Formalisation and Implementation of a Standard Access Control Mechanism for Web Services*

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

[1] Tiani "Spirit" GmbH, Guglgasse, 6 - 1110 Vienna, Austria
massimiliano.masi@tiani-spirit.com

[2] Università degli Studi di Firenze, Viale Morgagni, 65 - 50134 Firenze, Italy
rosario.pugliese@unifi.it

[3] IMT Advanced Studies Lucca, Piazza S. Ponziano, 6 - 55100, Lucca, Italy
francesco.tiezzi@imtlucca.it

WORKING DRAFT – December 17, 2011

Abstract. Controlling accesses to resources and services plays a central role in the security of service-oriented architectures. To this aim, the OASIS consortium has defined a new standard, called XACML, for policy-based access control. An XACML policy is an XML document specifying credentials and requirements needed to access a resource. The XACML standard defines a language for expressing policies and a complete workflow to achieve access control. However, the XML syntax of the policy language and the lack of a formal semantics make designing XACML access control policies a difficult and error-prone task. In this paper, we propose a formal account of XACML to clarify all ambiguous and intricate aspects of the standard. Specifically, we provide XACML 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 have developed a software tool, supporting the specification and evaluation of policies and access requests, whose implementation fully relies on our formal development.

Keywords: Policy Based Access Control, XACML, formal semantics, CASE tools

* This work has been partially sponsored by the EU project ASCENS (257414).

Table of Contents

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]). The existence of so many models can be explained by the fact that the newer, probably more complicated, models arise from the need to address changes in organizational structures, technologies, etc., and not from deficiencies in the security provided by earlier models.

For example, the Discretionary Access Control model is one of the first models proposed, and it is still the basis of UNIX systems: an access control decision is solely performed on the basis either of a concrete identity of a subject or of a group membership. In this model, each resource has its own set of permissions that must be checked every time the resource is accessed. This tends to increase the complexity of managing access control in enterprise settings. Therefore, the Role Based Access Control (RBAC) [1] model has been introduced. In RBAC, each authenticated subject owns a role. A list of such roles is given to each resource and the access control decision is made on the basis of the matching between the user's role and the roles defined in the list. A major drawback of this model is that a precise semantic of roles must be defined among organizations, which can be unrealistic in service-oriented architectures, where no agreement on the capabilities assigned to roles can be achieved in advance by the different involved entities. To cope with the scalability problems posed by the previous models, the Policy Based Access Control (PBAC) [2] has been introduced. 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. PBAC is by now the de-facto standard model for enforcing access control policies in service-oriented architectures.

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 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

[4] We will refer from now on to [3] as *the standard*.

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 of policies and the verification of 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 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

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

implementations. At the same time, its mathematical foundations enable the development of reasoning tools (as we briefly discuss in Section 7).

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 present the XACML standard by describing the underlying access control model and informal semantics as defined in [3], and by illustrating the main features of the policy language through some examples from an healthcare project. 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. In Section 5, we show how the formal semantics definitions apply to the policy examples introduced in Section 2, while in Section 6 we describe our Java-based implementation of the formal semantics. In Section 1, we touch upon comparisons with closest related work. Finally, in Section 7, we touch upon directions for future work.

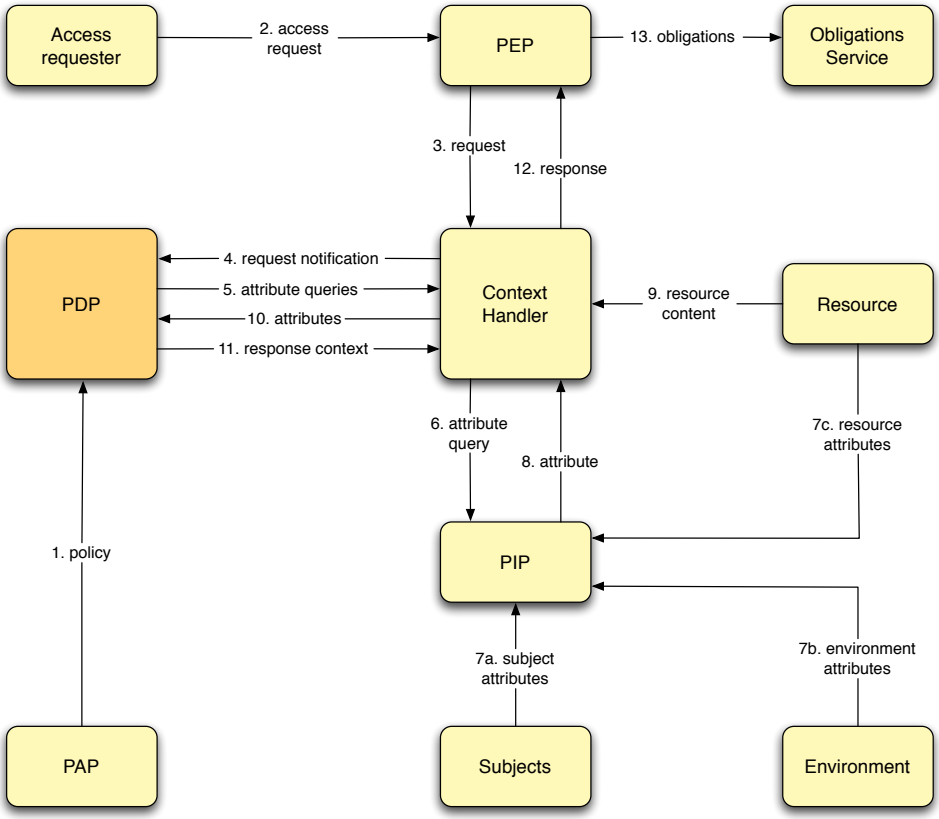

Fig. 1. The XACML workflow

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 in order to access the resource. In this section, we provide an overview of the XACML standard that covers its access control model, the languages for expressing policies and requests, and its informal semantics.

2.1 The XACML model

The data-flow diagram shown in Figure 1 describes how the major actors involved in the XACML model interact each other. The Policy Administration Points (PAPs) write policies and policy sets and make them available to the PDP (step 1), which is on duty to decide whether to give access to resources or not. The communications between PAPs and the PDP may be facilitated by a *policy repository*; however, the XACML specification does not require it. 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 (à la *façade* pattern [28]) by a PEP, which reuses claims within the service invocation made by the *access requester* (step 2). PEPs can have many different forms, e.g. they may be part of a remote-access gateway, a Web server or 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.

Thus, once the PEP receives an access request, it instantiates a new context for handling the corresponding XACML request, which contains the capabilities of the requestor encoded using the language defined by the standard (step 3). Then, the context handler sends the XACML request to the PDP (step 4).

The authorization decision is made by the PDP by checking the *matching* between values of the request and values from the retrieved policies. To carry out the access request evaluation, the PDP can combine the evaluation results of more policies or request for new attributes to the Policy Information Point (PIP) (steps 5-10). Then, the decision is encoded into a *decision statement* and sent to the context handler (step 11) that, in its own turn, converts it into the native response format of the PEP and sends the resulting response to it (step 12).

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, policy evaluation errors). The response can also contain some obligations that the PEP must respect before enforcing the decision of the PDP (step 13).

This paper mainly focusses on steps 4 and 11 of the XACML workflow and, in particular, on the evaluation procedure performed by the PDP to take authorization decisions.

2.2 A glimpse of the XACML language

In this section, we provide an informal presentation of the language for expressing access policies and requests proposed by XACML.

The basic XACML element is the 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 given the fact that there can be (permit/deny) conflicts in the rule decisions.

The Target of a policy is composed of four sub-elements: Subjects, Actions, Resources, and Environments. Each category is composed by a set of target elements, each of which contains an attribute identifier, a typed value and a match-

ing 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.

A Policy can also contain a set of Obligations indicating the actions that the PEP shall enforce after receiving the response.

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 (such targets refine the applicability of each policy). If the Target of the PolicySet is empty, the PolicySet applies to any request.

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.

2.3 Examples from the healthcare domain

Many projects are now using XACML as a methodology to enforce access control. The EU Project epSOS [4], for instance, defines a set of policies for the enforcement of the *patient privacy consent* [29]. As an example, for each patient, epSOS defines the policies in Listings 1 and 2.

