orthogonal. "

2.4.2 The Scope of this submission

The initial submission addresses the following scope issues of the RFP

1. to de-couple access control decision logic from application logic

   The submission supports the separation of access control logic from the application. The submitters are discussing extending this support in the final submission by allowing applications to register dynamic attribute evaluators via a standard interface.

2. to provide a standard interface for requesting access control decisions.

   The AccessDecision interface provides this functionality.

In addition, this submission extends this scope to provide a framework that supports the following:

1. replaceable authorization engines (PolicyEvaluatorLocator, PolicyEvaluatorLocatorAdmin, and PolicyEvaluator)
2. custom integration of multiple authorization engines (PolicyEvaluatorLocatorAdmin and DecisionCombinator)
3. use of dynamic attributes in access decisions (DynamicAttributeService)
4. The application of pre-defined access policy to a resource. (PolicyEvaluatorAdmin)

The submission does ***not*** address the following scope issue of the RFP:

1. to provide a standard interface for the definition of access control rules

   The submitters could not agree on IDL for the definition of access control rules. This is primarily because there are so many diverse ways that people express access control policies and accommodation from a single IDL interface for this diversity is not an easy task. The final submission may include an example of how the CORBAsecurity required rights model could be used to provide for the definition of access control rules **if** access control policy uses a "required rights" model. In general, the administration of access control policy was felt to be out of scope of this submission.

3. DfResourceAccessDecision module

```
//File: DfResourceAccessDecision.idl
//

#ifndef _DF_RESOURCE_ACCESS_DECISION_IDL_
#define _DF_RESOURCE_ACCESS_DECISION_IDL_

#include "Security.idl"

#pragma prefix "omg.org"

module DfResourceAccessDecision {

interface AccessDecision {
…
};

interface DynamicAttributeService {
…
};

interface PolicyEvaluatorLocator {
…
};

interface DecisionCombinator {
…
};

interface PolicyEvaluator {
…
};

interface AccessDecisionAdmin {
…
};

interface PolicyEvaluatorLocatorAdmin {
…
};

interface PolicyEvaluatorAdmin {
…
};

};

#endif // _DF_RESOURCE_ACCESS_DECISION _IDL_
```

The DfResourceAccessDecision contains four interfaces defined below and has type dependencies on the CORBA Security Service and the CORBAMed NamingAuthority modules.

```
#include <Security.idl>
```

The types declared within the Security service and used by the RAD are:

```
Security::AttributeList
```

These types are used for consistency with CORBASec and have the same meaning when used in RAD interfaces. They are typedef'd in this specification for ease of use.

#pragma prefix "omg.org"

In order to prevent name pollution and name clashing of IDL types this module (and all modules defined in this specification) uses the pragma prefix that is the omg DNS name.

3.1 Types

There are a number of structured types used widely through out the DfResourceAccessDecision Model. These types are described in this section:

3.1.1 Basic Types & Types used from the CORBA Security service

```
//*******************************************************
//         Basic Types
//*******************************************************

typedef sequence<boolean> BooleanList;

typedef Security::AttributeList AttributeList;
```

BooleanList

A sequence of boolean used as a return value when multiple decisions are requested. This type is used as a return value in the multiple_access_allowed() method of the AccessDecision interface.

AttributeList

The Security::AttributeList is defined as follows in CORBA Security 1.2 (ptc/98-01-02). The AttributeList is provided as an input parameter by the "application" client when a request for an access decision is made. The AttributeList used for access decisions may be modified to include dynamic attributes by use of the get_dynamic_attributes() method of the DynamicAttributeService interface. As a convenience to the reader, the structure of a Security::AttributeList is replicated below.

```
typedef sequence<octet> Opaque;

// security attributes
typedef unsigned long SecurityAttributeType;


struct ExtensibleFamily {
    unsigned short        family_definer;
    unsigned short        family;
};
struct AttributeType {
    ExtensibleFamily      attribute_family;
    SecurityAttributeType attribute_type;
};


struct SecAttribute {
    AttributeType         attribute_type;
    Opaque                defining_authority;
```

```
    Opaque                value;
    // the value of this attribute can be
    // interpreted only with knowledge of type
};

typedef sequence <SecAttribute> AttributeList;
```

3.1.2 Types that identify and manage information about secured resources

```
//*********************************************************
//   Types that identify a secured resource
//*********************************************************

struct ResourceNameComponent {
    string     name_string;
    string     value_string;
};
typedef sequence<ResourceNameComponent> ResourceNameComponentList;

typedef string ResourceNamingAuthority;

struct ResourceName {
    ResourceNamingAuthority   resource_naming_authority;
    ResourceNameComponentList resource_name_component_list;
};

typedef ResourceName      ResourceNamePattern;

typedef string              Operation;
typedef sequence<Operation> OperationList;
```

ResourceNameComponent

A datum element of this type is invalid if the name_string member has empty value.

ResourceNameComponent

A datum element of type ResourceNameComponentList is invalid if it is empty or any of its sub-elements is invalid.

ResourceNamingAuthority

A datum element of type ResourceNamingAuthority is invalid if it is empty or has invalid syntax.

ResourceName

A ResourceName is used to identify a **secured resource**. A ResourceName contains a unique identifier for the naming authority and a sequence of ResourceNameComponents. Each ResourceNameComponent includes a name and value string. This combination of naming authority and name/value pairs allows for categorization and grouping of resources if desired. A datum of type ResourceName is invald if either resource_name_authority or resource_name_component_list is invalid.

ResourceNamePattern

A ResourceNamePattern is used in Administrative interfaces to allow generalized regular expressions to be provided in the value_string of a ResourceNameComponent for the purpose of administering groups of secured resources. The regular expression syntax is defined by 9945-2:1993 (ISO/IEC) Information Technology-Portable Operating System Interface (POSIX)-Part2: Shell and Utilities IEEE/ANSI Std 1003.2-1992 & IEEE/ANSI 1003.2a-1992 Section 2.8, pages

77-91, "Regular Expression Notation". A datum of type ResourceNamePattern is invald if either resource_name_authority or resource_name_component_list is invalid.

Operation

A datum element of this type is invalid if it is empty.

OperationList

An OperationList is used to identify a list of operations that may be performed on a secured resource.

3.1.3 Types associated with evaluating Access Policy

```
//*****************************************************
//   Types associated with evaluating Access Policy
//*****************************************************
typedef string               PolicyName;
typedef sequence<PolicyName>  PolicyNameList;

const PolicyName NO_ACCESS_POLICY = "NO_ACCESS_POLICY";

struct NamedPolicyEvaluator {
       string          evaluator_name;
       PolicyEvaluator policy_evaluator;
};
typedef sequence<NamedPolicyEvaluator> PolicyEvaluatorList;

struct PolicyDecisionEvaluators {
    PolicyEvaluatorList   policy_evaluator_list;
    DecisionCombinator   decision_combinator;
};
```

PolicyName

A PolicyName is a string used to identify an access policy for a secured resource. This type is only used in the PolicyEvalutorAdmin interface. It is used as an input parameter to the replace_policy(), add_policy(), and set_default_policy() methods of the PolicyEvaluatorAdmin interface. PolicyNames are assigned by the administrative interface of the policy engine and cannot be modified or controlled by the RAD. There is one standard PolicyName of "NO_ACCESS_POLICY". See the PolicyEvaluatorAdmin interface for usage. A datum element of this type is invalid if it is empty.

PolicyNameList

A PolicyNameList is a sequence of PolicyNames. It is returned from the list_policy() method of the PolicyEvaluatorAdmin interface. A datum element of this type is invalid if it is empty or any of its sub-elements is invalid.

NamedPolicyEvaluator

A NamedPolicyEvaluator is a structure that contains the name of the Policy Evaluator and the object reference for the policy evaluator. The evaluator_name will be null in implementations that choose not to name evaluators. Providing named evaluators allows an implementation to apply precedence logic based on evaluator names when making an access decision.

PolicyEvaluatorList

A PolicyEvaluatorList is a sequence of NamedPolicyEvaluator. The PolicyEvaluatorLocatorAdmin interface allows the association of a list of NamedPolicyEvaluator(s) with a ResourceName. This type is returned from get_policy_evaluators() and set_default_evaluators() and is used as an input parameter in the set_evaluators(), add_evaluators(), delete_evaluators(), and set_default_evaluators() methods of this interface. The PolicyEvaluatorList returned from the PolicyEvaluatorLocator is passed to the DecisionCombinator returned from the PolicyEvaluatorLocator.

PolicyDecisionEvaluators

The PolicyDecisionEvaluators struct contains a PolicyEvaluatorList and the DecisionCombinator. This is the type returned from the get_policy_decision_evaluators() method of the PolicyEvaluatorLocator interface. This structure contains the references of all the objects that may be consulted during an access decision.

