Resource Access Decision Facility Specification

April 2001
Version 1.0

Contents

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

Associated OMG Documents

In addition to the CORBA Transportation specifications, the CORBA documentation set includes the following:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBA Languages*, a collection of language mapping specifications. See the individual language mapping specifications.
- *CORBAservices: Common Object Services Specification*, a collection of OMG's Object Services specifications.
- *CORBAfacilities: Common Facilities Specification*, a collection of OMG's Common Facility specifications.
- *CORBA Manufacturing*: Contains specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
- *CORBA Healthcare*: Comprised of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
- *CORBA Finance*: Targets a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Telecoms*: Comprised of specifications that relate to the OMG-compliant interfaces for telecommunication systems.

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue, Suite 201
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Acknowledgments

The following companies submitted and/or supported parts of this specification:

2AB, INC.

Baptist Health Systems of South Florida

CareFlow/Net, Inc.

Concept Five

DASCOM, Inc.

IBM

Inprise

Los Alamos National Laboratory

National Institute of Standards (NIST)

National Security Agency (NSA)

Philips Medical Systems

The submitters would like to thank the following individuals for their active participation during the development of the specificaiton and detailed review of each version of the specification.

John Barkley - NIST

Bob Blakley - IBM

Overview 1

Contents

This chapter contains the following topics.

1.1 Introduction

The Resource Access Decision (RAD) Facility 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 facility is intended to be used by security-aware applications.

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

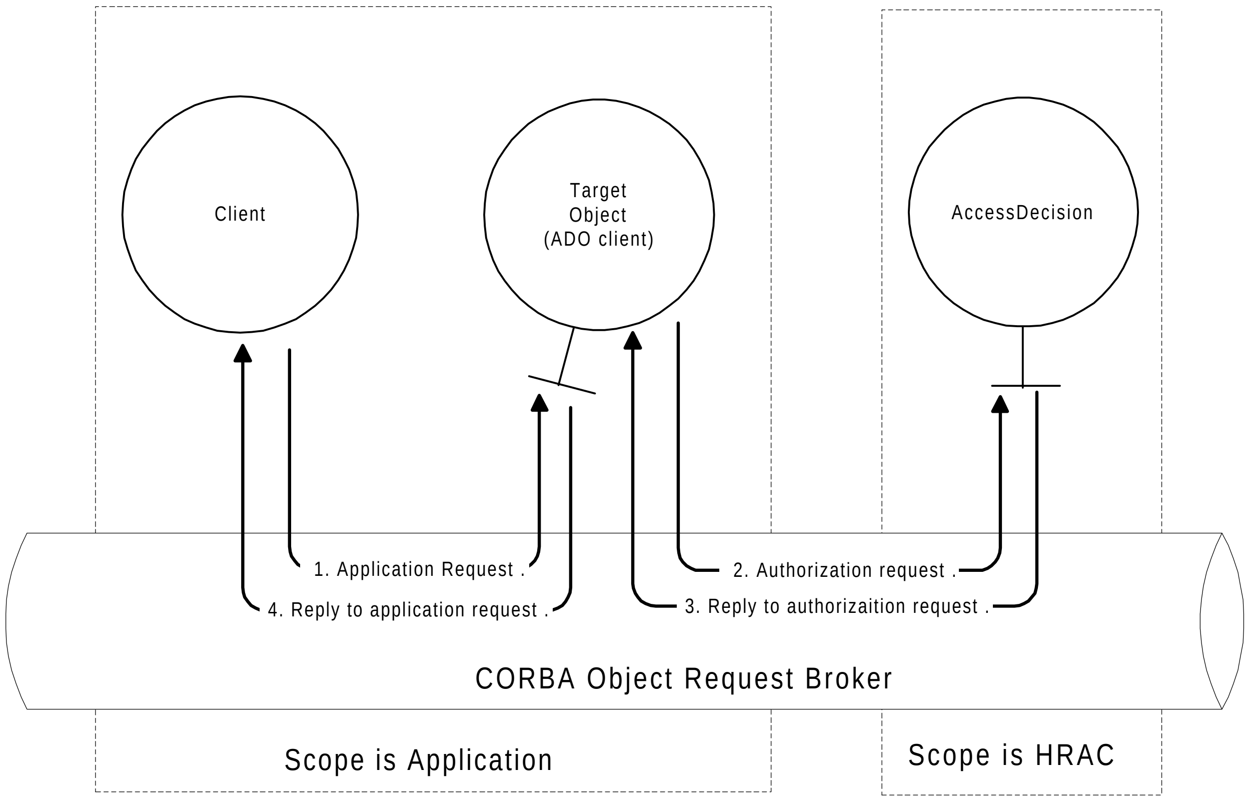

Client
Target Object (ADO client)
AccessDecision
1. Application Request .
2. Authorization request .
3. Reply to authorizaition request .
4. Reply to application request .
CORBA Object Request Broker
Scope is Application
Scope is HRAC

Figure 1-1 Application flow schema

The sequence of the interaction, illustrated by Figure 1-1, 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 (ADO) 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 specification) 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, “PolicyNameListIterator Interface,” on page 2-12.

1.2 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 a 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 specification by heavy dotted lines. In some cases there are types that occur within the scope of this specification 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 specification 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 that 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**.

1.2.1 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 its 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. This combinator is responsible for taking the results of the **PolicyEvaluators evaluate()** method and making a final access decision.

