Formalisation and Implementation of the XACML Access Control Mechanism*

Massimiliano Masi[1,2], Rosario Pugliese[2], and Francesco Tiezzi[3]

[1] Tiani "Spirit" GmbH, Guglgasse, 6 - 1110 Vienna, Austria
[2] Università degli Studi di Firenze, Viale Morgagni, 65 - 50134 Firenze, Italy
[3] IMT Advanced Studies Lucca, Piazza S. Ponziano, 6 - 55100, Lucca, Italy

Abstract. We propose a formal account of XACML, an OASIS standard adhering to the Policy Based Access Control model for the specification and enforcement of access control policies. To clarify all ambiguous and intricate aspects of XACML, we provide it with a more manageable alternative syntax and with a solid semantic ground. This lays the basis for developing tools and methodologies which allow software engineers to easily and precisely regulate access to resources using policies. To demonstrate feasibility and effectiveness of our approach, we provide a software tool, supporting the specification and evaluation of policies and access requests, whose implementation fully relies on our formal development.

Keywords: PBAC, XACML, formal semantics, CASE tools

1 Introduction

Nowadays, web services are increasingly used by enterprises and organizations to expose their data to business partners. In this context, resources and services are spread among different administrative domains, thus controlling accesses to them has become a crucial issue. Access control mechanisms are currently used to mitigate the risks of unauthorized access to resources and systems, which could jeopardise the secrecy of sensitive data and cause loss of competitive advantages. These mechanisms may take several forms, use different technologies and involve varying degrees of complexity. Anyway, they are implementations of one of the several access control models proposed in the literature (see, e.g., [1,2]).

We focus on the Policy Based Access Control (PBAC) model [2], that is by now the de-facto standard model for enforcing access control policies in service-oriented architectures. In this model, a resource is governed by a document that exactly specifies what subject credentials and requirements must be fulfilled in order to obtain access. A widely used implementation of PBAC is given by the eXtensible Access Control Markup Language (XACML) [3], an OASIS standard now at version 2.0[4]. It defines a language for the definition of policies and access requests, and a workflow to achieve policy enforcement. XACML is currently

* This work has been partially sponsored by the EU project ASCENS (257414).
[4] We will refer from now on to [3] as *the standard*.

used as a basis for enforcing access control in many large scale projects (see, e.g., [4,5]) and standards (see, e.g., [6,7]).

However, designing XACML access control policies is a difficult and error-prone task. The language has an XML syntax which makes writing XACML policies awkward by using common editors. To make the definition of XACML policies easier also for those end users that are not accustomed with the complexity of the overall policy language, many companies have equipped their products with ad-hoc policy editors (e.g. [8,9]). Such editors are certainly suitable to develop simple and repetitive policies, but might turn out to be cumbersome and ineffective when dealing with complex policies as indeed they tend to hide all the possibilities available in the policy language. Most of all, XACML comes without a formal semantics. The standard is written in prose and contains quite a number of loose points that may give rise to different interpretations and lead to different implementation choices. Some of these loose points are due to an extensive use of the keyword "SHOULD", as per the IETF rfc2119 [10], to indicate recommended requirements that can be for some reason ignored. This leaves the difficult task of understanding the full implications of the various choices to the implementers. Of course, this has to be avoided, since otherwise the portability of XACML policies across different platforms would be considerably undermined.

In this paper we introduce a formal semantics of XACML 2.0[5] that clarifies all ambiguous and intricate aspects of the standard. To hide the complexity introduced by XML, we propose an alternative syntax. This way, we get a tiny language with solid mathematical foundations that lays the basis for developing tools and methodologies that can be easily used by software engineers to precisely define access controls policies on resources. To demonstrate feasibility and effectiveness of our approach, by relying on the formal semantics, we have implemented our language using Java. We have thus obtained a software tool that supports the specification and evaluation of policies and access requests.

Related work. As a result of the widespread use of XACML in (web) service-oriented systems and international projects, many attempts of formalisation have been made. A largely followed approach is based on 'transformational' semantics, where XACML policies are translated into terms of some formalism. For example, [11] uses description logic expressions as target formalism, [12] exploits the process algebra CSP [13], and [14] the model-oriented specification language VDM++ [15]. The main advantage of this approach is the possibility of analysing policies by means of off-the-shelf reasoning tools that may be already available for the considered formalisms. From the semantics point of view, this approach provides some alternative high-level representations of policies, which in their turn have their own semantics. This makes it more difficult to understand the formal meaning of policies with respect to our formal semantics, which directly associates mathematical objects (i.e. 4-tuples of request sets) to policies. These concepts are easier and more understandable than terms, like e.g. description

[5] At the time of writing, the new version XACML 3.0 is under first review and, hence, is continuously changing. We suppose that a full adoption of this new version in production projects will take quite some time.

logic expressions, resulting from automatic translations, also because such translations unavoidably produce terms more complex than necessary. Therefore, our semantics can be conveniently exploited by software engineers to drive XACML implementations. At the same time, its mathematical foundations enable the development of reasoning tools (as we briefly discuss in Section 6).

A similar approach is proposed in [16], where the policies are first specified by means of the description language RW [17], then are analysed through a model checking technique, and finally are translated in XACML. Advantages and disadvantages with respect to our approach are as before.