3.1.4 Types used to request access decisions

```
//***************************************************
//      Types used to request an Access Decision
//***************************************************

struct AccessDefinition {
    ResourceName   resource_name;
    Operation      operation;
};
typedef sequence<AccessDefinition> AccessDefinitionList;

enum DecisionResult {ACCESS_DECISION_ALLOWED,
                    ACCESS_DECISION_NOT_ALLOWED,
                    ACCESS_DECISION_UNKNOWN
};
```

AccessDefinition

The AccessDefinition struct is provided to allow multiple access definitions to be defined. It contains the ResourceName and the operation name for the secured resource access being requested. AccessDefinition is used as an input parameter to the access_allowed() method of the AccessDecision interface and the evaluate() method of the PolicyEvaluator interface. A datum element of this type is invalid if either of its members is invalid.

AccessDefinitionList

AccessDefinitionList is the type used to request multiple access decisions in a single operation. It is used as an input parameter to the multiple_access_allowed() method of the AccessDecision interface and the multiple_evaluate() method of the PolicyEvaluator interface.

DecisionResult

DecisionResult is an enum with three possible values. The values are:

ACCESS_DECISION_ALLOWED: the policy evaluated for this ResourceName, operation and Attribute list indicates that access is ALLOWED.

ACCESS_DECISION_NOT_ALLOWED: the policy evaluated for this ResourceName, operation and Attribute list indicates access is NOT_ALLOWED.

ACCESS_DECISION_UNKNOWN: the policy evaluated for this ResourceName, operation and Attribute list indicates an access decision cannot be made.

This type is used as a result in access decisions where access policy is applied. This is the type returned from the evaluate() method of the PolicyEvaluator.

3.1.5 Exceptions

The following exceptions are used in this module

```
//*****************************************************
//*             Exception Data types
//*****************************************************
struct ExceptionData {
    short   error_code;
    string  reason;
};
enum InternalErrorType {Fatal, NotFatal};

//******************************************************
//    Exception thrown by the Access Decision Object
//******************************************************

exception InternalError{InternalErrorType ed;};

//******************************************************
//    Exception thrown by Internal non-admin interfaces
//******************************************************

exception ComponentError{
    ExceptionData ed;
    InternalErrorType it;
};

//******************************************************
//         Exceptions thrown by Admin Interfaces
//******************************************************

exception PatternConflict {ExceptionData ed;};
exception PatternDuplicate {ExceptionData ed;};
exception PatternNotRegistered {ExceptionData ed;};
exception PatternInUse {ExceptionData ed;};
exception InputFormatError {ExceptionData ed;};
exception ResourceNameNotFound {ExceptionData ed;};
exception NoAssociation {ExceptionData ed;};
exception InvalidPolicy {ExceptionData ed;};
exception DuplicateEvaluatorName {ExceptionData ed;};
exception InvalidResourceName {};
exception InvalidResourceNamePattern {};

exception InvalidPolicyEvaluatorList {
    ExceptionData          ed;
    NamedPolicyEvaluator   first_invalid_element;
};

exception InvalidPolicyNameList {
    ExceptionData  ed;
    PolicyName     first_invalid_element;
};
```

ExceptionData

The ExceptionData structure is included in most RAD exceptions. The contents of the error_code and reason are implementation dependent.

InternalError

The InternalError exception is reserved for internal logic errors and is not used as a reason code for rejecting a request. This is the only exception that is thrown by the AccessDecision object. ADO clients are not exposed to the security reason for not allowing access. Indicating RadFatal means that the ADO client should discontinue using the ADO.

ComponentError

The ComponentError exception may be thrown by non-administrative interfaces to alert the AccessDecision object when a component encounters an internal error. If the ComponentError is RadFatal, the AccessDecision object must determine if it can continue to process without the component. If it cannot, it must throw a RadInternalError with RadFatal. If the Access Decision Object can continue to function without this component or if the exception error type was RadNotFatal, it is implementation dependent what the ADO returns to the client.

PatternConflict

The PatternConflict exception is thrown by the PolicyEvalutatorLocatorAdmin when a register_reousrce_name_pattern() detects a pattern that conflicts with an existing registered pattern and the implementation does not support conflicting patterns.

PatternDuplicate

The PatternDuplicate exception is thrown by the PolicyEvaluatorLocatorAdmin when an register_resource_name_pattern() detects a duplicate pattern registration.

PatternNotRegistered

The PatternNotRegistered exception is thrown is thrown by PolicyEvaluatorLocatorAdmin operations when an when an attempt is made to use a pattern in an administrative interface without registering the pattern first.

PatternInUse

The PatternInUse exception is thrown by PolicyEvaluatorLocatorAdmin unregister_resource_name_pattern when an attempt is made to unregister a pattern that is currently in use by the HRA

InputFormatError

The InputFormatError exception is thrown by the administrative interface operations when an input parameter is provided in a format that is unacceptable to the RAD implementation. The error_code and reason are implementation dependent.

ResourceNameNotFound

The ResourceNameNotFound exception is thrown by PolicyEvaluatorAdmin interface operations when a ResourceName has not been defined. Not all implementations will require predefinition of ResourceNames. For those implementations that do not require pre-definition, this exception will not be thrown.

NoAssociation

The NoAssociation exception is thrown by the PolicyEvaluatorAdmin interface delete_policies() operation when a an assoicaition between the ResourceName and PolicyName does not exist.

InvalidPolicy

The InvalidPolicy exception is thrown by the PolicyEvaluatorAdmin interface operations when an attempt is made to associated an Invalid PolicyName with a ResourceName or to set a default Policy that is invalid.

DuplicateEvaluatorName

The DuplicateEvaluatorName exception is thrown by the PolicyEvaluatorLocatorAdmin interface operations when an attempt is made to use those operations to add an evaluator that has the same value of its data member *evalautor_name* but different value of its data member *policy_evalautor* as some other named policy evalautor associated or to be associated (after the current operation was supposed to complete) with a resource name pattern.

InvalidResourceName

This exception is raised when the provided resource name is invalid. Please refer to the specification of type ResourceName for the description of valid and invalid datum elements of type ResourceName.

InvalidResourceNamePattern

This exceptions is thrown by corresponding operations when a resource name pattern, provided as an operation argument, has invalid syntax. Please refer to the specification of ResourceNamePattern data type for description of invalid values for ResourceNamePattern.

InvalidPolicyNameList

This exceptions is raised when the provided Policy NameList has invalid value. Please refer to the specification of PolicyNameList data type for a description of valid PolicyNameList datum elements.

first_invalid_element is first policy name in the invalid list which caused the list to be invalid. If the value of this data member is nil than the list is invalid not because of a particular element, but because of some other reason (for example, because the list is empty).

InvalidPolicyEvaluatorList

This exception is raised when a policy evalautor list, passed as a part of an operation arguments, is invalid. Please refer to the specification of PolicyEvalautorList data type for a description of invalid PolicyEvalautorLis datum elements of that type.

first_invalid_element is first named policy evalautor in the invalid list which caused the list to be invalid. If the value of this data member is nil than the list is invalid not because of a particular element, but because of some other reason (for example, because the list is empty).

3.2 *AccessDecision interface*

```
//**************************************************
//     interface AccessDecision
//**************************************************

interface AccessDecision {

    boolean access_allowed(
        in  ResourceName   resource_name,
        in  Operation      operation,
        in  AttributeList  attribute_list
    )
    raises (InternalError);

    BooleanList multiple_access_allowed(
        in  AccessDefinitionList  access_requests,
        in  AttributeList         attribute_list
    )
    raises (InternalError);

};
```

The Access Decision object is used to request decisions on access based on a ResourceName, an Operation, and a list of SecAttributes. This submission provides a framework for the support of many policy evaluators. It is out of the scope of this submission to mandate how policy is defined or evaluated using the information provided by the client at the time access decisions are requested. This is the only interface that is necessary for a client to be familiar with in order to obtain access decisions from the RAD.

The AccessDecision object sometimes passes exceptions to callers indicating that it's encountered an internal error and is not able to make an access decision. This is different from the behavior of many operating systems, which have a default-deny or a default-grant policy when an internal failure occurs, but don't report the failure to their callers. This difference arises because RAD is an access decision service, not an access control service. In all cases, the application which calls RAD is responsible for enforcing the policy decision which RAD makes. Therefore, the RAD client application is the right place to make the policy enforcement decision about what should be done when RAD is not able to make a policy decision.

`access_allowed()`

A single access decision is requested and a boolean is returned. The InternalError exception is reserved for internal logic errors and should **not** be used as a reason code for rejecting a request.. ADO clients are not exposed to the security reason for not allowing access. Indicating Fatal means that the ADO client should discontinue using the ADO.

Preconditions

1. "resource_name" is valid.
2. "operation" is valid.

Postconditions

1. return == authorization decision for the requested operation on the specified resource name by a principal with the specified security attributes.

`multiple_access_allowed()`

Multiple access decisions are requested in a single method invocation and a sequence of booleans are returned. The boolean sequence maps one to one in the same order to the provided sequence of ResourceName/operation pairs. The InternalError exception is reserved for internal logic

errors and should **not** be used as a reason code for rejecting a request. ADO clients are not exposed to the security reason for not allowing access. Indicating Fatal means that the ADO client should discontinue using the ADO

Preconditions

1. All elements of "access_request" are valid.

Postconditions

1. The length of the returned list is the same as of "access_requests" list.
2. Each element of the returned list is an authorization decision for the corresponding request in the "access_requests" list. I.e. first element of the returned list is an authorization decision for the first element of access_request", and so on. Where an authorization decision == authorization decision for the requested operation on the specified resource name by a principal with the specified security attributes.