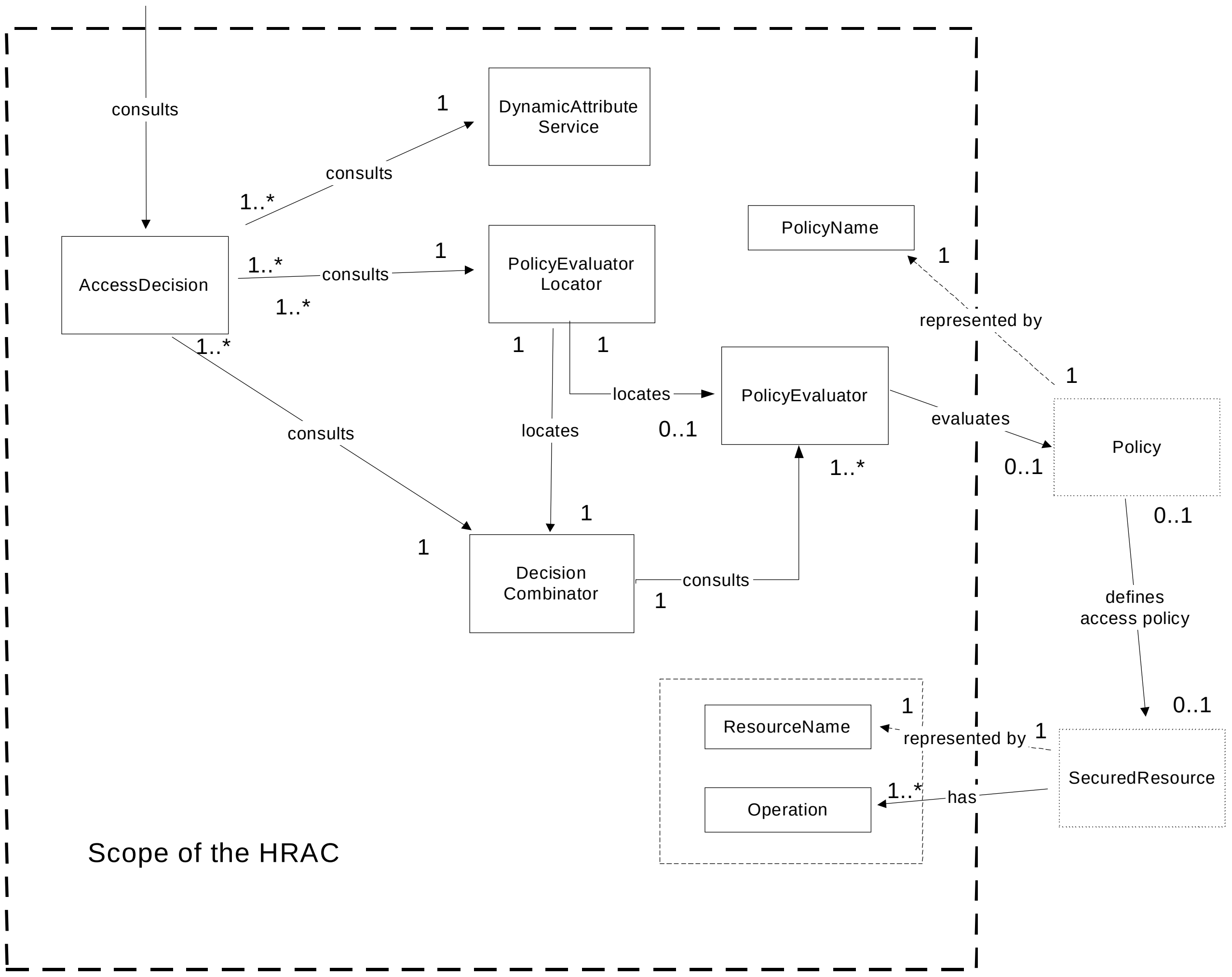


Figure 1-2 Access Decision Model

1.2.2 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 specification. This Administrative model clearly indicates this by placing Policy administration outside the dotted line that delineates the scope of the RAD specification.

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**. **PolicyEvaluators** have an endless series of options for implementation. For this reason, the interface is public and evaluators may be "plugged-in" to a 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, a 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.