Other formalisation approaches, more similar to ours, defines the semantics of XACML policies in a more direct way. For example, [18] proposes a semantics based on (multi-terminal) binary decision diagrams, which permit efficiently carrying out the proposed analysis techniques (i.e. property verification and change-impact analysis), but are not suitable as an implementation guide. Instead, [19] formalises a subset of XACML, called Core XACML. The semantics is given through an inductively defined policy evaluation function. Differently from our approach, each policy is evaluated only w.r.t. a single request and, most of all, Core XACML ignores some important XACML features, such as rule conditions, matching functions, some combining algorithms, and the indeterminate value.

There are by now many XACML implementations (see e.g. [20]). In particular, SUN XACML [21] and HERAS^{AF} [22], that are widely used in software in production, implement a Policy Decision Point (PDP) and a library for the development of Policy Enforcement Point (PEP)s. Differently from our implementation, they parse policies in XML format deployed in the policy repository. Moreover, they evaluate each request by visiting parts of the generated DOM tree, while we evaluate the requests by executing Java classes implementing the semantics representations of the policies. XEngine [23] is another notable implementation. It aims at highly efficient request processing, achieved by converting XACML policies into numerical representations. Instead, our main goal is the development of an XACML implementation driven by a formal semantics. Another implementation of an access control mechanism is PERMIS [24], a modular infrastructure specifically devised for Grid systems and integrated in modern toolkits (like, e.g., [25,26]). However, PERMIS relies on an ad-hoc, non-standard policy language which is less expressive than XACML [27].

To sum up, differently from related works, our formalisation has a twofold aim: it serves as a guide for implementers and, at the same time, paves the way for the development of analysis tools.

Summary of the rest of the paper. In Section 2, we give a glimpse of the XACML standard by describing the underlying access control model and the main features of the policy language. In Section 3, we introduce an alternative syntax for XACML, which we then use in Section 4 as the basis to define the formal semantics. We illustrate our approach through an example from an healthcare project. In Section 5, we describe our Java-based implementation of the formal semantics. Finally, in Section 6, we touch upon directions for future work.

2 The XACML standard

In the access control model underlying XACML, each resource can be paired with one or more policies, namely XML documents expressing the capabilities that a requestor needs to have for accessing the resource. Specifically, policies and policy sets are retrieved from a Policy Administration Point (PAP) by a PDP, which is on duty to decide whether to give access to resources or not. The policies and policy sets retrieved by the PDP represent the complete policy for the specified resources.

A request to access a resource is created by a PEP, which reuses claims within the service invocation made by an *access requester*. PEPs can have many different forms, e.g. they may be part of a remote-access gateway, a Web server, an email user-agent, etc. Thus, we cannot expect that in an enterprise all PEPs issue access requests to a PDP directly in a common format. Therefore, the requests and responses handled by the PDP must be converted in a canonical form, i.e. the so-called XACML *context*. The obvious benefit of this approach is that policies may be written and analyzed independently of the specific environment in which they have to be enforced.

The authorization decision is made by the PDP by checking the *matching* between values of the request and values from the retrieved policies. The decision taken by the PDP can be one among permit, deny, not-applicable and indeterminate: the meaning of the first two values is obvious, while the third means that the PDP does not have any policy that applies to the request and the fourth means that the PDP is unable to evaluate the access request (reasons for such inability include, e.g., missing attributes, network errors, evaluation errors).

Let us now consider the languages for expressing policies and requests provided by the standard. The basic element of the policy language is Policy. A Policy is composed of a Target, which identifies the set of capabilities that the requestor must expose, and some Rules. Every Rule contains the facts for the access control decision and has an Effect, which can be either Permit or Deny. A Policy also specifies a combining algorithm that defines what is the final decision for a request when there are (permit/deny) conflicts in the rule decisions.

A Target is composed of four sub-elements: Subjects, Actions, Resources, and Environments. Each category is composed of a set of target elements, each of which contains an attribute identifier, a value and a matching function. Such information is used to check whether the policy is applicable to a given request. Specifically, the matching function retrieves a value from the designed attribute in the request and matches it with the values specified in the target element, according to the function's semantics. If, for all four categories, at least a matching of a target element succeeds, then the policy is applicable to the request.

Besides the Effect, a Rule may specify a Target and some Conditions, i.e. a set of standardly-defined functions that operate on values coming from the request. The Effect is propagated to the upper level policy if the Target of the rule matches and if the Conditions are satisfied.

Policies can be combined together into a PolicySet, which specifies an algorithm that defines the policy set decision in case the contained policies cause

permit/deny conflicts. A PolicySet also contains a Target, which is checked for matching with the access request before the targets of the included policies are.

A Policy/PolicySet can also contain a set of Obligations indicating the actions that the PEP shall enforce after receiving the response. However, since such actions do not play any role in the evaluation procedure, Obligations are not considered in this paper.

An XACML Request, instead, is the request in a canonical form (created by the PEP or the context handler) made of attribute/value pairs. The elements specifying such pairs are grouped according to the same four categories used for the policies, i.e. Subject, Action, Environment and Resource.

3 An alternative syntax of XACML

The XACML standard, as explained in the previous section, defines an XML-based language that permits both writing policies [3, Section 5] and representing contexts (i.e. access requests and responses) [3, Section 6] in a way independent of the specific formats used by PEPs. However, the XML syntax of this language, on the one hand, can make the task of writing policies difficult and error-prone, and, on the other hand, is not adequate for formally defining the semantics of the language and reasoning on it. Therefore, in this section, we provide an alternative syntax of the XACML policy language through a BNF-like grammar (a similar grammar for context representation can be found in [28]).

