A Formal Software Engineering Approach to Policy-Based Access Control

Andrea Margheri\textsuperscript{1,2}, Massimiliano Masi\textsuperscript{2}, Rosario Pugliese\textsuperscript{1}, and Francesco Tiezzi\textsuperscript{3}

\textsuperscript{1} Università degli Studi di Firenze, Viale Morgagni, 65 - 50134 Firenze, Italy
\text{\{andrea.margheri,rosario.pugliese\}@unifi.it}

\textsuperscript{2} Università di Pisa, Largo Bruno Pontecorvo, 3 - 56127 Pisa, Italy
margheri@di.unipi.it

\textsuperscript{3} Tiani “Spirit” GmbH, Guglgasse, 6 - 1110 Vienna, Austria
massimiliano.masi@tiani-spirit.com

\textsuperscript{4} IMT Advanced Studies Lucca, Piazza S. Ponziano, 6 - 55100, Lucca, Italy
francesco.tiezzi@imtlucca.it

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 recently defined a standard, called XACML, for policy-based access control. An XACML policy is an XML document specifying the capabilities and credentials 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 an Eclipse-based 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

\textsuperscript{*} This work has been partially sponsored by the EU project ASCENS (257414).
# Table of Contents

A Formal Sw. Engineering Approach to Policy-Based Access Control .......................... 1
   A. Margheri, M. Masi, R. Pugliese and F. Tiezzi

1 Introduction ......................................................... 4

2 The eXtensible Access Control Markup Language (XACML) ................................. 6
   2.1 The underlying model ...................................... 6
   2.2 A glimpse of the XACML language .................... 8
   2.3 XACML informal semantics .............................. 9

3 Case Studies ........................................................ 15
   3.1 The eHealth domain ...................................... 15
   3.2 A Cloud IaaS Scenario .................................. 26

4 The Formal Access Control Policy Language (FACPL) ........................................ 28
   4.1 Syntax of policies ........................................ 28
   4.2 Policy examples ........................................... 32
      eHealth domain ........................................... 32
      Cloud domain ............................................. 34
   4.3 Syntax of requests and responses .................... 36
   4.4 Request and response examples ....................... 38
   4.5 FACPL vs XACML .......................................... 39

5 FACPL formal semantics ........................................... 41
   5.1 Semantics of the PDP decision process .............. 41
      Semantics without obligations ......................... 42
      Semantics with obligations ............................ 46
   5.2 Semantics of the PEP enforcement process .......... 49

6 Examples ............................................................. 51
   6.1 The decision process .................................... 51
      Deny-all policy ........................................... 51
      Privacy consent policy ................................ 53
      Privacy consent policy set .............................. 59
   6.2 The enforcement process ................................. 61

7 FACPL Supporting Tools ........................................... 62
   7.1 The FACPL library ........................................ 62
   7.2 The FACPL development environment ................. 68
   7.3 IaaS case study implementation ..................... 70

8 Related Work ....................................................... 71

9 Concluding remarks .............................................. 73

References ............................................................ 76

A Combining algorithms .............................................. 77
   A.1 A possible custom algorithm ......................... 79

B Cloud Case Study Policies ....................................... 79
Listings

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Deny-all policy</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>Privacy consent policy</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>Privacy consent policy set</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>Request by Dr. Marley</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>Decision for request in Listing 4 with respect to policy in Listing 2</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>Decision for request in Listing 4 with respect to policy set in</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Listing 3</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>Request by Mr. Elliot</td>
<td>23</td>
</tr>
<tr>
<td>8</td>
<td>Decision for request in Listing 7 with respect to policy in Listing 2</td>
<td>25</td>
</tr>
<tr>
<td>9</td>
<td>Decision for request in Listing 7 with respect to policy set in</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Listing 3</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>Instation of TYPE_1 VM</td>
<td>34</td>
</tr>
<tr>
<td>11</td>
<td>Freezing of VM</td>
<td>35</td>
</tr>
<tr>
<td>12</td>
<td>Load balancing condition</td>
<td>35</td>
</tr>
<tr>
<td>13</td>
<td>Release policy</td>
<td>36</td>
</tr>
<tr>
<td>14</td>
<td>XACML Load Balancing Condition</td>
<td>40</td>
</tr>
<tr>
<td>15</td>
<td>The algorithm permit-overrides with formal parameters</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sequence and r</td>
<td>44</td>
</tr>
<tr>
<td>16</td>
<td>The algorithm permit-overrides with obligations handling</td>
<td>47</td>
</tr>
<tr>
<td>17</td>
<td>Java patient summary policy</td>
<td>66</td>
</tr>
<tr>
<td>18</td>
<td>The combining algorithm deny-overrides</td>
<td>77</td>
</tr>
<tr>
<td>19</td>
<td>Combining algorithm deny-unless-permit</td>
<td>77</td>
</tr>
<tr>
<td>20</td>
<td>Combining algorithm permit-unless-deny</td>
<td>78</td>
</tr>
<tr>
<td>21</td>
<td>Combining algorithm first-applicable</td>
<td>78</td>
</tr>
<tr>
<td>22</td>
<td>Policy-Combining algorithm only-one-applicable</td>
<td>78</td>
</tr>
<tr>
<td>23</td>
<td>Energy saver policy</td>
<td>79</td>
</tr>
<tr>
<td>24</td>
<td>Load balancer policy</td>
<td>80</td>
</tr>
<tr>
<td>25</td>
<td>Load balancer creation policy</td>
<td>82</td>
</tr>
<tr>
<td>26</td>
<td>Release policy</td>
<td>92</td>
</tr>
</tbody>
</table>
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, often 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.

Models based on access control lists are the most basic form of access control, and they are still the basis of many 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, and on the basis of the requested action (such as read, write, etc.). In these models, each resource has its own list of permissions that must be checked every time an access to the resource is requested. This, however, 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 semantics 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] model has been introduced. In this model, a protected resource is governed by one or more policies that exactly specify what capabilities and credentials a requestor needs to fulfill in order to obtain access. PBAC overcomes scalability problems of previous models by resorting to credentials (called attributes) exhibited in the requests, concerning the requestor, the environment, the resource itself, etc., to base the corresponding access decisions. In fact, this form of fine-grained access control does not require to know the user population in advance. Moreover, PBAC provides support for security-relevant contextual information, such as time, location, or environmental state at the time the access requests are made, which are sometimes important for the access control decisions. These context parameters capture the dynamically changing access requirements and, hence, are critical to the effectiveness of the resulting access control scheme. PBAC is by now the de-facto standard model for enforcing access control policies in service-oriented architectures.
A widely used instantiation of the PBAC model is given by the eXtensible Access Control Markup Language (eXtensible Access Control Markup Language (XACML) \[3\]), developed by the international standardization consortium OASIS\[5\]. It defines a language for expressing 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 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 some of the possibilities available in the policy language. Most of all, XACML comes without a formal semantics. The specification document 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. Such deficiency is further worsened by 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 (version 3.0) that clarifies all ambiguous and intricate aspects of the specification. In order to get around the complexity introduced by XML, we propose an alternative syntax. This way, we get a tiny language, called Formal Access Control Policy Language (FACPL), with solid mathematical foundations that lay the basis for developing tools, methodologies, and analysis techniques (i.e., policy redundancies, ordering, constraint analysis, etc.) that can be easily used by software engineers to precisely define access control policies on resources. To demonstrate feasibility and effectiveness of our approach, we have developed a Java-based implementation of FACPL, by faithfully relying on its semantics rules, and an Eclipse-based environment supporting the specification of policies. We have thus obtained a complete software architecture for evaluating access requests with respect to a collection of policies.

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 \[11\], and in Section 3 by illustrating the main features of the policy language through some examples borrowed from an eHealth project and from a Cloud scenario. In Section 4 we introduce an alternative syntax for XACML,\[5\] http://www.oasis-open.org
which we then use in Section 5 as the basis to define the formal semantics. In Section 6, we show how the formal semantics definitions apply to the policy examples introduced in Section 2, while in Section 7, we describe our Java-based implementation of the formal semantics. In Section 8, we discuss more closely related work. Finally, in Section 9, we touch upon directions for future work.

2 The eXtensible Access Control Markup Language (XACML)

In the access control model underlying XACML, each resource can be paired with one or more policies, namely XML documents expressing the capabilities and credentials that a requestor needs to fulfill in order to access the resource. In this section, we provide an overview of XACML that covers its access control model, the languages for expressing policies and requests, and its informal semantics.

2.1 The underlying model

The data-flow diagram shown in Figure 1 describes how the major roles involved in the XACML model interact with each other. The Policy Administration Points (PAPs) write policies and policy sets and make them available to the Policy Decision Point (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, which can select policies by following specific criteria on the base of the requested resource or of environmental constraints; however, the XACML specification does not require it. The policies and policy sets retrieved by the PDP represent the set of policies to be used for the current requests.

A request to access a resource is created (à la façade pattern) by a Policy Enforcement Point (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, an email user-agent, etc. Thus, we cannot expect that in an enterprise setting all PEPs issue access requests to a PDP directly in a common format. Therefore, the requests and responses to be handled by the PDP must be converted by the context handler in a canonical form, i.e., the so-called XACML request. The obvious benefit of this approach is that policies may be written and evaluated independently of the specific environment in which they have to be enforced. Notably, at implementation level, different solutions can be applied: the context handler can either be part of the PEP (implicit context) or be an external independent entity (explicit context). Anyway, these low-level aspects are out of the scope of the XACML specification.

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 as attribute elements (step 3). Attributes are identified by a category, a name and a type. The context handler in this phase may add
environmental attributes to the request. Then, it 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 and can ask to the Policy Information Point (PIP) for new attributes (steps 5-10). The decision statement computed by the PDP can include additional actions, called obligations (resp. advices) that must (resp. should) be understood and discharged by the PEP in order to enforce the PDP decision. The decision statement is then sent to the context handler (step 11) that 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
effects). The internal PDP decision process is based on the so-called extended
indeterminate values, i.e. indeterminate-P, indeterminate-D and indeterminate-
DP, which specify the potential decisions that could have been taken if there
would not have been a problem during the evaluation leading to the indetermi-
nate result. Specifically, the first value indicates that a policy (or a part of it)
evaluated to indeterminate could have been evaluated to permit but not to deny.
The second value has the dual meaning. The last value means that the potential
decision could have been both permit or deny. Extended indeterminate values
allow the PDP to obtain additional information about policy evaluation which
can be exploited, e.g., for improving the treatment of errors. They are used only
internally by the PDP and are converted to indeterminate before being sent to
the PEP.

If the response received by the PEP contains obligations and/or advices, the
PEP has to discharge them in order to enforce the decision (step 13). Thus, the
authorization outcome of the PEP could be different from the decision taken by
the PDP.

This paper mainly focuses on steps 4, 11, and 13 of the XACML workflow
and, in particular, on the evaluation process performed by the PDP to take
authorization decisions and on the obligations/advices discharging process per-
formed by the PEP.

2.2 A glimpse of the XACML language

In this section, we provide an informal presentation of the XML-based language
for expressing access policies and requests proposed by XACML. The elements
emphasised in sans-serif, e.g. Rule, are XML elements, while elements’ attributes
are shown in italics.

The basic elements of the policy language are the Rules. A Rule specifies the
logic for the access control decision by means of an effect, that can be either
permit or deny; a Target, that indicates to which requests the rule applies; a Con-
dition, that is an expression on values coming from the request’s attributes for
refining the applicability established by the target; and some ObligationExpres-
sions and AdviceExpressions, that define the obligations and advices associated
to the rule.

A Target consists of a conjunctive sequence of AnyOf elements. Each AnyOf
element contains a disjunctive sequence of AllOf elements, each of which contains
a conjunctive sequence of Match elements. A Match element specifies an attribute
identifier, a typed value, and a matching function. Such information is used to
determine whether the parent element of the Target (e.g., a Rule) is applicable to a
given request. Specifically, the matching function retrieves a set of values from
the designed attribute in the request and matches them with the value specified
in the Match element, according to the function’s semantics. If, for all AnyOf
elements, at least the evaluation of an AllOf element is positive (i.e. all matchings
of its Match elements succeed), the Target’s parent element is applicable to the
request. Notably, if the Target is empty, the parent element applies to any request.
A formal sw. eng. approach to Policy-Based Access Control

A set of Rules can be combined into a Policy that, besides its own Target and obligations/advices, specifies a combining algorithm that, from the set of rules' decisions, computes what is the policy decision corresponding to a given request. Finally, a set of policies can be combined together into a PolicySet that, again, specifies a combining algorithm, a target and some obligations/advices. A PolicySet can be also nested within another one, thus creating an hierarchical structure.

Rules, policies and policy sets can specify obligations and advices, which indicate the actions (and their arguments) that the PEP must be able to discharge in order to enforce permit or deny authorization decisions. Obligations and advices have attached an option for applicability, called FulfillOn and AppliesTo respectively, which must match the decision calculated by the PDP for the parent element. To discharge an obligation, the PEP must be able to process the corresponding action and its argument values, as computed by the PDP, and perform successfully such an action. Instead, advices may be safely ignored if an error occurs while discharging them. In case of a permit (deny) decision, the PEP grants (forbids) access only if all associated obligations are successfully discharged; otherwise the enforced decision depends on the chosen enforcement algorithm.

Let us now consider the language provided by XACML for expressing context data, i.e. requests and responses. A Request is the request written in the canonical form (created by the PEP or by the context handler) composed of attribute/value pairs. The elements specifying such pairs are grouped by category, identifying e.g. the subject, the action, etc. Category identifiers are used in combination with attribute identifiers to retrieve attribute values from the request. Finally, a Response contains the authorization decision produced by the PDP. A decision includes the effect resulting from the evaluation of a request against a given set of policies and, possibly, the associated sets of obligations and advices.