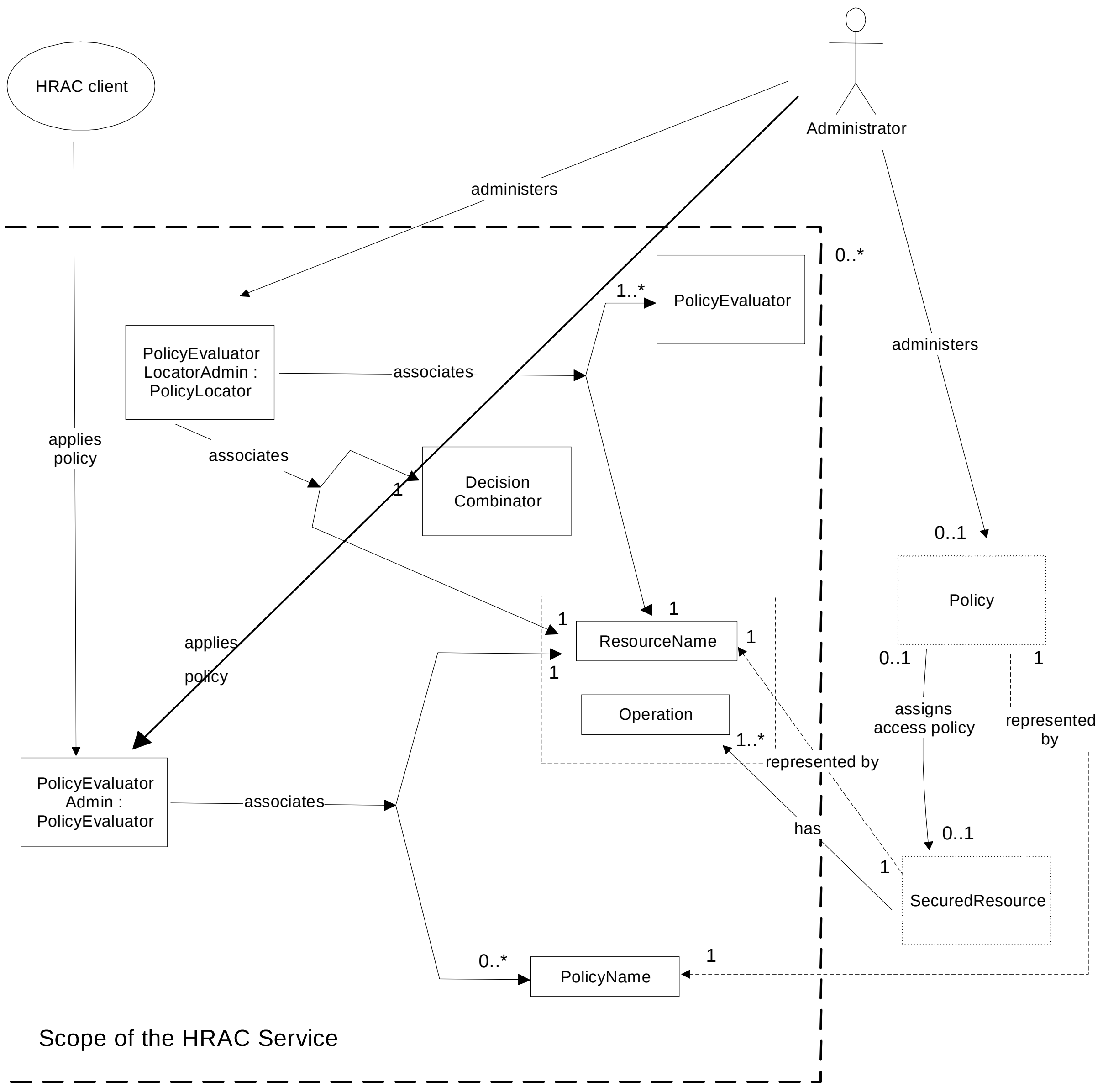


Figure 1-3 Administrative Model

1.2.3 Information Model

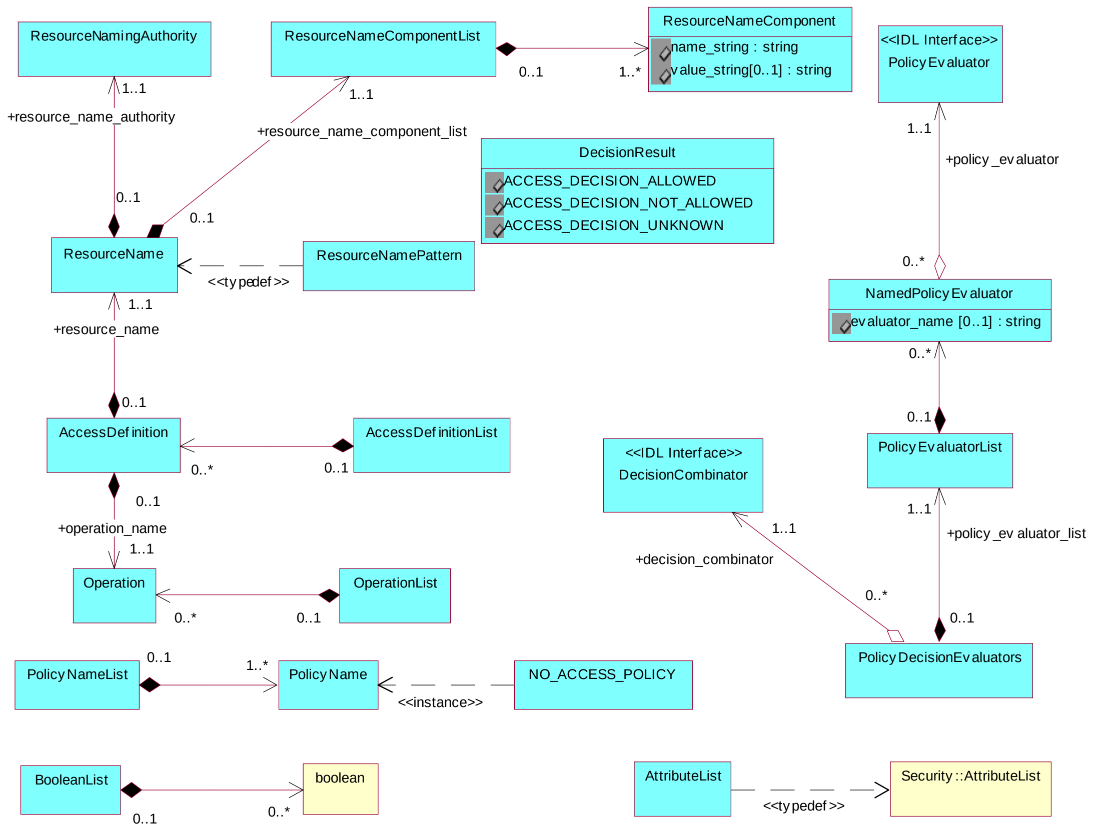


Figure 1-4 Information Model

The information model of RAD is designed to be simple to implement and to use.

1.2.4 Computational Model

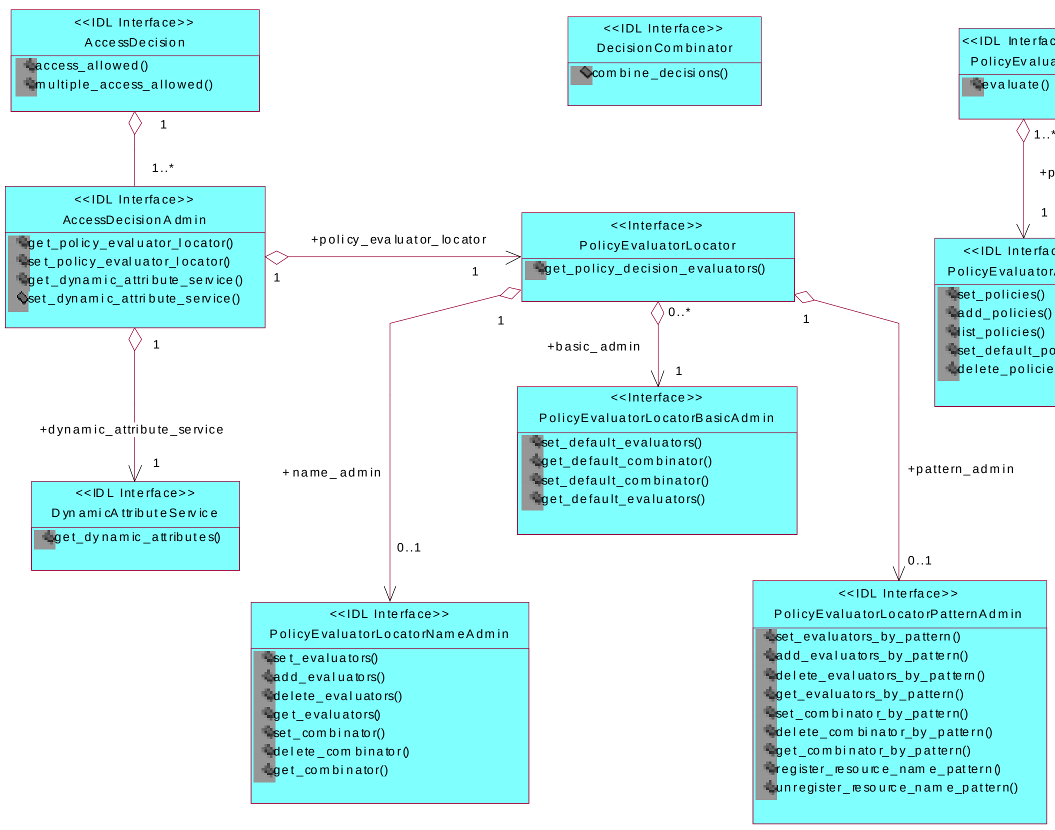


Figure 1-5 Computational Model

The computational model of RAD consists of two interface groups:

- Runtime interfaces: **AccessDecision**, **DynamicAttributeService**, **PolicyEvaluator**, **PolicyEvaluatorLocator**, and **DecisionCombinator**.
- Administrative interfaces: **AccessDecisionAdmin**, **PolicyEvaluatorAdmin**, and **PolicyEvaluatorLocatorBasicAdmin**, **PolicyEvaluatorLocatorNameAdmin**, **PolicyEvaluatorLocatorPatternAdmin**.

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.

DfResourceAccessDecision Module 2

Contents

This chapter contains the following topics.

2.1 OMG 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 {

interface PolicyNameListIterator {
...
};
interface AccessDecision {
...
};

interface DynamicAttributeService {
...
};

interface PolicyEvaluatorLocator {
...
};

interface DecisionCombinator {
...
};

interface PolicyEvaluator {
...
};

interface AccessDecisionAdmin {
...
};

interface PolicyEvaluatorLocatorBasicAdmin {
...
};

interface PolicyEvaluatorLocatorNameAdmin {
...
};

interface PolicyEvaluatorLocatorPatternAdmin {
```

```
...
};

interface PolicyEvaluatorAdmin {
...
};

};

#endif // _DF_RESOURCE_ACCESS_DECISION _IDL_
```

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

```
#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.

2.2 Types

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

2.2.1 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;
```

2.2.2 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 {
   ResourceNamingAuthorityresource_naming_authority;
   ResourceNameComponentListresource_name_component_list;
};

typedef ResourceName ResourceNamePattern;

typedef sequence<string> OperationList;
```

ResourceNameComponent

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

ResourceNameComponentList

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

ResourceNamingAuthority

A **ResourceNamingAuthority** is used to identify an authority whose defined the semantics of the naming scheme used in the components of the corresponding **resource_name_component_list** data member.

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 invalid 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 invalid if either **resource_name_authority** or **resource_name_component_list** is invalid.

OperationList

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

2.2.3 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 **PolicyEvaluatorAdmin** interface. It is used as an input parameter to the **replace_policy()**, **add_policy()**, and **set_default_policy()** methods of the **PolicyEvaluatorAdmin** interface. **PolicyName**s 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 **PolicyName**s. 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. A datum element of type **NamedPolicyEvaluator** is invalid if its data member "**policy_evaluator**" has value **nil.PolicyEvaluatorList**.