Our alternative syntax of the XACML policy language is reported in Table 1. As usual, square brackets are used to indicate optional items (that is, everything that is set within the square brackets may be present just once, or not at all).

The manipulable values, ranged over by value, can have simple types (e.g. boolean, string, integer) or complex types (i.e. the values are XML elements that may contain other elements and/or attributes). For the sake of simplicity, we present an untyped version of the language, because the treatment of types would be standard and, anyway, their addition is not relevant for our studies.

To base an authorization decision on some characteristics of the request, like e.g. the subject's identity or the resource's identifier, XACML provides facilities to identify specific values (called *attribute* values) contained in the request context. This approach is supported by means of *attribute designators* and *attribute selectors*. The former ones are pointers to specific attributes of targets (e.g. subjects or resources) in the request context, while the latter ones provide a more general retrieval mechanism based on XPath [29] expressions over the request context. For the sake of presentation, in our XACML's syntax, we represent both designators and selectors by means of *(structured) names*, ranged over by name. For example, the following designator (drawn from [28])

```
<SubjectAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:2.0:subject:role"
                            DataType="http://www.w3.org/2001/XMLSchema#string" />
```

is represented by the name subject.role.

To permit specifying conditions, the language is also equipped with *expressions*, ranged over by expression, which are defined by functions that operate

Table 1. XACML policies syntax

$PDPpolicies$	::= $\{Palg\,;Policies\}$	(Retrieved policies)
$Palg$	::= only-one-applicable | $Ralg$	(Policy-combining alg.)
$Ralg$	::= deny-overrides | permit-overrides	(Rule-combining alg.)
	| first-applicable	
	| ordered-deny-overrides	
	| ordered-permit-overrides	
$Policies$	::=	(Policies)
	$\{Palg\,;$ target :$\{\,[\,Targets\,]\,\}\,;Policies\}$	(policy set)
	| $\langle Ralg\,;$ target :$\{\,[\,Targets\,]\,\}\,;$ rules :$\{Rules\}\rangle$	(policy)
	| $Policies \quad Policies$	
$Targets$	::= $MatchId$(value,name)	(Targets)
	| $Targets \vee Targets$	
	| $Targets \wedge Targets$ | $Targets \sqcap Targets$	
$MatchId$	::= string-equal | integer-equal	(Match functions)
	| string-regexp-match	
	| integer-greater-than | ...	
$Rules$	::= ($Effect\,[\,;$ target :$\{\,Targets\,\}\,]$	(Rules)
	$[\,;$ condition :{expression}$\,]\,)$	
	| $Rules \quad Rules$	
$Effect$	::= permit | deny	(Effects)

on values and names. The complete list of functions provided by XACML is reported in [3, Appendix A.3], while the examples shown in the rest of the paper will exploit the syntax of expressions (reported in [28]) implemented by the tool described in Section 5.

For efficiency of evaluation and ease of management, the overall security policy in force across an enterprise is expressed as multiple independent components. Then, the top-level term $\{Palg\,;Policies\}$ of the XACML policy syntax is a simplified form of policy set (i.e. without target). Given a request, the PDP evaluates the policies in $Policies$ (possibly retrieved from a repository or a PAP) as if they are organised as a single policy set, according to a specified policy-combining algorithm $Palg$. The algorithms provided by XACML for combining the values resulting from policies evaluation – which can be permit, deny, not-applicable and indeterminate – are the following (we refer to [28] for a more precise account):

- deny-overrides: if any policy in the considered set evaluates to deny, then the result of the policy combination is deny;
- permit-overrides: it is similar to the previous algorithm, but this time permit takes precedence over the other results;
- first-applicable: the combined result is that resulting from the evaluation the first policy whose target is applicable to the request;
- ordered-deny-overrides/ordered-permit-overrides: like deny-overrides/permit-overrides, but policies are evaluated in the same order as they occur;
- only-one-applicable: it only applies to policies/policy sets and ensures that one and only one policy is applicable by virtue of its target.

The policies that can be evaluated by the PDP, and hence aggregated by a policy set, can be simple policies of the form $\langle Ralg\,;\mathsf{target}:\{\,[\,Targets\,]\,\}\,;\mathsf{rules}:\{Rules\}\rangle$ or, recursively, policy sets of the form $\{Palg\,;\mathsf{target}:\{\,[\,Targets\,]\,\}\,;\,Policies\}$. Both polices and policy sets specify the algorithm for combining the results of the evaluation of the contained elements and a target to which the policy/policy set applies. The algorithms for simple policies are the same as those for policy sets (but for only-one-applicable) and behave similarly.