Listing 1. epSOS deny-all policy

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<Policy PolicyId="policy-2.16.17.710.790.2.4.1.2"
        RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:
                            rule-combining-algorithm:permit-overrides"
        xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os">
   <Description>Opt-out</Description>
   <Target/>
   <Rule RuleId="urn:oasis:names:tc:xacml:2.0:example:SimpleRule1"
         Effect="Deny"/>
</Policy>
```

The aim of the policy in Listings 1 is to deny access to all resources for any requestor (in healthcare jargon, the patient did an *opt-out*). The target of the policy is empty, hence the policy applies to every XACML request. The value of the policy is determined by the evaluation of the policy's rules according to the specified combining algorithm. In this case, there is only one rule and the algorithm is permit-overrides. The enclosed rule does not specify any target, thus the rule inherits the target of the enclosing policy. This means that, for every request, the outcome of the policy will always be Deny (i.e. the rule's effect).

The policy in Listing 2 expresses the patient privacy consent for the epSOS initiative. In this project, each role (e.g. doctor, nurse, pharmacist) has *permissions* for performing a certain *coded action* [30] for a certain purpose (e.g. healthcare treatment, statistics, emergency).

Listing 2. epSOS privacy policy

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<Policy
    PolicyId="policyId1"
    RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:
                        rule-combining-algorithm:permit-overrides">
  <Target>
   <Subjects>
    <Subject>
      <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:
                             function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
          medical doctor
        </AttributeValue>
        <SubjectAttributeDesignator
            AttributeId="urn:oasis:names:tc:xacml:2.0:subject:role"
            DataType="http://www.w3.org/2001/XMLSchema#string"/>
      </SubjectMatch>
      <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:
                             function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
          TREATMENT
        </AttributeValue>
        <SubjectAttributeDesignator
            AttributeId="urn:oasis:names:tc:xspa:1.0:subject:purposeofuse"
            DataType="http://www.w3.org/2001/XMLSchema#string"/>
      </SubjectMatch>
    </Subject>
   </Subjects>
   <Resources>
    <Resource>
      <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:
                              function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
            34133-9
        </AttributeValue>
        <ResourceAttributeDesignator
            AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
            DataType="http://www.w3.org/2001/XMLSchema#string"/>
      </ResourceMatch>
    </Resource>
   </Resources>
  </Target>
  <Rule RuleId="permitId" Effect="Permit">
    <Description>
       Matches all the READ operations to requests containing the
       correct permissions
    </Description>
    <Target>
     <Actions>
       <Action>
         <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:
                               function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
              Read
          </AttributeValue>
          <ActionAttributeDesignator
              AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
              DataType="http://www.w3.org/2001/XMLSchema#string"/>
         </ActionMatch>
       </Action>
```

```
    </Actions>
  </Target>
  <Condition>
   <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-subset">
     <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-bag">
       <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
           urn:oasis:names:tc:xspa:1.0:subject:hl7:permission:PRD-003
       </AttributeValue>
       <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
           urn:oasis:names:tc:xspa:1.0:subject:hl7:permission:PRD-005
       </AttributeValue>
       <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
           urn:oasis:names:tc:xspa:1.0:subject:hl7:permission:PRD-010
       </AttributeValue>
       <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
           urn:oasis:names:tc:xspa:1.0:subject:hl7:permission:PRD-016
       </AttributeValue>
     </Apply>
     <SubjectAttributeDesignator
         AttributeId="urn:oasis:names:tc:xspa:1.0:subject:hl7:permission"
         DataType="http://www.w3.org/2001/XMLSchema#string"/>
   </Apply>
  </Condition>
 </Rule>
 <Rule RuleId="denyRule" Effect="Deny" />
</Policy>
```

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[6] 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[7] 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.

We conclude by reporting two sample XACML requests in Listings 3 and 5 from the epSOS project, and the corresponding XACML decisions made by a PDP in Listings 4 and 6.

Listing 3. epSOS request by Dr. Marley

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<Request>
    <Subject
        SubjectCategory="#access-subject">
        <Attribute
            AttributeId="countrycode"
            DataType="#string">
            <AttributeValue>GR</AttributeValue>
        </Attribute>
        <Attribute
            AttributeId="subject-id"
            DataType="#string">
```

[6] In the international coding scheme LOINC [31], the code 34133-9 identifies a *patient summary*.

[7] These permissions are string-encoded values from the Hl7 RBAC catalogue [30] and represent actions to be performed by a given role. They are grouped together by using a *bag*, i.e. an unordered collection that may contain duplicate values.

```
            <AttributeValue>Dr. Marley</AttributeValue>
        </Attribute>
        <Attribute
            AttributeId="organization"
            DataType="#string">
            <AttributeValue>HOSPITAL</AttributeValue>
        </Attribute>
        <Attribute
            AttributeId="organization-id"
            DataType="#string">
            <AttributeValue>2624</AttributeValue>
        </Attribute>
        <Attribute
            AttributeId="role"
            DataType="#string">
            <AttributeValue>medical doctor</AttributeValue>
        </Attribute>
        <Attribute
            AttributeId="purposeofuse"
            DataType="#string">
            <AttributeValue>TREATMENT</AttributeValue>
        </Attribute>
        <Attribute
            AttributeId="starttime"
            DataType="http://www.w3.org/2001/XMLSchema#integer">
            <AttributeValue>1299231601160</AttributeValue>
        </Attribute>
        <Attribute
            AttributeId="permission"
            DataType="#string">
            <AttributeValue>PRD-003</AttributeValue>
            <AttributeValue>PRD-006</AttributeValue>
            <AttributeValue>PRD-004</AttributeValue>
            <AttributeValue>PRD-005</AttributeValue>
            <AttributeValue>PRD-010</AttributeValue>
            <AttributeValue>PPD-046</AttributeValue>
            <AttributeValue>PRD-016</AttributeValue>
        </Attribute>
    </Subject>
    <Resource>
        <Attribute
            AttributeId="resource-id"
            DataType="#string">
            <AttributeValue>34133-9</AttributeValue>
        </Attribute>
    </Resource>
    <Action>
        <Attribute
            AttributeId="action-id"
            DataType="#string">
            <AttributeValue>Read</AttributeValue>
        </Attribute>
    </Action>
    <Environment/>
</Request>
```

The request in Listing 3 is made by Dr. Marley, a medical doctor of a greek HOSPITAL, for reading a resource with code 34133-9 (i.e. a patient summary) for TREATMENT purposes. The evaluation of such request with respect to the epSOS policy in Listing 2 produces the positive response in Listing 4.

Listing 4. Decision for request in Listing 3 w.r.t. policy in Listing 2

```
<Response>
  <Result ResourceId="34133-9">
  <Decision>Permit</Decision>
```

```
  <Status>
    <StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
  </Status>
  </Result>
</Response>
```

Listing 5. epSOS request by Mr. Elliot

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<Request>
    <Subject
        SubjectCategory="#access-subject">
        <Attribute
            AttributeId="countrycode"
            DataType="#string">
            <AttributeValue>GR</AttributeValue>
        </Attribute>
        <Attribute
            AttributeId="subject-id"
            DataType="#string">
            <AttributeValue>Mr. Elliot Pharmacist</AttributeValue>
        </Attribute>
        <Attribute
            AttributeId="organization"
            DataType="#string">
            <AttributeValue>HOSPITAL</AttributeValue>
        </Attribute>
        <Attribute
            AttributeId="organization-id"
            DataType="#string">
            <AttributeValue>2624</AttributeValue>
        </Attribute>
        <Attribute
            AttributeId="role"
            DataType="#string">
            <AttributeValue>pharmacist</AttributeValue>
        </Attribute>
        <Attribute
            AttributeId="purposeofuse"
            DataType="#string">
            <AttributeValue>TREATMENT</AttributeValue>
        </Attribute>
        <Attribute
            AttributeId="starttime"
            DataType="http://www.w3.org/2001/XMLSchema#integer">
            <AttributeValue>1299231604331</AttributeValue>
        </Attribute>

        <Attribute
            AttributeId="permission"
            DataType="#string">
            <AttributeValue>PRD-006</AttributeValue>
            <AttributeValue>PRD-004</AttributeValue>
            <AttributeValue>PRD-010</AttributeValue>
            <AttributeValue>PPD-046</AttributeValue>
        </Attribute>
    </Subject>
    <Resource>
        <Attribute
            AttributeId="resource-id"
            DataType="#string">
            <AttributeValue>34133-9</AttributeValue>
        </Attribute>
    </Resource>
    <Action>
        <Attribute
            AttributeId="action-id"
```

Table 1. Policy sets evaluation

Target	*Policy values*	*Policy set value*
match	at least one policy value is permit, deny or indeterminate	specified by the policy combining algorithm
match	all policy values are not-applicable	not-applicable
no-match	—	not-applicable
indeterminate	—	indeterminate

```
60              DataType="#string">
61              <AttributeValue>Read</AttributeValue>
62          </Attribute>
63      </Action>
64      <Environment/>
65  </Request>
```

The request in Listing 5 is similar to the previous one, but it is made by a pharmacist, Mr. Elliot. In this case, the evaluation produces the negative decision in Listing 6.