.

3.3 *DynamicAttributeService interface*

```
//****************************************************
//     interface DynamicAttributeService
//****************************************************

interface DynamicAttributeService {

    AttributeList get_dynamic_attributes(
        in  AttributeList   attribute_list,
        in  ResourceName   resource_name,
        in  Operation      operation
    )
    raises (ComponentError);
};
```

The DynamicAttributeService interface is used to obtain a new list of SecAttributes that are applicable to an access decision. This service may encapsulate calls to a relationship service and/or application specific logic to determine how the original AttributeList provided by the client should be modified.

`get_dynamic_attributes()`

This method takes the parameters provided by the client of the AccessDecision object; the AttributeList, the ResourceName, and the operation and determines what (if any) dynamic attributes should be added to the AttributeList. In addition, the returned AttributeList may be modified by this service. The service may add or remove SecAttributes to this list. It is the returned list of SecAttributes that is used as the basis of access decisions by the RAD.

Preconditions

1. "resource_name" is valid.
2. "operation" is valid.

Postconditions

No postconditions.

3.4 PolicyEvaluatorLocator interface

```
//****************************************************
//     interface PolicyEvaluatorLocator
//****************************************************

interface PolicyEvaluatorLocator {

   readonly attribute PolicyEvaluatorLocatorAdmin pel_admin;

   PolicyDecisionEvaluators get_policy_decision_evaluators(
       in  ResourceName    resource_name
   )
   raises (ComponentError);

};
```

The PolicyEvaluatorLocator interface is used to locate the PolicyEvaluators and the DecisionCombinator associated with a ResourceName. This submission provides a framework for the support of one or more policy evaluators for a single resource.

`readonly attribute PolicyEvaluatorLocatorAdmin`

The PolicyEvaluatorLocator's administrative interface can be obtained via this attribute.

`get_policy_decision_evaluators()`

A PolicyDecisionEvaluators structure which contains a list of PolicyEvaluator object references and the DecisionCombinator object reference for the resource is returned to the client.

Preconditions

1. "resource_name" is valid.

Postconditions

1. The returned references are not nil.
2. No elements of "policy_evaluator_list" in the returned datum have same value of "evalautor_name."

3.5 DecisionCombinator interface

```
//****************************************************
//     interface DecisionCombinator
//****************************************************

interface DecisionCombinator{

    boolean combine_decisions(
        in  ResourceName   resource_name,
        in  Operation      operation,
        in  AttributeList  attribute_list,
        in  PolicyEvaluatorList policy_evaluator_list
    )
    raises (ComponentError);
};
```

The DecisionCombinator interface is used to encapsulate the policy for the way that decisions of multiple PolicyEvaluators is combined. DecisionCombinators may be simple or arbitrarily complex. A default combinator may be used for all access decisions, or combinators may be chosen specifically for access decisions on specific secured resources.

Functions consisting of a global combinator operator are easy to implement; an example of such a policy is:

AND ((Evaluator_1 = ACCESS_DECISION_ALLOWED),

(Evaluator_2 = ACCESS_DECISION_ALLOWED), ...)

This policy can be expressed as an application of a global combinator ("AND" in this case) to the results returned by ALL the PolicyEvaluator objects passed to the DecisionCombinator.

The thing which makes this kind of policy easy to implement is that it's not necessary to know anything about the result returned by any specific PolicyEvaluator object, and hence the PolicyEvaluator objects can all be treated the same and can be called in any order.

The disadvantages of this kind of policy are:

- They aren't very expressive (there are lots of kinds of real-world policies which can't be expressed using only a global combinator)
- They are inefficient. It's always necessary to call all the PolicyEvaluator objects passed to the DecisionCombinator object in order to make a decision. An important goal of the DecisionCombinator design is to support complex policies which can be efficiently evaluated. A policy like the following can't be expressed using only a global combinator, but should be implementable as a DecisionCombinator object:

(Evaluator_1 result is ACCESS_DECISION_ALLOWED) OR

((Evaluator_2 result is ACCESS_DECISION_ALLOWED) AND

(Evaluator_3 result is (ACCESS_DECISION_ALLOWED OR

ACCESS_DECISION_UNKNOWN)))

Note that this policy can be short-circuit evaluated: if the DecisionCombinator calls Evaluator_1 and it returns ACCESS_DECISION_ALLOWED as a decision result, then it doesn't need to call Evaluator_2 and Evaluator_3 at all. However, In order to support evaluation of this policy, the DecisionCombinator object needs to be able to match the PolicyEvaluator objects passed to it as input to the formal parameters in this expression. This is why the DecisionCombinator interface accepts as input a structure containing both a reference to a PolicyEvaluator object and the name of that PolicyEvaluator object; it uses the PolicyEvaluator name to figure out which evaluators to call in which order; it uses the PolicyEvaluator object's reference to call the object and request a decision result, and then it uses the PolicyEvaulator object's name again to plug the decision result into the policy combinator expression above.

`combine_decisions()`

The DecisionCombinator is responsible for determining what PolicyEvaluators (from the list passed to it) must be called and how the results are to provide a boolean result. This is the result that will be returned by the AccessDecision object to the original client of the RAD facility.

Preconditions

1. "resource_name" is valid.
2. "operation" is valid.
3. "policy_evaluator_list" is valid.

Postconditions

No postconditions.

3.6 PolicyEvaluator interface

```
//*****************************************************
//     interface PolicyEvaluator
//*****************************************************

interface PolicyEvaluator {

    readonly attribute PolicyEvaluatorAdmin pe_admin;

    DecisionResult evaluate(
        in  ResourceName   resource_name,
        in  Operation      operation,
        in  AttributeList  attribute_list
    )
    raises (ComponentError);

};
```

The PolicyEvaluator interface is used to obtain an access decision based on an encapsulated policy for the ResourceName/operation when provided a list of effective Security Attributes for the requestor. This submission provides a framework for the support of one or more policy evaluators for a single resource.

`readonly attribute PolicyEvaluatorAdmin`

If the PolicyEvaluator has an associated administrative interface, it can be obtained via this attribute. If an administrative interface is not available for this evaluator, this attribute will be nil.

`evaluate()`

A single access decision is requested based on access policy(s) this evaluator determines is appropriate for the named resource. The decision is based on the ResourceName, the operation, and the effective Security Attributes. The SecAttributes passed to the AccessDecision object by the client in access_allowed() may have been modified by the DynamicAttributeService get_dynamic_attributes() method before the PolicyEvaluator is called. The DecisionResult is a ternary result. The DecisionResult is as follows:

ACCESS_DECISION_ALLOWED: the policy evaluated for this ResourceName, operation and Attribute list indicates that access is ALLOWED.

ACCESS_DECISION_NOT_ALLOWED: the policy evaluated for this ResourceName, operation and Attribute list indicates access is NOT_ALLOWED.

ACCESS_DECISION_UNKNOWN: the policy evaluated for this ResourceName, operation and Attribute list indicates an access decision cannot be made.

Preconditions

1. "resource_name" is valid.
2. "operation" is valid.

Postconditions

1. return == authorization decision for the requested operation on the specified resource name by a principal with the specified security attributes.

3.7 AccessDecisionAdmin interface

```
//*****************************************************
//     interface AccessDecisionAdmin
//*****************************************************

interface AccessDecisionAdmin {

       PolicyEvaluatorLocator get_policy_evaluator_locator();

       void   set_policy_evaluator_locator (
           in  PolicyEvaluatorLocator policy_evaluator_locator
       );

       DynamicAttributeService get_dynamic_attribute_service();

       void   set_dynamic_attribute_service(
           in  DynamicAttributeService dynamic_attribute_service
       );
};
```

The Access Decision Admin object is provided to allow a standard mechanism for replacement of the vendor provided PolicyEvaluatorLocator and the DynamicAttributeService.

get_policy_evaluator_locator()

This operation returns the PolicyEvaluatorLocator used by the access decision object.

set_policy_evaluator_locator()

This operation sets the PolicyEvaluatorLocator used by the access decision object.

get_dynamic_attribute_service()

This operation returns the DynamicAttributeService used by the access decision object.

set_dynamic_attribute_service()

This operation sets the DynamicAttributeService used by the access decision object.