PolicyEvaluatorList

A **PolicyEvaluatorList** is a sequence of **NamedPolicyEvaluator**. The administrative interfaces of **PolicyEvaluatorLocator** interface allow the association of a list of **NamedPolicyEvaluator**(s) with a **ResourceName**. This type is returned from **get_policy_decision_evaluators()** and **set_default_evaluators()** and is used as an input parameter in the **set_evaluators**, **add_evaluators()**, **delete_evaluators()**, **set_evaluators_by_pattern()**, **add_evaluators_by_pattern()**, **delete_evaluators_by_pattern()**, and **set_default_evaluators()** operations. The **PolicyEvaluatorList** returned from the **PolicyEvaluatorLocator** is passed to the **DecisionCombinator** returned from the **PolicyEvaluatorLocator**. A datum element of type **PolicyEvaluatorList** is invalid if it is empty or any of its elements is invalid.

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.

2.2.4 Types Used to Request Access Decisions

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

struct AccessDefinition {
    ResourceName   resource_name;
    string         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**.

2.2.5 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 RadInternalError{InternalErrorType ed;};
```

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

exception RadComponentError{
    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 ResourceNameNotFound {ExceptionData ed;};
exception NoAssociation {ExceptionData ed;};
exception InvalidPolicy {ExceptionData ed;};
exception DuplicateEvaluatorName {ExceptionData ed;};
exception InvalidResourceName {ExceptionData ed;};
exception InvalidResourceNamePattern {ExceptionData ed;};
exception TooMany { };

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.

RadInternalError

The **RadInternalError** 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. Indicating Fatal means that the ADO client should discontinue using the ADO.

RadComponentError

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

PatternConflict

The **PatternConflict** exception is thrown by the **PolicyEvaluatorLocatorAdmin** when a **register_resource_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 a **register_resource_name_pattern()** detects a duplicate pattern registration.

PatternNotRegistered

The **PatternNotRegistered** exception is thrown by **PolicyEvaluatorLocatorAdmin** operations 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 RAD.

ResourceNameNotFound

The **ResourceNameNotFound** exception is thrown by **PolicyEvaluatorAdmin** interface operations when a **ResourceName** has not been defined. Not all implementations will require pre-definition 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 an association 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 associate 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 **evaluator_name** but different value of its data member **policy_evaluator** as some other named policy evaluator 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 exception 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 exception 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, then the list is invalid not because of a particular element, but because of some other reason (for example, because the list is empty).

TooMany

This exception is raised for **list_policies()** if the number of **PolicyName**s found (based on the **seq_max** and **iter_max** argument) exceeds the implementations limit. The implementation may be optionally configurable by the implementation.

InvalidPolicyEvaluatorList

This exception is raised when a **PolicyEvaluatorList** contains invalid elements. Please refer to the specification of **PolicyEvaluatorList** data type for a description of invalid **PolicyEvaluatorList** datum elements of that type.

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

2.3 PolicyNameListIterator Interface

```
//*******************************************************************
//     interface PolicyNameListIterator
//*******************************************************************

interface PolicyNameListIterator {
    unsigned long how_many();
    boolean next_one(
            out PolicyName name);
     boolean next_n(
            in unsigned long how_many,
            out PolicyNameList list);
     void destroy();
 };
```

The **PolicyNameListIterator** is used to manage the list of Policy names that may be returned from a **list_policies()** operation of the **PolicyEvaluatorAdmin** interface.

how_many( )

returns the number of policy names held by the iterator at this time.

next_one( )

returns true if a **PolicyName** is returned in the out parameter. Returns false if there are no more policy names.

next_n( )

returns the next **n** policy names held by the iterator. Returns true if a **PolicyNameList** is returned in the out parameter. Returns false if there are no more policy names.

destroy( )

destroys the iterator. The iterator will destroy itself if all policy names are retrieved from the iterator; however, a client should destroy the iterator using this operation when they are finished if they have not retrieved all the policy names.

2.4 AccessDecision Interface

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

```
interface AccessDecision {

    boolean access_allowed(
        in  ResourceName resource_name,
        in  string             operation,
        in  AttributeList      attribute_list
)
    raises (RadInternalError);

    BooleanList multiple_access_allowed(
        in  AccessDefinitionListaccess_requests,
        in  AttributeList          attribute_list
)
    raises (RadInternalError);

};
```

The singleton **AccessDecision** object is used to request decisions on access based on a **ResourceName**, an operation, and a list of **SecAttributes**. This specification provides a framework for the support of many policy evaluators. It is beyond the scope of this specification 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 has 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 that calls RAD is responsible for enforcing the policy decision that 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 **RadInternalError** exception is reserved for internal logic errors and should not be used as a reason code for rejecting a request. As a security consideration, ADO clients are not provided with the specific reason for not allowing access.

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 **RADInternalError** 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. For example, first element of the returned list is an authorization decision for the first element of access_request, and so on.

2.5 DynamicAttributeService Interface

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

interface DynamicAttributeService {

    AttributeList get_dynamic_attributes(
        in   AttributeList    attribute_list,
        in   ResourceNameresource_name,
        in   string           operation
)
    raises (RadComponentError);
};
```

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 **SecAttributes** to the list or may remove **SecAttributes** from 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.

2.6 PolicyEvaluatorLocator Interface

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

interface PolicyEvaluatorLocator {

    readonly attribute PolicyEvaluatorLocatorBasicAdmin basic_admin;
    readonly attribute PolicyEvaluatorLocatorNameAdmin name_admin;
    readonly attribute PolicyEvaluatorLocatorPatternAdmin pattern_admin;

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

};
```

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

readonly attribute PolicyEvaluatorLocatorBasicAdmin basic_admin

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

readonly attribute PolicyEvaluatorLocatorNameAdmin name_admin

The interface for administrating associations between resource names and policy evaluators as well as between resource names and decision combinators can be obtained via this attribute.

readonly attribute PolicyEvaluatorLocatorPatternAdmin pattern_admin

The interface for administrating associations between resource name patterns and policy evaluators as well as between resource name patterns and decision combinators can be obtained via this attribute. If an implementation of a policy evaluator locator does not implement support for resource name patterns this attribute must be null.

get_policy_decision_evaluators()

A **PolicyDecisionEvaluators** structure is returned to the client. A **PolicyDecisionEvaluators** structure contains a **PolicyEvaluatorList** and the **DecisionCombinator**.

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 "evaluator_name."

2.7 DecisionCombinator Interface

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

interface DecisionCombinator{

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

The **DecisionCombinator** interface is used to encapsulate a policy for combining decisions of multiple consulted **PolicyEvaluators** that may disagree. **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 that 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 that 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 **PolicyEvaluator** 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.

2.8 PolicyEvaluator Interface

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

interface PolicyEvaluator {

    readonly attribute PolicyEvaluatorAdmin pe_admin;

    DecisionResult evaluate(
        in  ResourceName    resource_name,
        in  string          operation,
        in  AttributeList   attribute_list
)
    raises (RadComponentError);

};
```

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 specification 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.

2.9 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.

