

# **A Comparison of Two Privacy Policy Languages: EPAL and XACML**

**Anne Anderson**

# A Comparison of Two Privacy Policy Languages: EPAL and XACML

**Anne Anderson**

SMLI TR-2005-147

September 2005

## **Abstract:**

Current regulatory requirements such as Sarbanes-Oxley, HIPAA, and the European Union Directive on Data Privacy make it increasingly important for enterprises to be able to verify and audit their compliance with privacy policies.

Two platform-independent languages that support directly-enforceable policies including "purposes" are IBM's Enterprise Privacy Authorization Language(EPAL) and the OASIS eXtensible Access Control Markup Language (XACML). This document gives a brief overview of directly-enforceable policy languages, and then compares EPAL and XACML to show where the two languages differ. The differences are used to compare the strengths and weaknesses of each language for expressing privacy policies and for authorization or access control policies.

The main findings of this analysis are:

- With two exceptions, EPAL 1.2 supports a small subset of the functionality offered by XACML 2.0. The two exceptions, a built-in policy "vocabulary" mechanism and "categories", could be supported in XACML 2.0 without changes to the language. Their implementation in EPAL 1.2 is problematic.
- EPAL 1.2 lacks significant features required for complex enterprise policies, both for privacy and for access control in general. It adds no privacy-specific functionality not already supported by XACML 2.0.
- XACML 2.0 is an approved OASIS Standard with an OASIS Standard profile for privacy policies. If EPAL were considered as an additional standard, it would be detrimental to industry functionality and interoperability.

This document examines in detail the differences between the two languages that support these findings.



Sun Labs  
16 Network Circle  
Menlo Park, CA 94025

**email address:**  
anne.anderson@sun.com

© 2005 Sun Microsystems, Inc. All rights reserved. The SML Technical Report Series is published by Sun Microsystems Laboratories, of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

#### TRADEMARKS

Sun, Sun Microsystems, the Sun logo, Java, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>. All technical reports are available online on our website, <http://research.sun.com/techrep/>.

# **A Comparison of Two Privacy Policy Languages: EPAL and XACML**

Anne Anderson  
Sun Microsystems Laboratories

## **Contents**

1. Introduction
2. Overview of EPAL and XACML
3. Comparison summary table
4. Detailed comparison
5. Conclusions
6. Terminology
7. References

## **1. Introduction**

Current regulatory requirements such as Sarbanes-Oxley [17], HIPAA [18], and the European Union Directive on Data Privacy [19] make it increasingly important for enterprises to be able to enforce and verify their compliance with privacy policies. Structured policy languages can play a major role by supporting automated enforcement of policies and auditing of access decisions. While many applications and platforms have their own languages for access control, these are rarely adequate for enforcing privacy policies, and the lack of a single standard makes compliance auditing a nightmare.

A standard structured language for supporting expression and enforcement of privacy policies must meet several requirements. First, in order to support the expression of privacy policies, the language must support not only constraints on who is allowed to perform which actions on which resources, but also must support constraints on the purposes for which data was collected or is to be used [16]. Second, in order to support automated enforcement, the language must enable the expression of directly-enforceable policies - policies that describe subjects, resources, actions, and access conditions using identifiers that can be mapped directly and automatically to the actual objects and operations used by computer applications. Third, in order to be suitable for use as a standard, the language must be platform-independent. Fourth, the language used for privacy policies must be the same as or integrated with the language used for access control policies, because both types

of policies will usually control access to the same resources and must not be in conflict.

*The Platform for Privacy Preferences 1.0 (P3P)* specification from the World Wide Web Consortium (W3C) [14] is a standard privacy policy language that is platform-independent and supports purpose requirements, but it does not support directly-enforceable policies and is not a general-purpose access control language. It is intended for use by user agents for the expression and matching of policies at the human user level; P3P policies are not sufficiently fine-grained and expressive to handle the description of privacy policies at the implementation level. So P3P is not a sufficient solution to the need for an enforceable and verifiable language.

Two platform-independent languages that support directly-enforceable policies including “purposes” are IBM’s *Enterprise Privacy Authorization Language (EPAL)* [1] and the OASIS *eXtensible Access Control Markup Language (XACML)* [3]. IBM submitted EPAL 1.2 to the W3C in November, 2003, for consideration as a privacy policy language standard, although the W3C has taken no action to date. XACML 2.0 is an approved OASIS Standard access control language with an approved OASIS Standard profile for privacy policies [4].

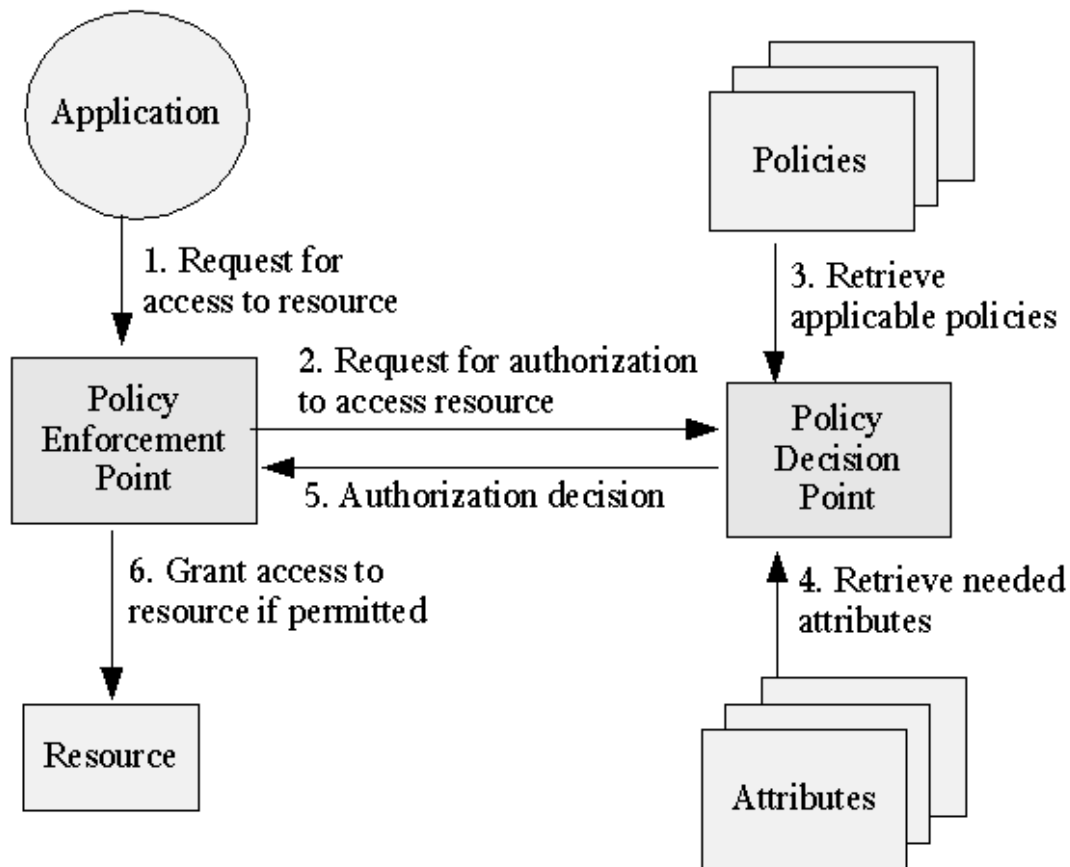
Given that we have two languages covering similar domains, several questions need to be answered:

- What are the differences between EPAL 1.2 and XACML 2.0?
- Which language is better for expressing privacy policies?
- Which language is better for expressing access control policies?
- Should EPAL become a standard for privacy policies?

This document gives a brief overview of EPAL and XACML, and then compares the two languages to show where they differ. The differences are used to compare the strengths and weaknesses of each language for expressing privacy policies and for expressing authorization or access control policies. [Conclusions](#), including answers to the questions above, are presented at the end.

## **2. Overview of EPAL and XACML**

Both EPAL and XACML share an abstract model for policy enforcement defined by the IETF [20][21] and ISO [22]. Illustration 1 shows the components of this model.



*Illustration 1 Policy enforcement model*

In this model, all requests for access to a protected resource go through an abstract component called a *Policy Enforcement Point*, or “PEP”. This component is called an *Access Enforcement Function*, or “AEF”, in the ISO standard. The PEP formulates a request for an authorization decision that includes a description of the request: who is making the request, what resource is being requested, what operation or action is to be performed on the resource, and what is the purpose(s) for the access request. This authorization decision request is sent to another abstract component called a *Policy Decision Point*, or “PDP”. This component is called an *Access Decision Function*, or “ADF”, in the ISO standard. The PDP retrieves the policies applicable to the request, along with any additional information required to evaluate those policies, evaluates the policies the information available, and returns an authorization decision the the PEP. In EPAL and XACML, the authorization decision can be one of *Permit*, *Deny*, or *NotApplicable*. *NotApplicable* means the PDP is unable to provide an authorization decision because it has no policies that apply to the information provided in the request. If the PDP is unable to

evaluate the policies or to retrieve required information for some reason, the PDP will return an error instead of an authorization decision. Based on the authorization decision, the PEP either grants the requested access to the resource or denies access. An authorization decision may be accompanied by *Obligations*, which are actions the PEP must take on conjunction with enforcing the authorization decision. A typical *Obligation* might be a requirement to log the access request.

The PDP has no control over the enforcement of the policy decision. This model for access control depends on having every request for protected resources go through a PEP, which is responsible for all enforcement. The implementation of a PEP is usually application- or platform-specific, as it is usually built into the application or the platform on which the application is built.

Any policy must be stated in terms of a specific vocabulary of terms. In EPAL and XACML, vocabulary terms are called *Attributes*, and are defined by the particular applications or domains that use policies – they are domain-specific. The PDP does not need to understand the domain-specific meaning of each Attribute used in a policy; it needs to understand only how to determine whether the values supplied for each Attribute in an authorization decision request satisfy the conditions for access specified in the policy. In order to do this, the PDP needs to know only a unique identifier for the Attribute and the generic data type of the values for that Attribute. It is the responsibility of each application or domain that will be using policies to define an Attribute to correspond to each of its policy vocabulary items. This means assigning a unique identifier for the item, the data type of the values for that item, and the meaning of each value that might be used for that item. The meaning of the values is not used by the PDP, however, and is significant only to the application or domain. Thus, the format (syntax) of an Attribute is domain-independent, even though the meaning (semantics) of the Attribute value is domain-specific. The policy language may itself define some standard Attributes that are available for use in any policy.

As an example of how Attributes are used to describe an access request and then used in a policy, consider the following access request, which uses a simple format to present a list of Attributes with their corresponding data types and values:

<b>AttributeId</b>	<b>Data type</b>	<b>Value</b>
RequesterName	string	"Mustafa"
RequesterAge	integer	34
ResourceId	URL	<a href="http://example.com/general/products/2357">http://example.com/general/products/2357</a>
Action	string	"view"
Purpose	string	"evaluate"

Now, a policy using this vocabulary of Attributes might look something like this:

```

Rule 1: Return "Permit" if
  (RequestedResource == "http://example.com/general/products/*")
Rule 2: Return "Permit" if
  (RequestedResource == "http://example.com/restricted/products/*") AND
  (RequesterAge ≥ 21)
Rule 3: Return "Deny" if
  (RequestedAction ≠ "view") OR
  ((Purpose ≠ "evaluate") AND (Purpose ≠ "purchase"))

```

In EPAL and XACML, a policy consists of one or more *Rules*. Rules contain conditions that specify the actual requirements for access to a resource or data. Conditions are specified using functions over values for specific Attributes. Rules and policies also contain a description of the requests to which the rule or policy applies. The languages specify how to resolve conflicts in the case that more than one rule or policy applies to a given request, since they might evaluate to different results.

