

# Formal Semantics of XACML v3.0

Vladimir Kolovski

Department of Computer Science, University of Maryland

College Park, MD 20740

`kolovski@cs.umd.edu`

March 1, 2008

## 1 Introduction

The first version of XACML was published in 2003, and since then there has been ongoing work on new versions – currently, XACML 3.0[4] is close to standardization. However, the language still lacks a formal and concise semantics that will enable further investigation into the formal and complexity properties of the language. To address this issue, in this paper, we present a proof-theoretic formalization of XACML v3 using natural deduction rules. The formalization covers core XACML and the Administrative Policy profile.

## 2 Syntax

To avoid the verbose XML representation of XACML, a lisp-like syntax is used, similarly to [6]. We use typewriter font to denote the names of the syntax elements as they occur in the XACML specification. Thus, `Policy` refers to the XACML syntactic element that contains `Rules`, whereas a policy can refer to a set of `Policy` or `PolicySet` elements. Also, terminal nodes in the syntax grammar below are denoted by a lower case starting letter.

```

S ::= (PolicySet Comb T S* id)
    | (Policy Comb T R* id)
R ::= (Rule Cond T Effect)
Comb ::= permit-Overrides
      | deny-Overrides
      | first-Applicable
      | only-One-Applicable
Effect ::= Permit | Deny

```

Table 1: Syntax of Policy Elements.

```

T ::= (Target DM*)
DM ::= (DisjunctiveMatch CM+)
CM ::= (ConjunctiveMatch M+)
M ::= (Match AV AD MatchFcn)
    | (Match AV AS MatchFcn)
AD ::= (AttributeDesignator cat attr-ID
      Type issuer? mustBePresent?)
AS ::= (AttributeSelector contextPath Type present?)
AV ::= (AttributeValue value Type)
MatchFcn ::= type-equal | type-greater-than |
           type-greater-than-or-equal | type-less-than |
           type-less-than-or-equal | type-regexp-match
Type ::= string | boolean | integer | ...

```

Table 2: Syntax of Targets and Matching Functions.

Table 3 shows the syntax of `Condition` elements in `Rules`. Note that arbitrary

nesting of functions is allowed in the `Condition` element. We only show a few of the supported functions in the table; a full listing can be found in the XACML 3.0 Specification [4].

```

Cond ::= (Condition Expr*)
Expr ::= Apply | AS | AV
      | Function | VariableReference | AD
Apply ::= (Fcn-ID Expr*)
Function ::= (Fcn-ID Expr*)
Fcn-ID ::= any-of | all-of | regexp-match ...

```

Table 3: Syntax of `Condition` element.

### 3 Proof-theoretic Semantics

In this section, a proof-theoretic semantics of XACML is presented using natural deduction rules. We use the following notation: for a syntactic element  $P$ ,  $X_P$  would refer to a child element of  $P$  that is of type  $X$ . For example, for a `Policy` element  $P$ ,  $T_P$  refers to its `Target`.

#### 3.1 Matching Functions

XACML is an attribute based access control language, so in its most basic form, it matches attribute values from a request with `Targets` in a policy. This section (Tables 5 and 6) presents the inference rules that determine if a `Request` matches a `Target`. They formalize the semantics of match evaluation given in Section 7 in the XACML 3.0 Specification [4].

Note that if a syntax error occurs during matching, in the specification Inde-

RQ	::=	(Request ATS*)
ATS	::=	(Attributes (AT content)* cat)
AT	::=	(Attribute AV* attr-ID issuer)
ARQ	::=	(Request Delegated Del-info Delegate ATS*)
Delegate	::=	((AT content)* delegate)
Del-info	::=	((AT content)* del-info)
Delegated	::=	((AT content)* delegated)

Table 4: Syntax of access requests. ARQ represents an administrative request (a special case of access requests).

`terminate` is returned as result. Since this chapter focuses on the semantic properties of XACML, it is assumed that the policy and request in question are syntactically correct. Observe that even with a syntactically correct policy set, an `Indeterminate` could be returned – e.g., if the *mustBePresent* attribute of an `AttributeDesignator` is true and there is no attribute in the request that matches it (Rule 3 in 5).

### 3.2 Rules, Policies, PolicySets

This section contains rules that capture the semantics of the four basic rule- and policy-combining algorithms: `permit-overrides`, `deny-overrides`, `first-applicable` and `only-one-applicable`. The rule- and policy-combining algorithms are very similar; the only difference occurs when an `Indeterminate` is returned while evaluating a child element. In a `PermitOverrides` rule-combining algorithm, an `Indeterminate` decision overrides a `Deny`, whereas in the equivalent policy-combining algorithm `Deny` decision would override `Indeterminate`.