2.3 XACML informal semantics

In this section, we present the evaluation of XACML policies, in a top-down fashion.

When the PDP receives a request, it starts the evaluation procedure on the basis of the retrieved policies. 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 (defined as a configuration parameter of the PDP) and the set of the retrieved policies/policy sets. The evaluation result will be returned as a 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 the request, the returned value is not-applicable, while if the target evaluates to indeterminate then it is returned an extended indeterminate value determined by evaluating the enclosed policies and applying the conversion rules shown in Table 2. Instead, if the target matches, the enclosed policies are evaluated and the results are combined according to the combining algorithm.
Table 1. Policy set evaluation (without obl. and adv.)

<table>
<thead>
<tr>
<th>Target</th>
<th>Policy Set Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>match</td>
<td>specified by the policy-combining algorithm</td>
</tr>
<tr>
<td>no-match</td>
<td>not-applicable</td>
</tr>
<tr>
<td>indeterminate</td>
<td>see Table 2</td>
</tr>
</tbody>
</table>

Table 2. Conversion of decision values into extended indeterminate values (to be applied in case of target indeterminate)

<table>
<thead>
<tr>
<th>Combining Algorithm Value</th>
<th>Policy Set Value or Policy Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>not-applicable</td>
<td>not-applicable</td>
</tr>
<tr>
<td>permit</td>
<td>indetP</td>
</tr>
<tr>
<td>deny</td>
<td>indetD</td>
</tr>
<tr>
<td>indeterminate</td>
<td>indetDP</td>
</tr>
<tr>
<td>indetDP</td>
<td>indetDP</td>
</tr>
<tr>
<td>indetD</td>
<td>indetD</td>
</tr>
</tbody>
</table>

algorithm specified in the policy set. How conflicts and special situations are managed, e.g., all policies are not applicable, it is specific to each combining algorithm.

The algorithms provided by XACML (see [3, Appendix C]) for combining the values resulting from rules, policies and policy sets evaluation are as follows:

- **deny-overrides**: if any policy in the considered set evaluates to deny, then the result of the policy set combination is deny. In other words, deny takes precedence, regardless of the result of evaluating any of the other policies in the policy set. Instead, if at least a policy evaluates to permit and all others evaluate to not-applicable, permit or indetP, then the result of the combination is permit. If all policies are found to be not-applicable to the decision request, then the policy set evaluates to not-applicable. Finally, a policy evaluation results in indetDP, indetD and indetP in the remaining cases, according to specific error situations.

- **permit-overrides**: this combination algorithm is the dual of the previous one, i.e. this time permit takes precedence over the other results.

- **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 in which they occur in the policy set.

- **deny-unless-permit**: this algorithm is similar to permit-overrides, because it is intended for those cases where a permit decision takes precedence over deny

---

6 The behaviour of the algorithms does not depend on the fact that they combine the results of evaluating rules, policies or policy sets. Therefore, for the sake of simplicity, we explain them as if they combine the results of evaluating policies enclosed in a policy set.
decisions; differently from permit-overrides, this algorithm returns neither not-applicable nor an (extended) indeterminate value.

- permit-unless-deny: this algorithm is the dual of the previous one, i.e. deny takes precedence over permit decisions.

- 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 halts and the policy set evaluates to indeterminate.

- 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 PDP decision process uses the extended indeterminate values. Therefore, the decision indeterminate possibly returned by the algorithms first-applicable and only-one-applicable is converted (as shown in Table 2) to indetDP for further combinations.

In case of decision permit or deny, any algorithm also returns the corresponding set of obligations and advices. This set is constructed from the obligations and advices returned by the enclosed items and may differ from one execution to the other of the evaluation process. For example, the algorithms permit-overrides or deny-overrides do not necessarily process the policies as they appear in a policy set. Therefore, if more than one policy can be evaluated, the resulting set of obligations and advices depends on where the policy evaluation stopped.

Remark 1. The use of extended indeterminate values has been introduced in the version 3.0 of XACML. This approach for handling the case where a target is indeterminate does not completely convince us. Indeed, on the one hand, it introduces additional evaluations just to obtain decision values that are used only internally to the PDP. In case of complex policies, this may considerably affect the evaluation performances. On the other hand, the treatment of the not-applicable case in Table 2 seems quite odd: if the target of the policy set is indeterminate and the combining algorithm applied to the enclosed policies returns not-applicable, then the policy set evaluates to not-applicable, rather than to an extended indeterminate value (e.g., indetDP). This significantly contrasts with the previous versions of XACML.

Anyway, PDP decisions sent to the PEP can only contain values permit, deny, not-applicable and indeterminate. Therefore, if it is needed, the extended indeterminate values indetP, indetD and indetDP are simply converted to indeterminate.
The evaluation of a single policy is similar to that of a policy set, as shown in Table 3. First the policy’s target is evaluated; then, if the target matches the request or an indeterminate results, the value of the policy is determined by the evaluation of the enclosed rules. The combining algorithms available in case of simple policies are the same as those for policy sets described before (except for only-one-applicable that cannot be used as rule-combining algorithm).

The evaluation of a rule, described in Table 4, consists of first evaluating its target and then, if it is needed, its condition. If the target does not match or evaluates to indeterminate then there is no need to evaluate the condition: the evaluation returns not-applicable or indetP/indetD, respectively. If the target matches, then the condition is evaluated and, if it is satisfied, the result of the evaluation is the effect specified in the rule, which is then propagated to the upper-level policy. If the condition is missing then it evaluates to true by default, while if it evaluates to neither true nor false (e.g. some error occurs during the evaluation) then the value indeterminate is returned. Notably, if the target or the condition returns indeterminate, then the evaluation of the rule returns indetP or indetD depending on the rule’s effect.

The evaluation of a target, described in Table 5, is determined by combining the results of the evaluation of its AnyOf elements. A target matches if all AnyOf enclosed elements do match. If at least one of them evaluates to no-match then the evaluation of the target returns no-match; otherwise, it returns indeterminate. According to Table 4, an AnyOf element matches if at least one of its AllOf elements matches. If no element matches and at least one of them is indeterminate then also the AnyOf is indeterminate. Otherwise, if all AllOf do not match, the result is no-match. Finally, as described in Table 7, an AllOf element matches if all match elements it encloses evaluate to true. Instead, if at least one of them evaluates to false the AllOf element is no-match; otherwise it is indeterminate.
Table 5. Target evaluation

<table>
<thead>
<tr>
<th>AnyOf Values</th>
<th>Target Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>All match</td>
<td>match</td>
</tr>
<tr>
<td>At least one no-match</td>
<td>no-match</td>
</tr>
<tr>
<td>No no-match and at least one indeterminate</td>
<td>indeterminate</td>
</tr>
</tbody>
</table>

Table 6. AnyOf evaluation

<table>
<thead>
<tr>
<th>AllOf Values</th>
<th>AnyOf Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>At least one match</td>
<td>match</td>
</tr>
<tr>
<td>All no-match</td>
<td>no-match</td>
</tr>
<tr>
<td>No match and at least one indeterminate</td>
<td>indeterminate</td>
</tr>
</tbody>
</table>

Table 7. AllOf evaluation

<table>
<thead>
<tr>
<th>Match Values</th>
<th>AllOf Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>All true</td>
<td>match</td>
</tr>
<tr>
<td>At least one false</td>
<td>no-match</td>
</tr>
<tr>
<td>No false and at least one indeterminate</td>
<td>indeterminate</td>
</tr>
</tbody>
</table>

Table 8. Match evaluation

<table>
<thead>
<tr>
<th>Results of Match Function Applications to Retrieved Request’s Values</th>
<th>Match Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>At least one true</td>
<td>true</td>
</tr>
<tr>
<td>All false</td>
<td>false</td>
</tr>
<tr>
<td>No true and at least one indeterminate</td>
<td>indeterminate</td>
</tr>
</tbody>
</table>

The evaluation of a match element with a matching function identified by `MatchId`, an attribute identifier `name` (specified by means of either an `attribute designator` or an `attribute selector`) and a value `value`, is as follows. If `name` does not identify any value in the request (e.g., there is no attribute matching with `name`), the result is either `false` or `indeterminate` depending on the `MustBePresent` option specified in the designator/selector: if `MustBePresent` is set to `false`, the match element evaluates to `false`, otherwise to `indeterminate`. The `indeterminate` value is returned also if some error occurs while retrieving the values of attribute `name` from the request. Otherwise, the function `MatchId` is applied between `value` and each value identified by `name`, and the evaluation result is determined as shown in Table 8. If at least one of those function applications evaluates to `true`, then the result is `true`; otherwise, if at least one of the applications returns `indeterminate`, then the result is `indeterminate`; finally, if all applications evaluate to `false`, then the result is `false`.

We conclude by discussing the evaluation of obligations and advices. They are evaluated only if their applicability option (i.e., `FullfillOn` for obligations and `AppliesTo` for advices) has the same value, `permit` or `deny`, as the current decision of their parent element (which can be either a rule, a policy or a policy set). If an obligation or advice is applicable, then the PDP retrieves all action’s arguments and evaluates their expressions. If an error occurs during the retrieval, the parent element is evaluated to `indeterminate`. As shown in Table 9, a value `permit` or `deny`
<table>
<thead>
<tr>
<th>Rule/Policy/Policy Set Value</th>
<th>Rule/Policy/Policy Set Final Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>permit</td>
<td>indetP if an error occurs in obl. with FullfillOn=permit or adv. with AppliesTo=permit</td>
</tr>
<tr>
<td></td>
<td>permit otherwise</td>
</tr>
<tr>
<td>deny</td>
<td>indetD if an error occurs in obl. with FullfillOn=deny or adv. with AppliesTo=deny</td>
</tr>
<tr>
<td></td>
<td>deny otherwise</td>
</tr>
<tr>
<td>not-applicable</td>
<td>not-applicable</td>
</tr>
<tr>
<td>indetDP</td>
<td>indetDP</td>
</tr>
<tr>
<td>indetP</td>
<td>indetP</td>
</tr>
<tr>
<td>indetD</td>
<td>indetD</td>
</tr>
</tbody>
</table>

can become an indetP or indetD [7] respectively, if at least one of the matching obligations and advices cause an error.

When the PDP completes the decision process, the resulting authorization decision, which can include obligations and advices, is sent to the PEP for the enforcing process. The PEP does not refer to the request for its process, but only to the decision statement received from the PDP. To enforce the authorization decision, the PEP must understand and discharge obligations and advices possibly included in the PDP decision. Errors occurring while processing advices may be safely ignored. Specifically, a PEP conforming to XACML shall forbid access unless all obligations are precisely discharged. In case of exceptions, e.g. some obligations cannot be discharged, the decision taken depends on the enforcement algorithm chosen for the PEP.

XACML (see [3, Section 7.2]) provides three enforcement algorithms for managing discharging of obligations:

- **base**: if the decision is permit (resp. deny) and all obligations are successfully discharged, then the PEP grants (resp. forbids) access, otherwise the PEP’s behaviour is undefined;
- **deny-biased**: if the decision is permit and all obligations are successfully discharged, then the PEP grants access, otherwise it forbids access;
- **permit-biased**: if the decision is deny and all obligations are successfully discharged, then the PEP forbids access, otherwise it grants access.

Basically, the PEP’s algorithms differ in the way they handle decisions not-applicable and indeterminate: these decisions are converted to deny by deny-biased and to permit by permit-biased, while the behaviour of the base algorithm is undefined in these cases. In the most common scenario, where the access must be granted only if the requestor is correctly authorized by the considered policies, the most suitable enforcement algorithm is deny-biased. The permit-biased algorithm, instead, should be used when granting the access has the priority over forbidding it, i.e. the access is denied only if applicable policies state it beyond

---

[7] In the XACML specification these effects are not clearly specified, they have been inferred from a careful study of the overall specification [3].
any reasonable doubt. This behaviour could be appropriate when there is an emergency and something should be allowed that under normal conditions would not be. Many use-cases requiring this kind of behaviour are related to human safety as when, for example, a doctor could have to override a patient privacy restriction for guaranteeing the patient’s safety. In fact, in case of emergency, if a request made by a doctor for accessing the healthcare data of a patient evaluates to an ‘uncertain’ value (i.e., not-applicable and indeterminate), it could be acceptable to grant the access. Moreover, in case the request evaluation returns the decision deny, such decision could have associated an obligation requiring the PEP to contact a close relative that has been delegated to authorize or not the access. If the obligation cannot be discharged, e.g. it is not possible to contact the relative, then we fall back into an uncertain situation and, hence, the access is granted.

The final decision returned by the enforcement algorithm is applied by the PEP to the requestor. Usually, a requestor expects only two possible decisions: the access is either granted or forbidden. In fact, these are the only results explicitly considered by the XACML specification. Anyway, XACML supports extensibility to accommodate other decision results (e.g., indeterminate or not-applicable) through the base algorithm that, for this reason, has been left partially unspecified. Returning decisions indeterminate or not-applicable can be useful for policy testing or when they imply other actions, e.g. reformulation and resubmission of the request. Such aspects are out of the scope of the XACML specification, since they may depend by the specific use cases.

3 Case Studies

In this section we present two different case studies for illustrating potentialities and effectiveness of our approach. The first case study concerns on the exchange of medical information to support patient’s treatment and prescription across different countries. While in the second case study is presented a policy-based Cloud platform able to instantiate two different kinds of virtual machine over a physical machines. Through the policies it is also applied in the platform different resource usage and adaptation techniques.