A privacy policy language specifies the format for policies, and specifies how the PDP is to evaluate those policies to compute an authorization decision. A privacy policy language may also specify a format for the authorization decision request and authorization decision, but this format is abstract: while it is possible to implement the abstract format directly, it is also possible to map various application-specific formats to the abstract format for use by the PDP. The format for Attributes is also abstract, allowing various application-specific attribute formats to be mapped to the abstract format used by the PDP. The privacy policy language itself does not specify how policies or Attributes are retrieved. Some Attributes are provided as part of the authorization decision request, but the PDP may have the capability of retrieving additional Attributes from other sources, such as from a registry or by calling back to the PEP.

### 3. Comparison summary table

While EPAL and XACML are very similar in structure and concept, the differences between the languages are significant, and greatly affect their usability and their ability to meet the requirements of an enterprise privacy policy language. This paper focuses on those differences rather than on the similarities.

The following table summarizes the differences between the two languages, and serves as an index to the subsequent sections, which compare each feature in detail. The "[Conclusions](#)" Section summarizes the impact of these differences on the usability of the two languages.

Feature	Difference between EPAL and XACML
<a href="#">Decision request</a>	EPAL supports a subset of XACML

<a href="#">Rule</a>	EPAL supports a subset of XACML
<a href="#">Applicability of rules</a>	EPAL supports a subset of XACML
<a href="#">Condition</a>	Equivalent
<a href="#">Nested policies</a>	EPAL does not support
<a href="#">Result conflicts</a>	EPAL supports a subset of XACML
<a href="#">Policy references</a>	EPAL does not support
<a href="#">Vocabulary</a>	XACML supports a subset of EPAL
<a href="#">Attribute mapping</a>	EPAL supports a subset of XACML
<a href="#">Attribute retrieval</a>	Equivalent
<a href="#">XML attribute values</a>	EPAL does not support
<a href="#">Hierarchical roles</a>	EPAL does not support
<a href="#">Hierarchical categories</a>	XACML does not support
<a href="#">Hierarchical resources / XML document resources</a>	EPAL supports a subset of XACML
<a href="#">Subjects with multiple attributes</a>	EPAL supports a subset of XACML
<a href="#">Multiple subjects</a>	EPAL does not support
<a href="#">Purpose attribute</a>	EPAL supports a subset of XACML
<a href="#">Error handling</a>	EPAL does not support
<a href="#">Revision number</a>	Equivalent
<a href="#">Data types</a>	EPAL supports a subset of XACML
<a href="#">Functions</a>	EPAL supports a subset of XACML
<a href="#">Obligations</a>	EPAL supports a subset of XACML
<a href="#">Multiple responses</a>	EPAL does not support
<a href="#">Status as a standard</a>	XACML is an OASIS Standard EPAL is not a standard

## 4. Detailed comparison

A detailed comparison between the two languages follows. Where terminology differs, but the concept is roughly the same in the given context, the terms are listed as *<XACML term>[<EPAL term>]*.

## Decision request

In both languages, the abstract decision request is a collection of attributes that describe a request for access. Each attribute has an identifier, a data type, and a value. A policy specifies its access rules conditioned on the values presented for these attributes. The attributes in a decision request are organized in different ways in the two languages, and certain types of attributes have different semantics. Both languages structure the conceptual decision request around a *subject[user-category]* who performs an *action* on a *resource[data-category]*. Another required part of an EPAL request, and optional part of an XACML request, is the *purpose* of the access. Another optional part of an XACML request is a set of attributes describing the *environment* in which the access is being performed (such as the time of day), while another optional part of an EPAL request is an unclassified collection of additional *container* attributes. *Container* attributes function in a way similar to the *environment* attributes of XACML, although *container* attributes may apply to the entities or operations represented by the *subject[user-category]*, *action*, or *resource[data-category]*.

In both languages, a concrete decision request format is provided, although, in both cases, implementations are not required to use that format; the format provided is merely one implementation of the abstract semantics of a decision request.

In both languages, there is only a weak relationship between the semantics of the request evaluated by a Policy Decision Point (PDP) and the syntax or contents of any particular message format sent from a Policy Enforcement Point (PEP) to a PDP. For example, attributes supplied by a PEP may be reformatted or augmented with additional attributes obtained by the PDP from other sources.

Illustration 2 presents a rough equivalence between the elements in an EPAL decision request and the elements in an XACML decision request.

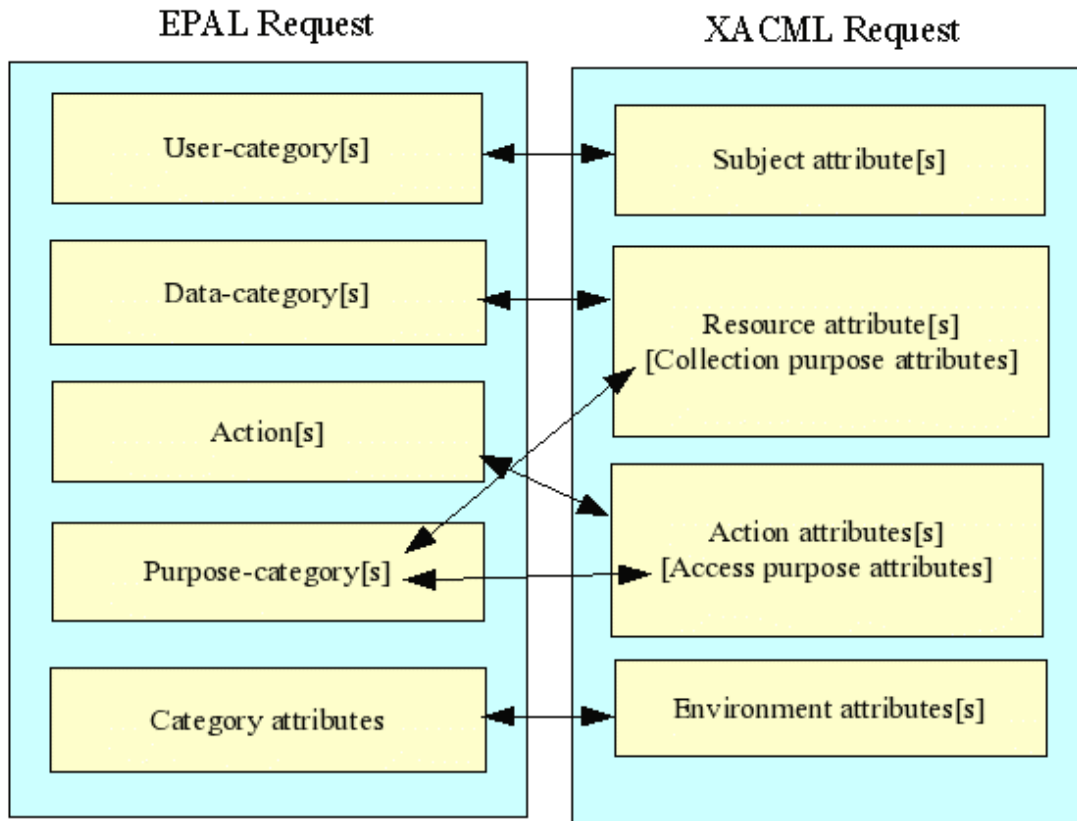


Illustration 2 Correspondence between EPAL and XACML decision requests

### EPAL decision request

An EPAL *simple request* consists of a single *subject[user-category]* attribute, a single *resource[data-category]* attribute, a single *action* attribute, a single *purpose* attribute, and an optional collection of additional *environment[container]* attributes. The *-category* and *purpose* attributes may have hierarchical semantics that are discussed separately below in the “Hierarchical categories” section.

EPAL also supports a *compound request* in which more than one *subject[user-category]* attribute, *resource[data-category]* attribute, *action* attribute, and/or *purpose* attribute can be specified. Each EPAL *subject[user-category]* attribute in a compound request must refer to the same subject entity: the multiple *subject[user-category]* attributes are just alternative names or attributes of the same user. Multiple *resource[data-categories]* attributes, *action* attributes, and *purpose* attributes in a compound request, however, may refer to different data *resources*, *actions*, and *purposes*. According to EPAL 1.2 Section 5.2, the semantics of a compound request are “I belong to multiple *user categories*. Are there any of my *user*

*categories* that allow me to perform all *actions* for all *purposes* on all *data categories*?” The response is *Permit[allow]* if true, and *Deny* (or *NotApplicable*) otherwise. For evaluation, the compound request is conceptually decomposed into multiple simple requests that are evaluated one by one. Implementations of the evaluation algorithm may avoid the decomposition so long as these semantics are preserved. *Subject[user-categories]* attributes are conceptually evaluated in the order in which they are specified in the policy's *vocabulary*. If the evaluation for any *subject[user-category]* attribute against all *resources [data-categories]*, *actions*, and *purpose* attributes returns *Permit[allow]*, then no further policy evaluation is done and a *Permit[allow]* response is returned along with any *Obligations* associated with the evaluation of that *subject[user-category]* attribute.

Additional attributes may be included in a decision request. These attributes are grouped into what are called *[containers]* in the policy's *vocabulary*. Different policy rules can identify the subset of the *[containers]* that they depend on; a decision request must supply values for all attributes in every *[container]* used by any rule that is applicable to that decision request. Each simple request in a decomposed compound request is evaluated using the same set of additional *environment[container]* attribute values, so these values must apply to all *subjects[user-categories]* and *resources[data-categories]*, *actions*, and *purposes* included in the request.

EPAL semantics do not support a policy where a requester must be a member of two *[user-categories]* at once. If an access request indicates that the requester is a member of two *[user-categories]*, then the access will be permitted if either of those *[user-categories]* permits access. For example, if *health care provider* and *hospital staff* are two different *[user-categories]*, then it is not possible to require that a requester be both a *health care provider* AND *hospital staff*.

### **XACML decision request**

An XACML decision request consists of four collections of attributes: a collection of *subject[user-categories]* attributes describing each *subject entity[user]* involved in making the access request, a collection of *resource[data-categories]* attributes describing the one resource being accessed, a collection of *action* attributes describing the one action to be performed on the one resource, and an optional collection of additional *environment [container]* attributes describing information related to the access request, but not associated specifically with any of the *subjects[users]*, the *resource[data-category]*, or the *action*. Two *purpose* attributes are defined in the *Privacy policy profile of XACML v2.0* [4]: a *resource[data-category]* attribute for describing the purpose(s) for which the data was collected, and an *action* attribute for describing the purpose(s) for which the data is being requested. All of these attributes are used simultaneously in evaluating a policy, and only a single policy evaluation is done per decision request.

Unlike EPAL, a single XACML decision request may involve multiple *subjects*, such as

- the human user on whose behalf the application is making the access request,
- the application class that is making the access request,
- the machine platform on which the application class is executing,
- the human user that will receive the resource being requested, etc.

Each of these *subject* entities may have specific attributes associated with it in the *subject [user-category]* collection. See the “[Multiple subjects](#)” section below for more details.

The *Multiple resource profile of XACML v2.0* [5] provides ways to package requests for multiple *resources[data-categories]* into a single decision request, but these are evaluated conceptually as if multiple separate requests had been submitted. As in EPAL, implementations need not perform separate evaluations so long as the semantics are preserved. Options allow the return of a collection of responses, one per individual decision request, or for the return of a single response. The option for return of a single response is equivalent to the EPAL use of multiple *resources[data-categories]* in a single decision request. In XACML, the ability to request access to multiple resources in a single decision request is especially useful for access to multiple nodes in a hierarchical resource, but may be used for any set of *resources[data-categories]*. The XACML option for returning separate responses for each resource can be used to allow access to only the subset of nodes for which access is permitted; to achieve this effect in EPAL, a separate EPAL decision request must be submitted for each node in the hierarchical resource.

The XACML specification says XACML supports only one *action* per request. While it violates the specification, there is nothing that actually prevents a deployer from treating the bag of values for the *action-id* attribute as a bag of different actions. Policies can be written based on the contents of that bag. Such policies must be written very carefully to avoid unintended results, however.

Each of the two XACML *purpose* attributes (one representing the purpose for which data was collected, the other representing the purpose for which data is being requested) will likewise be returned as a bag of values, so a single XACML request can use multiple values for the *purpose* attributes to describe multiple purposes, and policies can be written based on that collection of purpose values. Again, such policies must be written carefully to avoid unintended results.