A *target* permits identifying the set of access requests that a rule, a policy or a policy set is intended to evaluate. Specifically, a target specifies the set of *subjects*, *resources*, *actions* and *environments* to which the corresponding rule/policy/policy set applies. In the original XML-based syntax of XACML, the target element may contain four elements, one for each of the above categories. However, the evaluation of these separate blocks of information shall be performed in the same way. In fact, in the XACML specification document, the evaluations of subjects, resources, actions and environments are defined by the same 'match table' [3, Section 7.6] and, also, the set of designators for each category is not fixed in advance. Therefore, to obtain a more compact notation, we have decided to represent a target as an expression built from *match elements*, i.e. terms of the form $MatchId(\mathsf{value},\mathsf{name})$, by exploiting an operator for logical disjunction, $\vee$, and two operators for logical conjunction, $\wedge$ and $\sqcap$. Each match element spells out a specific value that the subject/resource/action/environment in the decision request (identified by a name) must match, according to a given matching function. Anyway, this target representation does not lead to a loss of information, because names can be structured and hence, as shown before in the designator example, can include the corresponding category. In a match element, *MatchId* specifies the (boolean) matching function to be used to compare the given literal value with the value of the attribute identified by the given name. XACML supports a wide range of (standard) matching functions (we refer to [3, Appendix A.3] for a complete account and to [28] for the list of functions supported by the tool described in Section 5). Notably, if the target of a policy (resp. policy set) is empty, the policy (resp. policy set) applies to any request context. Instead, if the target of a rule is absent, the rule inherits the target of its enclosing policy.

The three logical operators used for expressing targets are defined over the set $\{\mathsf{match},\mathsf{no\text{-}match},\mathsf{indeterminate}\}$. Basically, they behave as standard conjunction and disjunction operators over $\{\mathsf{match},\mathsf{no\text{-}match}\}$ (where match and no-match are dealt with as true and false, respectively) and the behaviours of the two conjunction operators $\wedge$ and $\sqcap$ only differ for the treatment of the value indeterminate. The decreasing order of precedence among them is as follows: $\wedge$, $\vee$ and $\sqcap$. A disciplined use of structured names and these logical operators permits properly expressing XACML targets: a target must be a term of the form $Subjects \sqcap Resources \sqcap Actions \sqcap Environments$, where each subterm, say $Subjects$, must have the form $Subject_1 \vee Subject_2 \vee \ldots \vee Subject_n$ and, finally, each $Subject_i$ must have the form $MatchId_1(\mathsf{value}_1,\mathsf{name}_1) \wedge \ldots \wedge$

$MatchId_m$(value$_m$,name$_m$). We believe our approach has many advantages like, e.g., a more compact syntax and a more intuitive and clearer semantics.

A single policy contains a (non-empty) set of rules of the form (*Effect* [; target :{ $Targets$ }][; condition :{expression}]), each specifying: 1. an *effect*, which indicates the rule-writer's intended consequence of a positive evaluation for the rule (the allowed values are permit and deny), 2. a *rule target*, which refines the applicability established by the target of the enclosing policy, and 3. a *condition*, which is a boolean expression that may further refine the applicability of the rule. Notably, in a rule, target and condition may be absent.

Regarding context requests, they are represented as terms of the form request :{ *Attributes* }, where *Attributes* consists of a set of (name,value) pairs. Such information indicate the subjects associated to the request, the resources for which the access is being requested, the action to be performed on the resources and the environmental properties. Again, to avoid dealing with separate blocks of information, we exploit structured names. As a matter of notation, we will use R_{all} to denote the set of all possible requests.

We conclude by showing the syntax of a policy[6], which expresses the *patient privacy consent* [30] for the EU Project epSOS [4]. In this project, each role (e.g. doctor, nurse, pharmacist) has *permissions* for performing a certain *coded action* [31] for a certain purpose (e.g. healthcare treatment, statistics, emergency).

```
⟨permit-overrides ;
   target :{ string-equal( "medical doctor" , subject.role)
                ∧ string-equal( "TREATMENT" , subject.purposeofuse)
             ⊓ string-equal( "34133-9" , resource.resource-id) } ;
   rules :{(permit ; target :{ string-equal( "Read" , action.action-id) } ;
            condition :{ string-subset(
                           string-bag( "PRD-003","PRD-005","PRD-010","PRD-016" ),
                           subject.permission) })
           (deny)  } ⟩
```

The policy specifies a subject and a resource in its target, according to which the policy applies to requests issued by a medical doctor with the purpose of accessing to a resource with a code identifier 34133-9[7] for an healthcare TREATMENT. If these capabilities are met, the rules enclosed in the policy are evaluated. The first rule has effect permit if the requestor aims at performing a Read action and has at least the permissions PRD-003, PRD-005, PRD-010 and PRD-016[8] for accessing the resource. The second rule has always effect Deny and is combined with the previous one in such a way that if the first rule evaluates to Permit then the policy permits the access to the resource, otherwise the access is denied.

[6] Due to lack of space, the corresponding XML description is relegated to [28].

[7] In the international code system LOINC [32], 34133-9 identifies a *patient summary*.

[8] These permissions are values from the Hl7 RBAC catalogue [31] grouped together by using a *bag*, i.e. an unordered collection that may contain duplicate values.

4 XACML formal semantics

We present in this section a semantics of XACML policies that formalises the informal one provided by the the standard.