3.1 The eHealth 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 [13]. To illustrate our approach, we use as a case study the policies in Listings 1 and 2 and the policy set in Listings 3 made by grouping the two previous policies. These policies are strongly inspired by those defined in the epSOS project. However, since the original epSOS policies follow the XACML 2.0 syntax [11] while we focus on the last version of the
language, we have converted them into the version 3.0 of XACML. Moreover, they are enriched with obligations for illustrating how these can be used and dealt with.

The policy in Listings 1 forbids access to all resources for any requestor.

**Listing 1.** Deny-all policy

```xml
<Policy xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   PolicyId="urn:oasis:names:tc:xacml:3.0:policy:deny_all"
   RuleCombiningAlgId="urn:oasis:names:tc:xacml:3.0:rule-combining-algorithm:permitoverrides"
   Version="1.0" MaxDelegationDepth="1">
  <Description>Opt-out</Description>
  <Target />
  <Rule RuleId="urn:oasis:names:tc:xacml:3.0:denyRule" Effect="Deny"/>
</Policy>
```

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. Therefore, for every request, the outcome of the policy evaluation will be the rule’s effect, i.e. deny.

The policy in Listing 2 expresses the permissive part of the patient privacy consent. In this domain, each role (e.g. doctor, nurse, pharmacist) has permissions to perform a certain coded action [14] for a certain purpose (e.g. healthcare treatment, statistics, emergency).

**Listing 2.** Privacy consent policy

```xml
<Policy xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   PolicyId="urn:oasis:names:tc:xacml:3.0:policy:privacy" Version="1.0"
   RuleCombiningAlgId="urn:oasis:names:tc:xacml:3.0:rule-combining-algorithm:permitoverrides"
   MaxDelegationDepth="1">
  <Target>
    <AnyOf>
      <AllOf>
        <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue>
            <DataType>http://www.w3.org/2001/XMLSchema#string</DataType>
            medical doctor
          </AttributeValue>
          <AttributeDesignator>
            <DataType>http://www.w3.org/2001/XMLSchema#anyURI</DataType>
            MustBePresent="false"
            Category="urn:oasis:names:tc:xacml:3.0:
```

To the best of our knowledge, no significative XACML 3.0 policies used in large scale projects are publicly available. Therefore, we have chosen the epSOS policies as a case study rather than using policies natively written in XACML 3.0 syntax.
A formal sw. eng. approach to Policy-Based Access Control

```xml
<Match Id="urn:oasis:names:tc:xacml:1.0:attribute-category:subject"/>

</AnyOf>
</Target>
</Rule RuleId="rule1" Effect="Permit">

<Description>
Matches all the READ operations to requests containing the correct permissions
</Description>

<Target>
<AnyOf>
<Match MatchId="urn:oasis:names:tc:xacml:1.0:attribute-category:resource">

<AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string">
Read
</AttributeValue>
</Match>

</AnyOf>
</Target>
</Rule>
```
The target of the policy specifies the credentials the subject must exhibit in order to access a given resource. Specifically, this policy applies to a subject with the medical doctor role that has the purpose of accessing a resource with code identifier 34133-9, which identifies a patient summary in the international
A formal sw. eng. approach to Policy-Based Access Control

If these capabilities are met, the rules enclosed in the policy are evaluated. The internal 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 for accessing the resource. The policy specifies the permit-oversides combining algorithm and an obligation that record the history of attempted requests. Thus, when the policy is evaluated as permit, a log operation, which records who has been authorized to access the resource, must be discharged in order to enforce the decision.

The definition of the patient privacy consent is then completed by grouping the previous two policies into the policy set reported in Listing 3.

Listing 3. Privacy consent policy set

```xml
<PolicySet xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17
   http://docs.oasis-open.org/xacml/3.0/core-schema-v3-schema-wd-17.xsd"
   PolicySetId="" Version="1.0"
   PolicyCombiningAlgorithm:"permit-oversides"
   MaxDelegationDepth="1">
  <Description>
    In case of permit authorization decision add redirect obligation
    for encryption service
  </Description>
  <Target />
  <PolicyIdReference>
    urn:oasis:names:tc:xacml:3.0:policy:deny:all
  </PolicyIdReference>
  <PolicyIdReference>
    urn:oasis:names:tc:xacml:3.0:policy:privacy
  </PolicyIdReference>
  <ObligationExpressions>
    <ObligationExpression
      FulfillOn="Permit"
      ObligationId="urn:oasis:names:tc:xacml:3.0:obligations:redirectto">
      <AttributeAssignmentExpression
        AttributeId="urn:epos:obligations:redirectto:url">
        <AttributeDesignator
          DataType="http://www.w3.org/2001/XMLSchema#anyURI"
          MustBePresent="false"
          Category="urn:oasis:names:tc:xacml:3.0:attribute-category:epos"/>
        <AttributeAssignmentExpression
          AttributeId="urn:oasis:names:tc:xacml:3.0:epos:redirectto:url"/>
      </AttributeAssignmentExpression>
      <ObligationExpression
        FulfillOn="Deny"
        ObligationId="urn:oasis:names:tc:xacml:obligation:mail">
        <AttributeAssignmentExpression
          AttributeId="urn:oasis:names:tc:xacml:3.0:attribute:mailto">
          <AttributeDesignator
            DataType="http://www.w3.org/2001/XMLSchema#string"
            MustBePresent="false"
            Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"/>
          <AttributeAssignmentExpression
            AttributeId="urn:oasis:names:tc:xacml:3.0:resource:resource-id:email"/>
      </AttributeAssignmentExpression>
  </ObligationExpressions>
</PolicySet>
```

These permissions are string-encoded values from the Hl7 RBAC catalogue 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.

The policy set has an empty target, thus it applies to any request. Indeed, the policy set is used to combine the decisions of the two policies and to specify a further obligation. The deny-all policy has always effect deny and is combined with the patient privacy policy by means of the algorithm permit-overrides. Therefore, if the second policy evaluates to permit, then the policy set grants access to the resource, otherwise the policy set forbids access. If the decision is permit, the obligation redirect defines an url to which the authorization decision is redirected. The url can be statically determined, e.g. by the context handler, and then added to the request, or dynamically retrieved through the PIP. This obligation could, e.g., be exploited to route the communication with the PEP through an encrypted channel. Instead, if the decision is deny, the patient is informed by email about an illegal attempt of accessing his personal healthcare data.

We conclude by reporting in Listings 4 and 7 two sample XACML requests, and in Listings 5, 6, 8 and 9 the corresponding XACML decisions made by a PDP.

**Listing 4. Request by Dr. Marley**

```xml
<!--Request xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17
  http://docs.oasis-open.org/xacml/3.0/xacml-core-v3-schema-wd-17.xsd"
  CombinedDecision="false" ReturnPolicyIdList="false">
  <Attributes>
    <Attribute IncludeInResult="false" Category="urn:oasis:names:tc:xacml:3.0:attribute-category:subject">
      <AttributeId="urn:oasis:names:tc:xacml:3.0:subject:countrycode">
        <AttributeValue(DataType="http://www.w3.org/2001/XMLSchema#string">GR</AttributeValue>
      </AttributeValue>
    </Attribute>
    <Attribute IncludeInResult="false" Category="urn:oasis:names:tc:xacml:3.0:attribute-category:subject">
      <AttributeId="urn:oasis:names:tc:xacml:3.0:subject:subject-id">
        <AttributeValue(DataType="http://www.w3.org/2001/XMLSchema#string">Dr. Marley</AttributeValue>
      </AttributeValue>
    </Attribute>
    <Attribute IncludeInResult="false" Category="urn:oasis:names:tc:xacml:3.0:attribute-category:subject">
      <AttributeId="urn:oasis:names:tc:xacml:3.0:subject:organization">
        <AttributeValue(DataType="http://www.w3.org/2001/XMLSchema#string">HOSPITAL</AttributeValue>
      </AttributeValue>
    </Attribute>
    <Attribute IncludeInResult="false" Category="urn:oasis:names:tc:xacml:3.0:attribute-category:subject">
      <AttributeId="urn:oasis:names:tc:xacml:3.0:subject:email">
        <AttributeValue(DataType="http://www.w3.org/2001/XMLSchema#string">eden@epso.org</AttributeValue>
      </AttributeValue>
    </Attribute>
  </Attributes>
</Request>
```
A formal sw. eng. approach to Policy-Based Access Control
The request in Listing 4 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 attributes in the request are organized into categories, i.e. subject, resource and action. The evaluation of such a request with respect to the policy in Listing 2 produces the positive response in Listing 5. The response also includes the obligation associated to the permit decision. This means that the PDP has been capable of retrieving and evaluating all attributes specified as arguments of the obligation’s action; in this example, the subject and resource identifiers have been successfully retrieved for the log action.

Listing 5. Decision for request in Listing 4 with respect to policy in Listing 2

The evaluation of the request in Listing 4 with respect to the policy set in Listing 3 is obtained by combining the decisions to the request for each component of the policy set, that is the decision for the privacy consent policy, reported in Listing 5, and that for the deny-all policy, which is obviously deny. The resulting
A formal sw. eng. approach to Policy-Based Access Control 23

Since the combining algorithm is permit-overrides, the final PDP decision is permit and includes the obligations fulfilled by the privacy policy (i.e. action log) and by the policy set (i.e. action redirect).

Listing 6. Decision for request in Listing 4 with respect to policy set in Listing 3

```
<Response xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
http://docs.oasis-open.org/xacml/3.0/xacml-core-v3-schema-wd-17.xsd">
  <Result>
    <Decision>Permit</Decision>
    <Status>
      <StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
    </Status>
    <Obligations>
      <Obligation ObligationId="urn:oasis:names:tc:xacml:obligation:log">
        <AttributeAssignment
          DataTypeId="http://www.w3.org/2001/XMLSchema#string"
          AttributeId="urn:oasis:names:tc:xacml:3.0:attribute:subject">
          <AttributeValue
            DataTypeId="http://www.w3.org/2001/XMLSchema#string">
            Dr. Marley
          </AttributeValue>
        </AttributeAssignment>
      </Obligation>
      <Obligation ObligationId="urn:oasis:names:tc:xacml:3.0:obligations:redirectto">
        <AttributeAssignment
          DataTypeId="http://www.w3.org/2001/XMLSchema#anyURI"
          AttributeId="urn:epsos:obligations:redirectto:uri">
          http://ncp/ESS/redirect
        </AttributeAssignment>
      </Obligation>
    </Obligations>
  </Result>
</Response>
```

The PDP response will be then enforced by the PEP, whose final decision will depend on its capability to discharge the two obligations and on the enforcement algorithm used.

The request in Listing 7 is similar to the previous one but it is made by a medical doctor, Dr. Elliot, who has different permissions from Dr. Marley.

Listing 7. Request by Mr. Elliot

```
<Request xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
  http://docs.oasis-open.org/xacml/3.0/xacml-core-v3-schema-wd-17.xsd">
  <Attributes>
    <Category"="urn:oasis:names:tc:xacml:3.0:attribute-category:subject">
      <AttributeIncludeInResult=false/>
      <AttributeId="urn:oasis:names:tc:xacml:3.0:subject:countrycode">
        <AttributeValue
          DataTypeId="http://www.w3.org/2001/XMLSchema#string">
          34133-9
        </AttributeValue>
      </AttributeId>
    </Category>
  </Attributes>
</Request>
```
The evaluation of the request reported in Listing 7 with respect to the policy reported in Listing 2 produces the decision reported in Listing 8.

Listing 8. Decision for request in Listing 7 with respect to policy in Listing 2

```xml
<Response xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
  xsi:schemaLocation="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17
http://docs.oasis-open.org/xacml/3.0/xacml-core-v3-schema-wd-17.xsd">
  <Result>
    <Decision>NotApplicable</Decision>
  </Result>
</Response>
```

The policy’s target matches the request, but the rule’s condition is not satisfied due to the lack of requestor’s permissions, hence the decision for this policy is not-applicable.

Finally, the evaluation of the request reported in Listing 7 with respect to the policy set in Listing 3 produces the decision in Listing 9.

Listing 9. Decision for request in Listing 7 with respect to policy set in Listing 3

```xml
<Response xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
  xsi:schemaLocation="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17
http://docs.oasis-open.org/xacml/3.0/xacml-core-v3-schema-wd-17.xsd">
  <Result>
    <Decision>Deny</Decision>
    <Status>
      <StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
    </Status>
    <Obligations>
      <Obligation ObligationId="urn:oasis:names:tc:xacml:obligation:mail">
        <AttributeAssignment>
          <AttributeId="urn:oasis:names:tc:xacml:3.0:attribute:mailto">
            <AttributeValue>
              patient@mail.com
            </AttributeValue>
          </AttributeId>
        </AttributeAssignment>
        <AttributeAssignment>
          <AttributeId="urn:oasis:names:tc:xacml:3.0:attribute:text">
            <AttributeValue>
              Your medical record has been requested by EpSOS
            </AttributeValue>
          </AttributeId>
        </AttributeAssignment>
      </Obligation>
    </Obligations>
  </Result>
</Response>
```
The resulting decision, i.e. deny, is indeed obtained by combining the not-applicable decision (resulting from the privacy policy) with the deny one (resulting from the deny-all policy) through the algorithm permitoverrides. The obligation log, fulfilled by the privacy consent policy and applicable to decision deny, is included in the final response, while the obligation redirect, defined in the policy set, is not included, because it is only applicable to decision permit.

3.2 A Cloud IaaS Scenario

Cloud computing fosters three well-known approaches to provide computational services [16]: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Service-as-a-Service (SaaS). Each of them is characterised by the offer provided to the users, ranging from the low-level access provided by IaaS, where users require the instantiation of general purpose virtual machines for running their processes, to the high-level access provided by SaaS, where users directly interact with application services (e.g., storage or mail services) without being aware of details concerning the infrastructure and platform supporting the provision of such services. The Cloud computing case study we use throughout this paper to illustrate our policy-based approach focusses on IaaS.