### **Decision request: summary**

XACML supports all the EPAL decision request functionality.

EPAL supports all the XACML decision request functionality except for

- the ability to request a separate response for each of multiple *resources[data-categories]* submitted in a single decision request,

- the ability to describe more than one subject entity involved in making an access request.

In order to obtain separate answers for multiple *resources[data-categories]* in EPAL, a separate decision request must be submitted for each. EPAL has no workaround for the lack of support for multiple simultaneous subjects.

In this aspect, therefore, EPAL supports only a subset of the functionality offered by XACML.

## **Rule**

In both EPAL and XACML, a rule specifies the actual conditions under which access is to be allowed or denied. A policy contains a collection of rules. Each rule contains a special set of predicates used to determine whether the rule applies to a given decision request, a specification of the result to be returned if the rule is satisfied, and an optional set of *Condition* predicates that must be true with respect to a given decision request if the rule is to be satisfied. If either the applicability predicates OR the *Condition* predicates evaluate to *false*, the rule is treated as not applicable to the given request. In both languages, the predicates in the applicability section are grouped by whether they refer to *subjects[users]*, *resources[data]*, or *actions*. The EPAL applicability section also has a special group for *purpose* predicates, which may be included in either the *resource[data]* or *action* predicates section in XACML. The XACML applicability section also has a special group for *environment[container]* attributes. In both languages, at least one predicate in each group must be true for the rule to be applicable.

In both languages, the *Conditions* consist of Boolean combinations of functions.

Illustration 3 shows the abstract components of a rule in both XACML and EPAL.

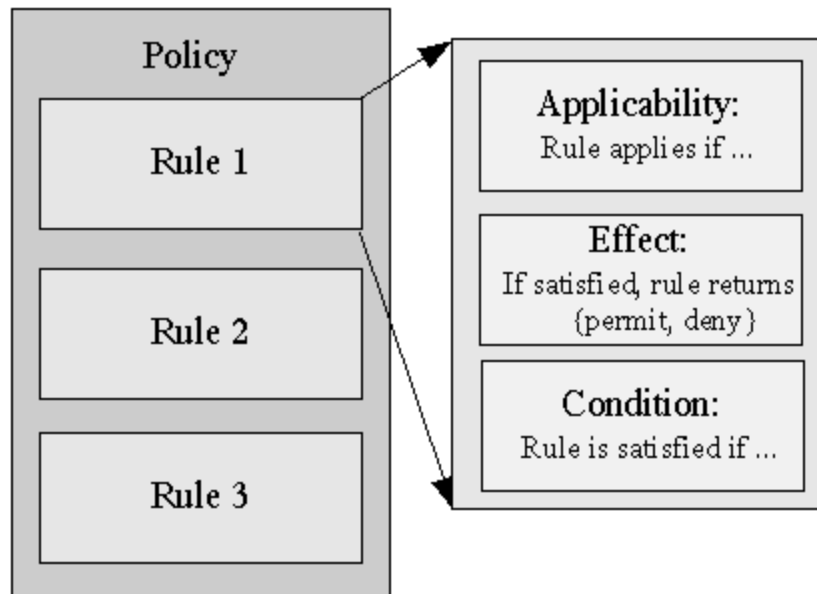


Illustration 3 Abstract components of a rule

#### EPAL rule

In the applicability section of an EPAL rule, only specific attributes may be used. There must be at least one of each of the following: *subject identifier[user-category]*, *resource identifier[data-category]*, *action*, and *purpose* attribute. The applicability section cannot refer to *environment[container]* attributes. The predicates in the EPAL applicability section are implicit: the values supplied must be equal to values in the request. The rule applies if at least one *subject identifier[user-category]*, one *resource identifier[data-category]*, one *action*, and one *purpose* attribute match the request.

In EPAL, the *Condition* predicates may refer only to *environment[container]* attributes; they may not refer to *subject identity[user-category]*, *resource identity[data-category]*, *action*, or *purpose identity[purpose]* attributes. Since the semantics of the applicability section specify that the rule is applicable if any one instance of a single *category/action/purpose* type (for example, *subject[user-category]*) matches the request, the fact that *Conditions* may not refer to these types of attributes means, for example, that an EPAL policy is unable to require a *subject[user]* to belong to more than one *[category]*.

An EPAL rule may contain *Obligations* to be returned if the rule applies and any *Conditions* are true. *Obligations* are discussed separately [below](#) in the “Obligations” section.

In EPAL rules, all attributes, *Conditions*, and *Obligations* are expressed as pointers into the

policy's *Vocabulary*, which is discussed separately [below](#) in the “Vocabulary” section.

## XACML rule

The applicability section of an XACML rule is called a *Target*. The predicates in this section may use any functions that return a Boolean result and compare the value of a request attribute's values to a literal value. The predicates may refer to any attributes. XACML does not distinguish between attributes indicating *group membership* or *identity [categories]* and other types of attributes in specifying whether an attribute should appear in the *Target* or in a *Condition* part of the rule.

The predicates in the *Target* are grouped by *subjects[users]*, *resources[data]*, *actions*, and *environment[containers]*. If the *Target* is missing, the rule is treated as being always applicable. If a particular type of group is missing (for example, there are no *subjects [users]*) the rule applies to all instances of that type of group (for example, to all *subjects [users]*). If there are multiple instances of groups of predicates of a given type, all predicates in at least one group for each type must be true. For example, if the applicability section of an XACML policy contained the following:

```
<Target>
  <Subjects>
    <Subject>
      <Predicate1>
      <Predicate2>
    </Subject>
    <Subject>
      <Predicate3>
      <Predicate4>
    </Subject>
  </Subjects>
  <Actions>
    <Action>
      <Predicate5>
      <Predicate6>
    </Action>
  </Actions>
</Target>
```

then the rule would apply to a given request if the following is true for that request:

```
((Predicate1 AND Predicate2) OR (Predicate3 AND Predicate4))
AND
(Predicate5 AND Predicate6)
```

In XACML, there is at most one *Condition*. If the *Condition* is missing, then the *Condition* is treated as implicitly true. If present, the *Condition* may contain either a single predicate

or a Boolean combination of predicates. A Boolean “and” combination of predicates may be used to achieve the effect of multiple *Conditions* in EPAL.

XACML also supports *Obligations*, but in XACML these are contained in the policy and not in the rule. *Obligations* can be associated with an individual rule, if needed, by creating a policy that contains only that one rule, plus the associated *Obligations*. XACML supports nesting of policies into *PolicySets*, and expressly made the *Policy* the unit of policy administration, leading to this requirement. The resulting functionality is equivalent.

### **Rules: summary**

EPAL supports all the rule functionality of XACML with the exception of

- being able to require a *subject[user]*, *resource[data]*, *action*, *environment [container]*, or *purpose* to belong to multiple groups,
- being able to use any attribute as part of the applicability section of a rule

XACML supports all of the rule functionality offered by EPAL.

In this aspect, therefore, EPAL supports only a subset of the functionality offered by XACML.

### **Applicability of rules**

This section describes the *applicability* section of a rule in XACML and in EPAL. This is the section that specifies the *Targets* or pre-conditions for the rule to apply to a particular authorization decision request.

#### **XACML applicability**

XACML attaches an optional pre-condition or *Target* to each policy and rule, specifying certain attributes in the request that must match specified values in order for the policy or rule to be applied. If no *Target* is specified, then the policy or rule always applies. A *Target* can reference any attribute. XACML *Targets* use a subset of the overall *Condition* functions that are suitable for use in matching. These include comparison operators (greater-than, etc.) as well as equality operators. This allows a *Target[scope]* to be used for indexing policies or for rapidly determining which policies or rules might apply to a request.

#### **EPAL applicability**

EPAL has similar concepts, although they are not identical. In EPAL, a policy has an

optional *Target[global-condition]*. that determines the applicability of the entire policy. EPAL does not limit *[global-condition]* functions to a subset of the available functions suitable for indexing as XACML does, and so a *[global-condition]* is not suitable for use in indexing policies.

Each EPAL rule has a *Target[scope]* that is a simple equality match on some set of *subject[user-categories]* attributes, *resource[data-category]* attributes, *actions*, and *purpose* attributes. An EPAL *Target[scope]* may not reference *environment[container]* attributes.

### **Applicability: summary**

In this aspect, XACML supports all the EPAL functionality. EPAL supports only a subset of the functionality provided by XACML.

### **Condition**

In both XACML and in EPAL, specific requirements for access to a *resource[data-category]* are expressed in a Boolean *Condition* element that may be recursively composed of Boolean expressions and functions. XACML and EPAL have minor differences in *Condition* syntax, but these are functionally equivalent.

### **Condition: summary**

In this aspect, XACML and EPAL are equivalent.

### **Nested policies**

For managing complex policies, it is useful to allow one policy to be made up of multiple sub-policies, which are evaluated and possibly authored and maintained separately. Sub-policies allow different departments to manage their own policies, for example, yet ensure that all policies are taken into account in reaching an authorization decision. Even where policies are administered centrally, the use of sub-policies allows policies to be grouped in meaningful ways into smaller units, making it easier and more efficient to select the policies that apply to particular *Targets* or to examine policies related to particular targets.

This leads to a requirement for nested policies. Since different sub-policies may return different *results[rulings]* for a given access decision request, there is an associated requirement for a conflict resolution mechanism. These conflict resolution mechanisms are discussed separately in the next section titled “Result conflicts”.

## EPAL nested policies

EPAL does not allow policies to be nested; each policy is separate, with no language-defined mechanism for combining results from multiple policies that may apply to a given request.

## XACML nested policies

XACML allows policies to be nested. A policy, including all its sub-policies, is evaluated only if the policy's *Target* is satisfied. In combination with the XACML ability to support policy references (see the “Policy references” section), nested policies can be used to support decentralized policy management. The XACML combining algorithms, discussed below in the “Result conflicts” section, allow policy writers to define how conflicting results from multiple policies are to be reconciled. For example, different departments within an organization can maintain their own policies for a given resource, and a master policy can reference each of these policies, requiring that all of them (or one of them, or some other combination) be satisfied. For example, policy “A” can be made up of two sub-policies “B1” and “B2”, and can require that both return a *result[ruling]* of *Permit[allow]* in order for “A” to return *Permit[allow]*. In turn, “B2” may be made up of three sub-policies “C1”, “C2”, and “C3”. “B2” may require that none of “C1”, “C2”, or “C3” return “Deny” if “B2” is to return “Permit[allow]”. “B1”, “B2”, “C1”, “C2”, and “C3” may be included directly in policy “A” or “B2”, respectively, or one or more of them may be included by reference.

## Nested policies: summary

XACML supports nested policies. EPAL does not.

## Result conflicts

Since policies in both XACML and EPAL may contain multiple rules, and since rules may evaluate to different *results[rulings]* given the same request, there must be a way to determine the *result[ruling]* to be returned from the policy evaluation if rules conflict. An XACML policy may also contain multiple sub-policies; since these also could return different *results[rulings]*, there must also be a way to resolve sub-policy conflicts. XACML and EPAL handle this problem differently

## EPAL result conflicts

In EPAL, rules are evaluated in policy document order. The *effect[ruling]* of the first rule that applies (see the “[Rule](#)” section) is returned as the *result[ruling]* of the policy. Subsequent rules are ignored. Barth, Mitchell, and Rosenstein [12] have described some of the language issues that arise from this choice of conflict resolution algorithm. One

significant issue is that two policies cannot be merged automatically (such as when two organizations merge).

Each EPAL policy also specifies a default *result[ruling]* that is to be returned if no rules apply to a given request.

### **XACML result conflicts**

In XACML, conflicting results are resolved by what is called a *combining algorithm*. A *combining algorithm* determines which rules or sub-policies are to be evaluated, how various combinations of results are to be resolved, and how any *Obligations* associated with policies are to be handled. A *combining algorithm* may also make use of optional parameters supplied to it in the policy. XACML defines several standard *combining algorithms*, but the mechanism is extensible to allow deployers to define additional algorithms. For example, one standard XACML *combining algorithm* says “return *Deny* if ANY sub-policy (or rule) returns a result of *Deny*.” Another algorithm says “return the *result[ruling]* of evaluating THE FIRST sub-policy that applies to the request.” Yet another algorithm says “permit access if AT LEAST ONE sub-policy or rule returns a result of *Permit[allow]*.” A third says “permit access only if none of the sub-policies or rules returns a result of *Deny*.”