Listing 6. Decision for request in Listing 5 w.r.t. policy in Listing 2

```
1 <Response>
2   <Result ResourceId="34133-9">
3       <Decision>NotApplicable</Decision>
4   </Result>
5 </Response>
```

Indeed, the policy in Listing 2 does not apply to subjects with the pharmacist role, but only to medical doctor ones. By the way, assuming that the policy would also apply to pharmacists' requests, the decision would be again negative, because the condition of the first rule enclosed within the policy would not be satisfied (indeed, a pharmacist has less permissions than a medical doctor).

2.4 XACML informal semantics

When the PDP receives a context request, it starts the evaluation procedure on the basis of the policies currently stored. It is worth noticing that the PDP performs the evaluation as if it has to evaluate a single policy set consisting of a combining algorithm (set as a configuration parameter of the PDP) and the set of the stored policies/policy sets. The evaluation results will be returned as a context response.

The evaluation of a policy set is described in Table 1. First, the target of the policy set is evaluated to determine if the policy set applies to the request. If the target does not match with the request, the returned value is not-applicable, while if it is indeterminate then the returned value is indeterminate. Instead, if the target matches, the enclosed policies are evaluated, then, the results are combined according to the policy set's combining algorithm. In particular, if all policies are not applicable then the policy set evaluates to not-applicable.

The evaluation of a single policy is similar to that of a policy set, as shown in Table 2. Again, first the policy's target is evaluated, then, in case of matching,

Table 2. Policies evaluation

Target	*Rule values*	*Policy value*
match	at least one rule value is permit, deny or indeterminate	specified by the rule combining algorithm
match	all rule values are not-applicable	not-applicable
no-match	—	not-applicable
indeterminate	—	indeterminate

Table 3. Rules evaluation

Target	*Condition*	*Rule value*
match	true	specified by the rule's effect (i.e. permit or deny)
match	false	not-applicable
match	indeterminate	indeterminate
no-match	—	not-applicable
indeterminate	—	indeterminate

Table 4. Target evaluation

Subjects	*Resources*	*Actions*	*Environments*	*Target value*
match	match	match	match	match
no-match	match or no-match	match or no-match	match or no-match	no-match
match or no-match	no-match	match or no-match	match or no-match	no-match
match or no-match	match or no-match	no-match	match or no-match	no-match
match or no-match	match or no-match	match or no-match	no-match	no-match
indeterminate	—	—	—	indeterminate
—	indeterminate	—	—	indeterminate
—	—	indeterminate	—	indeterminate
—	—	—	indeterminate	indeterminate

the value of the policy is determined by the evaluation of the enclosed rules.

The evaluation of a rule, described in Table 3, consists of evaluating firstly its target and then, if it is needed, its condition. If the target does not match or evaluates to indeterminate then the rule is not-applicable or indeterminate, respectively, and there is no need to evaluate the condition. If the target matches, then the condition is evaluated and, if it is satisfied, the effect specified in the rule determines the rule value. Notably, if the condition is absent then it evaluates to true, while if its expression evaluates to neither true nor false (e.g. some error occurs during the evaluation) then the condition value is indeterminate.

The evaluation of a target, described in Table 4, is determined by combining the results of the evaluation of its match elements . Specifically, a target matches if *all* categories it encloses (i.e. subjects, resources, actions and environments) match. Instead, if at least one category evaluates to indeterminate then the target is indeterminate; otherwise, the target does not match. According to Table 5, each

Table 5. Category evaluation

Category element values	*Category value*
at least one element value is match	match
all element values are no-match	no-match
no element value is match and at least one is indeterminate	indeterminate

Table 6. Category element evaluation

Match element values	*Category element value*
all match element values are true	match
at least one match element is false	no-match
no element value is false and at least one is indeterminate	indeterminate

target category matches if *at least one* of its elements matches; if all elements do not match then also the category does not match, otherwise it is indeterminate. As described in Table 6, each category element matches if *all* match elements it encloses evaluate to true. Instead, if at least one of them evaluates to false, the category element does match; otherwise, it is indeterminate.

Finally, the evaluation of a match element, with matching function identified by *MatchId*, attribute identifier name and value value, is as follows. If name does not identify any value in the request (i.e. there is no attribute with matching name), the result is false. Otherwise, the function *MatchId* is applied between value and each value identified by name: if at least one of those function applications evaluates to true, then the result is true; otherwise, if at least one of the applications results is indeterminate, then the result is indeterminate; finally, if all applications evaluate to false, then the result is false.

3 An alternative syntax of XACML

The XACML standard, as shown 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 adeguate for formally defining the semantics of the language and reasoning on it. Therefore, in this section, we provide an alternative syntax of XACML through two BNF-like grammars.

3.1 Policies syntax

Our alternative syntax of the XACML policy language is reported in Table 7. As usual, squared 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, integer, string, ...) or complex types (in this case, the values are XML elements that may contain other elements and/or attributes). For the

Table 7. XACML policies syntax

$PDPpolicies$	$::= \{Palg\,;Policies\}$	(Retrieved policies)
$Palg$	::= only-one-applicable | $Ralg$	(Policy-combining alg.)
$Ralg$	::= deny-overrides | permit-overrides | first-applicable | ordered-deny-overrides | ordered-permit-overrides	(Rule-combining alg.)
$Policies$	::=	(Policies)
	$\{Palg\,;$target :$\{\,[\,Targets\,]\,\}\,;Policies\}$	(policy set)
	| $\langle Ralg\,;$target :$\{\,[\,Targets\,]\,\}\,;$rules :$\{Rules\}\rangle$	(policy)
	| $Policies \quad Policies$	
$Targets$	::= $MatchId$(value,name)	(Targets)
	| $Targets \vee Targets$	
	| $Targets \wedge Targets \quad \mid \quad Targets \sqcap Targets$	
$MatchId$	::= 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)

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 [32] 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 Listing 2)

```
<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, while the following selector (drawn from Example 4.2.4.2 in [3])