We consider a IaaS provider that offers to customers a range of pre-configured virtual machines (VMs), providing different amounts of dedicated computing capacity in order to meet different computing needs. Each type of VM features a specific Service Level Agreement (SLA) that the provider commits to guarantee. Thus, the allocation of the right amount of resources needed to instantiate new VMs (while respecting committed SLAs) is a key aspect of the considered IaaS provider. As is common for Cloud systems, virtualisation is accomplished using an hypervisor, i.e., a software entity managing the execution of VMs.

For the sake of presentation, our IaaS provider relies only on two hypervisors running on top of two physical machines. The provider offers strongly defined types of VMs, like most of popular IaaS providers (consider, e.g., the instance types M1 Small and M1 Medium provided by Amazon EC2). Two types of VMs, namely TYPE1 and TYPE2, are in the provider’s service portfolio. Each type of VM has an associated SLA describing the hardware resources needed to instantiate the VM (e.g., CPU performance, size of memory and storage) by means of an aggregated measure: TYPE1 requires the allocation of one unit of resources, while TYPE2 requires two units.

The two types of VMs have different guarantees when the system is highly loaded. Specifically, if the system does not have enough resources for allocating a new TYPE2 VM, an appropriate number of TYPE1 VMs already instantiated will be frozen and moved to a queue of suspended VMs. This queue is periodically checked with the aim of trying to reactivate suspended VMs. When a VM is frozen, according to the Insurance [17] SLA approach for resource provisioning in Cloud computing systems, the VM’s owner will receive a credit that can be used, e.g., for activating new VMs or for paying computational time.

A graphical representation of the data-flow in our implementation of the scenario is shown in Figure 2. Clients interacts with the Cloud system via a
Web portal that, following a multi-tenancy architecture, sends VM instantiation requests to the Cloud manager through SOAP messages. This means that the manager exposes its functionalities to users by means of a Web service. Then, the manager evaluates the received requests with respect to a set of policies defining the logic of the system. In particular, such policies specify the credentials the clients have to provide in order to access the service (access control policies), the resource allocation strategy (resource-usage policies), and the actions to be performed to fulfill the requests by also taking into account the current system state, which include the system re-configuration actions in case of high load (adaptation policies). It is worth noticing that all policies are written by using XACML, however for sake of presentation we report the complete policies in the Appendix B. By means of a similar workflow, clients can request the shutdown of VMs, which involves the release of the allocated resources.

The administrator of the Cloud system can access a dedicated panel for managing the governing policies. Indeed, he can change at run-time the current policies with other ones, obtaining in this way a fully configurable and adaptable system. The core of the Cloud manager is the Policy Enforcement Point (PEP), which evaluates client requests according to the available policies in the Policy Repository (PR) and the environmental information about the Cloud system. The sub-component Policy Decision Point (PDP) has the duty of calculating if a request can be granted or rejected, and determining the actions needed to enforce the decisions, such as creation, freezing and shutdown of VMs. The enforcing is executed by the PEP by sending to the hypervisors the commands corresponding to the obtained actions. Notably, policies are independent from the specific kind of hypervisors installed on the system, such as XEN [http://www.xen.org] or Linux-KVM [http://www.linux-kvm.org], i.e., the actions returned by the PDP are converted by the PEP into the appropriate commands accepted by
the used hypervisors. Thus, in principle, the policy engine we have developed
could be integrated with any IaaS system provided that the adequate action
translation is also defined.

4 The Formal Access Control Policy Language (FACPL)

The XACML specification, as shown in the previous section, defines an XML-
based language that permits writing both policies and contexts (i.e. access re-
quests and responses) \[3, Section 6\] in a way independent of the specific formats
used by PEPs. However, the XML syntax of this language, on the one hand,
can make the task of writing policies difficult and error-prone, and, on the other
hand, is not adequate for formally defining the semantics of the language and
reasoning on it.

In this section, we present an alternative language named Formal Access
Control Policy Language (FACPL) for writing access policies, requests and re-
responses. Their syntax is defined through EBNF grammars. The semantics is
defined by structural induction on the syntax and provides a formal account of
the authorization process by precisely describing each evaluation step.

4.1 Syntax of policies

FACPL syntax for writing access control policies is reported in Table \[10\] As
usual in EBNF-like grammars, the question mark (?) stands for optional items,
the star (\(\ast\)) for (possibly empty) sequences, and the plus (+) for non-empty
sequences. For the sake of readability, whenever an element is missing, we also
omit the possibly related keyword; thus, e.g., we simply write (deny) in place of
rule (deny target : condition : obl : ). To facilitate the task of writing policies,
also the FACPL editor presented in Section \[7.2\] permits omitting such keywords.

The Policy Authorization Framework (PAF) top-level term includes a PDP
and a PEP.

A PDP \(\{\text{Palg policies : Policy}^+\}\) is a simplified form of policy set with an
empty target. The PDP evaluates the policies \(\text{Policy}^+\) as if they are organised
as a single policy set, according to the policy-combining algorithm \(\text{Palg}\).

A policy can be either a simple policy of the form

\((\text{Ralg target : Target}^\ast rules : \text{Rule}^+ obl : \text{Obligation}^\ast)\)

or a policy set of the form

\(\{\text{Palg target : Target}^\ast policies : \text{Policy}^+ obl : \text{Obligation}^\ast\}\)

Notably, a policy set can recursively enclose other policy sets and policies as follow

\(\{\text{Palg target : Target}
\text{ policies : } \{\text{Palg target : Target policies : Policy obl : Obligation}\}
\text{ rules : Rule obl : Obligation}\}\)
Table 10. Syntax of FACPL policies

<table>
<thead>
<tr>
<th>Policy Authorization Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>( PAF ::= {pd : PDP, \ pep : PEP} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Policy Decision Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>( PDP ::= {Palg \ policies : Policy^+} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Policy-combining algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Palg ::= \text{only-one-applicable}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule-combining algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Ralg ::= \text{deny-overrides}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Policies: atomic policies and policy sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Policy ::= {Ralg \ target : Target^7 \ rules : Rule^+ \ obl : Obligation^*}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Rule ::= {Effect \ target : Target^7 \ condition : BoolExpr^7 \ obl : Obligation^*} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Effect ::= \text{permit}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Target ::= \text{MatchId}(Value,Name)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Matching functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{MatchId ::= equal</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Name ::= \text{Identifier/Identifier} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Obligations</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Obligation ::= [Effect \ Type \ PepAction(\text{Expression}^*)] )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type: mandatory or optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Type ::= \text{M}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Policy Enforcement Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>( PEP ::= \text{base}</td>
</tr>
</tbody>
</table>

Both policies and policy sets specify the algorithm for combining the results of the evaluation of the contained elements, a target to which the policy/policy set applies and a set of obligations that define supplemental, mandatory or optional actions for the enforcing process. The available combining algorithms...\(^{10}\) Notably, optional obligations correspond to XACML advices.
Table 11. Target operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>match</th>
<th>no-match</th>
<th>indeterminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>∧</td>
<td>match</td>
<td>no-match</td>
<td>indeterminate</td>
</tr>
<tr>
<td></td>
<td>no-match</td>
<td>no-match</td>
<td>no-match</td>
</tr>
<tr>
<td></td>
<td>indeterminate</td>
<td>no-match</td>
<td>indeterminate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>match</th>
<th>no-match</th>
<th>indeterminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>∨</td>
<td>match</td>
<td>no-match</td>
<td>indeterminate</td>
</tr>
<tr>
<td></td>
<td>no-match</td>
<td>match</td>
<td>match</td>
</tr>
<tr>
<td></td>
<td>indeterminate</td>
<td>match</td>
<td>indeterminate</td>
</tr>
</tbody>
</table>

are the same as XACML, described in Section 2.3, but for permit-overrides and deny-overrides that here coincide with the corresponding ordered versions, i.e. the enclosed elements of a policy or a rule are always evaluated in the order in which they appear. This permits removing the non-determinism from the authorization process in order to guarantee a single meaning to each policy. It is worth noticing that this definition of the algorithms not only is compliant with the ordered version of the XACML algorithms, but also with the standard one. Therefore, all XACML combining algorithms can be expressed in FACPL. FACPL’s implementation provides all these algorithms and also the opportunity of implementing customized algorithms, as described in Section 7.

A single policy contains a (non-empty) set of rules of the form

\[(\text{Effect} \ target : \text{Target}^m \ condition : \text{BoolExpr}^n \ \text{obl} : \text{Obligation}^a)\]

each specifying: an effect, which indicates the rule-writer’s intended consequence of a positive evaluation for the rule (the allowed values are permit and deny), a rule target, which refines the applicability established by the target of the enclosing policy, a condition, which is a boolean expression that may further refine the applicability of the rule, and a set of obligations, which are additional actions that must be discharged by the PEP. Notably, target, condition and obligations may be missing.

A target permits to identify the set of access requests that a rule, a policy or a policy set is intended to evaluate. Namely, a target specifies the set of subjects, resources, actions, environments or other categories’ attributes to which the corresponding rule/policy/policy set applies. A target is a boolean expression built from match elements, i.e. terms of the form MatchId(\(Value, Name\)), by means of conjunction, \(\land\), and disjunction, \(\lor\). Operator \(\land\) takes precedence over operator \(\lor\). The behaviour of the two logical operators over the set of values \{match, no-match, indeterminate\} is defined in Table 11. Basically, over \{match, no-match\}, they behave as logical conjunction and disjunction operators, where match and no-match are dealt with as true and false, respectively.

If the target of a rule, policy or policy set is empty, then such rule, policy or policy set applies to any request.

In a match element, MatchId specifies the function to be used to compare the given literal \(Value\) with the value of the attribute identified by the given
Name. The function returns either a boolean value according to the result of comparison or, in case of error, the value indeterminate.

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

In order to identify attribute values contained in a request, FACPL uses (structured) names of the form Identifier/Identifier, where the first identifier stands for a category name and the second for an attribute name. A structured name can represent a XACML attribute designator used for retrieving values from requests. For example, the structured name subject/hl7:permission represents the following XACML designator (drawn from Listing 2)

```xml
<s:AttributeDesignator
  DataType="http://www.w3.org/2001/XMLSchema#anyURI"
  MustBePresent="false"
  Category="urn:oasis:names:tc:xacml:3.0:attribute-category:subject"
  AttributeId="urn:oasis:names:tc:xacml:1.0:subject:hl7:permission"/>
```

Thus, the evaluation of a structured name involves the retrieval from the request of one or more values related to the designed attribute.

Remark 2. Besides attribute designators, XACML also provides attribute selectors that retrieve attribute values evaluating XPath [18] expressions. In FACPL, path expressions without additional operators (for, e.g., arithmetic or node comparison) can be represented by structured names. For example, the following selector (drawn from Example 4.2.4.2 in [3])

```xml
<s:AttributeSelector
  MustBePresent="false"
  Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
  DataType="http://www.w3.org/2001/XMLSchema#date"/>
```

can be rendered as resource/record.patient.patientDoB. Support to all XPath functions is out of scope for the FACPL language.

Remark 3. In the original XML-based syntax of XACML (see examples in Section 3), a target may contain a sequence of AnyOf elements, each of which may contain a sequence of AllOf elements, each of which may contain a sequence of Match elements. FACPL target syntax has no limitation on expressions’ depth level, thus it seems more expressive than the original one as it permits writing terms that exceed the three level structure AnyOf – AllOf – Match. Indeed, a disciplined use of structured names and logical operators permits properly expressing XACML targets: a target must be a term of the form

\[\text{AnyOf}_1 \land \text{AnyOf}_2 \land \ldots \land \text{AnyOf}_n\]

11 A "identifier" is a sequence of characters, i.e. letters, digits, symbols ω : and ., that cannot begin with a digit.
where each \( \text{AnyOf} \) must have the form
\[
\text{AllOf}_1 \lor \text{AllOf}_2 \lor \ldots \lor \text{AllOf}_m
\]
and, finally, each \( \text{AllOf}_i \) must have the form
\[
\text{Match}_1(\text{Value}_1, \text{Name}_1) \land \ldots \land \text{Match}_k(\text{Value}_k, \text{Name}_k)
\]
We believe that FACPL syntax for targets provides a more compact notation and a more intuitive and clearer semantics while, at the same time, does not lead to loss of information.

Rule conditions are specified through boolean expressions \( \text{BoolExpr} \), which are formed simply by functions that operate on values and structured names. The complete definition of expressions is reported in Table 12.

Basically, FACPL provides the main logical, relational and arithmetic operators, which operate on boolean, string, integer, double and date values. Moreover, bags (i.e., unordered collections that may contain duplicated values) of strings are also supported.

Rules, policies or policy sets can specify obligations. An \text{Obligation} has the form
\[
[\text{Effect Type } \text{PepAction}(\text{Expression}^*)]
\]
where \( \text{Effect} \) defines the effect, permit or deny, for applying the obligation, while \( \text{PepAction}(\text{Expression}^*) \) specifies the identifier and the arguments of the action to be performed by the PEP. An obligation can be \text{Mandatory}, identified by \text{Type M}, or \text{Optional}, identified by \text{O}. Notationally, we will use \( o \) to range over (non-evaluated) obligations.

The PEP enforces the PDP decision by discharging the obligations, if present, and executing the chosen enforcement algorithm. The algorithms available for the PEP are those presented in the XACML specification and previously described in Section 2.3. In addition, FACPL also provides the algorithm \text{debug}, which is a variant of the algorithm \text{base} that retains the distinction among extended indeterminate values. The set of action identifiers understood by the PEP results from an agreement between the PEP implementation and policy writers (hence, \text{PepAction} has been intentionally left unspecified).