Our semantics is given in a denotational style, i.e. it is defined by a function $[\![\cdot]\!]_R$ that, given a policy/policy set (or a *PDPpolicies* term) and a set R of context requests (with $R \subseteq R_{all}$), returns a *decision* tuple of the form

$$(\,\mathsf{permit} : R_p\,;\ \mathsf{deny} : R_d\,;\ \mathsf{not\text{-}applicable} : R_n\,;\ \mathsf{indeterminate} : R_i\,)$$

where $R_p \cup R_d \cup R_n \cup R_i = R$. Intuitively, R is partitioned into four sets according to the results of the requests evaluation. Notably, R is a subset of the set R_{all} of all requests, thus it can contain e.g. all possible requests, only requests with a given structure or a single request. The definition of $[\![\cdot]\!]_R$ relies on an auxiliary function $(\!|\cdot|\!)_R$ that, given a target, returns a *matching* tuple of the form

$$(\,\mathsf{match} : R_m\,;\ \mathsf{no\text{-}match} : R_n\,;\ \mathsf{indeterminate} : R_i\,)$$

where $R_m \cup R_n \cup R_i = R$, i.e. R is partitioned according to the results of the target evaluation. We will use a projection operator $\cdot\downarrow_{\mathsf{v}}$ that, given a tuple, returns the set corresponding to the value v. Moreover, we will use r to denote a context request and, when convenient, we shall regard r as a set, writing e.g. $(\mathsf{name},\mathsf{value}) \in r$ to mean that $(\mathsf{name},\mathsf{value})$ is an attribute of the request r. As shown in [28], this representation of requests easily permits dealing with multivalued attributes and with the fact that attribute designators and selectors may select bags of values from a request context.

The semantics of a match element $\mathit{MatchId}(\mathsf{value},\mathsf{name})$ of a target is a matching tuple determined by comparing value with the values within the request attributes by means of the matching function $\mathit{MatchId}$.

$$\begin{aligned}
&(\!|\mathit{MatchId}(\mathsf{value},\mathsf{name})|\!)_R = \\
&\quad(\,\mathsf{match} : \{r \in R \mid \exists\,(\mathsf{name},\mathsf{value}') \in r : \mathit{MatchId}(\mathsf{value},\mathsf{value}') = \mathsf{true}\};\\
&\quad\ \mathsf{no\text{-}match} : \{r \in R \mid \forall\,(\mathsf{name},\mathsf{value}') \in r :\\
&\qquad\qquad\qquad \mathit{MatchId}(\mathsf{value},\mathsf{value}') = \mathsf{false}\};\\
&\quad\ \mathsf{indeterminate} : \{r \in R \mid \exists\,(\mathsf{name},\mathsf{value}') \in r :\\
&\qquad\qquad\qquad\qquad \mathit{MatchId}(\mathsf{value},\mathsf{value}') = \mathsf{indeterminate}\\
&\qquad\qquad\quad \wedge\ \nexists\,(\mathsf{name},\mathsf{value}') \in r :\\
&\qquad\qquad\qquad\qquad \mathit{MatchId}(\mathsf{value},\mathsf{value}') = \mathsf{true}\}\,)
\end{aligned}$$

Notably, the use of the universal quantification in the definition of the $\mathsf{no\text{-}match}$ set implies that requests that do not contain attributes named name are inserted into the $\mathsf{no\text{-}match}$ set[9]. The definitions of the matching functions supported by XACML are reported in [3, Appendix A.3]. For example, the function $\mathsf{string\text{-}equal}$ returns true if and only if both argument values are strings of equal length and are

[9] We assume that the `MustBePresent` parameter of every selector/designator has always the default value `false`, which prescribes to return an empty bag when the specified attribute is absent from the request.

equal byte-by-byte according to the function integer-equal (defined by the IEEE standard [33]); otherwise the function string-equal returns false. The matching tuples returned by the evaluation of the match elements within a given target are then combined according to the semantics of the operators $\vee$, $\wedge$ and $\sqcap$, as e.g. in

$$
\begin{array}{l}
(\!|Targets_1 \vee Targets_2|\!)_R = \\
\quad (\ \mathsf{match} : (\!|Targets_1|\!)_R \downarrow_{\mathsf{match}} \ \cup (\!|Targets_2|\!)_R \downarrow_{\mathsf{match}}; \\
\quad\ \ \mathsf{no\text{-}match} : (\!|Targets_1|\!)_R \downarrow_{\mathsf{no\text{-}match}} \ \cap (\!|Targets_2|\!)_R \downarrow_{\mathsf{no\text{-}match}}; \\
\quad\ \ \mathsf{indeterminate} : ((\!|Targets_1|\!)_R \downarrow_{\mathsf{indeterminate}} \setminus (\!|Targets_2|\!)_R \downarrow_{\mathsf{match}}) \\
\qquad\qquad\qquad \cup\ ((\!|Targets_2|\!)_R \downarrow_{\mathsf{indeterminate}} \setminus (\!|Targets_1|\!)_R \downarrow_{\mathsf{match}})\ )
\end{array}
$$

The semantics of a rule with effect permit is defined as follows:

$$
\begin{array}{l}
[\![\, (\mathsf{permit}\,;\mathsf{target}:\{\, Targets\,\};\mathsf{condition}:\{\mathsf{expression}\})\,]\!]_R = \\
\quad (\ \mathsf{permit} : \{r \in (\!|Targets|\!)_R \downarrow_{\mathsf{match}} \mid\ \mathsf{expression} \cdot r = \mathsf{true}\}\,; \\
\quad\ \ \mathsf{deny} : \emptyset\,; \\
\quad\ \ \mathsf{not\text{-}applicable} : \{r \in (\!|Targets|\!)_R \downarrow_{\mathsf{match}} \mid\ \mathsf{expression} \cdot r = \mathsf{false}\} \\
\qquad\qquad\qquad \cup\ (\!|Targets|\!)_R \downarrow_{\mathsf{no\text{-}match}}\,; \\
\quad\ \ \mathsf{indeterminate} : \{r \in (\!|Targets|\!)_R \downarrow_{\mathsf{match}} \mid\ \mathsf{expression} \cdot r = \mathsf{indeterminate}\} \\
\qquad\qquad\qquad \cup\ (\!|Targets|\!)_R \downarrow_{\mathsf{indeterminate}}\ )
\end{array}
$$