2.10 PolicyEvaluatorLocatorBasicAdmin Interface

```
//*****************************************************
//    interface PolicyEvaluatorLocatorBasicAdmin
//*****************************************************

interface PolicyEvaluatorLocatorBasicAdmin {

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

    PolicyEvaluatorList get_default_evaluators();

    DecisionCombinator get_default_combinator ();

    void set_default_combinator(
        in   DecisionCombinator decision_combinator
    );
};
```

The **PolicyEvaluatorLocatorBasicAdmin** object is used to administrate default associations between **PolicyEvaluators** and **ResourceNames** as well as default associations between **DecisionCombinators** and **ResourceNames**.

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. The default evaluators will be returned by the **PolicyEvaluatorLocator get_policy_decision_evaluators()** operation when no **PolicyEvaluators** have been explicitly assigned for a **ResourceName**.

Preconditions

No preconditions.

Postconditions

1. default_evaluators == "policy_evaluator_list"

get_default_evaluators()

The default set of policy evaluators provided is returned.

Preconditions

No preconditions.

Postconditions

1. return == default_evaluators.

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()** operation for these resources.

Preconditions

No preconditions.

Postconditions

1. default_combinator == "decision_combinator".

2.11 PolicyEvaluatorLocatorNameAdmin Interface

```
//*****************************************************
//    interface PolicyEvaluatorLocatorNameAdmin
//*****************************************************

interface PolicyEvaluatorLocatorNameAdmin {

    PolicyEvaluatorList get_evaluators(
        in   ResourceName resource_name
    )
    raises (InvalidResourceName);

    void set_evaluators (
        in   PolicyEvaluatorList policy_evaluator_list,
        in   ResourceName resource_name
    )
    raises (InvalidPolicyEvaluatorList,
            InvalidResourceName,
            DuplicateEvaluatorName);

    void add_evaluators (
        in   PolicyEvaluatorList policy_evaluator_list,
        in   ResourceName resource_name
)
    raises (InvalidResourceName,
             InvalidPolicyEvaluatorList,
             DuplicateEvaluatorName);

    void delete_evaluators (
        in   PolicyEvaluatorList policy_evaluator_list,
        in  ResourceName resource_name
)
    raises (InvalidResourceName,
             InvalidPolicyEvaluatorList,
             DuplicateEvaluatorName);

    DecisionCombinator get_combinator (
        in  ResourceName resource_name
)
    raises (InvalidResourceName);

    void set_combinator (
        in  DecisionCombinatordecision_combinator,
        in  ResourceName resource_name
)
    raises (InvalidResourceName);

    void delete_combinator (
        in  ResourceName resource_name
)
    raises (InvalidResourceName);
};
```

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

get_evaluators()

The list of **PolicyEvaluators** associated with the **ResourceName** is returned.

Preconditions

No preconditions.

Postconditions

1. return == “resource_name”.registered_ evaluator_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”.registered_evaluator_list == 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”.registered_evaluator_list == union (policy_evaluator_list, “resource_name”.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” : “resource_name”.registered_evaluator_list = “resource_name”.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 **ResourceName** provided, the return will be nil (it will not return the default combinator).

Preconditions

No preconditions.

Postconditions

1. return == “resource_name”.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”.registered_ decision_combinator == “decision_combinator”

delete_combinator()

The **DecisionCombinator** for the **ResourceName** 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” will be associated with the default combinator.

2.12 PolicyEvaluatorLocatorPatternAdmin Interface

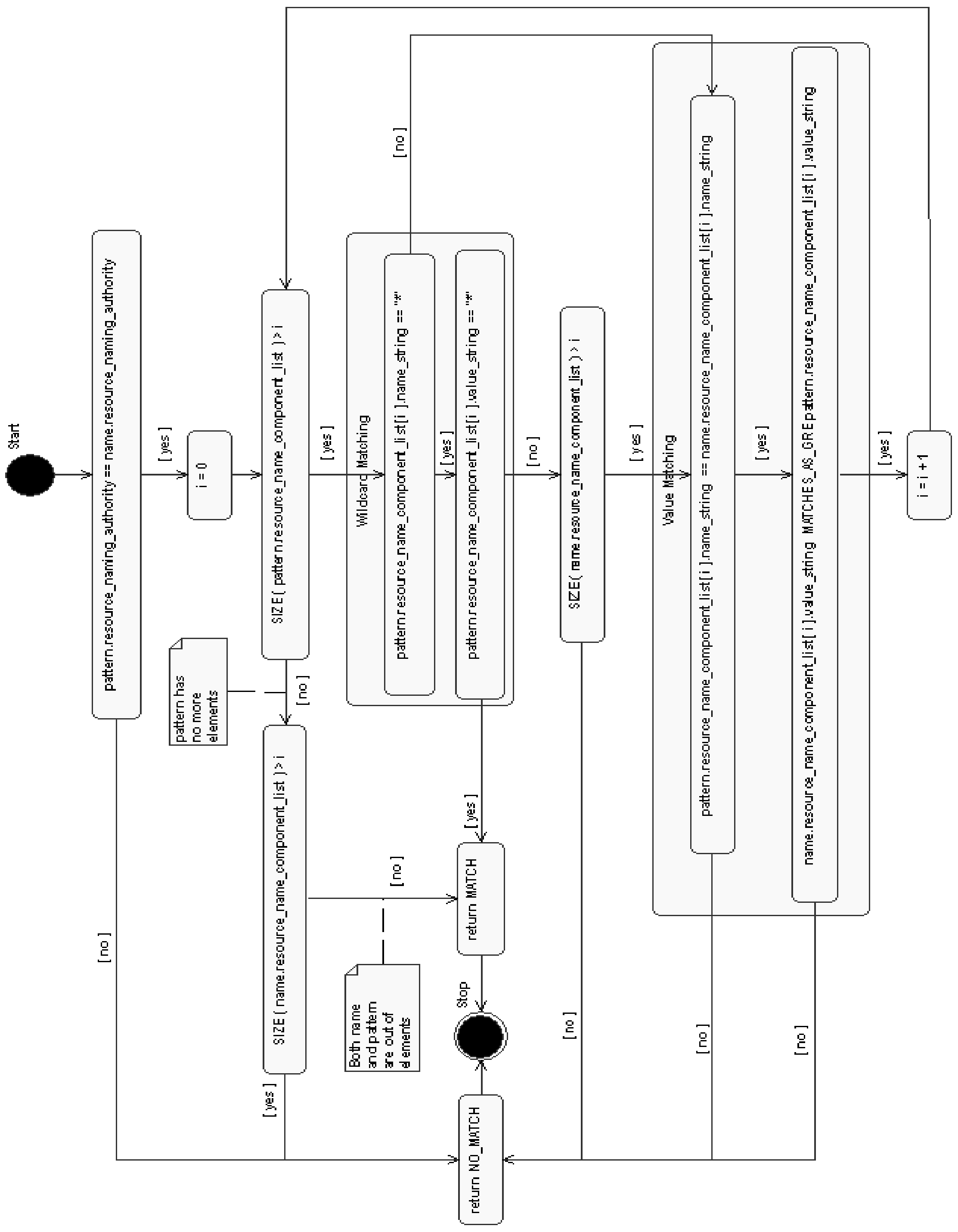
```
//*****************************************************
//    interface PolicyEvaluatorLocatorPatternAdmin
//*****************************************************

interface PolicyEvaluatorLocatorPatternAdmin {

    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_by_pattern (
        in   ResourceNamePattern pattern
    )
    raises (InvalidResourceNamePattern,
            PatternNotRegistered);

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

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

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

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

        void set_combinator_by_pattern (
                in  DecisionCombinatordecision_combinator,
                in  ResourceNamePattern pattern
        )
        raises (InvalidResourceNamePattern,
                        PatternNotRegistered);

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

       DecisionCombinator get_default_combinator( );
};
```

The **PolicyEvaluatorLocatorPatternAdmin** object is used to associate **PolicyEvaluators** with a **ResourceNamePattern**. The object is also used to associate the appropriate **DecisionCombinator** with the **ResourceNamePattern**. This specification provides a framework for the support of one or more policy evaluators for a single resource pattern.