4.2 Policy examples

In this section we describe through FACPL policies the case studies presented in Section 3. The policies for the eHealth domain will be used in Section 6 for a formal example of request evaluation, while the ones for the Cloud domain in Section 7.3 to implement the IaaS platform.

\text{eHealth domain}. We show here how the policy examples described in Section 3 are written using the syntax of FACPL presented in Table 10.
Table 12. Syntax of FACPL expressions

<table>
<thead>
<tr>
<th>Expressions</th>
<th>Expression ::= BoolExpr</th>
<th>StringExpr</th>
<th>ArithExpr</th>
<th>DateExpr</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Boolean expressions</strong></td>
<td>$BoolExpr ::= Name$</td>
<td>$true$</td>
<td>$false$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{and}(BoolExpr, \ldots, BoolExpr)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{or}(BoolExpr, \ldots, BoolExpr)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{not}(BoolExpr)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{equal}(Expression, Expression)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{greater-than}(Expression, Expression)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{less-than}(Expression, Expression)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{less-than-or-equal}(Expression, Expression)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{at-least-one-member-of}(BagExpr, BagExpr)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{subset}(BagExpr, BagExpr)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>String expressions</strong></td>
<td>$StringExpr ::= Name$</td>
<td>$StringValue$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$BagExpr ::= \text{string-bag}(StringExpr, \ldots, StringExpr)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic expressions</strong></td>
<td>$ArithExpr ::= Name$</td>
<td>$IntegerValue$</td>
<td>$DoubleValue$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{add}(ArithExpr, ArithExpr)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{subtract}(ArithExpr, ArithExpr)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{multiply}(ArithExpr, ArithExpr)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{divide}(ArithExpr, ArithExpr)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{mod}(ArithExpr, ArithExpr)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{abs}(ArithExpr)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Date and time expressions</strong></td>
<td>$DateExpr ::= Name$</td>
<td>$DateTimeValue$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

$$\langle \text{permit-overrides target : equal(“medical doctor”, subject/role)}$$

$$\quad \wedge \text{equal(“TREATMENT”, subject/purposeofuse)}$$

$$\quad \wedge \text{equal(“34133-9”, resource/resource-id)}$$

$$\text{rules : ( permit target : equal(“Read”, action/action-id)}$$

$$\text{condition : subset(}$$

$$\text{string-bag(“PRD-003”, “PRD-005”, “PRD-010”, “PRD-016”),}$$

$$\text{subject/hl7 : permission)})$$

$$\text{obl : [ permit M log(subject/subject-id, resource/resource-id) ] } \rangle$$
Then, the two previous policies organised in the policy set reported in Listing 3 can be rewritten as follows:

\[
\{\text{permit-override}\}
\]
\[
\{\text{permit-override rules : (deny)}\}
\]
\[
\{\text{permit-override rules : (permit ...)}\}
\]
\[
\{\text{obl : [permit M log(subject/subject-id, resource/resource-id)]} \}
\]
\[
\{\text{obl : [permit M redirect(epsos/redirect-url)] }\}
\]
\[
\{\text{deny M mail(resource/resource-id.email, “Your medical record has been requested by epSOS”)}\}
\]

**Cloud domain.** For the Cloud scenario we have developed two different approaches for managing, instantiating and releasing client requests. The first one concentrates the workload on hypervisor HYPER 1, while hypervisor HYPER 2 is only used when the primary one is fully loaded. Thus, by keeping the secondary hypervisor in stand-by mode until its use becomes necessary, energy can be saved. The second approach, instead, balances the workload between the two hypervisors.

The **energy saving** policies are presented below. Specifically, this specification defines a PDP containing a policy set, for supervising instantiation requests (specifying action **CREATE**), and a policy, for supervising release requests (specifying action **RELEASE**). The policy set contains a policy for each type of VM that, in its own turn, achieves the prioritized choice between the two hypervisors by specifying the combining algorithm **deny-unless-permit** and by relying on the rules order. The policy managing the instantiation of TYPE 1 VMs is as follows:

---

**Listing 10.** Instation of TYPE 1 VM

\{
\text{permit-override policies : (deny-unless-permit}
\text{target : equal("P_1", subject/profile-id)|| equal("P_2", subject/profile-id)}
\text{& & equal("TYPE_1", resource/vm-type)
\text{rules :}
\text{(permit target : less-than-or-equal(1, system/hyper1.availableResources)}
\text{obl : [permit M create("HYPER_1", system/vm-id, "TYPE_1")])
\text{})(permit target : less-than-or-equal(1, system/hyper2.availableResources)}
\text{obl : [permit M create("HYPER_2", system/vm-id, "TYPE_1")])
\text{obl : [deny O warning("Not enough available resources for TYPE_1 VMs")])}
\}

---

The policy’s target indicates that instantiation of TYPE 1 VMs can be required by clients having P 1 or P 2 as profile. The policy’s combining algorithm
evaluates the enclosed rules according to the order they occur in the policy; then, if one of them evaluates to permit, the evaluation terminates. Rule `hyper_1` evaluates to permit only if the hypervisor `HYPER_1` has at least one unit of available resources and, in this case, returns an obligation requiring the PEP to create a VM in this hypervisor. Rule `hyper_2`, governing VMs creation on `HYPER_2`, is similar. If no rule evaluates to permit, then the combining algorithm returns deny and, hence, the policy’s (optional) obligation will be executed by the PEP to notify the Cloud administrator that there are not enough resources in the system to instantiate a new `TYPE_1` VM. In this way, the administrator can decide to upgrade the system by adding new resources (e.g., a new physical machine).

The policy for creating `TYPE_2` VM has the same structure of the previous policy, obviously in the policy target is checked `TYPE_2` identifier and client profile `P_2`. The policy encloses a set of four rules: two are as those in `TYPE_1` policy only for `TYPE_2` VM, while the other two, one for each hypervisor, are applied when there is not enough available load on the hypervisors. For example, the rule `hyper_1_freeze` is applied to freeze some `TYPE_1` VMs running on the first hypervisor when a request for creating a new `TYPE_2` VM arrives and all the hypervisors are fully loaded. The rule inherits the target of the enclosing policy and specifies the effect permit, a condition refining the applicability of the rule to a request, and a set of obligations.

**Listing 11. Freezing of VM**

```
(permit
target: ...
condition: or(and(equal(0, system/hyper1.availableResources),
   less-than-or-equal(2, system/hyper1.vm1-counter)),
   and(equal(1, system/hyper1.availableResources),
   less-than-or-equal(1, system/hyper1.vm1-counter)))
obl: [permit M freeze("HYPER_1", subtract(2, system/hyper1.availableResources),
   "TYPE_1")]
   [permit M create("HYPER_1", system/vm-id, "TYPE_2")]
)
```

The condition in the listing above checks if there are enough `TYPE_1` VMs to freeze for increasing the available resources in the hypervisor and granting the request; recall that each `TYPE_1` VM requires 1 unit of resources, while each `TYPE_2` VM requires 2 units.

The policies for the **load balancing** approach are the same as before except that a condition on the hypervisors’ load is added to each instantiation rule. This condition permits applying a rule for a certain hypervisor only if its amount of available resources is greater than or equal to the amount of available resources of the other hypervisor. For example, the rule `hyper_1` is extended as follows.

**Listing 12. Load balancing condition**

```
( permit
target: ...
condition: less-than-or-equal(system/hyper2.availableResources, system/hyper1.availableResources)
obl: ... )
```
Finally, the RELEASE policy permits to delete a VM on an hypervisor updating the available load. The policy is as follows

```
< permit-override
  target:
  equal("RELEASE", action/action-id) &
  (equal("P_1", subject/profile-id)||equal("P_2", subject/profile-id))
rules:
  (permit
   condition: at-least-one-member-of(resource/vm-id, system/hyper1.vm-ids)
   obl: [permit M release("HYPER_1", resource/vm-id)]
  )
  (permit
   condition: at-least-one-member-of(resource/vm-id, system/hyper2.vm-ids)
   obl: [permit M release("HYPER_2", resource/vm-id)]
  )
>
```

The policy is enclosed in a policy set with the TYPE_1 and the TYPE_2 policies.

The complete code of the policies energy saver and load balancer is reported in the Listings 23 and 24 in Appendix B.

### 4.3 Syntax of requests and responses

FACPL syntax for writing access requests and responses is reported in Table 13.

A request, ranged over by r, consists of a (unique) identifier and a set of attributes, i.e., name-value pairs. Such pairs indicate the subjects associated to the request (e.g., 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 can be requested), the action to be performed on the resources (e.g., read, write), the environmental properties (e.g., current date and time) and all other attributes needed to evaluate the request. This latter information may be added by the context handler or PIP to the original request provided by the requestor. We exploit structured names in order to avoid dealing with attribute’s categories and identifiers as separate information. When convenient, we shall regard a request r simply as a set, writing e.g. \((Name, Value) \in r\) to mean that \((Name, Value)\) is an attribute of the request \(r\). In passing, notice that this representation of requests permits easily dealing with multivalued attributes (as pointed out in Remark 4) and with the fact that attribute designators may pick up bags of values from a request.

Multiple requests can be submitted to the PEP (which, in its own turn, forwards them to the PDP), by simply grouping them in request sets (ranged over by \(R\)). As a matter of notation, we will use \(R_{all}\) to denote the set of all possible requests generated by the grammar in Table 13.

A response contains the results of evaluation of a request set. We have two different kinds of responses: authorization decisions, i.e. responses by the PDP, and enforceable decisions, i.e. responses by the PEP. Both kinds of responses partition the requests set on the basis of the result (i.e., permit, deny, not-applicable
Table 13. Syntax of FACPL requests and responses

<table>
<thead>
<tr>
<th>Set Type</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request sets</td>
<td>RequestSet ::= { Request}</td>
</tr>
<tr>
<td>Requests</td>
<td>Request ::= request : Identifier (Name, Value)+</td>
</tr>
<tr>
<td>Authorization decisions</td>
<td>AD ::= ( permit : RequestWithObls</td>
</tr>
<tr>
<td></td>
<td>deny : RequestWithObls</td>
</tr>
<tr>
<td></td>
<td>not-applicable : { Identifier}</td>
</tr>
<tr>
<td></td>
<td>indetP : { Identifier}</td>
</tr>
<tr>
<td></td>
<td>indetD : { Identifier}</td>
</tr>
<tr>
<td></td>
<td>indetDP : { Identifier}</td>
</tr>
<tr>
<td>Sets of requests</td>
<td>RequestWithObls ::= { (Identifier, EvalObligation)}</td>
</tr>
<tr>
<td>Evaluated obligations</td>
<td>EvalObligation ::= [ Type PepAction(Value) ]</td>
</tr>
<tr>
<td>Enforceable decisions</td>
<td>ED ::= ( permit : { Identifier}</td>
</tr>
<tr>
<td></td>
<td>deny : { Identifier}</td>
</tr>
<tr>
<td></td>
<td>not-applicable : { Identifier}</td>
</tr>
<tr>
<td></td>
<td>indetP : { Identifier}</td>
</tr>
<tr>
<td></td>
<td>indetD : { Identifier}</td>
</tr>
<tr>
<td></td>
<td>indetDP : { Identifier}</td>
</tr>
</tbody>
</table>

...and the extended indeterminate values indetP, indetD and indetDP of their evaluation w.r.t. the policies managed by the considered policy authorization framework. The only difference is that, each request identifier in the sets corresponding to decisions permit and deny of PDP responses has attached a (possibly empty) sequence of evaluated obligations, each being a pair made of a type (either M or O) and an action identifier with a list of value arguments. Instead, the responses resulting from the PEP enforcement process only contain the request identifier after the discharge of obligations possibly returned by the PDP. Notably, the set corresponding to a decision value only contains request identifiers (and not whole requests).

For the sake of readability, if a set contains one element only, the set notation will be omitted. Thus, e.g. we will write (deny) instead of \{(deny)\} and permit : req1 instead of permit : \{ req1 \}. Moreover, if the set corresponding to a value within an authorization or an enforceable decision is empty, then it will be omitted together with the value. Thus, e.g. we will write ( permit : req1 ) instead of ( permit : \{ req1 \} deny : \emptyset not-applicable : \emptyset indetP : \emptyset indetD : \emptyset indetDP : \emptyset ).
4.4 Request and response examples

eHealth Domain. The request reported in Listing 4 can be written in FACPL as follows:

\[
\text{request : req1 (subject/countrycode, “GR")}
\]
\[
\text{(subject/subject-id, “Dr. Marley")}
\]
\[
\text{(subject/organization, “HOSPITAL")}
\]
\[
\text{(subject/organization-id, “2624")}
\]
\[
\text{(subject/role, “medical doctor")}
\]
\[
\text{(subject/purposeofuse, “TREATMENT")}
\]
\[
\text{(subject/hl7:permission, “PRD-003")}
\]
\[
\text{(subject/hl7:permission, “PRD-006")}
\]
\[
\text{(subject/hl7:permission, “PRD-004")}
\]
\[
\text{(subject/hl7:permission, “PRD-005")}
\]
\[
\text{(subject/hl7:permission, “PRD-010")}
\]
\[
\text{(subject/hl7:permission, “PRD-046")}
\]
\[
\text{(subject/hl7:permission, “PRD-016")}
\]
\[
\text{(resource/resource-id, “34133-9")}
\]
\[
\text{(action/action-id, “Read")}
\]
\[
\text{(epsos/redirect-url, “http://ncp/ESS/redirect")}
\]

The corresponding PDP response in Listing 6 can be written in FACPL as follows:

\[
\text{( permit : \{ req1, [ M log(“Dr. Marley”, “34133-9")])}
\]
\[
\text{[ M redirect(“Dr. Marley”, “http://ncp/ESS/redirect")]) }\}
\]

while the corresponding PEP response, assuming that the obligations log and redirect are successfully discharged, is ( permit : req1 ).