3.8 PolicyEvaluatorLocatorAdmin interface

```
//*****************************************************
//     interface PolicyEvaluatorLocatorAdmin
//*****************************************************

interface PolicyEvaluatorLocatorAdmin {

    void register_resource_name_pattern(
        in  ResourceNamePattern pattern
    )
    raises (InvalidResourceNamePattern,
            PatternDuplicate,
            PatternConflict);

    void unregister_resource_name_pattern(
        in  ResourceNamePattern pattern
```

```
    )
    raises (InvalidResourceNamePattern,
            PatternNotRegistered,
            PatternInUse);

    PolicyEvaluatorList get_evaluators(
        in  ResourceNamePattern pattern
    )
    raises (InvalidResourceNamePattern,
            PatternNotRegistered);

    void set_evaluators (
        in  PolicyEvaluatorList policy_evaluator_list,
        in      ResourceNamePattern pattern
    )
    raises (InputFormatError,
            PatternNotRegistered,
            DuplicateEvaluatorName);

    PolicyEvaluatorList set_default_evaluators(
        in  PolicyEvaluatorList policy_evaluator_list
    )
    raises (DuplicateEvaluatorName, InvalidPolicyEvaluatorList);

    void add_evaluators (
        in  PolicyEvaluatorList policy_evaluator_list,
        in  ResourceNamePattern pattern
    )
    raises (InvalidResourceNamePattern,
            PatternNotRegistered,
            InvalidPolicyEvaluatorList,
            DuplicateEvaluatorName);

    void delete_evaluators (
        in  PolicyEvaluatorList policy_evaluator_list,
        in  ResourceNamePattern pattern
    )
    raises (InvalidResourceNamePattern,
            PatternNotRegistered,
            InvalidPolicyEvaluatorList,
            DuplicateEvaluatorName);

    DecisionCombinator get_combinator (
        in  ResourceNamePattern pattern
    )
    raises (InvalidResourceNamePattern,
            PatternNotRegistered);

    void set_combinator (
        in  DecisionCombinator decision_combinator,
        in  ResourceNamePattern pattern
    )
    raises (InvalidResourceNamePattern,
            PatternNotRegistered);

    void delete_combinator (
        in  ResourceNamePattern pattern
    )
    raises (InvalidResourceNamePattern,
            PatternNotRegistered);

    DecisionCombinator get_default_combinator ();

    void set_default_combinator(
        in  DecisionCombinator decision_combinator
    );


};
```

The PolicyEvaluatorLocatorAdmin object is used to associate PolicyEvaluators with a ResourceName. The object is also used to associate the appropriate DecisionCombinator with the ResourceName. This submission provides a framework for the support of one or more policy evaluators for a single resource.

Patterns are used to group resources without requiring the PolicyEvaluatorLocator administrator to enumerate all the resources individually; this is accomplished by associating lists of PolicyEvaluator objects with ResourceNamePatterns, and checking whether a supplied resource name matches any of the Patterns with which it has associated PolicyEvaluators. This section describes how RAD objects decide whether a Pattern matches a resource name. Throughout the section, we use the shorthand phrase "exactly matches" to mean "is exactly the same string as". Patterns have a specific format:

- A Pattern must include a ResourceNamingAuthority.
- A Pattern must include a list of ResourceNameComponent strings.
- Each ResourceNameComponent consists of a name_string and a value_string.

A resource name matches a Pattern only if:

- the resource name's ResourceNamingAuthority exactly matches the Pattern's ResourceNamingAuthority AND
- each component of the resource name matches some component of the Pattern AND
- no component of the resource name conflicts with any component of the Pattern

Two kinds of ResourceNameComponents can occur in a pattern.

The first kind is a component value pattern. It has the form:

- name_string is a string
- value_string is a regular expression

A resource name component matches a component value pattern only if its

- name_string exactly matches the pattern's name_string and its value_string matches the component value pattern's value_string regular expression.

A resource name component conflicts with a component value pattern if its

- name_string exactly matches the pattern's name_string, but its value_string does not match the component value pattern's value_string regular expression.

(2) The second kind of ResourceNameComponent which can occur in a pattern is a component wildcard pattern:

- name_string is "*"
- value_string is "*"

Every component of a resource name matches a component wildcard pattern. No component of a resource name conflicts with a component wildcard pattern.

More formally, Given

name : ResourceName

pattern : ResourceNamePattern

The following algorithm determines whether the name matches the pattern:

if (

name.resource_naming_authority is not the same string as pattern.resource_naming_authority

) // name and pattern have different authorities

OR

(pattern.resource_name_component_list includes a component &pattern_component AND

name.resource_name_component_list includes a component &name_component AND

&name_component.name_string is the same string as &pattern_component.name_string AND

&name_component.value_string does not match the regular expression &pattern_component.value_string

) // resource name component conflicts with a pattern component

OR

(name.resource_name_component_list includes a component &name_component AND

name_component.name_string is &name AND

pattern.resource_name_component_list does not include a component <"*","*"> AND

pattern.resource_name_component_list does not include any component whose .name_string the same string as &name

) // resource name contains a component which does not match any pattern component

then

name does not match pattern

else

name matches pattern

`register_resource_name_pattern()`

Before a ResourceNamePattern can be used in the administrative interfaces, it must be registered. This allows the administration of name patterns separately from the administration of the association of patterns to evaluators and combinators. Since a ResourceName is a ResourceNamePattern, ResourceNames must also be registered if these administrative interfaces are administer evaluators and combinators.

Implementations may or may not support overlapping patterns; that is, an implementation may choose to allow registration of two patterns both of which match at least one name, or they may choose not to allow such registrations. An implementation which does not support overlapping patterns shall raise the PatternConflict exception when this method is used to register a pattern which overlaps with another previously registered pattern. Implementors should document whether their implementations support overlapping patterns or not.

Preconditions

No preconditions.

Postconditions

1. "resource_name_pattern" is registered.

`unregister_resource_name_pattern()`

ResourceNamePatterns may be unregistered. A ResourceNamePattern must not have any evaluators or combinators associated with it when it is unregistered.

Preconditions

No preconditions.

Postconditions

1. "resource_name_pattern" is unregistered.

`get_evaluators()`

The list of PolicyEvaluators associated with the ResourceNamePattern is returned.

Preconditions

No preconditions.

Postconditions

1. return == "resource_name_pattern".registered_ evalautor_list

`set_evaluators()`

A list of PolicyEvaluators is assigned for the named resource. If the resource had existing PolicyEvaluators assigned, they are removed and the entire list is replaced with the ones provided in this method. The replacement of evaluators for a resource which previously had none results in the added list of evaluators being the only evaluators consulted on an access decision (system default evaluators are no longer consulted unless a system default evaluator is a member of the replacement list).

These evaluators will be the PolicyEvaluators returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method.

Preconditions

No preconditions.

Postconditions

1. “resource_name_pattern”.registered_evaluator_list == policy_evaluator_list

`set_default_evaluators()`

The list of PolicyEvaluators provided is set as the default evaluators for any ResourceName for which PolicyEvaluators have not been explicitly assigned. Default evaluators are overridden by the add_evaluators() or replace_evaluators() methods.   The default evaluators will be returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method when no PolicyEvaluators have been explicitly assigned for a ResourceName.

Preconditions

No preconditions.

Postconditions

1. default_evaluators == “policy_evaluator_list”

`add_evaluators()`

A list of PolicyEvaluators is added to the list of evaluators for the named resource. These evaluators will be in the list of PolicyEvaluators returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method. The addition of evaluators to a ResourceName which previously had none results in the added list of evaluators being the only evaluators consulted on an access decision (system default evaluators are no longer consulted unless a system default evaluator is a member of the added list).

Preconditions

No preconditions.

Postconditions

1. "resource_name_pattern".registered_evaluator_list == union (policy_evaluator_list, "resource_name_ pattern".registered_evaluator_list)

`delete_evaluators()`

The list of PolicyEvaluators is removed from the list of evaluators for the named resource. These evaluators will not be in the list of PolicyEvaluators returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method.

Preconditions

No preconditions.

Postconditions

1. for the "resource_name_pattern" : "resource_name_ pattern".registered_evaluator_list = "resource_name_ pattern".registered_evaluators - "policy_evaluator_list"

`get_combinator()`

The DecisionCombinator specified for the named resource is returned. If a combinator has not been specified for the ResourceNamePattern provided, the return will be nil (it will not return the default combinator).

Preconditions

No preconditions.

Postconditions

1. return == "resource_name_pattern".registered_ decision_combinator

`set_combinator()`

A DecisionCombinator is specified for the named resource. This combinator will be returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method. The DecisionCombinator provided replaces any previous combinator specified for the secured resource.

Preconditions

No preconditions.

Postconditions

1. "resource_name_pattern".registered_ decision_combinator == “decision_combinator”

delete_combinator()

The DecisionCombinator for the ResourceNamePattern is removed. The default combinator will now be returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method.

Preconditions

No preconditions.

Postconditions

1. Resource names matching only "resource_name_ pattern" will be associated with the efault combinator.

get_default_combinator()

The DecisionCombinator provided is returned.

Preconditions

No preconditions.

Postconditions

1. return == default_combinator.

set_default_combinator()

The DecisionCombinator provided is set as a default. This combinator is now the combinator used when a DecisionCombinator has not been explicitly specified for a secured resource. This combinator will be returned by the PolicyEvaluatorLocator get_policy_decision_evaluators() method for these resources.

Preconditions

No preconditions.

Postconditions

1. default_combinator == “decision_combinator”.