Patterns are used to group resource names without requiring the **PolicyEvaluatorLocator** administrator to enumerate all the resources names 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**.
- Two kinds of **ResourceNameComponents** can occur in a pattern.


Start
pattern.resource_naming_authority == name.resource_naming_authority
[yes]
[no]
i = 0
SIZE (pattern.resource_name_component_list) > i
[yes]
[no]
pattern has no more elements
Wildcard Matching
pattern.resource_name_component_list[i].name_string == "*"
[yes]
[no]
pattern.resource_name_component_list[i].value_string == "*"
[yes]
[no]
SIZE (name.resource_name_component_list) > i
[yes]
[no]
Both name and pattern are out of elements
SIZE (name.resource_name_component_list) > i
[yes]
[no]
Value Matching
pattern.resource_name_component_list[i].name_string == name.resource_name_component_list[i].name_string
[yes]
[no]
name.resource_name_component_list[i].value_string MATCHES_AS_GRE pattern.resource_name_component_list[i].value_string
[yes]
[no]
i = i + 1
return MATCH
Stop
return NO_MATCH

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.

The second kind of **ResourceNameComponent** that can occur in a pattern is a component wildcard pattern:

- **name_string** exactly matches "*" and
- **value_string** exactly matches "*"

Every component of a resource name matches a component wildcard pattern.

A resource name matches a pattern if and only if the algorithm shown in the figure on page 2-27 returns MATCH.

The algorithm has two inputs: resource name ("name") and resource name pattern ("pattern"). It also assumes availability of two functions:

- SIZE - returns number of elements in a sequence,
- MATCHES_AS_GRE - returns "yes" if the resource "name" matches the resource name "pattern," where the "pattern" is interpreted according to the regular expression syntax specified in the definition of **ResourceNamePattern** in Section 2.2.2, "Types that identify and manage information about secured resources," on page 2-4.

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**, **ResourceName**s must also be registered if these administrative interfaces are used to 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 that 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_by_pattern ()

The list of **PolicyEvaluators** associated with the **ResourceNamePattern** is returned.

Preconditions

No preconditions.

Postconditions

1. return == “resource_name_pattern”.registered_ evaluator_list

set_evaluators_by_pattern ()

A list of **PolicyEvaluators** is assigned for the resources that will match **ResourceNamePattern**. 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

add_evaluators_by_pattern ()

A list of **PolicyEvaluators** is added to the list of evaluators for the resources that will match **ResourceNamePattern**. These evaluators will be in the list of **PolicyEvaluators** returned by the **PolicyEvaluatorLocator get_policy_decision_evaluators()** method. The addition of evaluators to a **ResourceNamePattern** 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_by_pattern ()

The list of **PolicyEvaluators** is removed from the list of evaluators for the resources that will match **ResourceNamePattern**. 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_by_pattern ()

The **DecisionCombinator** specified for by the **ResourceNamePattern** 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_by_pattern ()

A **DecisionCombinator** is specified for the resources that will match **ResourceNamePattern**. 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_by_pattern ()

The **DecisionCombinator** for the **ResourceNamePattern** is removed. The default combinator will now be returned by the **PolicyEvaluatorLocator get_policy_decision_evaluators()** method for those resource that used to match the specified **ResourceNamePattern** and do not match any other **ResourceNamePattern** set by **set_combinator_by_pattern()** operation.

Preconditions

No preconditions.

Postconditions

1. Resource names matching only "resource_name_pattern" will be associated with the default combinator.

get_default_combinator ()

The **default DecisionCombinator** is returned.

2.13 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(
             in unsigned long seq_max,
             in unsigned long iter_max,
             out PolicyNameListIterator iter
        ) raises (TooMany);

        PolicyName  set_default_policy(
            in  PolicyName  policy_name
        )
        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 specification) provides a mechanism to allow policies to be associated with “names” that 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 programmatically 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”.
2. if PolicyName == NO_ACCESS_POLICY, then no policy exists for the resource.

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 names of all policies supported by this instance of **PolicyEvaluator** is returned to the client. The number of policy names to be returned in the sequence should not exceed **iter_max**; the number of policy names to be held in the **PolicyNameListIterator** should not exceed **iter_max**. The **TooMany** exception is thrown if the number of policy names that exist (and are requested) exceeds the implementation max.

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 that 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"

2.14 Conformance Classes

There are two conformance classes: "RAD without Patterns" and "RAD with Patterns."

An implementation of Resource Access Decision (RAD) facility compliant to conformance class "RAD without Patterns" must implement all of the interfaces defined in this specification except interface **PolicyEvaluatorLocatorPatternAdmin**. In this case **pattern_admin** attribute of **PolicyEvaluatorLocator** interface implementation must be /return value null.

An implementation of Resource Access Decision facility compliant to conformance class "RAD with Patterns" must implement all of the interfaces defined in this specification. In this case **pattern_admin** attribute of **PolicyEvaluatorLocator** interface implementation must return an object reference for a **PolicyEvaluatorLocatorPatternAdmin**.

OMG IDL A

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

#ifndef _DF_RESOURCE_ACCESS_DECISION_IDL_
#define _DF_RESOURCE_ACCESS_DECISION_IDL_

#include "Security.idl"

#pragma prefix "omg.org"

module DfResourceAccessDecision {

typedef sequence<boolean> BooleanList;
typedef Security::AttributeList AttributeList;

interface DynamicAttributeService;
interface DecisionCombinator;
interface PolicyEvaluator;
interface PolicyEvaluatorAdmin;
interface PolicyEvaluatorLocatorBasicAdmin;
interface PolicyEvaluatorLocatorNameAdmin;
interface PolicyEvaluatorLocatorPatternAdmin;

//*****************************************************************
//    Types that identify a secured resource
//*****************************************************************
struct ResourceNameComponent {
      stringname_string;
      stringvalue_string;
};
typedef sequence<ResourceNameComponent> ResourceNameComponentList;

typedef string ResourceNamingAuthority;
```

```
struct ResourceName {
            ResourceNamingAuthorityresource_naming_authority;
            ResourceNameComponentListresource_name_component_list;
};
typedef ResourceName ResourceNamePattern;

typedef sequence<string> 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 {
      PolicyEvaluatorListpolicy_evaluator_list;
      DecisionCombinatordecision_combinator;
};

//*******************************************************************
//    Types used to request an Access Decision
//*******************************************************************
struct AccessDefinition {
      ResourceNameresource_name;
      string   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;
      stringreason;
};
enum InternalErrorType { Fatal, NotFatal };
```

```
//*****************************************************************
//    Exception thrown by the Access Decision Object
//*****************************************************************
exception RadInternalError {InternalErrorType ed;};