Remark 4 (Multivalued attribute representation). It is worth noticing that in the request above there are many attributes with the same structured name \text{subject/hl7:permission}: this is an example of how a multivalued attribute, i.e. an attribute with multiple values, is rendered in FACPL. In fact, although FACPL syntax does not permit to directly express multivalued attributes, they can be simply rendered as a collection of separate single-valued attributes sharing the same name. This design choice is sound with respect to XACML. Indeed, the XACML specification requires to evaluate multivalued attributes as they are split into separate single-valued attributes (see [3, Section 7.3.3]). This means that a multivalued attribute is in fact a compact notation for grouping single-valued attributes with the same name. \qed
The request reported in Listing 7 can be rewritten in FACPL as follows:

\[
\text{request : req2 (subject/countrycode, “GR”)}
\]

\[
\text{(subject/subject-id, “Mr. Elliot Pharmacist”)}
\]

\[
\text{(subject/organization, “HOSPITAL”)}
\]

\[
\text{(subject/organization-id, “2624”)}
\]

\[
\text{(subject/role, “pharmacist”)}
\]

\[
\text{(subject/purposeofuse, “TREATMENT”)}
\]

\[
\text{(subject/hl7:permission, “PRD-006”)}
\]

\[
\text{(subject/hl7:permission, “PRD-004”)}
\]

\[
\text{(subject/hl7:permission, “PRD-010”)}
\]

\[
\text{(subject/hl7:permission, “PRD-046”)}
\]

\[
\text{(resource/resource-id, “34133-9”)}
\]

\[
\text{(action/action-id, “Read”)}
\]

\[
\text{(epos/redirect-url, “http://ncp/ESS/redirect”)}
\]

The corresponding PDP response in Listing 9 can be written in FACPL as follows:

\[
\text{(deny : \langle req2, [M mail(“patient@mail.com”, “Your medical record has been requested by epSOS”)] \rangle )}
\]

while the corresponding PEP response, assuming that the obligation mail is successfully discharged, is \((\text{deny : req2} )\).

Cloud Domain. In the Cloud Scenario when a client requests a service, a FACPL request is generated; in addition to its identifier, this latter request contains the authentication credentials and the specification of the type of the requested service. These information are organised as attributes, i.e., pairs name-value. Names are structured, where the category identifier are subject, resource, action, and system), while the attribute names are, e.g., profile-id and action-id). A sample request for creating a TYPE_2 VM follows.

\[
\text{request : Create_Type2_VM}
\]

\[
\text{(subject/profile-id, “P_2”)}
\]

\[
\text{(resource/vm-type, “TYPE_2”)}
\]

\[
\text{(action/action-id, “CREATE”)}
\]

4.5 FACPL vs XACML

In the XACML XML-based language, access to a resource is regulated by one or more policies defining the capabilities and the credentials needed by a requestor for accessing the resource. Structural elements of polices (e.g., targets) are expressed as XML elements (denoted by pairs of start and end tags with the content in between), while the others (i.e., identifiers of functions, combing algorithms, etc.) are expressed as tag attributes. The XML format brings the benefit of enabling cross-platform interoperability, but can make the task of writing policies difficult and error-prone and is not adequate for formally defining the semantics
of the language and reasoning on it. The FACPL language, instead, relies on a BNF syntax and is equipped with a rigorous semantics given in a denotational style, which permits to formalise the authorization process of access requests. The interoperability is anyhow ensured by the possibility of exporting FACPL policies as XACML ones.

Another significant advantage of FACPL with respect to XACML is that the former provides a more compact notation for writing policies. Indeed, the markup style of XACML is more verbose, even for expressing elementary checks on request values (in fact, to conform to the XACML XML Schema, it is required to explicitly specify the whole XML structure enclosing the check). Moreover, each identifier should be prefixed by a (quite long) XACML namespace. Besides adversely affecting policies’ length, the markup style significantly undermines their readability and, hence, their specification and maintenance. For example, the condition for balancing machines’ load, reported in the previous listing, corresponds to the following XML code. It is 12 lines length in XACML, while it takes only 1 line in FACPL (without considering the automatic wrapping).

Listing 14. XACML Load Balancing Condition

```xml
<CCondition>
  <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-less-than-or-equal">
    <AttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#anyURI" MustBePresent="false">
      <Attribute Category="urn:oasis:names:tc:xacml:3.0:attribute-category:system">
        <AttributeIdId="urn:oasis:names:tc:xacml:3.0:system:hy er1.availableResources" />
      </Attribute>
    </AttributeDesignator>
  </Apply>
</CCondition>
```

Notably, the improvement of the language usability is due not only to the adoption of a non-XML format, but also to the design of specific lightweight linguistic constructs, as e.g. structured names for identifying attributes. For example, the second `AttributeDesignator` element in the code above is rendered in FACPL simply as `system/hyper2.availableResources`.

These differences become more evident when conditions contain expressions. E.g., the condition for freezing a `TYPE_1` machine takes 4 lines in FACPL (lines 50-53 in Listings [23] in Section [4.2]) and 42 lines in XACML. We report in Table [14] a comparison, in terms of code length, between the FACPL policies of the cloud case study and the FACPL corresponding ones (both groups of policies can be downloaded from `http://rap.dsi.unifi.it/facpl/FCloud_Policies/`).

The data shows that the use of FACPL can bring a relevant advantage to policy writers as well as to all users that anyhow need to understand the meaning of policies (e.g. developers, consumers, lawyers, etc.). Finally, note that all FACPL software tools are freely available and open-source.
Table 14. FACPL vs. FACPL on the Cloud case study (load balancer policies)

<table>
<thead>
<tr>
<th>Policy</th>
<th>Num. of lines XACML</th>
<th>FACPL</th>
<th>Saved lines</th>
<th>XACML</th>
<th>FACPL</th>
<th>Saved characters XACML</th>
<th>FACPL</th>
<th>Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLA_Type1</td>
<td>162</td>
<td>22</td>
<td>89.42%</td>
<td>3.607</td>
<td>164</td>
<td>89.27%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLA_Type2</td>
<td>449</td>
<td>36</td>
<td>89.68%</td>
<td>12.715</td>
<td>1.364</td>
<td>89.27%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Release_Policies</td>
<td>111</td>
<td>15</td>
<td>86.72%</td>
<td>4.193</td>
<td>438</td>
<td>89.55%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overall policies</td>
<td>648</td>
<td>101</td>
<td>84.41%</td>
<td>23.436</td>
<td>3.030</td>
<td>87.07%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5 FACPL formal semantics

We present in this section the semantics of the process for evaluating FACPL requests. The semantics is given in a denotational style and mimics the informal one expressed in natural language by the XACML specification. As a matter of notation, in the semantics, given a request \( r \), the corresponding request with evaluated obligations will be written as \( \langle r, \text{EvalObligation}^* \rangle \) instead of \( \langle \text{Id}, \text{EvalObligation}^* \rangle \), where \( \text{Id} \) is the identifier of \( r \).

The semantics of a policy authorization framework \( \{ \text{pdp} : PDP \ pep : PEP \} \) is a function \( \{ \text{pdp} : PDP \ pep : PEP \} \) that, given a set \( R \) of requests (with \( R \subseteq R_{all} \)), returns an enforceable decision, that is a tuple \( ED \) of the form

\[
(\text{permit} : R'_p \ \text{deny} : R'_d \ \text{not-applicable} : R'_n \ \text{indetP} : R'_{ip} \ \text{indetD} : R'_{id} \ \text{indetDP} : R'_{idp})
\]

where \( (R'_p, R'_d, R'_n, R'_{ip}, R'_{id}, R'_{idp}) \) is a partition of \( R \). More specifically, the set of requests \( R \) is first evaluated by the PDP with respect to the available policies and policy sets. The response resulting from the PDP decision process is an authorization decision, that is a tuple \( AD \) of the form

\[
(\text{permit} : \{ \langle r_i, EO_i \rangle \}_{r_i \in R'_p} \ \text{deny} : \{ \langle r_j, EO_j \rangle \}_{r_j \in R'_d} \ \text{not-applicable} : R'_n \ \text{indetP} : R'_{ip} \ \text{indetD} : R'_{id} \ \text{indetDP} : R'_{idp})
\]

where, again, \( (R'_p, R'_d, R'_n, R'_{ip}, R'_{id}, R'_{idp}) \) is a partition of \( R \). \( AD \) is then passed as input to the PEP enforcement process that finally returns the enforceable decision. Intuitively, tuples \( AD \) and \( ED \) partition the set of initial requests \( R \) into six subsets according to the results of evaluating each request. Moreover, in \( AD \), each request in the sets corresponding to decisions \( \text{permit} \) and \( \text{deny} \) has attached a (possibly empty) sequence \( EO \) of evaluated obligations. The overall FACPL authorization process can be thus summarized as

\[
\{ \text{pdp} : PDP \ pep : PEP \} \downarrow R = \{ PEP \}_{PDP} R
\]

Notably, \( R \) is only required to be a subset of the set \( R_{all} \) of all requests generated by the grammar in Table 13, thus it can contain e.g. all possible requests, only requests with a given structure or only one specific request.

5.1 Semantics of the PDP decision process

For the sake of presentation, we introduce the semantics of the PDP decision process in two steps: first, we define the semantics while ignoring obligations, then we refine the previous definitions for appropriately dealing with them.
Semantics without obligations. The semantics is given by the function \([PDP]\) , which is defined by structural induction on the syntax of PDP and relies on some additional functions for evaluating targets and obligations.

As a matter of notation, we will use the projection operator \(\downarrow_v\) that, given a tuple, returns the field corresponding to the value \(v\).

Let first introduce the function \([Target]\) defining the semantics of targets evaluation. This function, given a set \(R\) of requests, returns a matching tuple, that is a tuple of the form

\[
(\text{match} : R_m \quad \text{no-match} : R_n \quad \text{indeterminate} : R_i)
\]

where \((R_m, R_n, R_i)\) is a partition of \(R\) made on the basis of the result of evaluation of the target. The function is inductively defined on the syntax of targets by the following clauses:

\[
[\text{MatchId}(Value, Name)]_R =
\begin{align*}
\text{match} & : \{ r \in R \mid \exists (\text{Name}, \text{Value}') \in r : \text{MatchId}(Value, \text{Value}') = \text{true} \} \\
\text{no-match} & : \{ r \in R \mid \forall (\text{Name}, \text{Value}') \in r : \\
& \qquad \text{MatchId}(Value, \text{Value}') = \text{false} \} \\
\text{indeterminate} & : \{ r \in R \mid \exists (\text{Name}, \text{Value}') \in r : \\
& \qquad \text{MatchId}(Value, \text{Value}') = \text{indeterminate}, \\
& \not\exists (\text{Name}, \text{Value}') \in r : \\
& \qquad \text{MatchId}(Value, \text{Value}') = \text{true} \}
\end{align*}
\]

\[
[\text{Target}_1 \land \text{Target}_2]_R =
\begin{align*}
\text{match} & : \lbrack [\text{Target}_1]_R \downarrow \text{match} \cap [\text{Target}_2]_R \downarrow \text{match} \rbrack \\
\text{no-match} & : \lbrack [\text{Target}_1]_R \downarrow \text{no-match} \cup [\text{Target}_2]_R \downarrow \text{no-match} \rbrack \\
\text{indeterminate} & : \lbrack [\text{Target}_1]_R \downarrow \text{indeterminate} \land [\text{Target}_2]_R \downarrow \text{no-match} \rbrack \\
& \cup \lbrack [\text{Target}_2]_R \downarrow \text{indeterminate} \land [\text{Target}_1]_R \downarrow \text{no-match} \rbrack
\end{align*}
\]

\[
[\text{Target}_1 \lor \text{Target}_2]_R =
\begin{align*}
\text{match} & : \lbrack [\text{Target}_1]_R \downarrow \text{match} \cup [\text{Target}_2]_R \downarrow \text{match} \rbrack \\
\text{no-match} & : \lbrack [\text{Target}_1]_R \downarrow \text{no-match} \cap [\text{Target}_2]_R \downarrow \text{no-match} \rbrack \\
\text{indeterminate} & : \lbrack [\text{Target}_1]_R \downarrow \text{indeterminate} \land [\text{Target}_2]_R \downarrow \text{no-match} \rbrack \\
& \cup \lbrack [\text{Target}_2]_R \downarrow \text{indeterminate} \land [\text{Target}_1]_R \downarrow \text{no-match} \rbrack
\end{align*}
\]

The semantics of a match element \(\text{MatchId}(Value, Name)\) is a matching tuple determined by comparing, through the corresponding matching function \(\text{MatchId}\), \(Value\) with the values associated to the request’s attributes identified by \(Name\). Notably, when the attributes are lacking in the request, an empty bag is returned and the matching fails. This, due to the use of the universal quantification in the definition of the set corresponding to \(\text{no-match}\), implies that requests that do not contain attributes named \(Name\) are inserted into the \(\text{no-match}\) set. The list of matching functions supported by FACPL is reported in Table 10. For example,
the function \texttt{equal} returns \texttt{indeterminate} if some error occurs during the evaluation, e.g. the types of the argument values are not correct; otherwise, it returns \texttt{true} if, e.g., in case of strings they have equal length and are equal byte-by-byte according to the function \texttt{integer-equal} (defined by the IEEE standard \cite{IEEE}) and \texttt{false} otherwise. The matching tuples returned by evaluating the match elements within a given target are then combined according to the semantics of the operators \& and \lor given in Table \cite{11}.