3.9 PolicyEvaluatorAdmin interface

```
//*******************************************************
//      interface PolicyEvaluatorAdmin
//*******************************************************

interface PolicyEvaluatorAdmin {

    void    set_policies(
        in  PolicyNameList policy_names,
        in  ResourceName   resource_name
    )
```

```
    raises (InvalidResourceName,
            ResourceNameNotFound,
            InvalidPolicyNameList);

    void    add_policies(
        in  PolicyNameList policy_names,
        in  ResourceName   resource_name
    )
    raises (InvalidResourceName,
            ResourceNameNotFound,
            InvalidPolicyNameList);

    void    delete_policies(
        in  PolicyNameList policy_names,
        in  ResourceName   resource_name
    )
    raises (InvalidResourceName,
            ResourceNameNotFound,
            InvalidPolicyNameList,
            NoAssociation);

    PolicyNameList list_policies();

    PolicyName set_default_policy(
        in  PolicyName  policy_names
    )
    raises (InvalidPolicy);
};
```

The PolicyEvaluatorAdmin interface is used to associate named access policies with secured resources. It is assumed that the administrative tool used to create and manage access policies (outside the scope of this submission) provides a mechanism to allow policies to be associated with "names" which are represented as PolicyName (a string). This PolicyEvaluatorAdmin interface allows those policies to be applied "by name" to a secured resource represented by a ResourceName.

This interface is primarily provided for the application that wishes to assign a policy to a newly created resource programatically at the time of resource creation. It does, however, require that the application have knowledge of the named policies in order to choose an appropriate policy for access decisions.

`set_policies()`

The policies identified by PolicyNameList is associated with the secured resource identified by the ResourceName. If a single PolicyName of NO_ACCESS_POLICY is specified, then all policy is removed for the resource. If a PolicyNameList is applied to a ResourceName that has existing policy, then the policy will be replaced by the policy identified by this PolicyNameList.

Preconditions

No preconditions.

Postconditions

1. "resource_name".applied_policie_names == "policy_names".

`add_policies()`

The policy identified by PolicyNameList is added to the list of policies used when making access decisions for the secured resource identified by the ResourceName. If a PolicyNameList is added to a resource that has existing policy, then the policy will be added to the list of policies that

control access decisions for the resource. An implementation is not required to support multiple policies for a resource. If the implementation does not support the application of multiple policies, then a InvalidPolicy exception shall be thrown for this method.

Preconditions

No preconditions.

Postconditions

1. "resource_name".applied_policy_names == union ("resource_name".applied_policy_names, "policy_names")

`list_policies()`

A list of all existing PolicyNames is returned to the client.

Preconditions

No preconditions.

Postconditions

1. return == all_existing_policy_names.

`set_default_policy()`

The policy identified by PolicyName is associated (as default) with any secured resource which has not yet been assigned an access policy.

Preconditions

No preconditions.

Postconditions

The order is significant.

1. return == default_policy_name
2. default_policy_name == "policy_name"

.

4. Conformance Classes

A conformant Resource Access Decision (RAD) facility must implement all of the interfaces defined in this submission.

5. Appendix - Use Case Example

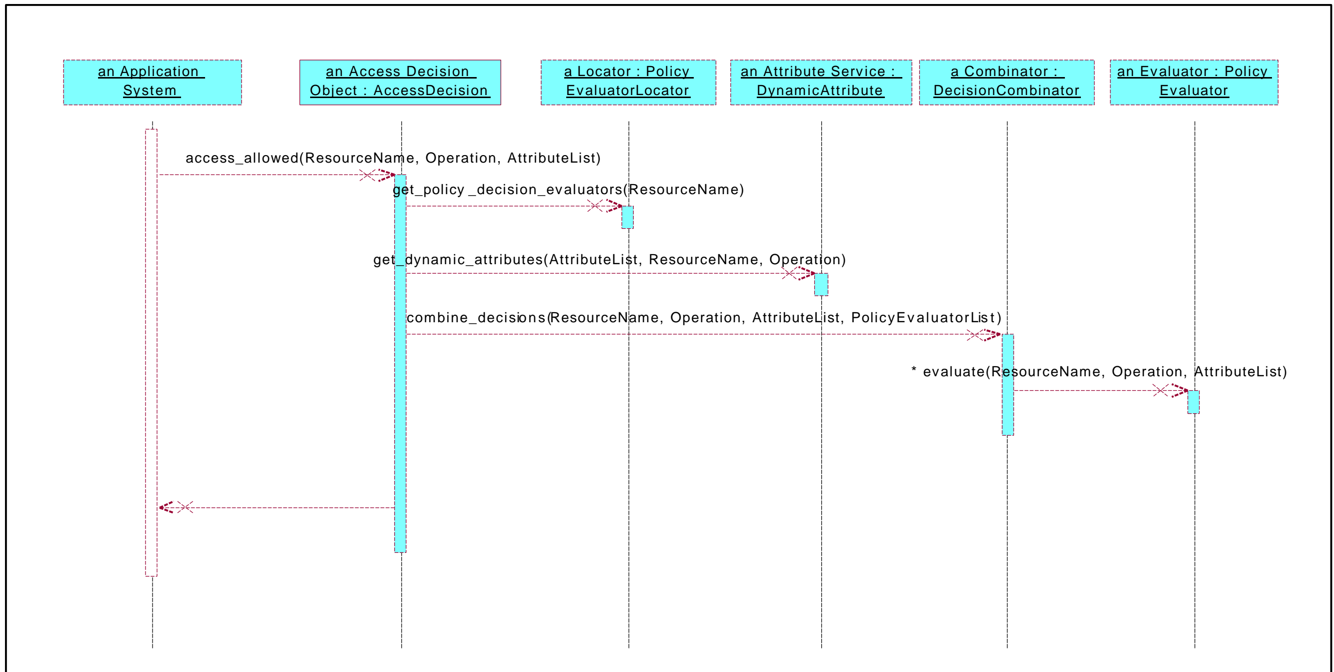
This appendix presents an example illustrating a healthcare scenario and the use of RAD to provide access control for the instances of healthcare information access implied by this scenario. The example consists of:

1. A description of the healthcare scenario which involves one or more accesses to healthcare information.
2. For each healthcare information access required by the scenario:
   A. A description of the actions of the healthcare application, the client of the Access Decision Object (ADO).
   B. A description of ADO actions with an Object Interaction Diagram (OID).

Before presenting the Use Case, a generic OID describing the ADO is provided.

5.1 Generic RAD Sequence Diagram

This section shows the generic sequence diagram for the RAD.



5.2 Healthcare Scenario: Out-patient Visit to Attending Physician

This scenario (see table 1) illustrates the interaction with a patient record as a result of a patient's visit with an attending physician at the hospital on an outpatient basis. In this example, the access control policy pertinent to this scenario is called the "Basic Hospital Patient Record Access Policy."

As described in more detail in the normative part of this document, an access control policy within RAD is realized by an evaluator applied to static attributes, dynamic attributes, and other factors, such as, time of day and location of the principal. An evaluator can be implemented as an interpreter of rules expressed in some scripting language, e.g., SQL, as a process for which the rules are encapsulated as part of the process, e.g., Java Classes, or as some combination of these methods.

Static attributes are used for describing relatively fixed properties of users and resources, such as, basic user role and resource creation date. The values of static attributes are typically set by a security administrator and are obtained by the application in an implementation specific manner, e.g., from the principal's credentials. While the use of a static attribute in policy is specified by a security administrator, the values of dynamic attributes are typically set as part of normal information processing. Unlike static attributes which are usually properties of (i.e.,metadata about) information content, values of dynamic attributes are information content which are necessary to make an access decision. Some examples of dynamic attributes, which may be contained in a patient record or elsewhere, are:

- A list of physicians, i.e., attending physicians, which are currently treating the patient.
- An authorization permitting the release of mental health information to designated parties.

Depending on the implementation, a dynamic attribute may be the value of the dynamic attribute or a reference to the value of the dynamic attribute. If a reference, then the dynamic attribute value is obtained by the evaluator if and when the evaluator determines that the value is needed to make the access decision.

RAD is able to support more than one access policy. This healthcare scenario describes RAD functionality using the Basic Hospital Patient Record Access Policy. Different developers may implement different access policy evaluators. Dynamic attributes may be associated with only one or several evaluators. New dynamic attributes may be added to the Dynamic Attribute Service of an RAD when new evaluators are installed. Once dynamic attributes are added to the Dynamic Attribute Service, they may be available for use by all evaluators. In addition to the Basic Hospital Patient Record Access Policy, other policies may specify access control requirements for HIV or mental health information resources which are part of the patient record.

The Basic Hospital Patient Record Access Policy used in this example specifies the conditions under which an attending physician can access a patient record. The policy specifies that attending physicians may read/update a patient record and/or modify certain authorization settings in a patient record. Within this policy, the term "update" when applied to clinical information refers to an append operation. Clinical information in the patient record once entered may not be modified.

Several static and dynamic attributes are used by the RAD evaluator which implements the Basic Hospital Patient Record Access Policy. Among these are the static attribute "role" and the dynamic attribute "principal/patient_relationship." The value of the static attribute role specifies the basic role of a user, such as, physician, nurse, and registrar. In this example, the value of role is obtained from the principal's credentials. The value of the dynamic attribute principal/patient_relationship specifies the relationship between the principal accessing the patient record and the patient who is the subject of the patient record being accessed, e.g., "primary_care," "attending," "consulting." In this example, the value of the principal/patient_relationship dynamic attribute is obtained by the Dynamic Attribute Service by accessing the content of the patient record which contains a list of attending physicians.