```
<AttributeSelector
     RequestContextPath="//md:record/md:patient/md:patientDoB/text()"
     DataType="http://www.w3.org/2001/XMLSchema#date" />
```

is expressed as record.patient.patientDoB.

To permit specifying conditions, the language is also equipped with *expressions*, ranged over by expression, which are defined by functions that operate 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 just exploit the syntax of expressions (reported in Appendix A) implemented by the tool described in Section 6.

For efficiency of evaluation and ease of management, the overall security policy in force across an enterprise is expressed as multiple independent policy components. Then, the top-level term $\{Palg\,;Policies\}$ of the XACML policy syntax is a simplified form of policy set (which, in fact, does not specify any target). Given a request, the PDP evaluates the policies in $Policies$ (possibly retrieved from a policy 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 as follows:

- deny-overrides: if any policy in the considered set evaluates to deny, then the result of the policy combination is deny. Similarly, if an error occurs while evaluating the target of a policy, or a policy evaluation results in indeterminate, then the policy set evaluates to deny. In other words, deny takes precedence, regardless of the result of evaluating any of the other policies in the policy set. If all policies are found to be not-applicable to the decision request, then the policy set evaluates to not-applicable. Finally, if at least a policy evaluates to permit and all others evaluate to either not-applicable or permit, then the result of the combination is permit.
- permit-overrides: this combination algorithm is similar to the previous one, but this time permit takes precedence over the other results. Notably, if an error occurs while evaluating the target of a policy, or a policy evaluates to indeterminate, then the policy set evaluates to indeterminate provided no other policy evaluates to permit or deny.
- first-applicable: in this case, policies are evaluated in the order of appearance in the policy set and the combined result is the same as the result of evaluating the first policy in the list of policies whose target is applicable to the decision request, if such result is either permit or deny. If all policies evaluate to not-applicable, then the policy set evaluates to not-applicable. If an error occurs while evaluating the target of a policy, or a policy evaluation results in indeterminate, then the evaluation of the algorithm shall halt and the policy set shall evaluate to indeterminate.
- ordered-deny-overrides and ordered-permit-overrides: the behavior of these algorithms is identical to that of deny-overrides and permit-overrides with one exception: the order in which the collection of policies is evaluated shall match the order as they occur in the policy set.
- only-one-applicable: this algorithm only applies to policies/policy sets and ensures that one and only one policy is applicable by virtue of its target.

> If no policy applies, then the result is not-applicable, but if more than one policy is applicable, then the result is indeterminate. When exactly one policy is applicable, the result of the combining algorithm is the result of evaluating the single applicable policy. If an error occurs while evaluating the target of a policy, or the policy evaluation results in indeterminate, then the policy set evaluates to indeterminate.

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 available in case of simple policies are the same as those for policy sets described above (except for only-one-applicable) and behave in a similar way. We refer the interested reader to [3, Appendix C] for the precise behaviour of the rule-combining algorithms.

A *target* permits identifying the set of decision 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 (see examples in Section 2.3), 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', see Table 5 and [3, Section 7.6] and, also, the set of designators for each category is not fixed in advance[8]. 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 in the examples above, 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 Appendix A for the list of functions supported by the tool described in Section 6). 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[9].

[8] For instance, in the Example 4.2.4.2 in [3], the designator parent-guardian.guardian-id is used to identify the subject, rather than using the standard designator subject-id.

[9] The target of a rule, differently from that of a policy/policy set, can be absent because every rule is enclosed within a policy and, hence the rule's target has the

Table 8. Target operators

$\vee$	match	no-match	indeterminate
match	match	match	match
no-match	match	no-match	indeterminate
indeterminate	match	indeterminate	indeterminate

$\wedge$	match	no-match	indeterminate
match	match	no-match	indeterminate
no-match	no-match	no-match	no-match
indeterminate	indeterminate	no-match	indeterminate

$\sqcap$	match	no-match	indeterminate
match	match	no-match	indeterminate
no-match	no-match	no-match	indeterminate
indeterminate	indeterminate	indeterminate	indeterminate

The three logical operators used for expressing targets are defined over the set $\{\mathsf{match}, \mathsf{no\text{-}match}, \mathsf{indeterminate}\}$ (Table 8). 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 these operators is as follows: $\wedge$, $\vee$ and $\sqcap$. It is worth noticing that a disciplined use of structured names and logical operators permits properly expressing XACML targets:[10] a target must be a term of the form

$$\mathit{Subjects} \sqcap \mathit{Resources} \sqcap \mathit{Actions} \sqcap \mathit{Environments}$$

where each of the above subterms, say $\mathit{Subjects}$, must have the form

$$\mathit{Subject}_1 \vee \mathit{Subject}_2 \vee \ldots \vee \mathit{Subject}_n$$

and, finally, each $\mathit{Subject}_i$ must have the form

$$\mathit{MatchId}_1(\mathsf{value}_1,\mathsf{name}_1) \wedge \ldots \wedge \mathit{MatchId}_m(\mathsf{value}_m,\mathsf{name}_m)$$

We believe our approach has many advantages like, e.g., a more compact syntax and a more intuitive and clearer semantics of targets (while, with the original

sole purpose of refining that of its enclosing policy. Thus, when a rule has to apply to the same requests identified by the target of its enclosing policy, the rule's target is omitted.

[10] Our alternative syntax of XACML is more expressive than the original one as it permits writing terms such as string-equal("nurse", subject.role) $\vee$ string-equal("read", action.action-id) and string-equal("1234", id), which cannot be expressed in XACML. This does not pose any problem when terms of our syntax are obtained by translating terms of the original XACML syntax. Instead, if our syntax is directly used to write XACML policies, a disciplined use of names and logical operators is required, which however can be enforced by means of simple static checks.

syntax, the semantics has to be deducted from the structure of the targets' information).

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.

It is worth noticing that, for the sake of presentation, we omit from policy sets, policies and rules their *descriptions*, because such information, written in natural language, do not play any role in the evaluation. For the same reason, also the *obligations*, which specify actions that must be fulfilled by the PEP in conjunction with the evaluation, are omitted. Finally, we do not consider policy *references* (and, hence, identifiers), *variable definitions* and *references*, and *combiner parameters*. Indeed, also these features do not significantly affect the expressiveness of the language and the policy evaluation procedure.

Policy examples. We show here how the policy examples described in Section 2.3 can be rewritten using the syntax defined in Table 7.

The policy reported in Listing 1 can be written as follows:

$$\langle \text{permit-overrides} \,;\, \text{target} :\{\,\} \,;\, \text{rules} :\{\,(\text{deny})\,\}\rangle$$

The policy reported in Listing 2 can be written as follows:

```
⟨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) } ⟩
```

3.2 Contexts syntax

Our alternative syntax of the XACML contexts is reported in Table 9 (again, squared brackets are used to indicate optional items). An XACML context is composed of two parts: the *request* and the *response*.

According to our syntax, a context request simply consists of a set of *attributes*, i.e. name/value pairs. Such information indicate the subjects associated to the request (e.g. the personal data of the human that requested the access, data about the application and the machine used for issuing the request), the resources for which the access is being requested (notably, multiple resources

Table 9. XACML contexts syntax

Context ::=	request :{ *Attributes* }[; response :{ *Results* }]	(Contexts)
Attributes ::=	(name,value)	(Request attributes)
	| *Attributes* *Attributes*	
Results ::=	([id :{value} ;] *Decision* [; status :{value}])	(Response results)
	| *Results* *Results*	
Decision ::=	permit | deny	(Decisions)
	| not-applicable | indeterminate	

can be requested), the action to be performed on the resources (e.g. read, write) and the environmental properties (e.g. current date and time). As in Section 3.1, we exploit again structured names in order to avoid dealing with subjects, resources, actions and environments as separate blocks of information. As a matter of notation, we will use R_{all} to denote the set of all possible requests generated by the grammar in Table 9.

A response, instead, contains the results of the request evaluation. Such results may be more than one, because multiple resources can be requested. In this case, each result contains a value that identifies the corresponding resource. A result also contains the authentication decision produced by the PDP and (optionally) the associated status (reporting e.g. information about possible errors occurred during the policies evaluation).

Context examples. The context request and response reported in Listings 3 and 4 can be rewritten as follows:

```
request :{ (subject.countrycode,"GR")
           (subject.subject-id,"Dr. Marley")
           (subject.organization,"HOSPITAL")
           (subject.organization-id,"2624")
           (subject.role,"medicaldoctor")
           (subject.purposeofuse,"TREATMENT")
           (subject.starttime,1299231601160)
           (subject.permission,"PRD-003")
           (subject.permission,"PRD-006")
           (subject.permission,"PRD-004")
           (subject.permission,"PRD-005")
           (subject.permission,"PRD-010")
           (subject.permission,"PRD-046")
           (subject.permission,"PRD-016")
           (resource.resource-id,"34133-9")
           (action.action-id,"Read") } ;
response :{ id :{"34133-9"} ; permit ; status :{ok} }
```

It is worth noticing that the attribute subject.permission in Listing 3 has multiple values, i.e. it is a so-called *multivalued attribute*. Since our syntax does not allow

to directly express such kind of attributes, we render multivalued attributes as a collection of separate single-valued attributes with the same name. This is sound because XACML requires to evaluate multivalued attributes as they are, indeed, split into separate single-valued attributes (see [3, Section 7.2.3]). This means that a multivalued attribute is just a compact notation for grouping single-valued attributes with the same name.

The context request and response reported in Listings 5 and 6 can be rewritten as follows:

request :{ (subject.countrycode,"GR")
(subject.subject-id,"Mr. Elliot Pharmacist")
(subject.organization,"HOSPITAL")
(subject.organization-id,"2624")
(subject.role,"pharmacist")
(subject.purposeofuse,"TREATMENT")
(subject.starttime,1299231604331)
(subject.permission,"PRD-006")
(subject.permission,"PRD-004")
(subject.permission,"PRD-010")
(subject.permission,"PRD-046")
(resource.resource-id,"34133-9")
(action.action-id,"Read") } ;
response :{ id :{"34133-9"} ; not-applicable }

4 XACML formal semantics

We present in this section a semantics of XACML policies that formalises the informal one expressed in natural language by 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, within the decision tuple, the set 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 generated by the grammar in Table 9, thus it can contain e.g. all possible requests, only requests with a given structure or only one specific request.

The definition of function $[\![\cdot]\!]_R$ relies on a 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.

As a matter of notation, 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 (i.e. a term of the form request :{ *Attributes* }) and, when convenient, we shall regard r as a set, writing e.g. (name,value) $\in r$ to mean that (name,value) is an attribute of the request r. This representation of requests easily permits dealing with multivalued attributes (as shown in Section 3.2) and with the fact that attribute designators and selectors may select bags of values from a request context.

The semantics of targets is defined as follows:

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

$$
\begin{array}{l}
(\!|\mathit{Targets}_1 \vee \mathit{Targets}_2|\!)_R = \\
\quad (\ \mathsf{match} : (\!|\mathit{Targets}_1|\!)_R \downarrow_{\mathsf{match}}\ \cup\, (\!|\mathit{Targets}_2|\!)_R \downarrow_{\mathsf{match}}; \\
\quad\ \mathsf{no\text{-}match} : (\!|\mathit{Targets}_1|\!)_R \downarrow_{\mathsf{no\text{-}match}}\ \cap\, (\!|\mathit{Targets}_2|\!)_R \downarrow_{\mathsf{no\text{-}match}}; \\
\quad\ \mathsf{indeterminate} : ((\!|\mathit{Targets}_1|\!)_R \downarrow_{\mathsf{indeterminate}} \setminus (\!|\mathit{Targets}_2|\!)_R \downarrow_{\mathsf{match}}) \\
\qquad\qquad\qquad \cup\, ((\!|\mathit{Targets}_2|\!)_R \downarrow_{\mathsf{indeterminate}} \setminus (\!|\mathit{Targets}_1|\!)_R \downarrow_{\mathsf{match}})\ )
\end{array}
$$

$$
\begin{array}{l}
(\!|\mathit{Targets}_1 \wedge \mathit{Targets}_2|\!)_R = \\
\quad (\ \mathsf{match} : (\!|\mathit{Targets}_1|\!)_R \downarrow_{\mathsf{match}}\ \cap\, (\!|\mathit{Targets}_2|\!)_R \downarrow_{\mathsf{match}}; \\
\quad\ \mathsf{no\text{-}match} : (\!|\mathit{Targets}_1|\!)_R \downarrow_{\mathsf{no\text{-}match}}\ \cup\, (\!|\mathit{Targets}_2|\!)_R \downarrow_{\mathsf{no\text{-}match}}; \\
\quad\ \mathsf{indeterminate} : ((\!|\mathit{Targets}_1|\!)_R \downarrow_{\mathsf{indeterminate}} \setminus (\!|\mathit{Targets}_2|\!)_R \downarrow_{\mathsf{no\text{-}match}}) \\
\qquad\qquad\qquad \cup\, ((\!|\mathit{Targets}_2|\!)_R \downarrow_{\mathsf{indeterminate}} \setminus (\!|\mathit{Targets}_1|\!)_R \downarrow_{\mathsf{no\text{-}match}})\ )
\end{array}
$$

$$
\begin{array}{l}
(\!|\mathit{Targets}_1 \sqcap \mathit{Targets}_2|\!)_R = \\
\quad (\ \mathsf{match} : (\!|\mathit{Targets}_1|\!)_R \downarrow_{\mathsf{match}}\ \cap\, (\!|\mathit{Targets}_2|\!)_R \downarrow_{\mathsf{match}}; \\
\quad\ \mathsf{no\text{-}match} : ((\!|\mathit{Targets}_1|\!)_R \downarrow_{\mathsf{no\text{-}match}} \setminus (\!|\mathit{Targets}_2|\!)_R \downarrow_{\mathsf{indeterminate}}) \\
\qquad\qquad\qquad \cup\, ((\!|\mathit{Targets}_2|\!)_R \downarrow_{\mathsf{no\text{-}match}} \setminus (\!|\mathit{Targets}_1|\!)_R \downarrow_{\mathsf{indeterminate}}); \\
\quad\ \mathsf{indeterminate} : (\!|\mathit{Targets}_1|\!)_R \downarrow_{\mathsf{indeterminate}}\ \cup\, (\!|\mathit{Targets}_2|\!)_R \downarrow_{\mathsf{indeterminate}}\ )
\end{array}
$$

The semantics of a match element $\mathit{MatchId}$(value,name) is a matching tuple determined by comparing value with the values within the request attributes by means of the matching function *MatchId*. The definitions of the matching functions supported by XACML are reported in [3, Appendix A.3]. For example, the function string-equal returns true if and only if both argument values are strings of equal length and are 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 $\sqcap$ given in Table 8.

The semantics of a rule is defined as follows:

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

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

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]. 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 (i.e. $(\!|Targets|\!)_R \downarrow_{\mathsf{no\text{-}match}}$) are evaluated as not-applicable, while those for which the policy's target is indeterminate (i.e. $(\!|Targets|\!)_R \downarrow_{\mathsf{indeterminate}}$) are evaluated as indeterminate. The remaining requests, namely those for which the policy's target matches (i.e. R_m), are partitioned according to the application of the combining algorithm $Ralg$ specified by the policy to the policy's rules (denoted by $Ralg(Rules)_{R_m}$). 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

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

Basically, such tuples are calculated by relying on the auxiliary functions $Ralg(Rules, r)$ whose definitions are given in [3, Appendix C]. For reader's convenience, we report in Listing 7 the pseudo-code defining the function deny-overrides$(Rules, r)$ and relegate to Appendix B.1 the definitions of the other rule-combining algorithms.

Listing 7. Rule combining algorithm deny-overrides

```
1  Boolean atLeastOnePermit = false;
2  Boolean atLeastOneError  = false;
3  Boolean potentialDeny = false;
4  foreach (rule ∈ Rules) {
5     DecisionTuple t = ⟦ rule ⟧{r};
6     if (r ∈ t↓deny) return deny;
7     if (r ∈ t↓permit) {
8        atLeastOnePermit = true;
9        continue;
10    }
11    if (r ∈ t↓not-applicable) continue;
12    if (r ∈ t↓indeterminate) {
13       atLeastOneError = true;
14       if (rule.effect == deny) potentialDeny = true;
15       continue;
16    }
17 }
18 if (potentialDeny) return indeterminate;
19 if (atLeastOnePermit) return permit;
20 if (atLeastOneError) return indeterminate;
21 return not-applicable;
```

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

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

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. For example,

the function deny-overrides$(Policies, r)$ is defined by the pseudo-code in Listing 8.

Listing 8. Policy combining algorithm deny-overrides

```
Boolean atLeastOnePermit = false;
foreach (policy ∈ Policies) {
   DecisionTuple t = ⟦ policy ⟧{r};
   if (r ∈ t↓deny) return deny;
   if (r ∈ t↓permit) {
      atLeastOnePermit = true;
      continue;
   }
   if (r ∈ t↓not-applicable) continue;
   if (r ∈ t↓indeterminate) return deny;
}
if (atLeastOnePermit) return permit;
return not-applicable;
```

Notice that, differently from the algorithm reported in Listing 7, this one never returns the value indeterminate. The definitions of the other policy-combining algorithms are relegated to Appendix B.2.

Finally, given a set R of context 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 $(\!|Targets|\!)_R \downarrow_{\mathsf{indeterminate}} = \emptyset$.

5 Examples

In this section, we show how the semantics definitions presented in Section 4 apply to the policy examples from the epSOS project, introduced in Sections 2.3 and 3.1.

5.1 epSOS deny-all policy

Consider the policy reported in Listing 1 rewritten with the syntax defined in Table 7 as follows:

$$\langle \mathsf{permit\text{-}overrides\,;target :\{\,\}\,;rules :\{\,(deny)\,\}} \rangle$$

Since the policy's target is empty (hence, the policy applies to all requests), its semantics is:

$$⟦\,\langle \mathsf{permit\text{-}overrides\,;target :\{\,\}\,;rules :\{\,(deny)\,\}} \rangle\,⟧_R = \mathsf{permit\text{-}overrides}((\mathsf{deny}))_R$$

that is it coincides with the decision tuple returned by the function application $\mathsf{permit\text{-}overrides}((\mathsf{deny}))_R$, which is defined as follows:

$$\begin{aligned}&\mathsf{permit\text{-}overrides}((\mathsf{deny}))_R = \\ &\quad(\,\mathsf{permit} : \{r \in R \mid \mathsf{permit\text{-}overrides}((\mathsf{deny}), r) = \mathsf{permit}\}\,; \\ &\quad\ \mathsf{deny} : \{r \in R \mid \mathsf{permit\text{-}overrides}((\mathsf{deny}), r) = \mathsf{deny}\}\,; \\ &\quad\ \mathsf{not\text{-}applicable} : \{r \in R \mid \mathsf{permit\text{-}overrides}((\mathsf{deny}), r) = \mathsf{not\text{-}applicable}\}\,; \\ &\quad\ \mathsf{indeterminate} : \{r \in R \mid \mathsf{permit\text{-}overrides}((\mathsf{deny}), r) = \mathsf{indeterminate}\}\ )\end{aligned}$$

Let us consider now the rule (deny). Its semantics is:

$$(\,\mathsf{permit} : \emptyset\,;\ \mathsf{deny} : R\,;\ \mathsf{not\text{-}applicable} : \emptyset\,;\ \mathsf{indeterminate} : \emptyset\,)$$

Indeed, the rule applies to all requests, since the target is absent, and $\mathsf{expression}{\cdot}r$ always evaluates to true, since also the condition expression is absent.

Thus, according to the semantics of the above rule, the algorithm $\mathsf{permit\text{-}overrides}((\mathsf{deny}), r)$ can be instantiated with rule (deny) as follows:

```
1  Boolean atLeastOneError = false;
2  Boolean potentialPermit = false;
3  Boolean atLeastOneDeny  = false;
4
5  if (r ∈ R) {
6     atLeastOneDeny = true;
7     continue;
8  }
9  if (r ∈ ∅) {
10    return permit;
11 }
12 if (r ∈ ∅) continue;
13 if (r ∈ ∅) {
14    atLeastOneError = true;
15    if (rule.effect == permit) potentialPermit = true;
16    continue;
17 }
18 if (potentialPermit) return indeterminate;
19 if (atLeastOneDeny) return deny;
20 if (atLeastOneError) return indeterminate;
21 return not-applicable;
```

Since condition $r \in R$ is always satisfied and condition $r \in \emptyset$ never holds, the above algorithm returns deny for every request r. Therefore, the decision tuple returned by $\mathsf{permit\text{-}overrides}((\mathsf{deny}))_R$, and hence also returned by $[\![\,\langle \mathsf{permit\text{-}overrides}\,;\mathsf{target}:\{\,\}\,;\mathsf{rules}:\{\,(\mathsf{deny})\,\}\rangle\,]\!]_R$, is as follows:

$$(\,\mathsf{permit} : \emptyset\,;\ \mathsf{deny} : R\,;\ \mathsf{not\text{-}applicable} : \emptyset\,;\ \mathsf{indeterminate} : \emptyset\,)$$

As expected, the semantics of the policy means that the policy evaluates to deny for all context requests.

5.2 epSOS privacy policy

Consider the policy reported in Listing 2 rewritten with the syntax defined in Table 7 as follows:

$$
\begin{array}{l}
\langle \text{permit-overrides}\,; \\
\quad \text{target} :\{\, \text{string-equal}(\text{“medical doctor”}, \text{subject.role}) \\
\qquad\qquad \wedge\ \text{string-equal}(\text{“TREATMENT”}, \text{subject.purposeofuse}) \\
\qquad\quad \sqcap\ \text{string-equal}(\text{“34133-9”}, \text{resource.resource-id})\,\}\,; \\
\quad \text{rules} :\{(\text{permit}\,; \text{target} :\{\, \text{string-equal}(\text{“Read”}, \text{action.action-id})\,\}\,; \\
\qquad\quad \text{condition} :\{\, \text{string-subset}(\\
\qquad\qquad\qquad\quad \text{string-bag}(\text{“PRD-003”},\text{“PRD-005”},\text{“PRD-010”},\text{“PRD-016”}), \\
\qquad\qquad\qquad\quad \text{subject.permission})\,\}) \\
\qquad\quad (\text{deny})\ \ \}\ \rangle
\end{array}
$$

To calculate the decision tuple returned by the application of function $[\![\cdot]\!]_R$ to this policy, we first evaluate its target. In particular, the evaluation of the first match element occurring in the policy's target is as follows:

$$
\begin{array}{l}
(\!|\text{string-equal}(\text{“medical doctor”}, \text{subject.role})|\!)_R = \\
\quad (\,\text{match} : \{r \in R \mid \exists\,(\text{subject.role,value}) \in r : \\
\qquad\qquad\qquad\qquad\qquad \text{string-equal}(\text{“medical doctor”,value}) = \text{true}\,\}; \\
\quad\ \ \text{no-match} : \{r \in R \mid \forall\,(\text{subject.role,value}) \in r : \\
\qquad\qquad\qquad\qquad\quad \text{string-equal}(\text{“medical doctor”,value}) = \text{false}\,\}; \\
\quad\ \ \text{indeterminate} : \{r \in R \mid \exists\,(\text{subject.role,value}) \in r : \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{string-equal}(\text{“medical doctor”,value}) = \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \text{indeterminate} \\
\qquad\qquad\qquad\qquad\quad \wedge\ \ \nexists\,(\text{subject.role,value}) \in r : \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{string-equal}(\text{“medical doctor”,value}) = \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \text{true}\,\}\,)
\end{array}
$$

The definition of the above decision tuple can be made more readable by exploiting the fact that function string-equal never returns indeterminate and returns true if and only if the two argument strings are identical:

$$
\begin{array}{l}
\quad (\,\text{match} : \{r \in R \mid (\text{subject.role,“medical doctor”}) \in r\}; \\
\quad\ \ \text{no-match} : \{r \in R \mid \forall\,(\text{subject.role,value}) \in r : \\
\qquad\qquad\qquad\qquad\quad \text{string-equal}(\text{“medical doctor”,value}) = \text{false}\,\}; \\
\quad\ \ \text{indeterminate} : \emptyset\,)
\end{array}
$$

Similarly, the evaluation of the other match elements occurring in the policy's target are as follows:

$$
\begin{array}{l}
(\!|\text{string-equal}(\text{“TREATMENT”}, \text{subject.purposeofuse})|\!)_R = \\
\quad (\,\text{match} : \{r \in R \mid (\text{subject.purposeofuse,“TREATMENT”}) \in r\}; \\
\quad\ \ \text{no-match} : \{r \in R \mid \forall\,(\text{subject.purposeofuse,value}) \in r : \\
\qquad\qquad\qquad\qquad\quad \text{string-equal}(\text{“TREATMENT”,value}) = \text{false}\,\}; \\
\quad\ \ \text{indeterminate} : \emptyset\,)
\end{array}
$$

$$(\!|\text{string-equal}(\text{“34133-9”}, \text{resource.resource-id})|\!)_R =$$

$$\begin{array}{l}(\ \text{match} : \{r \in R \mid (\text{resource.resource-id}, \text{“34133-9”}) \in r\};\\ \quad \text{no-match} : \{r \in R \mid \forall\, (\text{resource.resource-id}, \text{value}) \in r : \\ \qquad\qquad\qquad\qquad \text{string-equal}(\text{“34133-9”}, \text{value}) = \text{false}\ \};\\ \quad \text{indeterminate} : \emptyset\ )\end{array}$$

Thus, the semantics of the target is obtained by combining the above decision tuples as follows:

$$\begin{array}{l}(\!|\text{string-equal}(\text{“medical doctor”}, \text{subject.role})\\ \qquad \wedge\ \text{string-equal}(\text{“TREATMENT”}, \text{subject.purposeofuse})\\ \quad \sqcap\ \text{string-equal}(\text{“34133-9”}, \text{resource.resource-id})|\!)_R =\end{array}$$

$$\begin{array}{l}(\ \text{match} : \{r \in R \mid (\text{subject.role}, \text{“medical doctor”}) \in r\}\\ \qquad\qquad \cap\ \{r \in R \mid (\text{subject.purposeofuse}, \text{“TREATMENT”}) \in r\}\\ \qquad\qquad \cap\ \{r \in R \mid (\text{resource.resource-id}, \text{“34133-9”}) \in r\};\\ \quad \text{no-match} : \{r \in R \mid \forall\, (\text{subject.role}, \text{value}) \in r : \\ \qquad\qquad\qquad\qquad \text{string-equal}(\text{“medical doctor”}, \text{value}) = \text{false}\ \}\\ \qquad\qquad \cup\ \{r \in R \mid \forall\, (\text{subject.purposeofuse}, \text{value}) \in r : \\ \qquad\qquad\qquad\qquad \text{string-equal}(\text{“TREATMENT”}, \text{value}) = \text{false}\ \}\\ \qquad\qquad \cup\ \{r \in R \mid \forall\, (\text{resource.resource-id}, \text{value}) \in r : \\ \qquad\qquad\qquad\qquad \text{string-equal}(\text{“34133-9”}, \text{value}) = \text{false}\ \};\\ \quad \text{indeterminate} : \emptyset\ )\end{array}$$

The resulting tuple can be also rewritten as follows:

$$\begin{array}{l}(\ \text{match} : \{r \in R \mid (\text{subject.role}, \text{“medical doctor”}) \in r\\ \qquad\qquad\qquad\qquad \wedge\ (\text{subject.purposeofuse}, \text{“TREATMENT”}) \in r\\ \qquad\qquad\qquad\qquad \wedge\ (\text{resource.resource-id}, \text{“34133-9”}) \in r\}\ =\ R_m;\\ \quad \text{no-match} : \{r \in R \mid \forall\, (\text{subject.role}, \text{value}) \in r : \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{string-equal}(\text{“medical doctor”}, \text{value}) = \text{false}\\ \qquad\qquad\qquad\qquad\qquad \vee\ \forall\, (\text{subject.purposeofuse}, \text{value}) \in r : \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{string-equal}(\text{“TREATMENT”}, \text{value}) = \text{false}\\ \qquad\qquad\qquad\qquad\qquad \vee\ \forall\, (\text{resource.resource-id}, \text{value}) \in r : \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{string-equal}(\text{“34133-9”}, \text{value}) = \text{false}\}\ =\ R_n;\\ \quad \text{indeterminate} : \emptyset\ )\end{array}$$

Now, we can evaluate the first rule enclosed within the policy w.r.t. the set of requets R_m, by starting from its target (and by immediately applying the simplifications used above):

$$(\!|\text{string-equal}(\text{“Read”}, \text{action.action-id})|\!)_{R_m} =$$

$$\begin{array}{l}(\ \text{match} : \{r \in R_m \mid (\text{action.action-id}, \text{“Read”}) \in r\}\ =\ R'_m;\\ \quad \text{no-match} : \{r \in R_m \mid \forall\, (\text{action.action-id}, \text{value}) \in r : \\ \qquad\qquad\qquad\qquad \text{string-equal}(\text{“Read”}, \text{value}) = \text{false}\}\ =\ R'_n;\\ \quad \text{indeterminate} : \emptyset\ )\end{array}$$

We proceed then to evaluate the policy's rule. Let e denote the expression

$$\begin{array}{l}\mathsf{string\text{-}subset}(\mathsf{string\text{-}bag}(\text{“PRD-003”}, \text{“PRD-005”}, \text{“PRD-010”}, \text{“PRD-016”}),\\ \qquad\qquad \mathsf{subject.permission})\end{array}$$

within the rule's condition; it is worth noticing that, for any request r, $\mathsf{e} \cdot r$ can evaluate to either true or false, since functions string-subset and string-bag never return indeterminate. Thus, the rule semantics is:

$$\begin{array}{l}[\![\,(\mathsf{permit}\,;\mathsf{target}:\{\,\mathsf{string\text{-}equal}(\text{“Read”}, \mathsf{action.action\text{-}id})\,\};\mathsf{condition}:\{\mathsf{e}\})\,]\!]_{R_m} =\\ \quad(\,\mathsf{permit}:\{r \in R'_m \mid \mathsf{e}\cdot r = \mathsf{true}\} \;=\; R''_m;\\ \quad\;\mathsf{deny}:\emptyset\,;\\ \quad\;\mathsf{not\text{-}applicable}:\{r \in R'_m \mid \mathsf{e}\cdot r = \mathsf{false}\} \;\cup\; R'_n \;=\; R''_n;\\ \quad\;\mathsf{indeterminate}:\emptyset\;)\end{array}$$

Let us consider now the rule (deny). Its semantics is:

$$\begin{array}{l}[\![\,(\mathsf{deny})\,]\!]_{R_m} =\\ \qquad(\,\mathsf{permit}:\emptyset\,;\;\mathsf{deny}:R_m\,;\;\mathsf{not\text{-}applicable}:\emptyset\,;\;\mathsf{indeterminate}:\emptyset\;)\end{array}$$

Indeed, as shown in the example in Section 5.1, the rule applies to all requests.

The above semantics definitions can be now exploited to define the function p-o(r), i.e. the instantiation of the permit-overrides algorithm with the policy's rules, by means of the following pseudo-code:

```
1 if (r ∈ R''_m) return permit;
2 if (r ∈ R_m) return deny;
3 return not-applicable;
```

Finally, since the function p-o(r) never returns not-applicable and indeterminate for $r \in R_m$, the decision tuple defining the semantics of the policy is:

$$\begin{array}{l}(\,\mathsf{permit}:\{r \in R_m \mid \mathsf{p\text{-}o}(r) = \mathsf{permit}\}\,;\\ \quad\;\mathsf{deny}:\{r \in R_m \mid \mathsf{p\text{-}o}(r) = \mathsf{deny}\}\,;\\ \quad\;\mathsf{not\text{-}applicable}:R_n;\\ \quad\;\mathsf{indeterminate}:\emptyset\;)\end{array}$$

Therefore, given a set R of requests, it holds that:

– The set of requests for which the policy evaluates to permit is R''_m, that is

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

Basically, these are all requests in R that are issued by a medical doctor, with appropriate permissions, for read accessing a patient summary for treatment purpose.

– The set of requests for which the policy evaluates to deny is $R_m \backslash R''_m$, that is

$$\begin{aligned}\{r \in R \mid\ & (\text{subject.role}, \text{“medical doctor”}) \in r \\ & \wedge\ (\text{subject.purposeofuse}, \text{“TREATMENT”}) \in r \\ & \wedge\ (\text{resource.resource-id}, \text{“34133-9”}) \in r \\ & \wedge\ (\ (\text{action.action-id}, \text{“Read”}) \notin r \\ & \quad \vee\ (\text{string-subset}(\\ & \qquad \text{string-bag}(\text{“PRD-003”}, \text{“PRD-005”}, \text{“PRD-010”}, \text{“PRD-016”}), \\ & \qquad \text{subject.permission})\ ) \cdot r = \text{false}\ )\ \}\end{aligned}$$

This means that the access is denied to a request if

1. it is issued by a medical doctor for accessing a patient summary for treatment purpose, but it does not specify that the action to be performed on the data is the reading (only);
2. it is issued by a medical doctor for read accessing a patient summary for treatment purpose, but the doctor does not have the appropriate permissions.

– The set of requests for which the policy evaluates to not-applicable is R_n, that is

$$\begin{aligned}\{r \in R \mid\ & \forall\, (\text{subject.role}, \text{value}) \in r : \\ & \qquad \text{string-equal}(\text{“medical doctor”}, \text{value}) = \text{false} \\ & \vee\ \forall\, (\text{subject.purposeofuse}, \text{value}) \in r : \\ & \qquad \text{string-equal}(\text{“TREATMENT”}, \text{value}) = \text{false} \\ & \vee\ \forall\, (\text{resource.resource-id}, \text{value}) \in r : \\ & \qquad \text{string-equal}(\text{“34133-9”}, \text{value}) = \text{false}\}\end{aligned}$$

This means that a request is not applicable to the given policy if

1. it is not been issued by a medical doctor;
2. or the purpose is not the treatment of a patient;
3. or it does not refer to a patient summary.

– The policy never evaluates to indeterminate.

6 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.

7 Concluding remarks

In this paper, we provide a formal semantics of the XACML standard and, to demonstrate the feasibility and effectiveness of the proposed approach, we have fully implemented the semantics as a Java tool.

The first, and most evident, advantage of a formal semantics for XACML, with respect to the informal one (written in 'natural' language) given in [3], is that it clarifies all ambiguous and intricate aspects of XACML. This way, by relying on this semantics, software engineers can precisely describe, and implement, access controls using policies on resources.

Besides this, our formalisation 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[11]) 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
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

[11] 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”.

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. Erich, G., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable. Addison-Wesley (1994)
29. The IHE Initiative: IT Infrastructure Technical Framework (2009) `http://www.ihe.net`.
30. Health Level Seven organization: Hl7 standards (2009) `http://www.hl7.org`.
31. The Regenstrief Institute: Logical observation identifiers names and codes (LOINC) `http://www.loinc.org`.
32. Clark, J., DeRose, S.: XML Path Language (XPath) version 1.0 (1999) `http://www.w3.org/TR/xpath`.
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

A Expressions and matching functions supported by the tool

The syntax of expressions implemented by the current version of the tool described in Section 6 is as follows

$$
\begin{aligned}
\mathit{Expression} ::= {} & \mathit{StringExpr} \mid \mathit{BagExpr} \\
& \mid \mathit{IntExpr} \mid \mathit{DateExpr} \mid \mathit{BoolExpr}
\end{aligned}
$$

$$
\mathit{StringExpr} ::= \mathsf{name} \mid \mathsf{string\text{-}value}
$$

$$
\mathit{BagExpr} ::= \mathsf{string\text{-}bag}(\mathit{StringExpr}, \ldots, \mathit{StringExpr})
$$

$$
\begin{aligned}
\mathit{IntExpr} ::= {} & \mathsf{name} \mid \mathsf{integer\text{-}value} \\
& \mid \mathsf{integer\text{-}add}(\mathit{IntExpr}, \mathit{IntExpr}) \\
& \mid \mathsf{integer\text{-}subtract}(\mathit{IntExpr}, \mathit{IntExpr}) \\
& \mid \mathsf{integer\text{-}multiply}(\mathit{IntExpr}, \mathit{IntExpr}) \\
& \mid \mathsf{integer\text{-}divide}(\mathit{IntExpr}, \mathit{IntExpr}) \\
& \mid \mathsf{integer\text{-}mod}(\mathit{IntExpr}, \mathit{IntExpr})
\end{aligned}
$$

$$
\mathit{DateExpr} ::= \mathsf{name} \mid \mathsf{dateTime\text{-}value}
$$

$$
\begin{aligned}
\mathit{BoolExpr} ::= {} & \mathsf{name} \mid \mathsf{true} \mid \mathsf{false} \\
& \mid \mathsf{string\text{-}equal}(\mathit{StringExpr}, \mathit{StringExpr}) \\
& \mid \mathsf{integer\text{-}equal}(\mathit{IntExpr}, \mathit{IntExpr}) \\
& \mid \mathsf{boolean\text{-}equal}(\mathit{BoolExpr}, \mathit{BoolExpr}) \\
& \mid \mathsf{and}(\mathit{BoolExpr}, \ldots, \mathit{BoolExpr}) \\
& \mid \mathsf{or}(\mathit{BoolExpr}, \ldots, \mathit{BoolExpr}) \\
& \mid \mathsf{not}(\mathit{BoolExpr}) \\
& \mid \mathsf{integer\text{-}greater\text{-}than}(\mathit{IntExpr}, \mathit{IntExpr}) \\
& \mid \mathsf{integer\text{-}greater\text{-}than\text{-}or\text{-}equal}(\mathit{IntExpr}, \mathit{IntExpr})
\end{aligned}
$$

$$
\begin{aligned}
& \mid \mathsf{integer\text{-}less\text{-}than}(\mathit{IntExpr}, \mathit{IntExpr}) \\
& \mid \mathsf{integer\text{-}less\text{-}than\text{-}or\text{-}equal}(\mathit{IntExpr}, \mathit{IntExpr}) \\
& \mid \mathsf{dateTime\text{-}greater\text{-}than}(\mathit{DateExpr}, \mathit{DateExpr}) \\
& \mid \mathsf{dateTime\text{-}greater\text{-}than\text{-}or\text{-}equal}(\mathit{DateExpr}, \mathit{DateExpr}) \\
& \mid \mathsf{dateTime\text{-}less\text{-}than}(\mathit{DateExpr}, \mathit{DateExpr}) \\
& \mid \mathsf{dateTime\text{-}less\text{-}than\text{-}or\text{-}equal}(\mathit{DateExpr}, \mathit{DateExpr}) \\
& \mid \mathsf{string\text{-}at\text{-}least\text{-}one\text{-}member\text{-}of}(\mathit{BagExpr}) \\
& \mid \mathsf{string\text{-}subset}(\mathit{BagExpr}, \mathit{BagExpr}) \\
& \mid \mathsf{string\text{-}regexp\text{-}match}(\mathit{StringExpr}, \mathit{StringExpr})
\end{aligned}
$$

The syntax is parameterized by four countable and pairwise disjoint sets of values: the set of *strings* (ranged over by string-value), the set of *integers* (ranged over by integer-value), the set of *dates* (ranged over by dateTime-value), and the set of *booleans* (i.e. {true, false}). In the paper, we use value to range over a generic value.

Expressions are built from values by applying functions; the definitions of such functions are standard and are given in [3, Appendix A.3].

The list of *matching functions* supported by the tool is defined according to the expression language defined above:

$$
\begin{aligned}
\mathit{MatchId} ::= {} & \text{string-equal} \mid \text{integer-equal} \mid \text{boolean-equal} \\
\mid {} & \text{and} \mid \text{or} \mid \text{not} \\
\mid {} & \text{integer-greater-than} \mid \text{integer-greater-than-or-equal} \\
\mid {} & \text{integer-less-than} \mid \text{integer-less-than-or-equal} \\
\mid {} & \text{dateTime-greater-than} \mid \text{dateTime-greater-than-or-equal} \\
\mid {} & \text{dateTime-less-than} \mid \text{dateTime-less-than-or-equal}
\end{aligned}
$$

B Combining algorithms

We report in this appendix the definitions of the combining algorithms not described in the paper. Specifically, we first present in Section B.1 the algorithms for combining rules and then, in Section B.2 those for combining policies.

B.1 Rule-combining algorithms

Listing 9. Rule combining algorithm permit-overrides