where $\mathsf{expression} \cdot r$ denotes the evaluation of the expression expression w.r.t. the request r according to the function definitions reported in [3, Appendix A.3]. The semantics of a rule with effect deny is similar, except that in the decision tuple the permit and deny sets are swapped. Notably, in a rule, the target and the condition are optional; if one or both of them are absent, the semantics of the rule is determined by the above definitions where $\mathsf{expression} \cdot r$ is true for any r if the expression is omitted, and $(\!|Targets|\!)_R \downarrow_{\mathsf{match}} = R$, $(\!|Targets|\!)_R \downarrow_{\mathsf{no\text{-}match}} = \emptyset$ and $(\!|Targets|\!)_R \downarrow_{\mathsf{indeterminate}} = \emptyset$, if the target is omitted.

The semantics of a policy is defined as follows:

$$
\begin{array}{l}
[\![\, \langle Ralg\,;\mathsf{target}:\{\, Targets\,\}\,;\mathsf{rules}:\{Rules\}\rangle\,]\!]_R = \\
\quad (\ \mathsf{permit} : Ralg(Rules)_{R_m} \downarrow_{\mathsf{permit}}\,; \\
\quad\ \ \mathsf{deny} : Ralg(Rules)_{R_m} \downarrow_{\mathsf{deny}}\,; \\
\quad\ \ \mathsf{not\text{-}applicable} : Ralg(Rules)_{R_m} \downarrow_{\mathsf{not\text{-}applicable}} \ \cup\ (\!|Targets|\!)_R \downarrow_{\mathsf{no\text{-}match}}\,; \\
\quad\ \ \mathsf{indeterminate} : Ralg(Rules)_{R_m} \downarrow_{\mathsf{indeterminate}} \ \cup\ (\!|Targets|\!)_R \downarrow_{\mathsf{indeterminate}}\ )
\end{array}
$$

where R_m stands for $(\!|Targets|\!)_R \downarrow_{\mathsf{match}}$. Basically, the requests for which the policy's target does not match are evaluated as not-applicable, while those for which the policy's target is indeterminate are evaluated as indeterminate. The remaining requests, i.e. those for which the policy's target matches, are partitioned by applying the algorithm $Ralg$ specified by the policy to the policy's rules. Similarly to the evaluation of rules, if the policy's target is empty then the policy is evaluated as above by letting $(\!|Targets|\!)_R \downarrow_{\mathsf{match}} = R$, $(\!|Targets|\!)_R \downarrow_{\mathsf{no\text{-}match}} = \emptyset$ and $(\!|Targets|\!)_R \downarrow_{\mathsf{indeterminate}} = \emptyset$. Functions $Ralg(Rules)_R$, given a set $Rules$ of

rules and a set R of requests, return decision tuples of the form

$$\begin{array}{l}(\,\mathsf{permit} : \{r \in R \mid Ralg(Rules, r) = \mathsf{permit}\}\,;\\ \;\;\mathsf{deny} : \{r \in R \mid Ralg(Rules, r) = \mathsf{deny}\}\,;\\ \;\;\mathsf{not\text{-}applicable} : \{r \in R \mid Ralg(Rules, r) = \mathsf{not\text{-}applicable}\}\,;\\ \;\;\mathsf{indeterminate} : \{r \in R \mid Ralg(Rules, r) = \mathsf{indeterminate}\}\;)\end{array}$$

Basically, such tuples are calculated by relying on the auxiliary functions $Ralg(Rules, r)$ whose definitions are given in [3, Appendix C].

The semantics definition of a policy set is similar to that of a single policy:

$$\begin{array}{l}[\![\,\{Palg\,;\mathsf{target}:\{\,Targets\,\}\,;Policies\}\,]\!]_R =\\ \;\;(\,\mathsf{permit} : Palg(Policies)_{R_m}\downarrow_{\mathsf{permit}}\,;\\ \;\;\;\;\mathsf{deny} : Palg(Policies)_{R_m}\downarrow_{\mathsf{deny}}\,;\\ \;\;\;\;\mathsf{not\text{-}applicable} : Palg(Policies)_{R_m}\downarrow_{\mathsf{not\text{-}applicable}}\;\cup\;(\!|Targets|\!)_R\downarrow_{\mathsf{no\text{-}match}}\,;\\ \;\;\;\;\mathsf{indeterminate} : Palg(Policies)_{R_m}\downarrow_{\mathsf{indeterminate}}\;\cup\;(\!|Targets|\!)_R\downarrow_{\mathsf{indeterminate}}\;)\end{array}$$

where R_m stands for $(\!|Targets|\!)_R\downarrow_{\mathsf{match}}$, and function $Palg(Policies)_R$ returns a decision tuple calculated by applying the algorithm $Palg$ to the enclosed policies. It is worth noticing that the definitions of the policy combining algorithms slightly differ from the corresponding rule combining algorithms.