Use Case Name	Out-patent Hospital Visit to Attending Physician
Goal in Context	Physician provides care to a visiting patient
Scope & Level	Summary
Preconditions	Patient records already exist in the system, there is already some kind of relationship between the patient and the physician (attending, consulting, admitting, etc.)
Success End Condition	Patient records are updated according to the visit results.
Failed End Condition	Patient records are not updated according to the visit results.
Primary Actors	Care providing physician
Secondary Actors	
Trigger	Patient visits corresponding physician.
Applicable Access Policy	Basic Hospital Patient Record Access
Diagram	Physician Log Into the System Read Patient Records Examine Patient Update Patient Records

Description	Step	Action

	1	Physician (or physician representative) logs into the information system unless it was done previously.
	2	Physician retrieves patient records and browses them.
	3	Physician examines the patient.
	4	Physician updates patient records.
Extensions	Step	Branching Action
	4 a	Physician changes authorization settings for the patient records (or their sub-set) according to the patient request and/or sensitivity of the information with which records are updated.
Variations	Step	Branching Action
		No variations
Related Information		
Priority		High
Performance		1 hour
Frequency		Many times per hour through the hospital
Channels to actors		Vision, speech, various instruments and devices in order to examine the patient; computer GUI to log into the system, brows and update patient records.
Open Issues		What authorization settings of the patient records can a related physician change?
		What if another related physician has limited access to records that are interesting in the context of the visit and the patient agrees those records can be disclosed?
Superordinate use cases		No superordinates
Subordinate use cases		Log into the system, Read Patient Records, Examine Patient, Update Patient Records, Change Authorization Settings for the Patient Record(s).

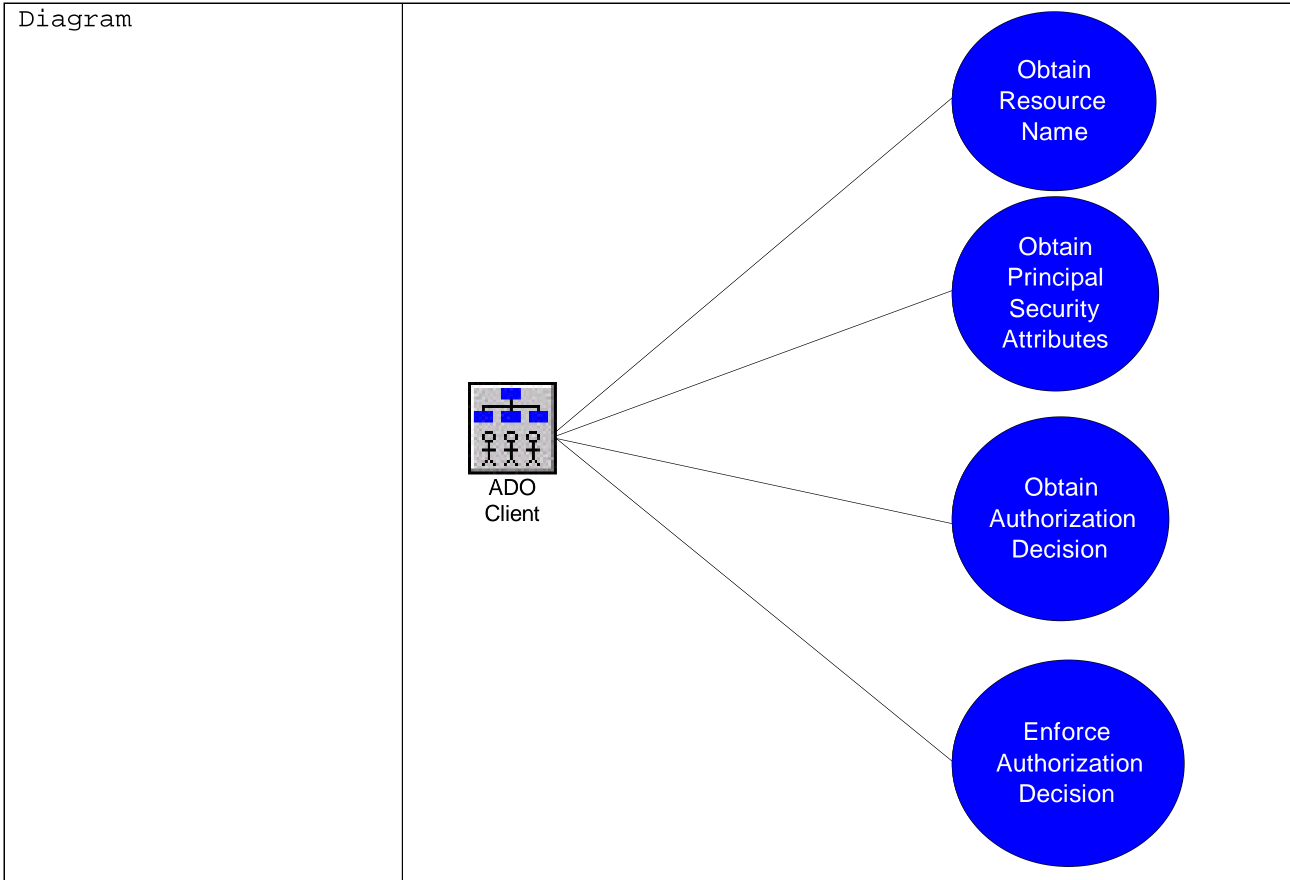
Table 1: Healthcare Scenario: Out-patient Visit to Attending Physician

As shown in table 1, there are three types of access to the patient record involved in this scenario: read, update, and change authorization.

The next section describes the actions of the application program (the ADO client) in reading the patient record including how the ADO is used to determine access according to the Basic Hospital Patient Record Access Policy.

5.2.1 ADO Client Actions: Read Patient Record

Use Case Name	ADO Client Actions: Read Patient Record
Goal in Context	Application program (ADO client) browses patient record.
Scope & Level	Subfunction
Preconditions	Patient records already exist in the system; physician has logged into application program; application program initiated successfully.
Success End Condition	The intended part of patient records are "read" accessed by the caregiver.
Failed End Condition	The intended part of patient records are not "read" accessed by the caregiver.
Primary Actors	1. Client program acting on behalf of the caregiver (Client) 2. CORBA-compliant application service (Service), which provides "read" access to the required information
Secondary Actors	1. Access Decision Object (ADO), which provides interface DfResourceAccessDecision::AccessDecision
Trigger	A caregiver is attempting to "browse" parts of the patient medical record.
Applicable Access Policy	Basic Hospital Patient Record Access: An attending physician may read any part of the patient record.

Diagram		
Description	Step	Action
	1	Application program (ADO client), acting on behalf of the physician, obtains the resource_name for the part of the patient record to be read and the static attribute_list.
	2	ADO client invokes access_allowed(resource_name, "read", attribute_list).
	3	If access_allowed() returns "true," then ADO client reads and displays requested part of the patient record to physician; otherwise, ADO Client displays error.
Extensions	Step	Branching Action
		No variations
Variations	Step	Branching Action
		No variations
Related Information		
Priority		High
Performance		
Frequency		Many times per hour through the hospital
Channels to actors		
Open Issues		
Superordinate use cases		Out-patent Visit to Attending Physician

Subordinate use cases	ADO Actions: Read Patient Record

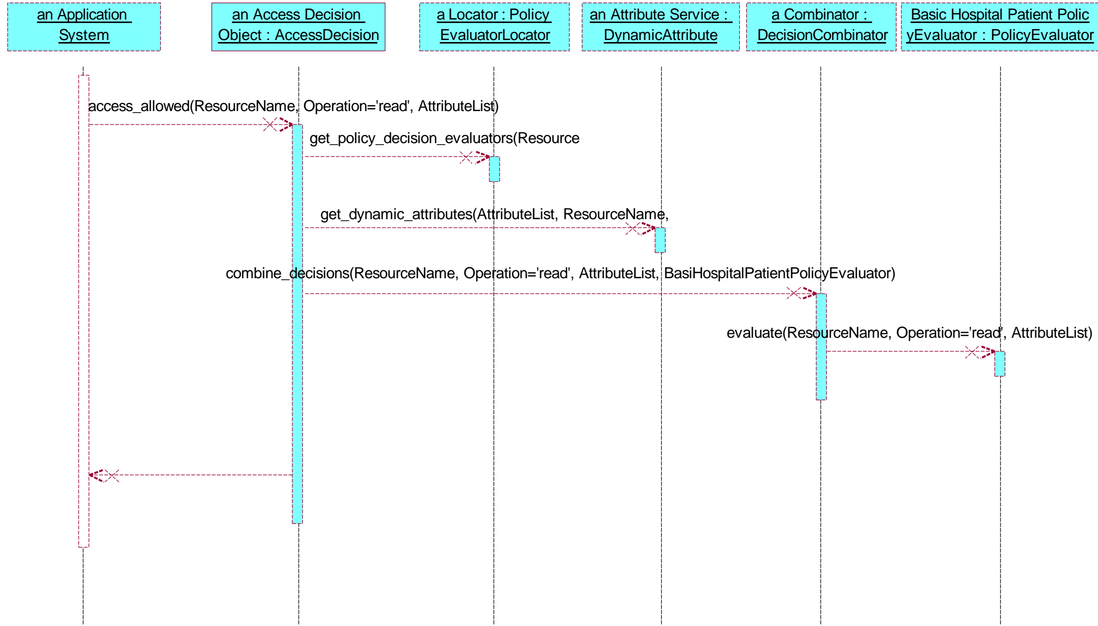
Table 2: ADO Client Actions: Read Patient Record

Table 2 describes the actions of the application program (ADO client) in providing the physician the capability of browsing resources contained in the patient record. The application program obtains from the physician the name of the resource to be read. It then obtains the static attributes from the physician's credentials. The application invokes the ADO which returns an indication of whether the physician is able to read the requested resource within the patient record. If the physician has read access to the resource, the application displays the resource for the physician.