//*****************************************************************
//    Exception thrown by Internal non-admin interfaces
//*****************************************************************
exception RadComponentError {
    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 ResourceNameNotFound { ExceptionData ed; };
exception NoAssociation { ExceptionData ed; };
exception InvalidPolicy { ExceptionData ed; };
exception DuplicateEvaluatorName { ExceptionData ed; };
exception InvalidResourceName { ExceptionData ed; };
exception InvalidResourceNamePattern { ExceptionData ed; };
exception TooMany {};
exception InvalidPolicyEvaluatorList {
    ExceptionData ed;
    NamedPolicyEvaluatorfirst_invalid_element;
};
exception InvalidPolicyNameList {
    ExceptionData ed;
    PolicyNamefirst_invalid_element;
};

//*****************************************************************
//    interface PolicyNameListIterator
//*****************************************************************

interface PolicyNameListIterator {
    unsigned long how_many();

    boolean next_one(
        out PolicyName name);

    boolean next_n(
        in unsigned long how_many,
        out PolicyNameList list);
    void destroy();
};

//*****************************************************************
//    interface AccessDecision
//*****************************************************************
interface AccessDecision {

    boolean access_allowed(
            in ResourceNameresource_name,
            in stringoperation,
            in AttributeListattribute_list
    )
    raises (RadInternalError);

    BooleanList multiple_access_allowed(
            in AccessDefinitionListaccess_requests,
            in AttributeListattribute_list
    )
    raises (RadInternalError);
};

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

    AttributeList get_dynamic_attributes(
            in AttributeListattribute_list,
            in ResourceNameresource_name,
            in stringoperation
    )
    raises (RadComponentError);
};

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

    readonly attribute PolicyEvaluatorLocatorBasicAdminbasic_admin;
    readonly attribute PolicyEvaluatorLocatorNameAdmin name_admin;
    readonly attribute PolicyEvaluatorLocatorPatternAdmin pattern_admin;

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

};
```

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

      boolean combine_decisions(
                  in ResourceNameresource_name,
                  in string     operation,
                  in AttributeListattribute_list,
                  in PolicyEvaluatorListpolicy_evaluator_list
      )
      raises (RadComponentError);

};

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

      readonly attribute PolicyEvaluatorAdmin pe_admin;

      DecisionResult evaluate(
                  in ResourceNameresource_name,
                  in stringoperation,
                  in AttributeListattribute_list
      )
      raises (RadComponentError);

};

//*****************************************************************
//    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 PolicyEvaluatorLocatorBasicAdmin
//******************************************************************
interface PolicyEvaluatorLocatorBasicAdmin {

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

    PolicyEvaluatorList get_default_evaluators();

    DecisionCombinator get_default_combinator();

    void set_default_combinator(
            in DecisionCombinator decision_combinator
    );

};

//******************************************************************
//    interface PolicyEvaluatorLocatorNameAdmin
//******************************************************************
interface PolicyEvaluatorLocatorNameAdmin {

    PolicyEvaluatorList get_evaluators(
            in ResourceName resource_name
    )
    raises (InvalidResourceName);

    void set_evaluators(
            in PolicyEvaluatorList policy_evaluator_list,
            in ResourceName resource_name
    )
    raises (InvalidPolicyEvaluatorList,
            InvalidResourceName,
            DuplicateEvaluatorName);

    void add_evaluators(
            in PolicyEvaluatorList policy_evaluator_list,
            in ResourceName resource_name
    )
    raises (InvalidPolicyEvaluatorList,
            InvalidResourceName,
            DuplicateEvaluatorName);

    void delete_evaluators(
            in PolicyEvaluatorList policy_evaluator_list,
                in ResourceName resource_name
        )
        raises (InvalidPolicyEvaluatorList,
                InvalidResourceName,
                DuplicateEvaluatorName);

        DecisionCombinator get_combinator(
                in ResourceName resource_name
        )
        raises (InvalidResourceName);

        void set_combinator(
                in DecisionCombinator decision_combinator,
                in ResourceName resource_name
        )
        raises (InvalidResourceName);

        void delete_combinator(
                in ResourceName resource_name
        )
        raises (InvalidResourceName);

};

//****************************************************************
//    interface PolicyEvaluatorLocatorPatternAdmin
//****************************************************************
interface PolicyEvaluatorLocatorPatternAdmin {

        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_by_pattern(
                in ResourceNamePattern pattern
        )
        raises (InvalidResourceNamePattern,
                PatternNotRegistered);

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

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

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

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

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

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

        DecisionCombinator get_default_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(
            in unsigned long seq_max,
            in unsigned long iter_max,
            out PolicyNameListIterator iter
    )
    raises (TooMany);

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

}; // end of DfResourceAccessDecision module
#endif
```

Use Case Example B

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

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

B.1 Generic RAD Sequence Diagram

This section shows the generic sequence diagram for the RAD.