`Only-One-Applicable` and `First-Applicable` are the same as for Rules.

$\frac{\text{attr-id}_{AD} = \text{attr-id}_{AT} \quad \text{type}_{AV_{AT}} = \text{type}_{AD} \quad \forall \text{issuer}_{AD}, \text{issuer}_{AT} : \text{issuer}_{AD} = \text{issuer}_{AT}}{AD, AT \models_m AV_{AT}}$	$\frac{\exists \text{ATS} \in \text{RQ}, \text{AT} \in \text{ATS} : \text{cat}_{AD} = \text{cat}_{ATS} \quad AD, AT \models AV_{AT}}{AD, \text{RQ} \models_m AV_{AT}}$
$\frac{\text{Value} = \text{xpath-select}(\text{RQ}, \text{contextPath}_{AS})}{AS, \text{RQ} \models_m \text{Value}}$	
$\frac{\forall \text{ATS} \in \text{RQ}, \forall \text{AT} \in \text{ATS} : AD, \text{RQ} \not\models_m AV_{AT} \quad \text{mustBePresent}_{AD} = \text{True}}{AD, \text{RQ} \models_m \text{Indeterminate}}$	
$\frac{AS, \text{RQ} \not\models_m \text{Value} \quad \text{mustBePresent}_{AS} = \text{True}}{AS, \text{RQ} \models_m \text{Indeterminate}}$	

Table 5: Matching AttributeDesignator and AttributeSelector with attributes in Request.

$\frac{\exists AD \in M : AD, \text{RQ} \models_m AV_{AT} \quad \text{fcn}_M(AV_M, AV_{AT}) = \text{True}}{M, \text{RQ} \models \text{True}}$	$\frac{\exists AD \in M : AD, \text{RQ} \models_m \text{Indeterminate}}{M, \text{RQ} \models \text{Indeterminate}}$
$\frac{\exists AS \in M : AS, \text{RQ} \models_m AV_{AT} \quad \text{fcn}_M(AV_M, AV_{AT}) = \text{True}}{M, \text{RQ} \models \text{True}}$	$\frac{\exists AS \in M : AS, \text{RQ} \models_m \text{Indeterminate}}{M, \text{RQ} \models \text{Indeterminate}}$
$\frac{\forall M_{CM} : M_{CM}, \text{RQ} \models \text{True}}{CM, \text{RQ} \models \text{True}}$	$\frac{\forall M_{CM} : M_{CM}, \text{RQ} \models \text{Indeterminate}}{CM, \text{RQ} \models \text{Indeterminate}}$
$\frac{\exists CM_{DM} : CM_{DM}, \text{RQ} \models \text{True}}{DM, \text{RQ} \models \text{True}}$	$\frac{\forall CM_{DM} : CM_{DM}, \text{RQ} \models \text{Indeterminate}}{DM, \text{RQ} \models \text{Indeterminate}}$
$\frac{\forall DM_T : DM_T, \text{RQ} \models \text{True}}{T, \text{RQ} \models \text{True}}$	$\frac{\exists DM_T : DM_T, \text{RQ} \models \text{Indeterminate}}{T, \text{RQ} \models \text{Indeterminate}}$

Table 6: Matching a Request(RQ) with a Target(T).

$$\begin{array}{c}
\frac{T_R, RQ \models \text{True} \quad \text{Cond}_R, RQ \models \text{True}}{R, RQ \models \text{Effect}_R} \qquad \frac{T_R, RQ \models \text{Indeterminate}}{R, RQ \models \text{Indeterminate}} \\
\\
\frac{R, RQ \not\models \text{Effect}_R \quad R, RQ \not\models \text{Indeterminate}}{R, RQ \models \text{NotApplicable}}
\end{array}$$

Table 7: Evaluating a Rule against a Request.

$$\begin{array}{c}
\frac{\exists i : R_i, RQ \models \epsilon \quad T, RQ \models \text{True}}{(\text{Policy } \epsilon\text{-Overrides } T R_1, \dots, R_n), RQ \models \epsilon} \\
\\
\frac{\begin{array}{c} \forall i : R_i, RQ \not\models \text{Indeterminate} \vee \text{Effect}_{R_i} = \bar{\epsilon} \\ \exists i : R_i, RQ \models \bar{\epsilon} \quad T, RQ \models \text{True} \\ \forall j : R_j, RQ \not\models \epsilon \end{array}}{(\text{Policy } \epsilon\text{-Overrides } T R_1, \dots, R_n), RQ \models \bar{\epsilon}} \\
\\
\frac{\begin{array}{c} \exists i : R_i, RQ \models \text{Indeterminate} \wedge \text{Effect}_{R_i} = \epsilon \\ T, RQ \models \text{True} \\ \forall j : R_j, RQ \not\models \epsilon \end{array}}{(\text{Policy } \epsilon\text{-Overrides } T R_1, \dots, R_n), RQ \models \text{Indeterminate}}
\end{array}$$