The next section describes the actions of the ADO when it is invoked by the application to determine if the physician has read access to the patient record resource.

5.2.2 ADO Actions: Read Patient Record

Use Case Name	ADO Actions: Read Patient Record
Goal in Context	ADO renders access decision for a resource which is part of the patient record.
Scope & Level	Subfunction
Preconditions	Patient records already exist in the system; Application program has invoked ADO.
Success End Condition	An access decision is returned by the ADO to the application program.
Failed End Condition	An exception occurred and an access decision is not returned by the ADO to the application program.
Primary Actors	1. Access Decision Object (ADO), which provides interface **DfResourceAccessDecision::AccessDecision**
Secondary Actors	1. Policy Locator Object(PL), which provides the interface **DfResourceAccessDecision::PolicyEvaluatorLocator** 2. Dynamic Attribute Service Object(DAS), which provides interface **DfResourceAccessDecision::DynamicAttributeService** 3. Policy Evaluator Object (PE), which provides the interface **DfResourceAccessDecision::PolicyEvaluator** 4. Decision Combinator Object(DCO), which provides the interface **DfResourceAccessDecision::DecisionCombinator**
Trigger	Application program (ADO client) invokes ADO.
Applicable Access Policy	Basic Hospital Patient Record Access: An attending physician may read any part of the patient record

<table>
<tr><td colspan="3">Diagram

</td></tr>
<tr><td>Description</td><td>Step</td><td>Action</td></tr>
<tr><td></td><td>1</td><td>ADO invokes get_policy_decision_evaluators(resource_name) which returns:
1. policy_evaluator_list that contains only one item: the NamedPolicyEvaluator consisting of the evaluator_name "Basic Hospital Patient Record Access Policy" and its object reference policy_evaluator.
2. A decision_combinator.</td></tr>
<tr><td></td><td>2</td><td>Using the static attribute_list provided by the ADO client, ADO invokes get_dynamic_attributes(attribute_list, resource_name, "read") which returns attribute_list', a list of all static and dynamic attributes required for policy_evaluator to make the access decision.</td></tr>
<tr><td></td><td>3</td><td>ADO invokes combine_decisions(resource_name, "read", attribute_list', policy_evaluator_list). Within combine_decisions(), the policy_evaluator with evaluator_name "Basic Hospital Patient Record Access Policy" is invoked returning "ACCESS_DECISION_ALLOWED". combine_decisions()returns "TRUE" to the ADO.</td></tr>
<tr><td></td><td>4</td><td>ADO returns the boolean result "TRUE".</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>Extensions</td><td>Step</td><td>Branching Action</td></tr>
<tr><td></td><td></td><td>No variations</td></tr>
<tr><td>Variations</td><td>Step</td><td>Branching Action</td></tr>
<tr><td></td><td></td><td>No variations</td></tr>
<tr><td colspan="2">Related Information</td><td></td></tr>
<tr><td colspan="2">Priority</td><td>High</td></tr>
</table>

Performance	
Frequency	Many times per hour through the hospital
Channels to actors	
Open Issues	
Superordinate use cases	ADO Client Actions: Read Patient Record
Subordinate use cases	

Table 3: ADO Actions: Read Patient Record

Table 3 describes the actions of the ADO in providing an access decision when invoked by the application in order to determine if the physician has the capability of browsing resources contained in the patient record. Given **resource_name**, a resource within the patient record, the operation "**read**", and **attribute_list**, a list of static attributes which contains the static role attribute **"physician"**, the ADO invokes **get_policy_decision_evaluators()** with the **resource_name** which returns:

1. **policy_evaluator_list** that contains only one item: the **NamedPolicyEvaluator** consisting of the **evaluator_name** "Basic Hospital Patient Record Access Policy" and its object reference **policy_evaluator**.
2. A **decision_combinator**.

The ADO obtains dynamic attributes by invoking **get_dynamic_attributes()** with the static attribute_list provided by the ADO client, **resource_name**, and the operation "**read**." Upon return, a combined list of static and dynamic attributes, consisting of the static role attribute **"physician"** and the dynamic relationship attribute **"attending"**, is now contained in **attribute_list'**.

The ADO then invokes **combine_decisions()** with **resource_name,** the operation **"read",** the combined list of static and dynamic attributes **attribute_list',** and **policy_evaluator_list**. Within **combine_decisions()**, the **policy_evaluator** with **evaluator_name** "Basic Hospital Patient Record Access Policy" is invoked returning **"ACCESS_DECISION_ALLOWED"** since the principal has both the static role attribute **"physician"** and the dynamic relationship attribute **"attending"**. Having invoked all evaluators in **policy_evaluator_list**, **combine_decisions()** returns **"TRUE"** to the ADO.

Finally, the ADO returns **"TRUE"** to the ADO client.

6. Appendix – Resource Names for PIDS

This section describes corresponding changes to Person Identification Service Specification (PIDS) (corbamed/98-02-29) in order for PIDS-compliant services to use RAD in a standard way.

6.1 Changes to Conformance Classes

The submission requires to add a new conformance class ‘PIDS using RAD’ in the list of conformance classes by appending the following bullet item after the last bullet item on page 63:

- “‘PIDS using RAD’ – An implementation of PIDS is conformant to this class if it is conformant to any of the above conformance classes and, in addition, it obtains from Resource Access Decision facility and enforces authorization decisions according to the description provided in section 11.8 of this specification.” New Section in Security Guidelines

The submission requires to add a new section (11.8) titled “Use of Resource Access Decision Facility” with the following text:

“Resource names used for obtaining access decisions from RAD facility by PIDS-compliant services, should be created in a predefined manner:

PIDS_RAD_Resource_Name ::= ‘IDL:omg.org/PersonIdService’ +
{"QualifiedPersonId.domain", <QualifiedPersonId.domain>} +
{"QualifiedPersonId.id", <QualifiedPersonId.id>}+
(, {"TraitName", TraitName})$^{+}$

Text below explains the expression above in English.

If a PIDS-compliant service uses Resource Access Decision facility (RAD), it shall:

- create RAD resource names according to the following rules:
  1. “resource_naming_authority” data member of ResourceName shall adhere to the syntax of NamingAuthority::AuthorityIdStr type. For the corresponding datum element of type AuthorityId, the value of *authority* shall be ‘IDL’. The value of *naming_entity* shall be ‘omg.org/PersonIdService’.
  2. First element of ResourceName data member *resource_name_component_list* is mandatory. It shall have value of *name_string* ‘QualifiedPersonId.domain’, and the value of *value_string* shall be the value of *domain* data member of the corresponding datum element of type QualifiedPersonId for the person whose traits are to be accessed.
  3. Second element of ResourceName data member *resource_name_component_list* is mandatory. It shall have value of *name_string* ‘QualifiedPersonId.id’, and the value of *value_string* shall be the value of *id* data member of the corresponding datum element of type QualifiedPersonId for the person whose traits are to be accessed.
  4. Third element of ResourceName data member *resource_name_component_list* is mandatory. It shall have value of *name_string* ‘TraitName’. The value of the corresponding *name_string*

data members shall be the name of the trait to be accessed and it shall adhere to the syntax of PersonIdService::TraitName data type.

5. All other elements of ResourceName data member *resource_name_component_list* are optional. They shall have value of *name_string* 'TraitName'. The value of the corresponding *name_string* data members shall be the name of the trait to be accessed and it shall adhere to the syntax of PersonIdService::TraitName data type.

- Create RAD operation name according to the following rules:

  1. When serving invocations of operations that semantically mean "get", *operation* in DfResourceAccessDecision::access_allowed() shall have value 'read'.

  2. When serving invocations of operations that semantically mean "set" or "register", *operation* in DfResourceAccessDecision::access_allowed() shall have value 'write'.

- Obtain security attributes of the invoking principal.

- Obtain resource access decision(s) by invoking either access_allowed() or multiple_access_allowed() on DfResourceAccessDecision::AccessDecision interface.

- Enforce the decision according to the semantics of the operation the PIDS-compliant service is serving.

- It is not mandated by this specification how exceptions caught during an attempt to invoke either access_allowed() or multiple_access_allowed() on DfResourceAccessDecision::AccessDecision interface are handled by PIDS-compliant service."