Finally, given a set R of access requests, the semantics of a top-level term $\{Palg\,;Policies\}$ is determined by applying the definition for policy sets and by letting $R_m = R$, $(\!|Targets|\!)_R\downarrow_{\mathsf{no\text{-}match}} = \emptyset$, and $(\!|Targets|\!)_R\downarrow_{\mathsf{indeterminate}} = \emptyset$.

We conclude the section by showing how the semantics definitions presented so far apply to the policy example from the epSOS project, introduced in Sections 3. Given a set R of requests, the permit set of the decision tuple returned by the application of function $[\![\cdot]\!]_R$ to this policy is as follows:

$$\begin{array}{l}\{r \in R \mid (\mathsf{subject.role}, \text{“medical doctor”}) \in r\\ \qquad\wedge\,(\mathsf{subject.purposeofuse}, \text{“TREATMENT”}) \in r\\ \qquad\wedge\,(\mathsf{resource.resource\text{-}id}, \text{“34133-9”}) \in r\\ \qquad\wedge\,(\mathsf{action.action\text{-}id}, \text{“Read”}) \in r\\ \qquad\wedge\,(\mathsf{string\text{-}subset}(\mathsf{string\text{-}bag}(\text{“PRD-003”}, \text{“PRD-005”}, \text{“PRD-010”}, \text{“PRD-016”}),\\ \qquad\qquad\qquad\mathsf{subject.permission}))\cdot r = \mathsf{true}\;\}\end{array}$$

As expected, these are all those requests in R that are issued by a medical doctor, with appropriate permissions, for read accessing a patient summary for treatment purpose. The deny set of the decision tuple consists of all requests in R that match with the policy's target but are not in the set above, i.e. they do not satisfy the target and condition of the first rule. The remaining requests in R belong to the not-applicable set, since the policy never evaluate to indeterminate. We refer the interested reader to [28] for a step-by-step computation of the above decision tuple and further examples.

5 Tools

The implementation of the formalisation presented in the previous sections is made in Java, by also using the ANTLR tool [34] for parsing generation. Our

tool "compiles" a policy written in the syntax proposed in Section 3 into a Java class following the semantics rules defined in Section 4. Thus, a repository storing some policies consists of a Java archive containing all the Java classes generated from the policies. A policy decision is then computed by executing the generated code with the requests passed as parameters to an entry method.

For long-lasting repositories where policy changes are infrequent, this approach is convenient, since no policy's XML Document trees need to be loaded in memory and parsed for each request. Instead this approach does not fit well in situations where the policy repository changes on-the-fly.

Specifically, we have defined two separate parsers: one for the proposed XACML syntax and another one for the rule condition expressions. Each parser is defined so that, every time a syntactic category is identified within a policy term, the corresponding Java method is included into the class under generation. The generated class exploits three lists for representing the matching tuples computed during the evaluation of targets. Indeed, when a target is found, the corresponding matching function is retrieved from a specific data structure, i.e. a 'function table' containing the code implementing all functions defined by the standard. The operators $\wedge$, $\vee$, and $\sqcap$ are used to maintain the lists of requests.

Rules are created according to the corresponding rule combining algorithm: if targets and conditions are satisfied, the algorithm is applied and the decision tuples are returned to the caller. Here, to deal with conditions, a factory method is used to load the current implementation of the expression evaluator. The strategy used in this version of the tool follows the same paradigm as the XACML syntax implementation: when a new condition is satisfied, a Java file is created on-the-fly and compiled. Policies and policy sets are implemented in a way similar to the implementation of rules, relying on the policy-combining algorithms. When targets, rules, and policies are evaluated, the resulting lists representing the decision tuples will be returned to the caller.

A web interface to the tool is available online at `http://rap.dsi.unifi.it/xacml_tools`. It permits to practice with the implementation by using sample policies. The web interface gives the possibility to create XACML requests and, then, to obtain the decision computed by the engine.

6 Concluding remarks

We defined a formal semantics of XACML that aims at clarifying all ambiguous and intricate aspects of the XACML standard and, hence, at conveniently driving implementations. To demonstrate the feasibility and effectiveness of our approach, we fully implemented the semantics as a Java tool.

Another significant advantage of our formalisation is that it paves the way for the development of reasoning tools supporting the analysis of XACML policies. For example, *equivalences* and *preorders* among (syntactically) different policies could be defined based on their semantics denotations and then used to more compactly store the policies or to more efficiently compute a decision. Thus, two policies could be considered as equivalent if their associated decision tuples

coincide or, simply, have the same permit set (indeed, sometimes it does not matter the reason why the access is not permitted, as e.g. with a *deny-biased* PEP [3, Section 7.1.2] that allows the access if the decision taken by the PDP is permit and denies the access in all other cases). We leave the investigation of policy relations as a future work.

We also intend to develop techniques, based on our formal semantics, for studying the application of the *least-privilege* concept [35], in order to determine the requests using the least amount of privilege necessary to satisfy a given XACML policy. To this aim, we will consider an approach where *weights* (indicating the access privilege level[10]) are associated to request data and are used to identify, within the permit set of the decision tuple associated to the considered policy, the requests with minimum total weight. We will also exploit our semantics as a basis for studying *separation of duty* aspects of XACML policies.

We also plan to extend our Java-based framework with other tools, e.g. for translating XACML policies written in the original XML format into polices written in our syntax, and vice versa, and for generating XACML requests, as variations of a template, to be input by the evaluation tool already available. We intend to determine the performances of our tool and to compare them with those of the most notable XACML implementations.