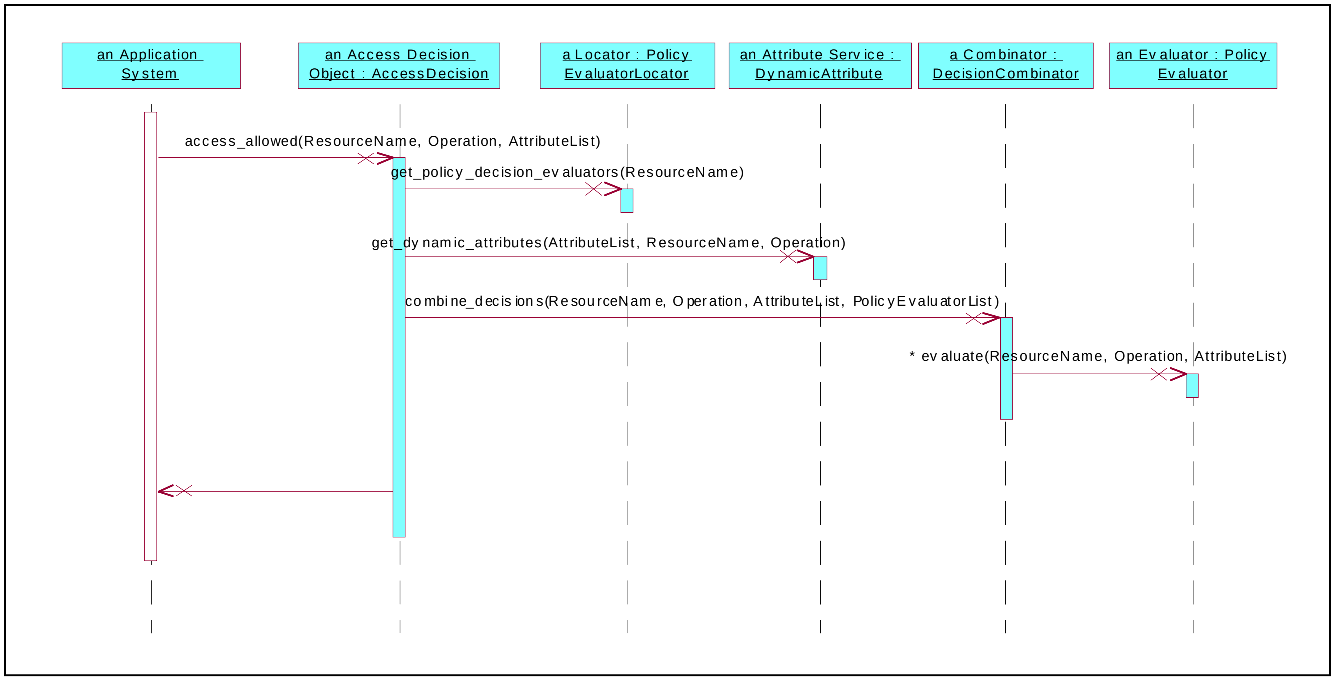

an Application System
an Access Decision Object : AccessDecision
a Locator : Policy EvaluatorLocator
an Attribute Service : DynamicAttribute
a Combinator : DecisionCombinator
an Evaluator : Policy Evaluator
access_allowed(ResourceName, Operation, AttributeList)
get_policy_decision_evaluators(ResourceName)
get_dynamic_attributes(AttributeList, ResourceName, Operation)
combine_decisions(ResourceName, Operation, AttributeList, PolicyEvaluatorList)
* evaluate(ResourceName, Operation, AttributeList)

Figure B-1 Generic RAD Sequence Diagram

B.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 that 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) 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 a 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 that 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 that 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 that contains a list of attending physicians.

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

Use Case Name	Out-patient Hospital Visit to Attending Physician
Goal in Contact	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.)

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

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 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 4a	Branching Action
	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 5	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.

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

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).

As shown in Table B-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.

B.2.1 ADO Client Actions: Read Patient Record

Table B-2 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.

Table B-2 ADO Client Actions: Read Patient Record

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.

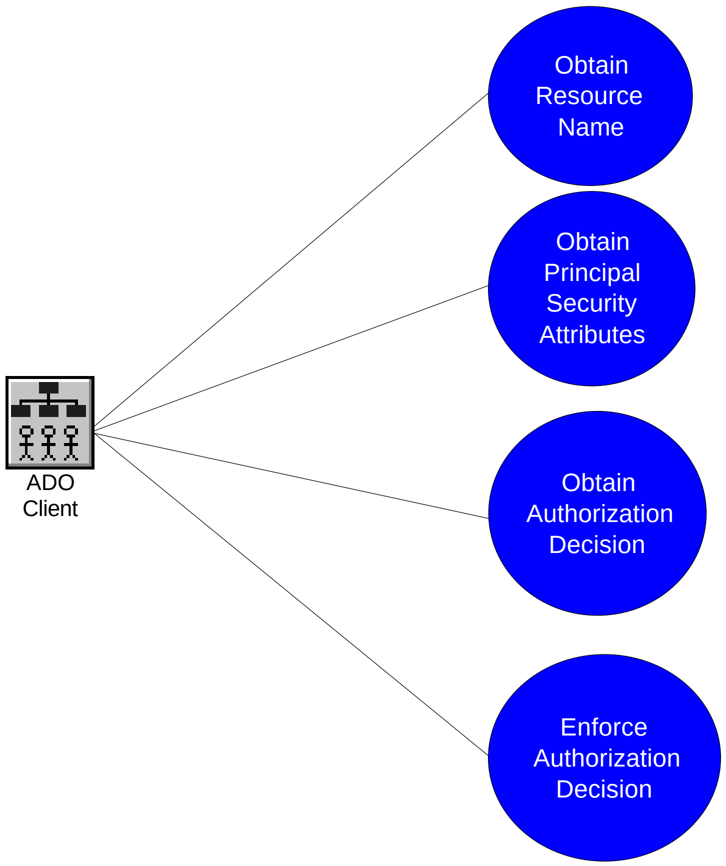

ADO Client
Obtain Resource Name
Obtain Principal Security Attributes
Obtain Authorization Decision
Enforce Authorization Decision

Figure B-2 ADO Client Actions Diagram

Table B-3 ADO Client Actions: Read Patient Record

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 B-3 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.

B.2.2 ADO Actions: Read Patient Record

Table B-4 Read Patient Record

Use Case Name	ADO Actions: Read Patient Records
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.

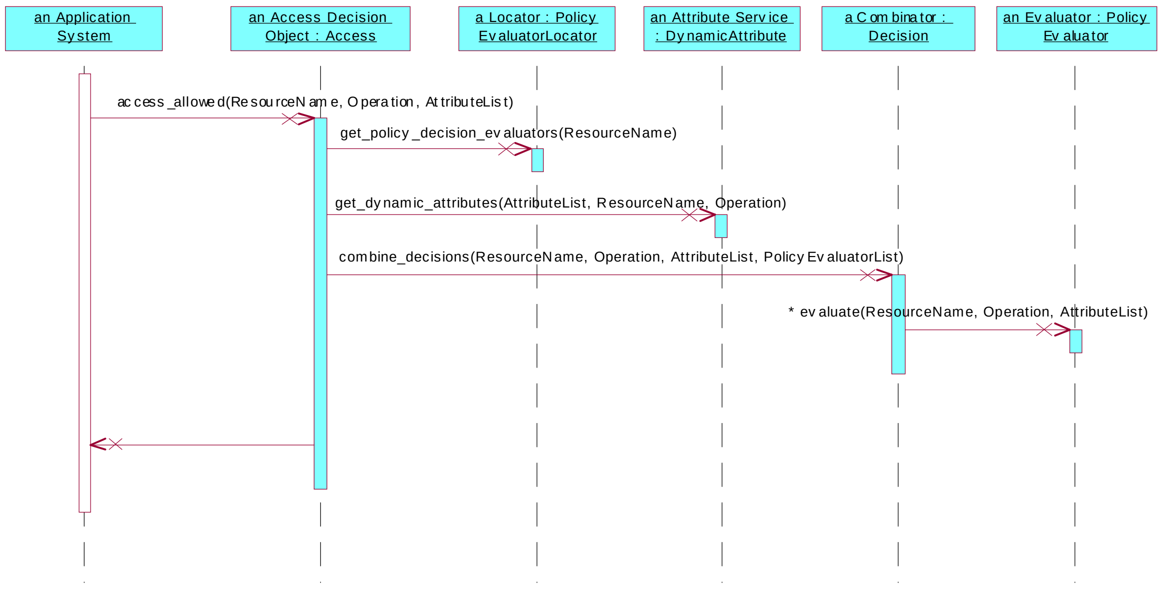

an Application System
an Access Decision Object : Access
a Locator : Policy EvaluatorLocator
an Attribute Service : DynamicAttribute
a Combinator : Decision
an Evaluator : Policy Evaluator
access_allowed(ResourceName, Operation, AttributeList)
get_policy_decision_evaluators(ResourceName)
get_dynamic_attributes(AttributeList, ResourceName, Operation)
combine_decisions(ResourceName, Operation, AttributeList, PolicyEvaluatorList)
* evaluate(ResourceName, Operation, AttributeList)

Figure B-3 Read Patient Record Diagram

Description	
Step	**Action**
1	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.
2	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.

Description	
Step	**Action**
3	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.
4	ADO returns the boolean result "TRUE".
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	ADO Client Actions: Read Patient Record

The above table 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 that 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.

Resource Names for PIDS C

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.

C.1 Changes to Conformance Classes

The specification 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."

C.1.1 Changes to Security Guidelines

The specification 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."

Index

Index