```
Boolean atLeastOneError = false;
Boolean potentialPermit = false;
Boolean atLeastOneDeny  = false;
foreach (rule ∈ Rules) {
   DecisionTuple t = ⟦ rule ⟧{r};
   if (r ∈ t↓deny) {
      atLeastOneDeny = true;
      continue;
   }
   if (r ∈ t↓permit) return permit;
   if (r ∈ t↓not-applicable) continue;
   if (r ∈ t↓indeterminate) {
      atLeastOneError = true;
      if (rule.effect == permit) potentialPermit = true;
      continue;
   }
}
if (potentialPermit) return indeterminate;
if (atLeastOneDeny) return deny;
if (atLeastOneError) return indeterminate;
return not-applicable;
```

Listing 10. Rule combining algorithm ordered-deny-overrides

```
The behavior of the algorithm is identical to that of
deny-overrides with one exception: the instruction
‘foreach (rule ∈ Rules)‘ traverses the rules within
Rules in the order in which they are listed in the policy.
```

Listing 11. Rule combining algorithm ordered-permit-overrides

```
The behavior of the algorithm is identical to that of
permit-overrides with one exception: the instruction
'foreach (rule ∈ Rules)' traverses the rules within
Rules in the order in which they are listed in the policy.
```

Listing 12. Rule combining algorithm first-applicable

```
foreach (rule ∈ Rules) {
    DecisionTuple t = ⟦ rule ⟧{r};
    if (r ∈ t↓deny) return deny;
    if (r ∈ t↓permit) return permit;
    if (r ∈ t↓not-applicable) continue;
    if (r ∈ t↓indeterminate) return indeterminate;
}
return not-applicable;

NB: the instruction 'foreach (rule ∈ Rules)' traverses the
rules within Rules in the order in which they are listed in
the policy
```

B.2 Policy-combining algorithms

Listing 13. Policy combining algorithm permit-overrides

```
Boolean atLeastOneError = false;
Boolean atLeastOneDeny = false;
foreach (policy ∈ Policies) {
    DecisionTuple t = ⟦ policy ⟧{r};
    if (r ∈ t↓deny) {
        atLeastOneDeny = true;
        continue;
    }
    if (r ∈ t↓permit) return permit;
    if (r ∈ t↓not-applicable) continue;
    if (r ∈ t↓indeterminate) {
        atLeastOneError = true;
        continue;
    }
}
if (atLeastOneDeny) return deny;
if (atLeastOneError) return indeterminate;
return not-applicable;
```

Listing 14. Policy combining algorithm ordered-deny-overrides

```
The behavior of the algorithm is identical to that of
deny-overrides with one exception: the instruction
```

```
'foreach (policy ∈ Policies)' traverses the policies within
Policies in the order in which they are listed in the policy
set.
```

Listing 15. Policy combining algorithm ordered-permit-overrides

```
The behavior of the algorithm is identical to that of
permit-overrides with one exception: the instruction
'foreach (policy ∈ Policies)' traverses the policies within
Policies in the order in which they are listed in the policy
set.
```

Listing 16. Policy combining algorithm first-applicable

```
foreach (policy ∈ Policies) {
   DecisionTuple t = ⟦ policy ⟧{r};
   if (r ∈ t↓deny) return deny;
   if (r ∈ t↓permit) return permit;
   if (r ∈ t↓not-applicable) continue;
   if (r ∈ t↓indeterminate) return indeterminate;
}
return not-applicable;

NB: the instruction 'foreach (policy ∈ Policies)' traverses the
policies within Policies in the order in which they are listed
in the policy set
```

Listing 17. Policy combining algorithm only-one-applicable

```
Boolean atLeastOne      = false;
Policy selectedPolicy = null;
foreach (policy ∈ Policies) {
   MatchingTuple t = (| policy.target |){r};
   if (r ∈ t↓indeterminate) return indeterminate;
   if (r ∈ t↓match) {
      if (atLeastOne) {
         return indeterminate;
      } else {
         atLeastOne = true;
         selectedPolicy = policy;
      }
   }
   if (r ∈ t↓no-match) continue;
}
if (atLeastOne) {
   return ⟦ selectedPolicy ⟧{r};
} else {
   return not-applicable;
}
```