References

1. Ferraiolo, D., Kuhn, R.: Role-based access control. In: NIST-NCSC National Computer Security Conference. (1992) 554–563
2. NIST: A survey of access control models (2009) http://csrc.nist.gov/news_events/privilege-management-workshop/PvM-Model-Survey-Aug26-2009.pdf.
3. OASIS XACML TC: eXtensible Access Control Markup Language (XACML) version 2.0 (2005) http://docs.oasis-open.org/xacml/2.0/XACML-2.0-OS-NORMATIVE.zip.
4. The epSOS project: A european ehealth project http://www.epsos.eu.
5. The Nationwide Health Information Network (NHIN): an American eHealth Project (2009) http://healthit.hhs.gov/portal/server.pt.
6. OASIS: Cross-Enterprise Security and Privacy Authorization (XSPA) Profile of XACML v2.0 for Healthcare v1.0 (2009) http://www.oasis-open.org.
7. OASIS Security Services TC: Assertions and protocols for the OASIS security assertion markup language (SAML) v2.02 (2005) http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf.
8. Namli, T., Dogac, A.: Implementation Experiences On IHE XUA and BPPC. Technical report, Software Research and Development Center, Middle East Technical University Ankara (December 2006)
9. Universidad de Murcia: UMU-XACML-Editor (2008) http://sourceforge.net/projects/umu-xacmleditor/.
10. Bradner, S.: Key words for use in rfcs to indicate requirement levels (1997)
11. Kolovski, V., Hendler, J.A., Parsia, B.: Analyzing web access control policies. In: WWW, ACM (2007) 677–686

[10] For example, the privilege level corresponding to datum “head physician” would be higher than the level of “nurse”, which would be higher than that of “anonymous”.

12. Bryans, J.: Reasoning about XACML policies using CSP. In: SWS, ACM (2005) 28–35
13. Hoare, C.: Commmunicating Sequential Processes. Prentice-Hall (1985)
14. Bryans, J., Fitzgerald, J.S.: Formal engineering of xacml access control policies in vdm++. In: ICFEM. Volume 4789 of LNCS., Springer (2007) 37–56
15. Fitzgerald, J., Larsen, P., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object-oriented Systems. Springer (2005)
16. Zhang, N., Ryan, M., Guelev, D.P.: Evaluating access control policies through model checking. In: ISC. Volume 3650 of LNCS., Springer (2005) 446–460
17. Zhang, N., Ryan, M., Guelev, D.P.: Synthesising verified access control systems in XACML. In: FMSE, ACM (2004) 56–65
18. Fisler, K., Krishnamurthi, S., Meyerovich, L.A., Tschantz, M.C.: Verification and change-impact analysis of access-control policies. In: ICSE, ACM (2005) 196–205
19. Tschantz, M.C., Krishnamurthi, S.: Towards reasonability properties for access-control policy languages. In: SACMAT, ACM (2006) 160–169
20. OASIS XACML TC: Available XACML Implementations (2011) `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml#other`. Last visited 21 September 2011.
21. Proctor, S.: SUN XACML (2011) `http://sunxacml.sf.net`. Last visited 21 September 2011.
22. The Herasaf consortium: HERAS^{AF} `http://www.herasaf.org`.
23. Liu, A.X., Chen, F., Hwang, J., Xie, T.: Xengine: a fast and scalable XACML policy evaluation engine. In: SIGMETRICS, ACM (2008) 265–276
24. ISSRG: The Modular PERMIS Project `http://sec.cs.kent.ac.uk/permis/`.
25. Foster, I.T.: Globus toolkit version 4: Software for service-oriented systems. J. Comput. Sci. Technol. **21**(4) (2006) 513–520
26. Barton, T., et al.: Identity federation and attribute-based authorization through the globus toolkit, shibboleth, gridshib, and myproxy. Technical report, National Center for Supercomputing Applications, University of Illinois (2006)
27. Chadwick, D.W., Zhao, G., Otenko, S., Laborde, R., Su, L., Nguyen, T.A.: Permis: a modular authorization infrastructure. Concurrency and Computation: Practice and Experience **20**(11) (2008) 1341–1357
28. Masi, M., Pugliese, R., Tiezzi, F.: Formalisation and Implementation of a Standard Access Control Mechanism for Web Services (full version). Technical report, Dipartimento di Sistemi e Informatica, Univ. Firenze (2011) Available at `http://rap.dsi.unifi.it/xacml_tools`.
29. Clark, J., DeRose, S.: XML Path Language (XPath) version 1.0 (1999) `http://www.w3.org/TR/xpath`.
30. The IHE Initiative: IT Infrastructure Technical Framework (2009) `http://www.ihe.net`.
31. Health Level Seven organization: Hl7 standards (2009) `http://www.hl7.org`.
32. The Regenstrief Institute: Logical observation identifiers names and codes (LOINC) `http://www.loinc.org`.
33. IEEE Computer Society: IEEE Standard for Binary Floating-Point Arithmetic (1985) IEEE Product No. SH10116-TBR.
34. Parr, T.J., Quong, R.W.: ANTLR: A Predicated-LL(k) Parser Generator. Software Practice and Experience **25** (1994) 789–810
35. Saltzer, J.H.: Protection and the Control of Information Sharing in Multics. Commun. ACM **17** (1974) 388–402