XACML sub-policies or rules can return a result of *Indeterminate* if some error occurs during evaluation such that the policy engine is unable to determine a valid result for evaluating the sub-policy or rule. Each *combining algorithm* specifies how to combine every possible combination of *Permit[allow]*, *Deny*, *NotApplicable*, and *Indeterminate*.

### **Result conflicts: summary**

XACML supports all the EPAL functionality. EPAL supports only one of the ways XACML provides for resolving result conflicts: return the result of the first applicable rule. In this aspect, therefore, EPAL supports only a small subset of the functionality offered by XACML.

## **Policy references**

### **EPAL policy references**

EPAL does not support *policy references*.

### **XACML policy references**

XACML allows nested sub-policies to be included either by reference or by physical

inclusion. For example, policy “A” may contain two sub-policies “B1” and “B2”. These sub-policies could either be physically included in policy “A” or one or both could be included by a reference to its *policy-id*, a unique identifier associated with each XAML policy. In combination with the XACML ability to support nested policies (see the “Nested policies” section), *policy references* can be used to support decentralized policy management.

### Policy references: summary

XACML supports *policy references*. EPAL does not.

## Vocabulary

A policy's *vocabulary* is the set of attributes used in specifying the policy. XACML and EPAL handle *vocabularies* differently.

### EPAL vocabulary

A major feature of EPAL 1.2 that is not in XACML is a *[vocabulary]* element in a policy. This element points to a separate file that defines not only the collection of attributes, but also the *Condition* predicates, and *Obligations* used in the the policy. Actual policies refer to all these items by reference to their definitions in the *[vocabulary]* element.

An EPAL *[vocabulary]* can specify several different types of attributes:

- A *subject[user-category]* attribute represents the identity of a “category of individuals that can access data” and that are “distinct from a privacy point of view”. The EPAL 1.2 specification states that “most policies should include a distinguished *user-category* called '*data-subject*' that represents the data subject who's [sic] personal data has been collected. This *user-category* can then be used to define the access rights of the individual, e.g., whether the individual can read and/or update its data.”
- A *resource[data-category]* attribute is “used to distinguish classifications of collected data that need to be treated differently from a privacy point of view.”
- *Purpose[purpose]* attributes “state the purposes for which data is used or is going to be used.”
- An *action* attribute specifies an action that may be permitted or denied on a *resource[data]*.
- *Environment[Container]* attributes are arbitrary other attributes associated with all *subjects[user-categories]*, *resources[data-categories]*, *actions*, and *purposes*.

*[Category]* and *purpose* attributes can be hierarchical. For example, the *[user-category]* called “*health care provider*” could have child *[user-categories]* called “*physician*”,

“nurse practitioner”, etc.

A *[vocabulary]* can specify multiple *[containers]*. Each separate *[container]* can be used to specify a collection of attributes that can be obtained together from a single source, or that might represent a subset of attributes that would be used by a given *Condition* in a policy rule. References to attributes in the predicates or functions of a *Condition* contain the attribute's *[container-id]* and *attribute-id*. EPAL 1.2 Section 3.7 states “We require that all the container instances that are needed for condition evaluation are available at the time of authorization, i.e., if a rule is applicable but the container instance needed to evaluate the condition is missing, this is defined to be an error.” Note that “container instances” must be available, not merely instances of specific attributes in the container that are referenced.

The *[vocabulary]* element also specifies the issuer of the attributes and other descriptive information. Each attribute in the *[vocabulary]* is tagged as to whether its retrieval is being audited. None of this information is used in policy evaluation. The *audit* tag indicates that the data retrieval mechanism should log the conditions under which the datum is being retrieved. The EPAL *[vocabulary]* element also defines *Obligations* that can be returned in association with that *[vocabulary]*. *Obligations* are discussed more thoroughly [below](#) in the “Obligations” section.

### **XACML vocabulary**

XACML policies do not describe the *[vocabularies]* they use. In the XACML model, *[vocabularies]* are defined separately from the policies and are outside the scope of the XACML language. Policies must include all the information required to identify (but not retrieve) and evaluate each attribute used in the policy.

Without extending the XACML language, it is easy to define a *[vocabulary]* attribute in XACML to hold identities of *[vocabularies]* used in a rule. Requesters can then be required to include the identifiers of the *[vocabularies]* for which they are providing attribute values. Rules will apply only if the rule and the decision request agree on the *[vocabularies]* being used. For example, a rule can state

```
Condition:  
  values for the "vocabulary" attribute must include "vocabulary#23"  
and  
  ....other conditions....
```

Each authorization decision request in this model must include values of the “vocabulary” attribute that represent all of the externally defined vocabularies that define attributes used in the request.

## Vocabulary: summary

*[vocabulary]* seems like a potentially useful concept, and the idea of *[containers]* to collect attributes that might be retrieved as a group is also potentially useful. EPAL has defined a schema for *[vocabulary]* that might be useful for use with XACML or other policy languages.

Specifying *[vocabularies]* in policies may be overly rigid in practice, however. For example, a policy may say that either a *Challenge/Response* Attribute or a *Private Key Authenticator* Attribute must be supplied. If the *[vocabulary]* lists both of these in the same *[container]*, then clients that support only a *Challenge/Response* or only an *Authenticator* will fail. It would be possible to create separate *[containers]* and separate rules for each combination of attributes, but for complex policies, the complexity of creating the *[containers]* may be significant; the designer of the *[vocabulary]* must be aware of the structure of the policies. Changes to policies may require changes to the structure of the *[vocabulary]*.

Another issue with *[vocabularies]* is that the corporate entities responsible for defining *[vocabularies]* may be different from the entities responsible for defining policies. Keeping the two coordinated may be difficult under these circumstances.

EPAL 1.2 Section 3.7 “Design Notes” states, “ensuring ... consistency between the policy and its environment is out of the scope of EPAL. Whenever an EPAL implementation needs a context that is not available, it returns an error.” *[Vocabularies]* help with this problem, but do not solve it. They list the attributes that are needed, but do not ensure that the values will actually be available.

XACML does not support the EPAL *[vocabulary]* functionality as a first-class concept, but is capable of supporting some of the functionality by defining and using an attribute to represent the identity of a vocabulary of attributes, which can be done without extending or modifying XACML itself. EPAL supports all the XACML functionality. In this aspect, therefore, XACML supports only a subset of the functionality supported by EPAL.

It is not clear, however, that *[vocabularies]* embedded in policies are worth the drawbacks listed above, especially given their inability to ensure consistency.

## Attribute mapping

In both languages, attributes are used to describe the access request. A policy then controls access by stating which combinations of values for these attributes are permitted or denied. Both languages support an attribute information model in which attributes are supplied to the policy evaluation engine based on a conceptual *identity + data type + value* form, such that the engine does not have to deal with external formats directly. This allows language

implementations to handle multiple external attribute formats by providing a component that can map these various external formats to the conceptual format used by the policy language. In both languages, using the wording of the EPAL 1.2 specification, it is the responsibility of the application using the language to “map its own terminology to the privacy-policy-specific one by means of a deployment mapping.”

### **EPAL attribute mapping**

EPAL retrieves attribute values based on a reference to their unique identifiers as specified in the policy's *[vocabulary]*, along with the identifier of the *[vocabulary] [container]* in which the attribute is defined.. Each attribute specification includes the data type of the attribute's values.

If the content of an actual resource being accessed is to be used in EPAL policies, there must be an *environment[container]* attribute defined to correspond to each piece of information in that resource. The linkage between these *environment[container]* attributes and instances of the actual resource must be defined and managed outside of the policy - a policy is unable to directly reference information in the data being accessed.

EPAL 1.2 treats attribute values as *bags* - unordered collections that may contain duplicates - of attribute values. XACML also uses bags for attribute values. While bags did not originate with XACML, the decision to define attribute values as bags was controversial, and there were many other options. The fact that EPAL (which was defined later) also uses bags suggests, but does not prove, that EPAL took this design choice from XACML, particularly since the previous version of EPAL [2] explicitly used the XACML syntax for its policy *Conditions*.

### **XACML attribute mapping**

Like EPAL, XACML attribute values are treated as bags. XACML retrieves attribute values based on a reference to the attribute's unique identifier, data type, and, optionally, issuer.

XACML, in addition, supports an attribute information model in which attributes may be referenced via XPath expressions over information included in the conceptual decision request. This allows XACML to support attributes whose values are instances of XML complexType, and allows XACML to obtain attribute values from an XML document that is the actual resource being accessed. An example is the ability to include a copy of a patient record in an access request, and permit access to it if the identity of the *subject [user-subject]* in the request matches the name contained in the “<PatientName>”1 element of the record. This functionality will be increasingly important for privacy policies as more and more data is encoded using XML.

## Attribute mapping: summary

XACML supports all of the functionality offered by EPAL. EPAL does not support all the functionality offered by XACML. In this aspect, therefore, EPAL supports only a subset of the functionality offered by XACML.

## Attribute retrieval

Both XACML and EPAL expect attribute value retrieval to be handled outside the scope of the language processor itself, and do not define when or how attribute values required by the evaluation of a policy are to be retrieved. The “decision request” formats in both languages are conceptual and need not be part of the actual implementation. Attributes in the conceptual request need not come from the Policy Enforcement Point (PEP), but may be retrieved on the Policy Decision Point (PDP) side from various sources.

The concept of having a “virtual” request, in which attributes are represented in a standard form “as if” they were present, but can actually be retrieved on request by the policy evaluator, was a controversial and difficult design decision for the OASIS XACML Technical Committee (TC). It has proven to be a real strength in applying XACML to many different types of application environments. The fact that EPAL 1.2 uses exactly the same approach to attribute retrieval suggests, but does not prove, that EPAL 1.2 took this design choice from XACML, particularly since the previous version of EPAL [2] explicitly used the XACML syntax for its policy *Conditions*.

The use of the EPAL [*vocabulary*] concept, which XACML does not have (but can partially support), can make it easier for the attribute retrieval component to know which attributes will be needed before evaluating a *Condition*. It also allows a collection of attributes to be retrieved together, which may allow optimizations. The EPAL 1.2 design notes for Section 3.7 “<container>: Abstract Definition of the Data to be Evaluated by Conditions” says “We require that all the container instances that are needed for condition evaluation are available at the time of authorization, i.e., if a rule is applicable but the container instance needed to evaluate the condition is missing, this is defined to be an error.” This means that attributes may be unavailable, even though defined in the [*vocabulary*]. It also means that values for all attributes defined in a given [*container*] must be retrieved whenever any one of them is retrieved, even if the *Condition* evaluation does not require all of the values. This seems like an unnecessary restriction in the language. If [*containers*] are to be used to optimize attribute retrieval, they must be designed carefully and in synchronization with the policy rules that will reference them.

XACML implementations are free to use optimizations in the retrieval of attributes, but they are not mandated by the language. An XACML implementation could, when a rule is determined to be applicable, attempt to retrieve values for all attributes referenced in the rule (determined by static analysis), but it is not an error if some of those attributes cannot

be retrieved. Failure to provide an attribute value is an error in XACML only if the subsequent policy evaluation actually requires an instance of that attribute's value. In most cases, a missing attribute value causes the rule *result[ruling]* to be *NotApplicable*. It is possible, however, to specify that an error is to be generated when an attribute value is not available when required during an evaluation.

### **Attribute retrieval: summary**

In this aspect, XACML and EPAL are virtually identical. EPAL mandates retrieval of all attributes in a *[container]* when any one of them is needed; XACML allows predictive retrieval of attributes, but does not mandate it.

### **XML attribute values**

EPAL does not support attributes that have values that are XML instances, and does not support XPath for referencing elements of attributes or of the resource that is being accessed. If a policy requires that one node of an XML resource must match a given value, then the EPAL *[vocabulary]* must define that specific node as a separate attribute with its own identifier. The attribute retrieval component that is out of scope for EPAL would have to handle the retrieval and mapping of all such attribute values.