By exploiting the target evaluation function, given a set \( R \) of requests, the semantics of a rule is defined by the following clauses:

\[
\begin{align*}
\forall (\text{permit} \ target : Target \ condition : \text{BoolExpr}) \| \forall (\text{deny} \ target : Target \ condition : \text{BoolExpr}) & = \\
& \begin{cases}
\begin{align*}
\{ r \in \llbracket Target \rrbracket & | r \models \text{BoolExpr} = \text{true} \} \\
\cup \llbracket Target \rrbracket & | r \models \text{BoolExpr} = \text{false}
\end{align*}
\end{cases}
\end{align*}
\]

where \( r \models \text{BoolExpr} \) denotes the evaluation of the expression \( \text{BoolExpr} \) with respect to the request \( r \) according to the function definitions reported in Table \cite{12}.

The result of such evaluation can be either \texttt{true}, \texttt{false} or \texttt{indeterminate}. Recall that target and condition of a rule are optional. If one or both of them are missing, the semantics of the rule is determined by the above definitions where, if the condition is omitted, we let \( \text{BoolExpr} \) be \texttt{true} (thus, \( r \models \text{BoolExpr} = \text{true} \) for any \( r \)) and, if the target is omitted, we let \( \llbracket Target \rrbracket = ( \text{match} : R \ \text{no-match} : \emptyset \ \text{indeterminate} : \emptyset ) \).

To define the semantics of policies, we exploit a family of functions of the form \( \text{Ralg}(\cdot) \) that, given a sequence \( \text{Rule}^{+} \) of rules and a set \( R \) of requests, return authorization decisions, that is tuples of the form

\[
\begin{align*}
\{ \text{permit} : \{ r \in R \ | \ \text{Ralg}(\text{Rule}^{+}, r) = \text{permit} \} \} & = \\
\{ \text{deny} : \{ r \in R \ | \ \text{Ralg}(\text{Rule}^{+}, r) = \text{deny} \} \} & = \\
\{ \text{not-applicable} : \{ r \in R \ | \ \text{Ralg}(\text{Rule}^{+}, r) = \text{not-applicable} \} \} & = \\
\{ \text{indetP} : \{ r \in R \ | \ \text{Ralg}(\text{Rule}^{+}, r) = \text{indetP} \} \} & = \\
\{ \text{indetD} : \{ r \in R \ | \ \text{Ralg}(\text{Rule}^{+}, r) = \text{indetD} \} \} & = \\
\{ \text{indetDP} : \{ r \in R \ | \ \text{Ralg}(\text{Rule}^{+}, r) = \text{indetDP} \} \} & = \\
\end{align*}
\]

\( \text{Ralg}(\text{Rule}^{+}, r) \) indicates the result of executing the rule-combining algorithm \( \text{Ralg} \) with input a sequence \( \text{Rule}^{+} \) of rules and a request \( r \). We report in Listing \cite{15} the pseudo-code defining the combining algorithm.
permit-overrides($Rule^+, r$) and relegate to the Appendix A the definitions of the other combining algorithms.

Listing 15. The algorithm permit-overrides with formal parameters $Sequence$ and $r$

```
Boolean atLeastOneErrorD = false;
Boolean atLeastOneErrorP = false;
Boolean atLeastOneErrorDP = false;
Boolean atLeastOneDeny = false;

foreach (elem ∈ Sequence) {
    AuthDecision t = [[ elem ]](r);
    if ($r$ ∈ $t$↓deny) {
        atLeastOneDeny = true;
        continue;
    }
    if ($r$ ∈ $t$↓permit) return permit;
    if ($r$ ∈ $t$↓not-applicable) continue;
    if ($r$ ∈ $t$↓indetD) {
        atLeastOneErrorD = true;
        continue;
    }
    if ($r$ ∈ $t$↓indetP) {
        atLeastOneErrorP = true;
        continue;
    }
    if ($r$ ∈ $t$↓indetDP) {
        atLeastOneErrorDP = true;
        continue;
    }
}
if (atLeastOneErrorDP) return indetDP;
if (atLeastOneErrorP & & (atLeastOneErrorD || atLeastOneDeny)) return indetDP;
if (atLeastOneDeny) return deny;
if (atLeastOneErrorD) return indetD;
return not-applicable;
```

Now, the semantics of a single policy can be defined as follow:

$$[[\{\text{Role target : Target rules : Rule}^+\}]_R =$$

- (permit : $Ralg(Rule^+)_Rm \downarrow_{\text{permit}}$
  - deny : $Ralg(Rule^+)_Rm \downarrow_{\text{deny}}$
  - not-applicable : $R_m \cup Ralg(Rule^+)_Rm \downarrow_{\text{not-applicable}}$
  - $Ralg(Rule^+)_Rm \downarrow_{\text{not-applicable}}$

- indetP : $Ralg(Rule^+)_Rm \downarrow_{\text{indetP}} \cup Ralg(Rule^+)_R \downarrow_{\text{permit}}$
- indetD : $Ralg(Rule^+)_Rm \downarrow_{\text{indetD}} \cup Ralg(Rule^+)_R \downarrow_{\text{deny}}$
- indetDP : $Ralg(Rule^+)_Rm \downarrow_{\text{indetDP}} \cup Ralg(Rule^+)_R \downarrow_{\text{indetDP}}$)
A formal sw. eng. approach to Policy-Based Access Control

where we let \([\text{Target}]_R = (\text{match} : R_m \ \text{no-match} : R_n \ \text{indeterminate} : R_i)\).

Basically, the requests for which the target does not match are evaluated as not-applicable, while the remaining requests are partitioned according to the application of the combining algorithm \(R_{alg}\) to the policy’s rules. Thus, for example, a request is granted access if it matches the target and the result of the combining algorithm is permit, instead it is forbidden access if it matches the target and the result of the combining algorithm is deny. Similarly to the evaluation of rules, if the policy’s target is empty then the policy is evaluated as above by letting \(R_m = R\) and \(R_n = R_i = \emptyset\).

The semantics of a policy set is defined in a way similar to that of a single policy. Thus, we exploit functions of the form \(\text{Palg}()\) that return authorization decisions calculated by applying the policy-combining algorithm \(\text{Palg}\) to the sequence \(\text{Policy}^+\) of policies and the set \(R\) of requests. The definition of the policy-combining algorithms coincides with that of the corresponding rule-combining algorithms, apart of course for that of algorithm only-one-applicable.

For example, \(\text{permit-overrides}(\text{Policy}^+, r)\) is again defined by the pseudo-code in Listing 15. Now, the semantics of a policy set is defined by the following clause:

\[
\begin{align*}
[\{\text{Palg target: Target policies: Policy}^+\}]_R = \\
\text{(permit : Palg(Policy)\(\downarrow\text{permit}\) } \\
\text{deny : Palg(Policy)\(\downarrow\text{deny}\) } \\
\text{not-applicable : } R_n \cup \text{Palg(Policy)}\(\downarrow\text{not-applicable}\) } \\
\text{indetP : Palg(Policy)\(\downarrow\text{indetP}\) } \\
\text{indetD : Palg(Policy)\(\downarrow\text{indetD}\) } \\
\text{indetDP : Palg(Policy)\(\downarrow\text{indetDP}\) }
\end{align*}
\]

where we let \([\text{Target}]_R = (\text{match} : R_m \ \text{no-match} : R_n \ \text{indeterminate} : R_i)\) and, if the target is empty, we also let \(R_m = R\) and \(R_n = R_i = \emptyset\). The clause is very similar to that defining the semantics of a single policy: we have policies and policy-combining algorithms instead of rules and rule-combining algorithms.

Finally, the semantics of a PDP top-level term \(\{\text{Palg policies: Policy}^+\}\) is determined by applying the clause for policy sets with an empty target:

\[
\begin{align*}
[\{\text{Palg policies: Policy}^+\}]_R = \\
\text{(permit : Palg(Policy)\(\downarrow\text{permit}\) } \\
\text{deny : Palg(Policy)\(\downarrow\text{deny}\) } \\
\text{not-applicable : Palg(Policy)\(\downarrow\text{not-applicable}\) } \\
\text{indetP : Palg(Policy)\(\downarrow\text{indetP}\) } \\
\text{indetD : Palg(Policy)\(\downarrow\text{indetD}\) } \\
\text{indetDP : Palg(Policy)\(\downarrow\text{indetDP}\) }
\end{align*}
\]
Semantics with obligations. Let us now refine the PDP decision process for taking the role of obligations into account. To this aim, we first introduce an auxiliary function \( \langle \cdot \rangle \), for evaluating a sequence of obligations (\( \epsilon \) denotes the empty sequence, \( \cdot \) is the concatenation operator between a single obligation, resp. evaluated obligation, and a sequence of obligations, resp. evaluated obligations) with respect to a given request:

\[
\begin{align*}
\langle \epsilon \rangle_r &= \epsilon \\
\langle o \cdot \{Obligation^*\} \rangle_r &= \\
&= \begin{cases} 
\langle eo \cdot \{Obligation^*\} \rangle_r & \text{if } r \models o = eo \text{ and } \langle \{Obligation^*\} \rangle_r \neq \text{undef} \\
\text{undef} & \text{otherwise}
\end{cases}
\end{align*}
\]

Intuitively, \( r \models o \) denotes the evaluation of the obligation \( o \) with respect to the request \( r \): it returns an evaluated obligation \( eo \) when all argument expressions of \( o \) can be successfully evaluated by using \( r \); otherwise, e.g. if an attribute used in an argument expression cannot be retrieved, it returns \( \text{undef} \). Thus, if no error occurs in the evaluation of any obligation of a sequence, function \( \langle \cdot \rangle \) returns a sequence of evaluated obligations (ranged over by \( EO \)); otherwise, it returns \( \text{undef} \) (i.e. the evaluation of the whole sequence is undefined).

Given a sequence \( Obligation^* \) of obligations to fulfill, we will use notation \( Obligation^*_{\text{Effect}} \), where \( \text{Effect} \) is \textit{permit} or \textit{deny}, to indicate the subsequence of \( Obligation^* \) only including those obligations having \( \text{Effect} \) as associated applicability effect.

Now, when taking obligations into account, while the function \( [\text{Target}] \) for targets evaluation does not change, the functions providing the semantics of rules, rule- and policy-combining algorithms, policies and policy sets are modified to return an authorization decision, that is a tuple \( AD \) where the elements of the sets corresponding to decisions \( \text{permit} \) and \( \text{deny} \) are pairs of the form \( (r, EO) \).
The semantics of rules is defined by the following clauses:

\[
\{(\text{permit target : Target condition : BoolExpr obl : Obligation}^+)\}_R = \\
(\text{permit }): \{ \langle r, \langle \text{Obligation}_{\text{permit}}^+ \rangle_r \rangle | r \in [\text{Target}]_R \uparrow_{\text{match}}, r \models \text{BoolExpr} = \text{true} \} \\
\text{not-applicable}: \{ r \in [\text{Target}]_R \uparrow_{\text{no-match}} \} \\
\text{indetP}: \{ r \in [\text{Target}]_R \uparrow_{\text{match}} | r \models \text{BoolExpr} = \text{indeterminate} \} \\
\text{indetD}: \{ r \in [\text{Target}]_R \uparrow_{\text{match}} | r \models \text{BoolExpr} = \text{true}, \} \\
(\langle \text{Obligation}_{\text{permit}}^+ \rangle_r, \text{undef} \})
\]

\[
\{(\text{deny target : Target condition : BoolExpr obl : Obligation}^+)\}_R = \\
(\text{deny }): \{ \langle r, \langle \text{Obligation}_{\text{deny}}^+ \rangle_r \rangle | r \in [\text{Target}]_R \downarrow_{\text{match}}, r \models \text{BoolExpr} = \text{true} \} \\
\text{not-applicable}: \{ r \in [\text{Target}]_R \downarrow_{\text{no-match}} \} \\
\text{indetP}: \{ r \in [\text{Target}]_R \downarrow_{\text{match}} | r \models \text{BoolExpr} = \text{indeterminate} \} \\
\text{indetD}: \{ r \in [\text{Target}]_R \downarrow_{\text{match}} | r \models \text{BoolExpr} = \text{true}, \} \\
(\langle \text{Obligation}_{\text{deny}}^+ \rangle_r, \text{undef} \})
\]

The set of granted requests is now refined. Indeed, in order for a request to be granted it is required that all rule’s obligations with applicability effect permit can be successfully fulfilled by exploiting the request. Literally, it is required that \(\langle \text{Obligation}_{\text{permit}}^+ \rangle_r\) returns a sequence of evaluated obligations, which will be associated to the request. Instead, if there is at least one obligation with applicability effect permit that cannot be fulfilled (i.e., function \(\langle \text{Obligation}_{\text{permit}}^+ \rangle_r\), is undefined), the request returns a decision indetP. Notably, only obligations with the same applicability effect as the rule’s effect are taken into account. The clause for the effect deny has a dual meaning.

Let us now consider \(\text{Ralg}(\text{Rule}^+, r)\), which indicates the result of executing a rule-combining algorithm \(\text{Ralg}\) with input a sequence \(\text{Rule}^+\) of rules and a request \(r\). When the decision is permit or deny, the algorithm returns, other than the decision, also a sequence of obligations. Similar observations also hold for policy-combining algorithms. Thus, the algorithm permit-overrides shown in Listing 15 when taking obligations into account, becomes that shown in Listing 16.