7. Appendix - Complete IDL

```
//File: DfResourceAccessDecision.idl
//

#ifndef _DF_RESOURCE_ACCESS_DECISION_IDL_
#define _DF_RESOURCE_ACCESS_DECISION_IDL_

#include "Security.idl"

#pragma prefix "omg.org"

module DfResourceAccessDecision {

//*******************************************************
//         Basic Types
//*******************************************************

typedef sequence<boolean> BooleanList;

typedef Security::AttributeList AttributeList;

interface DynamicAttributeService;
interface DecisionCombinator;
interface PolicyEvaluator;
interface PolicyEvaluatorLocator;
interface PolicyEvaluatorLocatorAdmin;
interface PolicyEvaluatorAdmin;



//*******************************************************
//   Types that identify a secured resource
//*******************************************************

struct ResourceNameComponent {
   string     name_string;
   string     value_string;
};
typedef sequence<ResourceNameComponent> ResourceNameComponentList;

typedef string ResourceNamingAuthority;

struct ResourceName {
   ResourceNamingAuthority       resource_naming_authority;
   ResourceNameComponentList resource_name_component_list;
};

typedef ResourceName      ResourceNamePattern;

typedef string            Operation;
typedef sequence<Operation> OperationList;

//**************************************************
//   Types associated with evaluating Access Policy
//**************************************************
typedef string                PolicyName;
typedef sequence<PolicyName>  PolicyNameList;

const PolicyName NO_ACCESS_POLICY = "NO_ACCESS_POLICY";

struct NamedPolicyEvaluator {
       string         evaluator_name;
       PolicyEvaluatorpolicy_evaluator;
};
typedef sequence<NamedPolicyEvaluator> PolicyEvaluatorList;
```

```
struct PolicyDecisionEvaluators {
   PolicyEvaluatorList   policy_evaluator_list;
   DecisionCombinator decision_combinator;
};

//****************************************************
//      Types used to request an Access Decision
//****************************************************

struct AccessDefinition {
   ResourceName   resource_name;
   Operation      operation;
};
typedef sequence<AccessDefinition> AccessDefinitionList;

enum DecisionResult {ACCESS_DECISION_ALLOWED,
                    ACCESS_DECISION_NOT_ALLOWED,
                    ACCESS_DECISION_UNKNOWN
};


//********************************************************
//*             Exception Data types
//********************************************************
struct ExceptionData {
   short  error_code;
   string reason;
};
enum InternalErrorType {Fatal, NotFatal};

//*********************************************************
//    Exception thrown by the Access Decision Object
//*********************************************************

exception InternalError{InternalErrorType ed;};

//*********************************************************
//    Exception thrown by Internal non-admin interfaces
//*********************************************************

exception ComponentError{
   ExceptionData ed;
   InternalErrorType it;
};

//*********************************************************
//         Exceptions thrown by Admin Interfaces
//*********************************************************

exception PatternConflict {ExceptionData ed;};
exception PatternDuplicate {ExceptionData ed;};
exception PatternNotRegistered {ExceptionData ed;};
exception PatternInUse {ExceptionData ed;};
exception InputFormatError {ExceptionData ed;};
exception ResourceNameNotFound {ExceptionData ed;};
exception NoAssociation {ExceptionData ed;};
exception InvalidPolicy {ExceptionData ed;};
exception DuplicateEvaluatorName {ExceptionData ed;};
exception InvalidResourceName {};
exception InvalidResourceNamePattern {};

exception InvalidPolicyEvaluatorList {
   ExceptionData         ed;
   NamedPolicyEvaluator  first_invalid_element;
};

exception InvalidPolicyNameList {
   ExceptionData  ed;
   PolicyName     first_invalid_element;
};
```

```
//****************************************************
//     interface AccessDecision
//****************************************************

interface AccessDecision {

   boolean access_allowed(
       in  ResourceName   resource_name,
       in  Operation      operation,
       in  AttributeList  attribute_list
   )
   raises (InternalError);

   BooleanList multiple_access_allowed(
       in  AccessDefinitionList  access_requests,
       in  AttributeList         attribute_list
   )
   raises (InternalError);

};

//*****************************************************
//     interface DynamicAttributeService
//*****************************************************

interface DynamicAttributeService {

   AttributeList get_dynamic_attributes(
       in  AttributeList   attribute_list,
       in  ResourceName    resource_name,
       in  Operation       operation
   )
   raises (ComponentError);
};

//*****************************************************
//     interface PolicyEvaluatorLocator
//*****************************************************

interface PolicyEvaluatorLocator {

   readonly attribute PolicyEvaluatorLocatorAdmin pel_admin;

   PolicyDecisionEvaluators get_policy_decision_evaluators(
       in  ResourceName    resource_name
   )
   raises (ComponentError);

};

//******************************************************
//     interface DecisionCombinator
//******************************************************

interface DecisionCombinator{

   boolean combine_decisions(
       in  ResourceName   resource_name,
       in  Operation      operation,
       in  AttributeList  attribute_list,
       in  PolicyEvaluatorList policy_evaluator_list
   )
   raises (ComponentError);
};

//*****************************************************
//     interface PolicyEvaluator
//*****************************************************

interface PolicyEvaluator {
```

```
   readonly attribute PolicyEvaluatorAdmin pe_admin;

   DecisionResult evaluate(
       in  ResourceName   resource_name,
       in  Operation      operation,
       in  AttributeList  attribute_list
   )
   raises (ComponentError);

};
//****************************************************
//
//            Management Interfaces
//
//****************************************************
//     interface AccessDecisionAdmin
//****************************************************

interface AccessDecisionAdmin {

       PolicyEvaluatorLocator get_policy_evaluator_locator();

       void    set_policy_evaluator_locator (
           in  PolicyEvaluatorLocator policy_evaluator_locator
       );

       DynamicAttributeService get_dynamic_attribute_service();

       void    set_dynamic_attribute_service(
           in  DynamicAttributeService dynamic_attribute_service
       );
};

//*****************************************************
//     interface PolicyEvaluatorLocatorAdmin
//*****************************************************

interface PolicyEvaluatorLocatorAdmin {

   void register_resource_name_pattern(
       in  ResourceNamePattern pattern
   )
   raises (InvalidResourceNamePattern,
           PatternDuplicate,
           PatternConflict);

   void unregister_resource_name_pattern(
       in  ResourceNamePattern pattern
   )
   raises (InvalidResourceNamePattern,
           PatternNotRegistered,
           PatternInUse);

   PolicyEvaluatorList get_evaluators(
       in  ResourceNamePattern pattern
   )
   raises (InvalidResourceNamePattern,
           PatternNotRegistered);

   void set_evaluators (
       in  PolicyEvaluatorList policy_evaluator_list,
       in      ResourceNamePattern pattern
   )
   raises (InputFormatError,
           PatternNotRegistered,
           DuplicateEvaluatorName);

   PolicyEvaluatorList set_default_evaluators(
       in  PolicyEvaluatorList policy_evaluator_list
   )
   raises (DuplicateEvaluatorName, InvalidPolicyEvaluatorList);
```

```
    void add_evaluators (
        in  PolicyEvaluatorList policy_evaluator_list,
        in  ResourceNamePattern pattern
    )
    raises (InvalidResourceNamePattern,
            PatternNotRegistered,
            InvalidPolicyEvaluatorList,
            DuplicateEvaluatorName);

    void delete_evaluators (
        in  PolicyEvaluatorList policy_evaluator_list,
        in  ResourceNamePattern pattern
    )
    raises (InvalidResourceNamePattern,
            PatternNotRegistered,
            InvalidPolicyEvaluatorList,
            DuplicateEvaluatorName);

    DecisionCombinator get_combinator (
        in  ResourceNamePattern pattern
    )
    raises (InvalidResourceNamePattern,
            PatternNotRegistered);

    void set_combinator (
        in  DecisionCombinator decision_combinator,
        in  ResourceNamePattern pattern
    )
    raises (InvalidResourceNamePattern,
            PatternNotRegistered);

    void delete_combinator (
        in  ResourceNamePattern pattern
    )
    raises (InvalidResourceNamePattern,
            PatternNotRegistered);

    DecisionCombinator get_default_combinator ();

    void set_default_combinator(
        in  DecisionCombinator decision_combinator
    );


};


//*******************************************************
//      interface PolicyEvaluatorAdmin
//*******************************************************

interface PolicyEvaluatorAdmin {

    void    set_policies(
        in  PolicyNameList policy_names,
        in  ResourceName   resource_name
    )
    raises (InvalidResourceName,
            ResourceNameNotFound,
            InvalidPolicyNameList);

    void    add_policies(
        in  PolicyNameList policy_names,
        in  ResourceName   resource_name
    )
    raises (InvalidResourceName,
            ResourceNameNotFound,
            InvalidPolicyNameList);

    void    delete_policies(
```

```
        in  PolicyNameList policy_names,
        in  ResourceName   resource_name
    )
    raises (InvalidResourceName,
            ResourceNameNotFound,
            InvalidPolicyNameList,
            NoAssociation);

    PolicyNameList list_policies();

    PolicyName set_default_policy(
        in  PolicyName policy_names
    )
    raises (InvalidPolicy);
};


};

#endif  // _DF_RESOURCE_ACCESS_DECISION_IDL_
```