XACML supports use of XPath as an option for dealing with attribute values that are instances of XML schema elements, as well as to support comparisons between a requested XML document node value and other values, either in or external to the document. In XACML, as in EPAL, most attribute retrieval is out of scope for the language itself, but XACML, with its support for XPath, is able to handle references to specific nodes in an XML instance as part of the policy itself. XACML, by handling this directly as an option, supports more flexible specification of attributes. It is interesting that an IBM representative to the XACML TC was the primary proponent for including XPath references in XACML.

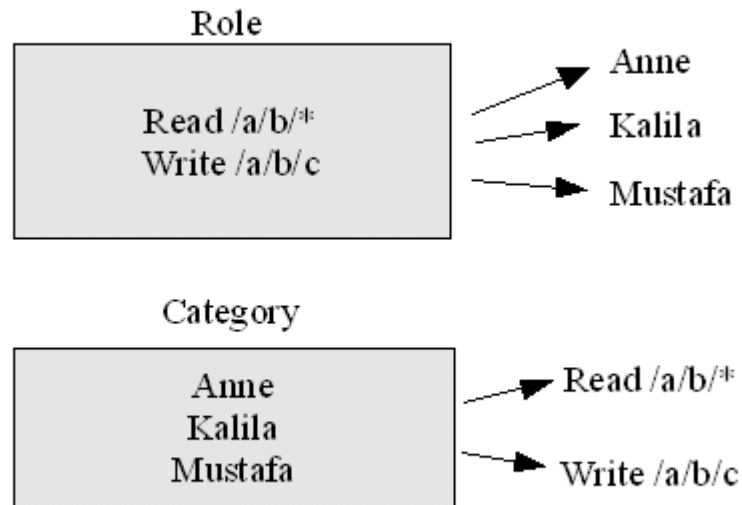
### **XML attribute values: summary**

XACML supports XML attribute values. EPAL does not.

### **Hierarchical roles**

A *role*, as defined by the 2004 *ANSI Standard for Role Based Access Control* [8], is a collection of permissions. It is intended to model the permissions needed to perform some function within an organization. A role differs from an *EPAL category* in that a *category* is a collection of entities, not a collection of permissions. A role may be associated with various entities, but does not contain or consist of those entities. A category may be associated with various permissions or prohibitions but does not contain or consist of those

permissions and prohibitions. Illustration 4 shows the difference between these two concepts. An ANSI Standard role also differs from a category in that a role contains only permissions, but not prohibitions (other role models have tried to deal with prohibitions). An EPAL category may be associated with both permissions and prohibitions.



*Illustration 4: Difference between a “role” and a “category”*

The ANSI Standard defines a *role hierarchy* as a partial order in which senior roles acquire the permissions of their juniors, and junior roles become associated with the user membership of their seniors. “General Hierarchical RBAC” is defined as supporting multiple inheritance, whereas “Limited Hierarchical RBAC” supports only single inheritance.

### **EPAL hierarchical roles**

EPAL does not support hierarchical roles as defined in the ANSI Standard. EPAL hierarchical *subjects[user-categories]* do not fit the ANSI model since the children of an EPAL *[category]* inherit permissions of their parent; parent *[categories]* do not inherit the permissions of their children. Instead, parents acquire the user membership of their juniors. A non-branching ANSI style role hierarchy could be modeled as an upside down EPAL *subject[user-category]* hierarchy, but in order to handle role hierarchies having multiple junior roles, an EPAL rule would need to specify each *[user-category]* in the hierarchy to which the rule applies individually.

## XACML hierarchical roles

The XACML 2.0 OASIS Standard includes the *Core and hierarchical role based access control (RBAC) profile of XACML 2.0* [7]. This profiles the use of XACML to support the ANSI Standard models for both “core” RBAC and hierarchical roles. This profile of XACML was designed in consultation with the authors of the ANSI Standard; it supports both General Hierarchical RBAC and Limited Hierarchical RBAC.

### Hierarchical roles: summary

XACML directly supports hierarchical roles. EPAL does not.

## Hierarchical categories

### EPAL hierarchical categories

For an explanation of the different between a *category* and a *role*, please refer to the previous section on “Hierarchical roles”.

EPAL uses special attribute types called *[categories]* for specifying a rule's *Target subjects [user-categories]*, *resources[data-categories]*, and *purposes*. *[Categories]* are different from *roles* in that they are collections of entities, not collections of permissions. *[Categories]* may be hierarchical. EPAL *actions*, *Obligations* and *environment [container]* attributes may not be hierarchical.

*Note: The EPAL 1.2 specification uses “child” and “parent” in its description of inheritance in hierarchical [categories]. I believe the text should read “descendants” and “ancestors” to be consistent with the semantics described, so I have interpreted the specification in that way.*

The semantics of EPAL hierarchical *[categories]* are described as:

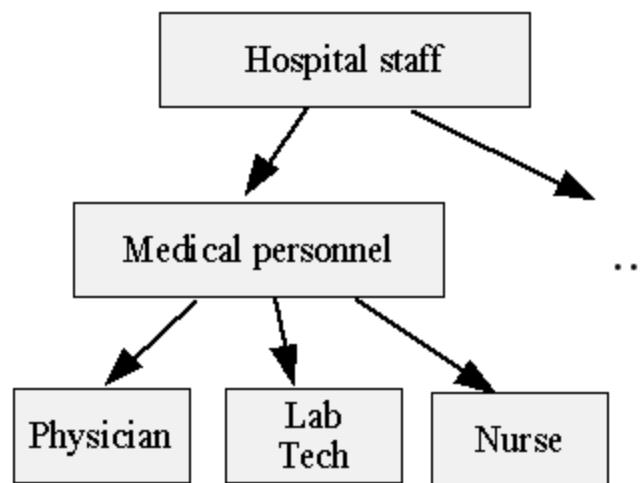
- “Groups”: a parent group *[category]* includes all members of its descendant groups *[categories]*.
- The descendants of any given node in the hierarchy inherit the permissions of the given node.
- All descendants and ancestors of any given node inherit the denials [prohibitions] of the given node.

Except for the upward inheritance of prohibitions, these semantics are similar to those of semantic class hierarchies: a *[category]* that is a child of another *[category]* inherits the semantics associated with its parent: it is a semantic subclass of its parent.

Note that these semantics differ from the standard definition of hierarchical roles, in which the parent of a node in the hierarchy inherits the permissions of its children.

- With hierarchical roles, if a member of a particular role has a given permission, then members of roles that are ancestors of that role will also have that permission. Hierarchical roles (in the ANSI Standard model) do not use prohibitions.
- With hierarchical *[categories]*, if a member of a particular *[category]* has a given permission, then members of *[categories]* that are descendants of that *[category]* will also have that permission. If a member of a particular *[category]* is denied some permission, then members of *[categories]* that are ancestors or descendants of that *[category]* will also be denied that permission.

Illustration 5 shows an example of hierarchical EPAL *subject[user-category]* attributes. The arrows indicate the direction in which permissions are inherited.



*Illustration 5 EPAL hierarchical user-categories*

In this example, all users having the “*Physician*”, “*Lab Tech*”, and “*Nurse*” *subject[user-category]* attributes inherit all permissions associated with users having the “*Medical personnel*” *subject[user-category]* attribute. All users having the “*Medical personnel*” *subject[user-category]* attribute inherit all permissions associated with users who have the “*Hospital staff*” *subject[user-category]* attribute. If a “*Physician*” is denied permission to do something, then “*Medical personnel*” and “*Hospital staff*” will also be denied that permission (but not “*Lab Tech*” or “*Nurse*”). If “*Medical personnel*” is denied permission to do something, then “*Hospital staff*”, “*Physician*”, “*Lab Tech*”, and “*Nurse*” will also be denied that permission, but descendants of any siblings of “*Medical personnel*” or their descendants.

If a particular EPAL rule applies to a given hierarchical *[category]*, then, if the rule returns an *effect[ruling]* of *Permit[allow]*, then the rule also applies to all descendants of the given hierarchical *[category]*. If the rule returns an *effect[ruling]* of *Deny*, then the rule also applies to all descendants and ancestors of the given hierarchical *[category]*. An example of rule applicability based on the previously illustrated hierarchy is shown in Illustration 6 .

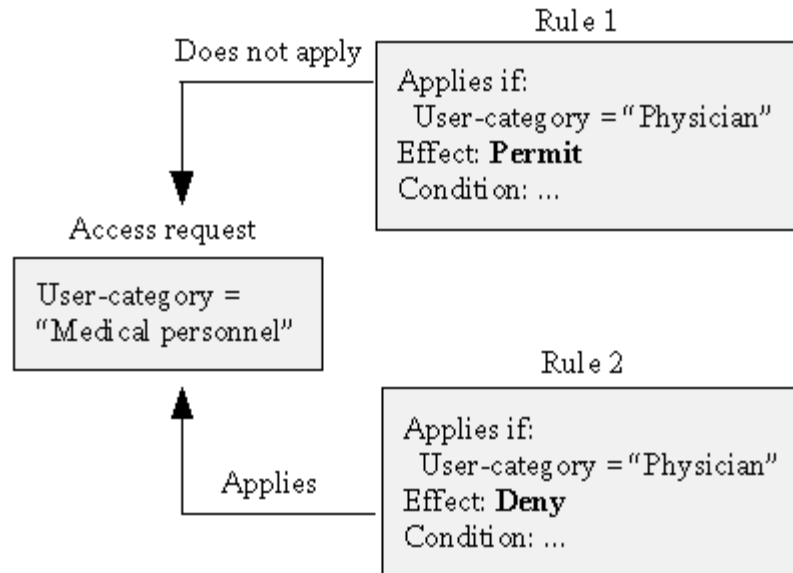


Illustration 6 EPAL rule applicability

In EPAL, a hierarchy is defined in the EPAL policy *[vocabulary]* using separate attributes linked together by a *parent* field.

A major problem with hierarchical *[categories]* in EPAL is that the semantics of hierarchical *[categories]* can conflict with the semantics of rule evaluation. For example, if a request comes from a member of *[User-Category]* “A”, and the first rule in the policy permits access to a resource to all members *[user-category]* “A”, but the second rule in the policy denies access to all members of *[user-category]* “A”, then the policy will return a value of *Permit[allow]*, since the *result[effect]* of the first applicable rule is used, even though the *[category]* semantics state that denials are inherited by all ancestors and descendants of the *[category]* that is denied. Similarly, if a particular requester is identified as a member of *[user-category]* “A” that is a child of *[user-category]* “B”, and a rule permits access to members of *[user-category]* B, then a result of *Permit[allow]* will be returned even if a subsequent rule specifically denies access to members of *[user-category]* “A”.

The processing of hierarchical attributes *[categories]* in EPAL is potentially very costly as,

conceptually, each combination of *[category]* attributes in all hierarchies must be evaluated. Optimizations are possible, but would be difficult given the semantics of *[categories]*.

EPAL hierarchies do not support multiple inheritance. For example, a “*nurse practitioner*” *[user-category]* cannot inherit from both a “*limited physician*” *[user-category]* and a “*nurse*” *[user-category]*.

### **XACML hierarchical categories**

XACML has no explicit built-in support for hierarchical *[categories]* of the type supported by EPAL, where parents consist of the collection of their children, and where prohibitions are inherited upward and downward, but permissions are inherited only downward. In order to support such hierarchical *[categories]*, an XACML rule currently would need to specify *Condition* functions individually for each attribute in the hierarchy to which the rule applies.

It would be possible to support general hierarchical groups of any type of attribute by defining three new attributes associated with any existing XACML attribute, such as the following:

- “urn:existing-attribute:group-and-ancestors”
- “urn:existing-attribute:group-and-descendants”
- “urn:existing-attribute:group-ancestors-and-descendants”

The XACML component responsible for retrieving attribute values (the “Context Handler”) must then be configured to know how to obtain all values for the “*existing attribute*” and for all its ancestors or descendants, respectively (just as the EPAL *[vocabulary]* provides this information to the EPAL processor). This assumes all the ancestor and descendant attributes are the same data type. Since XACML does not specify how attributes are obtained or from which sources, this functionality can be added without any extensions to the XACML language, and in fact was added for hierarchical resources in the *Hierarchical resource profile of XACML v2.0* [6] .