Table 8:  $\epsilon$ -Overrides Rule Combining Algorithm.  $\epsilon$  stands for Permit or Deny, and  $\bar{\epsilon}$  for the opposite.

$$\frac{\begin{array}{c} \text{Effect} \in \{\text{Permit}, \text{Deny}, \text{Indeterminate}\} \quad T, RQ \models \text{True} \\ \exists R_i \text{ s.t. } R_i, RQ \models \text{Effect} \\ \forall R_j \text{ s.t. } j < i : \\ R_j, RQ \models \text{NotApplicable} \end{array}}{(\text{Policy First-Applicable } T R_1, \dots, R_n), RQ \models \text{Effect}}$$

Table 9: First-Applicable Rule Combining Algorithms.

$\frac{\exists i : R_i, RQ \models \text{Indeterminate} \quad T, RQ \models \text{True}}{(\text{Policy Only-One-Applicable } T R_1, \dots, R_n), RQ \models \text{Indeterminate}}$
$\frac{\exists i, j : (R_i, RQ \models \text{Deny} \vee R_i, RQ \models \text{Permit}) \wedge (R_j, RQ \models \text{Deny} \vee R_j, RQ \models \text{Permit}) \wedge i \neq j \quad T, RQ \models \text{True}}{(\text{Policy Only-One-Applicable } T R_1, \dots, R_n), RQ \models \text{Indeterminate}}$
$\frac{\exists i : R_i, RQ \models \text{Effect} \quad \forall j \text{ s.t. } j \neq i : R_j, RQ \models \text{NotApplicable} \quad T, RQ \models \text{True}}{(\text{Policy Only-One-Applicable } T R_1, \dots, R_n), RQ \models \text{Effect}}$

Table 10: Only-One-Applicable Rule Overriding Algorithm.

$\frac{\text{Comb} \neq \text{Deny-Overrides} \quad \exists i : P_i, RQ \models \text{Indeterminate} \quad T, RQ \models \text{True} \quad \forall j : P_j, RQ \models \text{Indeterminate} \vee P_j, RQ \models \text{NotApplicable}}{(\text{PolicySet Comb } T P_1, \dots, P_n), RQ \models \text{Indeterminate}}$
$\frac{P, RQ \not\models \text{Permit} \quad P, RQ \not\models \text{Deny} \quad P, RQ \not\models \text{Indeterminate}}{P, RQ \models \text{NotApplicable}}$
$\frac{\exists i : R_i, RQ \models \text{Indeterminate} \quad T, RQ \models \text{True} \quad \forall j : R_j, RQ \not\models \text{Permit} \wedge R_j, RQ \not\models \text{Deny}}{(\text{Policy Comb } T R_1, \dots, R_n), RQ \models \text{Indeterminate}}$

Table 11: Generic Indeterminate rule for policies and rules.

$\frac{\exists i : P_i, RQ \models \text{Permit} \quad T, RQ \models \text{True}}{(\text{PolicySet Permit-Overrides } T P_1, \dots, P_n), RQ \models \text{Permit}}$
$\frac{\exists i : P_i, RQ \models \text{Deny} \quad T, RQ \models \text{True} \quad \forall j : P_j, RQ \not\models \text{Permit}}{(\text{PolicySet Permit-Overrides } T P_1, \dots, P_n), RQ \models \text{Deny}}$
$\frac{\exists i : P_i, RQ \models \text{Deny} \vee P_i, RQ \models \text{Indeterminate} \quad T, RQ \models \text{True}}{(\text{PolicySet Deny-Overrides } T P_1, \dots, P_n), RQ \models \text{Deny}}$
$\frac{\exists i : P_i, RQ \models \text{Permit} \quad T, RQ \models \text{True} \quad \forall j : P_j, RQ \not\models \text{Deny} \wedge P_j, RQ \not\models \text{Indeterminate}}{(\text{PolicySet Deny-Overrides } T P_1, \dots, P_n), RQ \models \text{Permit}}$

Table 12: Permit-Overrides and Deny-Overrides Policy Combining Algorithms.

### 3.2.1 Condition Element and arbitrary functions

In XACML V3.0, a total of 237 different functions are allowed in a **Condition**. The types of functions range from datatype comparison and conversion functions to higher order bag functions. Considering that the semantics of these functions has already been covered in [3], we omit their discussion here.