XACML will not assign any particular inheritance semantics to such hierarchical group attributes: semantics must be associated explicitly by the policy writer. For example, if a policy writer wants a *Deny* rule to apply to all requests from any member of a given attribute hierarchy, then the policy writer must explicitly use the “urn:existing-attribute:group-ancestors-and-descendants” attribute in stating the *Deny* rule *Conditions*. As a second example, if a policy writer wants a *Permit[allow]* rule to apply to all requests from members of any group associated with an attribute, or of any of its descendants, then the policy writer must explicitly use the “urn:existing-attribute:group-and-descendants” in stating the *Permit[allow]* rule *Conditions*.

If such hierarchical groups are provided, XACML will not run into the conflict between inheritance semantics and rule evaluation order that EPAL encounters unless the policy writer chooses a *combining algorithm* (such as *first-applicable*) that introduces such conflicts. It is up to the policy writer to associate the desired inheritance and override semantics to a particular attribute hierarchy.

### **Hierarchical categories: summary**

XACML does not directly support hierarchical [*categories*], although an implementation can support them without requiring changes in the language. The inheritance rules offered by XACML in this case can be tailored to the requirements of the policy writer. EPAL directly supports hierarchical [*categories*], but only in particular ways, and the EPAL functionality has inherent semantic conflicts.

### **Hierarchical resources / XML document resources**

Some resources, such as XML documents or file systems, are structured as a hierarchy. It is useful in policies to be able to associate *Permit[allow]* or *Deny Conditions* with individual nodes in the hierarchy or with entire subtrees of the hierarchy rather than having to repeat the same *Condition* for each node separately.

For example, it might be permissible for a physician to view everything in the patient visit history section in a patient's health record, but might not be permissible for that same physician to view anything in the patient's payment history section. Each section might consist of many nodes, and without specific support for resources structured as hierarchies, it would be necessary to repeat the *Permit[allow] Condition* for each node in the patient visit history section and to repeat the *Deny Condition* for each node in the payment history section.

It can also be useful to accept requests for an entire resource hierarchy (or subtree of a hierarchy), but to return separate responses for each node in the hierarchy, allowing support for filters that will return to the requester only those portions of the resource to which the requester has access.

In yet other cases, it can be useful to accept requests for an entire resource hierarchy (or subtree of a hierarchy), but to return a single response of *Permit[allow]* only if access to every node is permitted.

### **EPAL hierarchical resources**

In EPAL, in order to support a hierarchical resource, the resource must be expressed as a hierarchy of *resource[data-categories]* attributes in the EPAL [*vocabulary*]. It would be possible to translate XML document schemas into such a hierarchy of attributes with some

attribute naming convention, which would then allow EPAL to express rule *Conditions* such as “permit access to all nodes below node X”. This would not allow rule *Conditions* such as “deny access to all nodes below node X”, since prohibitions are inherited up as well as down the hierarchy. With such a translation, EPAL could also allow a request for access to an entire XML document, and to obtain a response of *Permit[allow]* only if access is permitted to all nodes in the document. It would not be possible in EPAL, however, to request access to an entire XML document and to obtain individual responses for each node. In order to obtain results for individual nodes, multiple requests would have to be submitted, one for each node in the document, and even then, access to a node will be denied if access to any of its ancestors or descendants is denied.

### **XACML hierarchical resources**

The XACML core syntax directly supports hierarchical *resources[data-categories]* that are XML documents, without requiring that each element in the document be expressed as a separate attribute, as in EPAL. XACML can also support other types of hierarchical *resources* that are not XML documents. Support for hierarchical XML *resources[data-categories]* is in the core specification for XACML 1.0 and 1.1, and has been expanded into a *Hierarchical resource profile of XACML v2.0* [6], which covers both XML documents and non-XML hierarchies. XACML 2.0 supports submitting a single request for an entire hierarchy, and getting back either a single response for the hierarchy as a whole or separate responses for each node in the hierarchy. This functionality is described in the *Multiple resource profile of XACML v2.0* [5].

Using these two profiles, XACML can return a fine-grained response to a request for access to an entire XML document or other hierarchy: the response will indicate exactly which elements of the hierarchy the *subject[user-category]* is permitted to access and which elements the *subject[user-category]* is not permitted to access. The individual nodes are identified using XPath expressions, so if desired, an XSLT [23] can then be used to expose only the permitted nodes of the XML document to the requester.

### **Hierarchical resources: summary**

In this aspect, EPAL functionality is extremely limited. XACML can support all the EPAL functionality. EPAL supports only a small subset of the functionality offered by XACML.

### **Subjects with multiple attributes**

In EPAL, as in XACML, a single subject may have multiple attributes *[user-categories]*, such as belonging to multiple roles or having an “*identity*” *[user-category]* attribute as well as an “*age*” *[container]* attribute. There are important differences in the ways such attributes are evaluated in the two languages, however.

## EPAL subjects with multiple attributes

In EPAL, there are two types of *subject[user]* attributes: *[user-category]* attributes and *[container]* attributes. The *[user-category]* attributes represent “categories of individuals” or “groups” to which the individual belongs. A *subject's[user's]* “*identity[data-subject]*” is also a *[user-category]* attribute. All other attributes, such as “age” or “*classification-level*” are *[container]* attributes. These two types of attributes are handled differently. The policy is conceptually evaluated once for each *[user-category]*, and a *Permit[allow]* result *[ruling]* is returned if ANY *subject[user-category]* attribute is permitted. Each such *[user-category]* evaluation may make use of any of the *[container]* attributes.

Since an EPAL policy is conceptually evaluated multiple times, there is a possibility that different evaluations will return different results. These differences are to be resolved using precedence rules, but EPAL 1.2 seems to be inconsistent in specifying these rules. EPAL 1.2 Section 5.2.3 “Operational Semantics” states that a policy is evaluated (conceptually) once for each *subject[user-category]* attribute specified in the decision request. If any *subject[user-category]* attribute evaluates to *Permit[allow]*, then a *result[ruling]* of *Permit[allow]* is returned as the authorization decision. If no *subject[user-category]* attribute evaluates to *Permit[allow]*, then, if any *subject[user-category]* attribute evaluates to *Deny*, a *result[ruling]* of *Deny* is returned. Yet the “Design Notes” for the same Section says “Note that processing of compound requests does not use precedence to resolve conflicts, i.e. a lowest-level 'Deny' on any combination of elements can overrule any highest-level 'allow'.” This is an inconsistency in a very important area of the language.

Since each *[user-category]* is evaluated separately, and since a *Permit[allow]* for any *[user-category]* results in a *Permit[allow]* result from the policy, there is no way to require a *subject[user]* to be a member of two *[user-categories]* simultaneously unless they are part of the same *[category]* hierarchy. For example, if “*research consortium member*” and “*physician*” are two different *[user-category]* attributes, then in order for a policy to require that a *subject[user]* be both a “*research consortium member*” and a “*physician*”, “*physician*” would have to be defined as an ancestor or descendant of “*research consortium member*”. Alternatively, either “*research consortium member*” or “*physician*” could be defined as a *[container]* attribute instead of as a *[user-category]*. A *[container]* attribute cannot be used to determine the applicability of a rule and cannot be part of a hierarchy of attributes. Note that *[container]* attributes are not explicitly associated with the *subject[user]*, as opposed to being associated with the *resource[data]*, *action*, or *environment*. To make associations, it is necessary to define some out-of-band agreement between the policy writer and the decision requester. Such an agreement might be expressed in the form of naming conventions for *[container]* attributes, such as “*data-age*” versus “*user-age*”.

## **XACML subjects with multiple attributes**

In XACML, each attribute is explicitly associated with either a *subject[user-category]*, *resource[data-category]*, *action*, or *environment[containers]*. There are no special types of attributes such as *[categories]*, and all attribute values included in or associated with a request are available during any given evaluation of a policy.

An XACML policy is explicitly evaluated once, with all attributes, including all *subject[user-categories]* attributes, available simultaneously. This allows an XACML policy to constrain access to a subject who must be in two different roles simultaneously; using the example above, an XACML policy can require that a requester be a “*research consortium member*” as well as a “*physician*” in order to gain access to a resource.

### **Subject with multiple attributes: summary**

In this aspect, XACML supports all the EPAL functionality. EPAL supports only a subset of the functionality offered by XACML.

## **Multiple subjects**

An access request done via computer inherently involves multiple principals or entities: there is the user under whose identity the computer application is running, there is the application itself (or a chain of application class methods terminating in the actual request), there is the platform on which the application is running (from which the request was received). There may be another user to whom the accessed resource will be sent, or one or more intermediaries through whom this request has been relayed. Some operations may require that there be two co-signers for a given request.

A system may need to constrain access based on attributes of more than one such principal. In EPAL and XACML, it is the *Subject[user]* that represents the principal(s) involved in making a request, so this introduces a requirement for supporting multiple *subjects[user]* associated with a single request.

### **EPAL multiple subjects**

EPAL does not support this concept of multiple *subjects[users]*: there is only one principal involved in each request. Although there may be multiple *subject[user-categories]* attributes in a request, they all refer to the same entity or principal.

### **XACML multiple subjects**

XACML supports multiple *subjects[users]* for a single access request, each representing a

separate principal or entity involved in making the request. Access can be granted based on attributes of a single such *subject[user]* or on attributes of some combination of *subjects[users]*. The XACML standard defines identifiers for many types of *subjects[users]*, but deployers can define and use additional types without changing either the language or an implementation of it. If multiple *subjects[users]* are not needed, then policies may make use of only a single *subject[user]*.

### **Multiple subjects: summary**

In this aspect, EPAL does not support the functionality provided by XACML.

### **Purpose attribute**

Privacy policies often state that a particular resource or item of data may be accessed only if the purpose for the access is consistent with the purpose for which the resource or item of data was collected and was intended to be used.

### **EPAL purpose attributes**

In addition to the *subject[user-category]*, *resource[data-category]*, and *action*, EPAL defines a top-level element called a *purpose*. A *purpose* is required in every request, and the values for *purpose* may be hierarchical.

This is one area where EPAL seems tailored to privacy, and not to general access control. While it may make sense to require a *purpose* for a privacy policy, a purpose is not needed for all access control policies. The EPAL 1.2 specification even acknowledges this explicitly: “Unlike access control, the <purpose> is part of an EPAL authorization query. Without knowing the purpose of an access, authorization cannot be decided. As a consequence, any system using EPAL must be able to determine a purpose before asking the EPAL engine to evaluate a given policy.” The EPAL 1.2 specification suggests, however, that every *[vocabulary]* include an *otherPurpose [category]* to allow for policies pertaining to *purposes* that the *[vocabulary]* did not define.

Since the EPAL *purpose* attributes are independent of a particular *resource[data-category]*, if a particular EPAL rule applies to multiple *resources[data-categories]*, then all *purposes* in the *rule* must apply to all *resources[data-categories]* equally.

### **XACML purpose attributes**

XACML 2.0 supports two *purpose* attributes as described in the *Privacy policy profile of XACML v2.0* [4]. The first is a *resource[data-category]* attribute for specifying the purpose for which the resource was collected. The second is an *action* attribute specifying

the purpose for which the resource is being requested. Use of these attributes is not required, but may be used where needed. The XACML *purpose* attributes are not inherently hierarchical, but could be treated as hierarchies as described above in the “Hierarchical categories” section.

### **Purpose attribute: summary**

In this aspect, XACML supports all the EPAL functionality. EPAL supports only a subset of the functionality provided by XACML.

### **Error handling**

Errors inevitably occur during the processing of policies. A language can either provide control over how errors are handled or can treat errors as situations that terminate language processing with an error result.