## 3.3 Multiple Resource and Hierarchical Profile

While an important part of XACML, the Multiple Resource [2] and Hierarchical Profile [1] profiles do not actually add any expressive power to the language. Instead, they introduce abbreviated syntax that can be used to express access requests covering multiple resource in a concise manner. The semantics of such requests is given by ‘unwrapping’ them into individual access requests, and then evaluating each request separately. Without any loss of generality, this report covers only *individual* access requests.

### 3.4 Administrative Profile

So far, we have only been discussing access policies, i.e., policies that specify the situations under which users are granted or denied access. An administrative policy, on the other hand, specifies who (and under what conditions) is authorized to write access policies. For example, an administrative policy might state that members of group Administrators are allowed to write access policies about Files. Following, we describe the basic processing model of administrative XACML.

When a new access request  $R$  is to be checked against a XACML policy, it is first applied against all access policies in the set. If some policy applies to  $R$  and yields an access decision, then the access decision needs to be authorized by a *trusted* policy using a process defined in [5] as *reduction*.

Reduction is performed as follows by applying the access request against any administrative policy  $P_s$  that is sibling of  $P$ , generating administrative requests  $ARQ$ . Then, using the administrative requests and other policies reduction *edges* are generated between the policies. The semantics of reduction is shown in Table 13. Finally, a search is performed through the reduction graph starting from the original access policy  $P$ , following reduction edges until a trusted policy is reached. The access decision of  $P$  is authorized only if the graph search reaches a trusted policy. The propagation rules (pertaining to the graph search) are shown in Table 14.

$$\begin{array}{c}
\forall P_i, P_j, s.t. parent(P_i) = parent(P_j) \\
RQ, P_i \models_{rede} AR \quad AR, P_j \models \epsilon \\
\hline
\epsilon P(P_i, P_j)
\end{array}
\qquad
\begin{array}{c}
\forall P_i, P_j, s.t. parent(P_i) = parent(P_j) \\
RQ, P_i \models_{rede} AR \\
AR, P_j \models \text{Indeterminate} \\
\hline
\epsilon I(P_i, P_j)
\end{array}$$

$$\begin{array}{c}
ARQ = (\text{Request Delegated Del-Info Delegate ATS-Admin}) \\
\hline
ARQ, P \models_{rede} (\text{Request ATS-Admin } \epsilon \text{ Issuer}_P \emptyset)
\end{array}$$

Table 13: Reduction Rules.  $\epsilon$  stands for a Permit or Deny.

$$\begin{array}{c}
RQ, P \models \epsilon \\
\exists path = (P, \dots, P_n) s.t. \forall 1 \leq i < n : \\
\epsilon P(P_i, P_{i+1}) \wedge issuer_{P_n} = trusted \\
\hline
RQ, P \models (auth)\epsilon
\end{array}
\qquad
\begin{array}{c}
RQ, P \models \epsilon \vee RQ, P \models \text{Indeterminate} \\
\exists path = (P, \dots, P_n) s.t. \forall 1 < i < n : \\
(\epsilon P(P_i, P_{i+1}) \vee \epsilon I(P_i, P_{i+1})) \wedge \\
issuer_{P_n} = trusted \\
\hline
RQ, P \models (auth)\text{Indeterminate}
\end{array}$$

Table 14: Propagation Rules

## References

- [1] Anne Anderson. Hierarchical resource profile of XACML, February 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).
- [2] Anne Anderson. Multiple resource profile of XACML, November 2007. Available at <http://www.oasis-open.org/committees/download.php/26154/xacml-3.0-hie%2Cmul%2Cdsig%2Cpriv-profiles-wd01.zip>.
- [3] Polar Humenn. The Formal Semantics of XACML, October 2003. Available at <http://lists.oasis-open.org/archives/xacml/200310/pdf000000.pdf>.
- [4] Erik Rissanen. eXtensible Access Control Markup Language (XACML) Version 3.0 (Core Specification and Schemas), May 2007. Available at <http://www.oasis-open.org/committees/download.php/23950/xacml-3.0-core-wd-02.zip>.
- [5] Erik Rissanen, Hal Lockhart, and Tim Moses. Xacml administrative policy version 1.0, working draft 17, May 2007. Available at <http://www.oasis-open.org/committees/download.php/23951/xacml-3.0-admininstration-v1-wd-17.zip>.
- [6] Michael Carl Tschantz and Shriram Krishnamurthi. Towards reasonability properties for access-control policy languages. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 160–169, New York, NY, USA, 2006. ACM Press.