### **EPAL error handling**

The EPAL specification provides no way of handling errors other than to “return an error”. For example, the specification requires that values be available for every attribute that is defined a particular *[container]* if a *Condition* requires one attribute in the *[container]*. However, since EPAL allows *[containers]* of attributes to be retrieved on an as-needed basis, and also allows attributes to be obtained via call-back mechanisms, errors due to missing attributes can still occur. If an attribute required to evaluate a policy is missing, EPAL 1.2 Section 3.7 says that the EPAL engine returns an error. The specification does not provide any way of handling errors generated during function evaluation, reference-not-found, etc., other than to “return an error”.

### **XACML error handling**

XACML supports a result of *Indeterminate* to indicate that a *result[ruling]* could not be determined due to some error. Errors might occur during the application of a function (an attempt to divide by 0, for example), or to an attribute value needed to evaluate a function that could not be obtained, or a referenced policy might be unavailable. In general, a missing attribute in an XACML *Target[scope]* makes the *Target[scope]*'s policy or rule *NotApplicable*, and a missing attribute in a *Condition* makes the *Condition*'s rule *Indeterminate*, but it is possible to modify this behavior if needed. XACML recognizes errors as a core concept, integrated with the XACML algorithms for combining results of different *rules* and policies. For example, an XACML policy using additive permissions may allow a *result[ruling]* of *Permit[allow]* to be returned because one sub-policy or rule evaluated to *Permit[allow]* even though evaluation of another sub-policy or rule resulted in an error (see the “Result conflicts” section for more information about XACML's algorithms for combining results). Policy-directed handling of errors, particularly those

that arise in a distributed environment, was a first-order design requirement for XACML.

### **Error handling: summary**

XACML supports policy-directed error handling as part of the language itself. EPAL does not.

### **Revision number**

EPAL supports the concept of a policy *revision-number* attribute along with a unique *policy identifier*.

XACML 2.0 also includes a revision number element in the XACML policy schema. All versions of XACML include use of unique *policy identifiers*.

### **Revision number: summary**

In this aspect, XACML and EPAL are equivalent.

### **Data types**

EPAL supports a strict subset of the XACML primitive data types. The schema makes it fairly clear that this set was selected from the XACML 1.0 types (Appendix B.4), since there is a comment in the EPAL 1.2 schema\_saying:

```
“<!-- Four data types (dayTimeDuration, yearMonthDuration, x500Name, and rfc822Name) are ignored since there are no corresponding single types. -->”
```

The four data types mentioned are all included in XACML, and x500Name and rfc822Name are actually defined in XACML.

### **Data types: summary**

In this aspect, XACML is a strict superset of EPAL. EPAL supports only a subset of the functionality supported by XACML. The EPAL data types appear to have been borrowed without attribution from XACML.

### **Functions**

EPAL 1.2 Appendix 5.1 “EPAL Functions” lists the functions supported in EPAL *Conditions*. This list appears to be taken from XACML 1.0 (Appendix A.14), with only minor changes. Exactly the same set of arithmetic functions is defined, with exactly the

same definitions (including support for two or more attribute values as arguments to the two *add* functions), and listed in exactly the same order as in the XACML specification. The string conversion functions have slightly different names, and include one function not in XACML (but it is the converse of an XACML function). The numeric conversion and bag functions are identical, with the exception that the conversions from a single-element bag to a single value have different names. The equality, string comparison, and numeric comparison predicates have exactly the same names, and only one minor difference in the order in which the predicates are specified. The bag predicates are almost identical, differing only slightly in name. The *regex-string-match* function has the same name (changed in XACML 2.0 to *string-regex-match* for consistency with other names), but the definition of the result is different, and is probably an error in EPAL: EPAL 1.2 says the first argument is a regular expression and the second argument is a general string, but then EPAL says the values are matched according to the “*fn:matches*” function in “*XQuery 1.0 and XPath 2.0 Functions and Operators*” [15]. “*fn:matches*” specifies that the first argument is the general string and the second argument is the regular expression. XACML does not have this error.

Another design choice that appears to be taken from XACML is the use of strongly typed functions: rather than a single *add* function, there is an *integer-add* and a *double-add*, for example. This choice is reflected in the XACML naming convention for functions, which EPAL retains: `<data type>-<function>`. This is not an obvious design decision, and involved much discussion and controversy in the XACML TC.

EPAL references a newer snapshot of *XQuery 1.0 and XPath 2.0 Functions and Operators* [15] than does XACML, so there may be minor differences in semantics in application of the functions.

XACML 2.0 has defined additional match functions that are not included in EPAL. EPAL also does not support the XACML “higher-order functions”.

### **Functions: summary**

In this aspect, EPAL supports only a subset of the functions supported by XACML. The EPAL function definitions appear to be an unacknowledged borrowing from XACML.

### **Obligations**

*Obligations* are values returned with a *result[ruling]* that represent operations that must be performed by the Policy Enforcement Point (PEP) in addition to enforcing the access decision.

There are four differences in the way *Obligations* work in EPAL and in XACML.

1. In an EPAL rule, *Obligations* are stated by referencing an *Obligation* that has been defined in the *[vocabulary]* element associated with the policy; in XACML, *Obligations* are completely defined in the policy containing the rule itself.
2. EPAL *Obligations* can include explicit parameters. This is needed because an EPAL rule references an *Obligation* that is defined in the *[vocabulary]*. References from different rules may require variations in that *Obligation*. XACML *Obligations* do not need parameters because XACML *Obligations* are specified in the policy where they apply, and can include any variations directly.
3. In an EPAL *simple request*, *Obligations* associated with only one rule will be returned. The response to a single XACML decision request, however, may return *Obligations* associated with multiple rules, since a policy's *combining algorithms* may require that multiple rules be evaluated. In XACML, *Obligations* associated with all rules that are evaluated and have an *effect[ruling]* consistent with the final *result[ruling]* are returned. The XACML *combining algorithms* specify exactly how *Obligations* are combined, so desired behavior can be obtained by creating a new *combining algorithm* if necessary.
4. In an EPAL *compound request*, *Obligations* associated with multiple rules may be returned; each set of *Obligations* is identified by the rule-id of the rule that specified those *Obligations*. The *Obligations* are not, however, associated with the specific *resource[data-category]*, *action*, or *purpose* that caused those *Obligations* to be returned. In an XACML request for multiple *resources[data-categories]*, a separate response for each *resource[data-category]* is returned, with its associated *Obligations*. XACML *Obligations* are identified by the *resource[data-category]* to which they apply, but not by the rule-ids of the rules that specified those *Obligations*.

The only functional difference between these two ways of handling *Obligations* is that, in XACML, *Obligations* are identified by the *resource[data-category]* to which they apply, whereas in EPAL, *Obligations* are identified by the rule from which they came. An XACML *Obligation* could, by convention, contain the rule-id of its rule, so XACML is capable of handling the EPAL functionality. EPAL, however, is not able to express the *resource[data-category]* with which a set of *Obligations* is associated. The remaining differences are a matter of style and of compatibility with other design choices in the language.

Note that the definition of *Obligations* in XACML was a contribution from the IBM representative to the XACML TC. At the time XACML 1.0 was under development, IBM has declared a patent it holds related to *Obligations* to OASIS, but did not offer royalty-free licensing terms for use in XACML implementations until August of 2005. As a result, the XACML TC made support for *Obligations* in XACML optional, even though the IBM

patent appears to cover only a specialized implementation.

### **Obligations: summary**

In this aspect, XACML supports all the functionality of EPAL. EPAL supports only a subset of the functionality provided by XACML.

## **Multiple responses**

### **XACML multiple responses**

XACML supports returning multiple responses to a single request when the request refers to accessing a hierarchical resource or to multiple resources. XACML returns a separate response for each element in the hierarchical resource, or for each separately requested resource. For example, the resource might be a *patient record* with two sub-sections: *account information* and *medical information*. A subject in role *case-supervisor* might be allowed to view the medical information in the patient record, but not the account information. In this case, XACML would return three responses: one with a *result[ruling]* of *Permit[allow]* for the *patient record* element (that is, *patient record* meta-data), one with a *result[ruling]* of *Deny* for *account information*, and one with a *result[ruling]* of *Permit[allow]* for *medical information*.

### **EPAL multiple responses**

EPAL does not support multiple responses to a single request.

### **Multiple responses: summary**

In this aspect, EPAL does not support functionality that is supported by XACML.

## **Status as a standard**

### **XACML status as a standard**

XACML 1.0 was approved as an OASIS Standard on 6 February 2003. XACML 1.1 was approved as an OASIS XACML TC Committee Draft on 24 July 2003; it was not progressed to OASIS Standard because the changes were relatively minor. XACML 2.0 was approved as an OASIS Standard on 1 February 2005. XACML 2.0 includes several profiles, including a *Privacy policy profile of XACML v2.0* [4], which were also approved as OASIS Standards on 1 February 2005.

## **EPAL status as a standard**

EPAL 1.2 has been submitted to the W3C for consideration, but no further action has been taken by the W3C. At this point, EPAL is a proprietary IBM specification with no standard status. It has not been accepted for progression to a standard by any standards body.

### **Status as a standard: summary**

XACML is an approved OASIS Standard. EPAL is a proprietary IBM specification that has not been accepted for progression to a standard by any standards body.

## **5. Conclusions**

We are now ready to address the questions raised in the introduction.

### **What are the differences between EPAL 1.2 and XACML 2.0?**

EPAL is not just XACML with a different namespace – EPAL has structured some of its functionality in significantly different ways from XACML. But, in almost every area, the functionality of XACML 2.0 is a superset of EPAL 1.2. Where the two languages differ, the EPAL differences often result in less functionality than XACML has. In many cases, the EPAL 1.2 differences from XACML make construction of flexible or scalable privacy policies impossible or difficult.

There are major features contained in XACML that are not in EPAL 1.2:

- The ability to combine results of multiple policies developed by potentially independent policy issuers.
- The ability to reference other policies as part of a given policy.
- The ability to specify conditions on multiple subjects that may be involved in making a request.
- The ability to return separate results for each node when access to a hierarchical resource is requested.
- Support for subjects who must simultaneously be in multiple independent hierarchical roles or groups.
- Policy-directed handling of error conditions and missing attributes.
- Support for attribute values that are instances of XML schema elements.
- Support for additional primitive data types (including X.500 Distinguished Names, RFC822 names, and IP addresses).

The only significant functions added by EPAL that are not in XACML are:

- The concept of a policy *[vocabulary]*. The most significant part of this functionality (identification of the sets of attributes used by a policy) can be supported by XACML, but has not been defined in a standard way to date. In complex policies, EPAL's association of specific groups attributes (*[containers]*) in a *[vocabulary]* with particular rule *Conditions* may actually add to complexity by requiring *[vocabulary]* managers to be aware of the structure of the rules in the policy.
- Hierarchical categories with defined inheritance characteristics. XACML supports different types of hierarchies (such as roles and hierarchical resources) directly, and is capable of supporting EPAL-style hierarchical categories with no change in the language. The EPAL hierarchical categories have inherent semantic conflicts.

There are numerous places where EPAL 1.2 appears to have borrowed concepts, design elements, terminology, functions, and data types from XACML. These are described in the detailed comparisons above. These borrowings are not acknowledged; there is no mention of or reference to XACML anywhere in the EPAL 1.2 specification.

### **Which language is better for expressing privacy policies?**

EPAL contains no privacy-specific features that are not already supported in XACML. EPAL lacks significant features that are included in XACML and that are important in many enterprise privacy policy situations.

XACML and the *Privacy profile of XACML* have already been accepted as OASIS standards. XACML has been widely deployed. XACML is designed for general-purpose access control as well as for privacy policies, so with XACML these two closely related policy types can be integrated. There are several publicly available implementations of XACML, in various languages, including at least one open source implementation [9] with a BSD license. There seems to be no reason to choose EPAL 1.2 over XACML as a privacy policy language.

### **Which language is better for expressing access control policies?**

XACML is a functional superset of EPAL 1.2. It was designed for enterprise access control, and specifically addresses issues that arise in distributed systems. It was developed in a public standards group, based on publicly gathered requirements from many sources. It went through an extensive public review prior to its adoption as an OASIS Standard. Its semantics have been formally described [10]. It has been revised based on actual experience with its use over a period of several years, reflecting a level of maturity in the language.

XACML contains significant features that are missing from EPAL 1.2. These are important

in many enterprise access control policy situations, just as in many privacy policy situations. Many of the minor differences between EPAL 1.2 and XACML actually make it harder to express access control policies in EPAL 1.2.

One feature of EPAL is clearly targeted solely at privacy policies, and not at access control in general. EPAL requires each request to include a *purpose* attribute. This attribute is not meaningful for all access control requests. XACML 2.0 supports *purpose* attributes (see the *Privacy policy profile of XACML v2.0* [4]), but does not require them in policies that do not need them.

The EPAL 1.2 specification itself states that it is not a general access control language:

“Unlike access control, the <purpose> is part of an EPAL authorization query. Without knowing the purpose of an access, authorization cannot be decided. As a consequence, any system using EPAL must be able to determine a purpose before asking the EPAL engine to evaluate a given policy.”

The two aspects of EPAL 1.2 that XACML does not explicitly address are

1. the specification of the *[vocabulary]* of attributes used by a given policy or rule. This feature has limitations described in the “Vocabulary” section above, but if desired, can be supported in XACML.
2. category hierarchies. This feature can be supported in an implementation of XACML without changes to the language as described in the “Hierarchical categories” section above.

XACML has already been accepted as an OASIS Standard. XACML has been widely deployed [24]. XACML is designed to support centralized or decentralized policy management. XACML is designed to support both general access control and privacy policies, allowing these closely related policy types to be integrated. There are several publicly available implementations of XACML, in various languages, including at least one open source implementation [9] with a BSD license. There seems to be no reason to choose EPAL 1.2 over XACML as an access control policy language.

## **Should EPAL become a standard?**

XACML is both a more comprehensive access control policy language than EPAL 1.2, and a full-featured privacy policy language. It has the important features required by both types of policies, including major features not supported by EPAL 1.2. It has been publicly reviewed and formally described [10]. XACML is already an approved OASIS Standard. There are several publicly available implementations of XACML, in various languages, including at least one open source implementation [9] with a BSD license. XACML has an active community of users and developers who are continuing to expand, improve, and

apply the language. See the list of references, including publicly announced adoptions and deployments, on the *OASIS XACML TC Home Page* [11] .

EPAL adds no significant functionality, and in particular no new privacy-specific functionality, over what is already supported in XACML, while XACML contains significant additional functionality missing from EPAL.

For all these reasons, it would be a mistake to standardize EPAL. Multiple, competing “standards” that address the same problem space, particularly when they are so similar, are a detriment to the industry.

Given the very strong similarities between EPAL 1.2 and XACML, the EPAL developers could have extended XACML or worked with the XACML TC to modify XACML if they felt additional or alternative functionality was needed. Instead, the EPAL authors chose to develop their specification outside a standards group. This decision is especially dismaying since the “Mission Statement” in Section 1.1 of the EPAL 1.2 specification includes “leverage existing standards and technologies” as one of its three mission components.

It is not too late for the EPAL authors to work with the XACML TC. The OASIS XACML Technical Committee has minuted an interest [13] in inviting the EPAL authors to work with the XACML TC, and the TC co-chairs have issued this invitation. The EPAL authors have replied that they are not available to participate in the XACML TC.

## 6. Terminology

- **Attribute:** a variable used in a policy to represent a *vocabulary* item. At the time a policy is evaluated, each Attribute that is required for the evaluation must have one or more values assigned to it. These values are typically supplied in the authorization decision request, but may come from external sources.
- **category:** as used in EPAL, a *category* is an identity associated with a collection of users, data (resources), or purposes intended to model the *categories* of users, data, and purposes referenced in high-level privacy policies. *Categories* can be hierarchical: a *category* can be associated with the identities of lower-level *categories* as well as with specific users, data, or purposes. In this case, the collections of users, data, or purposes associated with the lower-level *categories* are considered part of the higher-level *category* that includes them just as if they had been included directly in the higher-level *category*. See the “Hierarchical categories” for more information. Compare to *role*, which is a collection of permissions.
- **directly-enforceable policy language:** a language is *directly-enforceable* if it

supports matching a specific access request against a policy, and determining whether that access request is to be permitted or denied. A P3P [14] policy is not directly-enforceable; it is designed to express a site's high-level practices, but not how those practices are applied to specific access requests for particular documents or other resources by particular people under particular conditions. EPAL and XACML are both *directly-enforceable*: a specific access request can be evaluated against an EPAL or XACML policy; the policy evaluation engine can determine whether that request should be permitted or denied in accordance with the policy.

- **policy**: a set of rules governing access to resources.
- **Policy Decision Point (PDP)**: an abstract entity in the policy enforcement model defined by the IETF [20][21] that is responsible for evaluating policies to produce authorization decisions. A PDP accepts an authorization decision request from a Policy Enforcement Point, retrieves applicable policies, and evaluates the policies against the authorization decision request to produce an authorization decision, which is returned in a response to the Policy Enforcement Point.
- **Policy Enforcement Point (PEP)**: an abstract entity in the policy enforcement model defined by the IETF [20][21] that is responsible for intercepting an attempted access to a protected resource, obtaining an authorization decision from the Policy Decision Point, and enforcing the decision with respect to the attempted access.
- **role**: an identity associated with a collection of permissions; in the ANSI Standard model for role based access control, roles are not associated with prohibitions. Roles are intended to model the permissions required to perform a particular function or role in an organization. Roles can be hierarchical: a role can be associated with the identities of other more junior roles as well as being associated directly with a set of permissions. The more senior role then includes all the permissions associated with the more junior roles just as if the permissions of the more junior roles had been associated directly with the more senior role. If an access requester is identified as having a particular role, then the policy should grant that requester the permissions associated with that role or with any of its more junior roles. Compare to *category*, which is a collection of individuals or resources.
- **Obligation**: a requirement specified in a policy that is to be fulfilled by the Policy Enforcement Point in conjunction with enforcing an authorization decision.
- **vocabulary**: a set of terms used to express policies in in a particular domain. The meaning of each item in the vocabulary is defined by the domain, which is responsible for assigning along with a unique identifier and a description of the values that item may have, and the data type of those values. Each vocabulary item is represented in the EPAL and XACML policy languages using an *Attribute*.

## 7. References

- [1] *Enterprise Privacy Authorization Language (EPAL)*, Version 1.2, 2003; the version submitted to the W3C. Available at <http://www.w3.org/Submission/2003/SUBM-EPAL-20031110/>.
- [2] *Enterprise Privacy Authorization Language (EPAL)*, Version 1.1; October 1, 2003; an older version that used XACML syntax. Available at <http://www.zurich.ibm.com/security/enterprise-privacy/epal/Specification/index.html>.
- [3] *eXtensible Access Control Markup Language (XACML)*, Version 2.0; OASIS Standard, February 1, 2005. Available at [http://www.oasis-open.org/committees/tc\\_home.php?wg\\_abbrev=xacml](http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml).
- [4] *Privacy policy profile of XACML v2.0*; OASIS Standard, February 1, 2005. Available at [http://docs.oasis-open.org/xacml/2.0/access\\_control-xacml-2.0-privacy\\_profile-spec-os.pdf](http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-privacy_profile-spec-os.pdf).
- [5] *Multiple resource profile of XACML v2.0*; OASIS Standard, February 1, 2005. Available at [http://docs.oasis-open.org/xacml/2.0/access\\_control-xacml-2.0-multi-profile-spec-os.pdf](http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-multi-profile-spec-os.pdf).
- [6] *Hierarchical resource profile of XACML v2.0*; OASIS Standard, February 1, 2005. Available at [http://docs.oasis-open.org/xacml/2.0/access\\_control-xacml-2.0-hier-profile-spec-os.pdf](http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-hier-profile-spec-os.pdf).
- [7] *Core and hierarchical role based access control (RBAC) profile of XACML v2.0*; OASIS Standard, February 1, 2005. Available at [http://docs.oasis-open.org/xacml/2.0/access\\_control-xacml-2.0-rbac-profile1-spec-os.pdf](http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-rbac-profile1-spec-os.pdf).
- [8] *Role Based Access Control*; ANSI INCITS 359-2004; ANSI Standard.
- [9] *Sun's XACML Open Source Implementation*; freely available under a BSD license at <http://sunxacml.sourceforge.net/>.
- [10] *The Formal Semantics of XACML*, by Polar Humenn, Syracuse Univ.; DRAFT. Report on implementing XACML in Haskell. Available at <http://lists.oasis-open.org/archives/xacml/200310/msg00094.html>.
- [11] *OASIS XACML TC Home Page*. Includes links to all specifications and schemas, references (including adoptions and deployments), and publicly available implementations of XACML. Available at <http://www.oasis->

- [open.org/committees/tc\\_home.php?wg\\_abbrev=xacml](http://open.org/committees/tc_home.php?wg_abbrev=xacml).
- [12] *Conflict and Combination in Privacy Policy Languages* (Summary), by Adam Barth, John C. Mitchell, and Justin Rosenstein, Workshop on Privacy in the Electronic Society, 28 October 2004. Available at <http://www.adambarth.org/papers/barth-mitchell-rosenstein-2004.pdf>.
- [13] Minutes of the XACML TC, February 5, 2004. Available at <http://lists.oasis-open.org/archives/xacml/200402/msg00039.html>.
- [14] *The Platform for Privacy Preferences 1.0 (P3P1.0) Specification*, W3C Recommendation, 16 April 2002. Available at <http://www.w3.org/TR/P3P/>.
- [15] *XQuery 1.0 and XPath 2.0 Functions and Operators*, W3C Working Draft, 2 May 2003. Available at <http://www.w3.org/TR/2003/WD-xpath-functions-20030502/>.
- [16] *Guidelines on the Protection of Privacy and Transborder Flows of Personal Data*, Organization for Economic Co-operation and Development, 23 September 1980. Available at [http://www.oecd.org/document/18/0,2340,en\\_2649\\_34255\\_1815186\\_1\\_1\\_1\\_1,00.html](http://www.oecd.org/document/18/0,2340,en_2649_34255_1815186_1_1_1_1,00.html).
- [17] *Sarbanes-Oxley Act of 2002*, U.S. Government Securities and Exchange Commission. Available at <http://www.sec.gov/about/laws/soa2002.pdf>.
- [18] *Health Insurance Portability and Accountability Act (HIPAA)*, U.S. Government Department of Health and Human Services, 1996. Available at <http://aspe.hhs.gov/admsimp/pl1104191.htm>.
- [19] *Directive on Data Privacy*, European Union, 1998. Available at [http://europa.eu.int/comm/justice\\_home/doc\\_centre/privacy/law/index\\_en.htm](http://europa.eu.int/comm/justice_home/doc_centre/privacy/law/index_en.htm)
- [20] *A Framework for Policy-based Admission Control*, by R. Yavatkar, et al., IETF RFC 2753, January 2000. Available at <http://www.ietf.org/rfc/rfc2753.txt>.
- [21] *Terminology for Policy-Based Management*, by A. Westerinen, et al., IETF RFC 3198, November 2001. Available at <http://www.ietf.org/rfc/rfc3198.txt>.
- [22] *ISO/IEC 10181-3:1966 Information technology -- Open Systems Interconnection -- Security frameworks for open systems: Access control framework*, ISO/IEC.
- [23] *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation, 16 November 1999. Available at <http://www.w3.org/TR/xslt>.

[24] *XACML references, "Products and Deployments"*, Anne Anderson, editor, OASIS XACML TC. Available at <http://docs.oasis-open.org/xacml/xacmlRefs.html#Products>.

## **About the Author**

Anne Anderson is a Senior Staff Engineer at Sun Microsystems Laboratories in Burlington, Massachusetts, where she works on various types of policy languages, including those for privacy, access control, web services, and business processes. She joined Sun in 1998 after working on distributed systems security, operating systems, relational databases, and compilers for several companies including Hewlett Packard/Apollo Computer and Unisys/Burroughs. She received a B.A. in Sociology and Anthropology from Swarthmore College in 1975, and a M.S. in Computer Science from San Diego State University in 1986.