Listing 16. The algorithm permit-overrides with obligations handling

```
1 Boolean atLeastOneErrorD = false;
2 Boolean atLeastOneErrorP = false;
3 Boolean atLeastOneErrorDP = false;
4 Boolean atLeastOneDeny = false;
5 List OblDeny = [];
6 foreach (elem ∈ Sequence) { 
7  AuthDecision t = [ elem ](r);
```
if \((r, EO) \in t_{\text{deny}}\) {  
  OblDeny.add(EO);
  atLeastOneDeny = true;
  continue;
}
if \((r, EO) \in t_{\text{permit}}\) return <permit, EO>;
if \((r \in t_{\text{not-applicable}}\) continue;
if \((r \in t_{\text{indetD}}\) {  
  atLeastOneErrorD = true;
  continue;
}
if \((r \in t_{\text{indetP}}\) {  
  atLeastOneErrorP = true;
  continue;
}
if \((r \in t_{\text{indetDP}}\) {  
  atLeastOneErrorDP = true;
  continue;
}
if \((\text{atLeastOneErrorDP})\) return indetDP;
if \((\text{atLeastOneErrorP}) \&\& (\text{atLeastOneErrorD} \mid \text{atLeastOneDeny})\)
  return indetDP;
if \((\text{atLeastOneErrorP})\) return indetP;
if \((\text{atLeastOneDeny})\) return <deny, OblDeny>;
if \((\text{atLeastOneErrorD})\) return indetD;
return not-applicable;

Since we consider only ordered combining algorithms, the sequence of obligations related to a request is deterministically identified. Therefore, the functions of the form \(R_{alg}(\cdot)\) are now defined as follows:

\[
R_{alg}(\text{Rule}^+)_{R} = \\
\begin{align*}
\text{permit} &: \{(r, EO) \mid r \in R, R_{alg}(\text{Rule}^+, r) = \langle \text{permit}, EO \rangle \} \\
\text{deny} &: \{(r, EO) \mid r \in R, R_{alg}(\text{Rule}^+, r) = \langle \text{deny}, EO \rangle \} \\
\text{not-applicable} &: \{r \in R \mid R_{alg}(\text{Rule}^+, r) = \text{not-applicable} \} \\
\text{indetP} &: \{r \in R \mid R_{alg}(\text{Rule}^+, r) = \text{indetP} \} \\
\text{indetD} &: \{r \in R \mid R_{alg}(\text{Rule}^+, r) = \text{indetD} \} \\
\text{indetDP} &: \{r \in R \mid R_{alg}(\text{Rule}^+, r) = \text{indetDP} \} \\
\end{align*}
\]

Of course, functions \(P_{alg}(\cdot)\) are defined similarly.
The semantics of a single policy can now be defined by the following clause:

\[ (\text{Target} : \text{Target}, \text{rules} : \text{Rule}^+, \text{obl} : \text{Obligation}^*) )_R = \]

( \text{permit} : \{(r, EO \cdot \{\text{Obligation}_{\text{permit}}^*\}_r) \mid (r, EO) \in \text{Ralg}(\text{Rule}^+)_{R_m} \} \downarrow \text{permit} \)

\text{deny} : \{(r, EO \cdot \{\text{Obligation}_{\text{deny}}^*\}_r) \mid (r, EO) \in \text{Ralg}(\text{Rule}^+)_{R_m} \} \downarrow \text{deny} \)

\text{not-applicable} : R_n \cup \text{Ralg}(\text{Rule}^+)_{R_m} \downarrow \text{not-applicable} \)

\text{indetP} : \text{Ralg}(\text{Rule}^+)_{R_m} \downarrow \text{indetP} \cup \text{Ralg}(\text{Rule}^+)_{R_i} \downarrow \text{permit} \)

\text{indetD} : \text{Ralg}(\text{Rule}^+)_{R_m} \downarrow \text{indetD} \cup \text{Ralg}(\text{Rule}^+)_{R_i} \downarrow \text{deny} \)

\text{indetDP} : \text{Ralg}(\text{Rule}^+)_{R_m} \downarrow \text{indetDP} \cup \text{Ralg}(\text{Rule}^+)_{R_i} \downarrow \text{indetDP} \)

where we let \([\text{Target}]_R = (\text{match} : R_m \text{ no-match} : R_n \text{ indeterminate} : R_i)\) and, if the target is empty, we also let \(R_m = R\) and \(R_n = R_i = \emptyset\). Notably, the sequence of obligations associated to a request to which access is granted or forbidden can include both obligations coming from rules (retrieved by the rule-combining algorithm) and from the policy itself. An indeterminate result can also be caused by an obligation wrongly evaluated by the PDP, see cases indetP and indetD.

The semantics with obligation handling of policy sets and PDP top-level terms is defined in a way similar to that of policies.

### 5.2 Semantics of the PEP enforcement process

The semantics of the enforcement process defines how the PEP discharges obligations and enforces the authorization decisions. It is captured by function \([\cdot]_{AD}\) that, given an authorization decision \(AD\) and chosen an enforcement algorithm among base, deny-biased and permit-biased (described in Section 2.3), and debug (introduced in Section 4.1), returns an enforceable decision, that is a tuple that partitions the requests according to the result of the enforcing process.

To define the semantics of the PEP, we use the function \([\cdot]_{PEP}\) that, given a sequence \(EO\) of evaluated obligations fulfilled by the PDP, returns a boolean value indicating if the mandatory obligations in the sequence have been successfully discharged or not. Notably, since errors caused by optional obligations can be safely ignored by the PEP, we only need to check failures of mandatory
obligations (i.e., of type M). The function is defined as follows:

\[
\langle \epsilon \rangle_{PEP} = \text{true}
\]

\[
\langle [O \ PepAction(\text{Value}^*)] \cdot EO \rangle_{PEP} = \langle EO \rangle_{PEP}
\]

\[
\langle [M \ PepAction(\text{Value}^*)] \cdot EO \rangle_{PEP} = \begin{cases} 
\text{false} & \text{if } \text{PepAction(\text{Value}^*)} \Downarrow \text{fail} \\
\langle EO \rangle_{PEP} & \text{otherwise}
\end{cases}
\]

where \(\text{PepAction(\text{Value}^*)} \Downarrow \text{fail}\) denotes that execution of the action failed. Of course, since \(\text{PepAction}\) is unspecified because obligation actions are out of scope of this work (and of the XACML specification as well), also the definition of the predicate \(\cdot \Downarrow \) is left unspecified.

Finally, function \(\Downarrow_{AD}\) is defined by the following clauses:

\[
\text{base}_{AD} =
\]

\[
\begin{array}{l}
\text{permit: } \{ r \mid (r, EO) \in AD \downarrow_{\text{permit}}, \langle EO \rangle_{PEP} \} \\
\text{deny: } \{ r \mid (r, EO) \in AD \downarrow_{\text{deny}}, \langle EO \rangle_{PEP} \} \\
\text{not-applicable: } AD \downarrow_{\text{not-applicable}} \\
\text{indetP: } AD \downarrow_{\text{indetP}} \cup AD \downarrow_{\text{indetD}} \cup AD \downarrow_{\text{indetDP}} \\
\end{array}
\]

\[
\text{denied}_{AD} =
\]

\[
\begin{array}{l}
\text{permit: } \{ r \mid (r, EO) \in AD \downarrow_{\text{permit}}, \langle EO \rangle_{PEP} \} \\
\text{deny: } \{ r \mid (r, EO) \in AD \downarrow_{\text{deny}}, \langle EO \rangle_{PEP} \} \\
\text{not-applicable: } AD \downarrow_{\text{not-applicable}} \cup AD \downarrow_{\text{indetP}} \cup AD \downarrow_{\text{indetD}} \cup AD \downarrow_{\text{indetDP}} \\
\end{array}
\]

\[
\text{debug}_{AD} =
\]

\[
\begin{array}{l}
\text{permit: } \{ r \mid (r, EO) \in AD \downarrow_{\text{permit}}, \langle EO \rangle_{PEP} \} \\
\text{deny: } \{ r \mid (r, EO) \in AD \downarrow_{\text{deny}}, \langle EO \rangle_{PEP} \} \\
\text{not-applicable: } AD \downarrow_{\text{not-applicable}} \\
\text{indetP: } AD \downarrow_{\text{indetP}} \cup \{ r \mid (r, EO) \in AD \downarrow_{\text{permit}}, \neg\langle EO \rangle_{PEP} \} \\
\text{indetD: } AD \downarrow_{\text{indetD}} \cup \{ r \mid (r, EO) \in AD \downarrow_{\text{deny}}, \neg\langle EO \rangle_{PEP} \} \\
\text{indetDP: } AD \downarrow_{\text{indetDP}} \\
\end{array}
\]

Notably, here we have completed the definition of (the result returned by) the \text{base} algorithm, which is not fully defined in the XACML specification (see Section 2.3), by returning \text{not-applicable} for all those requests that belong to the
corresponding set in the authorization decision and by returning \texttt{indetDP} for all those requests belonging to the three sets corresponding to the extended indeterminate values and for all those requests whose mandatory obligations cannot be successfully discharged. Moreover, we propose a variant of such algorithm, called \texttt{debug}, that does not merge the sets of requests corresponding to the extended indeterminate values into a single set. Extended indeterminate values are thus retained for enabling policy debug, as they provide a deeper knowledge about the progress of the PDP decision process.

6 Examples

In this section, we show how the semantics definitions presented in Section 5 apply to the policy examples inspired by those of the epSOS project, introduced in Sections 3.1 and 4.2. We first present the PDP decision process, then the PEP enforcement process.

In the sequel we will use the following notations:

- given a set \( R \) of requests, \( \overline{R} \) denotes the set \( \{ \langle r, \epsilon \rangle | r \in R \} \), i.e. the set of requests in \( R \) paired with an empty sequence of evaluated obligations;
- given a set \( R \) of requests with evaluated obligations, notation \( \underline{R} \) denotes the set \( \{ r | \exists EO : \langle r, EO \rangle \in R \} \), i.e. the set of requests in \( R \) without the attached obligations.

To simplify the presentation of the examples, we also assume that all attribute values within the requests in the considered set \( R \) have type string.

6.1 The decision process

We present in details the evaluation of the PDP decision process for the deny-all policy and the privacy consent policy introduced in Section 3.1. Then we show how these authorization decisions are combined by the enclosing policy set reported in Listing 3. We proceed in two steps: firstly we present a semantics that ignores obligations, then we show how obligations are taken into account.

**Deny-all policy.** Consider the policy reported in Listing 1 and rewritten in Section 4.2 with the syntax defined in Table 10 as follows:

\[
\langle \text{permit-overrides rules : (deny)} \rangle
\]

Since the policy’s target is empty (hence the policy applies to all requests), its semantics, parameterized with respect to a set \( R \) of requests, is:

\[
\llangle \langle \text{permit-overrides rules : (deny)} \rangle \rrangle_R = \text{permit-overrides((deny))}_R
\]
That is the policy’s semantics coincides with the authorization decision returned by the function application $\text{permit-overrides}((\text{deny}))_R$, i.e.

\[
\begin{align*}
\text{permit} &: \{ r \in R \mid \text{permit-overrides}((\text{deny}), r) = \text{permit} \} \\
\text{deny} &: \{ r \in R \mid \text{permit-overrides}((\text{deny}), r) = \text{deny} \} \\
\text{not-applicable} &: \{ r \in R \mid \text{permit-overrides}((\text{deny}), r) = \text{not-applicable} \} \\
\text{indetP} &: \{ r \in R \mid \text{permit-overrides}((\text{deny}), r) = \text{indetP} \} \\
\text{indetD} &: \{ r \in R \mid \text{permit-overrides}((\text{deny}), r) = \text{indetD} \} \\
\text{indetDP} &: \{ r \in R \mid \text{permit-overrides}((\text{deny}), r) = \text{indetDP} \}
\end{align*}
\]

In order to further elaborate the previous term, let us now consider the rule $(\text{deny})$. Given a set $R$ of requests, its semantics is given by the tuple:

\[
\begin{align*}
\text{permit} &: \emptyset \\
\text{deny} &: R \\
\text{not-applicable} &: \emptyset \\
\text{indetP} &: \emptyset \\
\text{indetD} &: \emptyset \\
\text{indetDP} &: \emptyset
\end{align*}
\]

Indeed, the rule applies to all requests, since target and condition are missing, and the effect is always deny. Therefore, by considering the semantics of the rule above, the algorithm $\text{permit-overrides}((\cdot,\cdot)$ can be instantiated (and partially evaluated) on the sequence of rules only containing $(\text{deny})$ and on a generic request $r$ as follows:

```plaintext
1 Boolean atLeastOneErrorD = false;
2 Boolean atLeastOneErrorP = false;
3 Boolean atLeastOneErrorDP = false;
4 Boolean atLeastOneDeny = false;
5 if (r \in \{r\}) { 
6   atLeastOneDeny = true;
7   continue;
8 } 
9 if (r \in \emptyset) { 
10   return permit;
11 }
12 if (r \in \emptyset) continue;
13 if (r \in \emptyset) { 
14   atLeastOneErrorD = true;
15   continue;
16 } 
17 if (r \in \emptyset) { 
18   atLeastOneErrorP = true;
19   continue;
20 } 
21 if (r \in \emptyset) { 
22   atLeastOneErrorDP = true;
23   continue;
24 }
25 if (atLeastOneErrorDP) return indetDP;
26 if (atLeastOneErrorD \&\& (atLeastOneErrorP \| atLeastOneDeny)) 
27   return indetDP;
28 if (atLeastOneErrorP) return indetP;
29 if (atLeastOneDeny) return deny;
```
if (atLeastOneErrorD) return indetD;
return not-applicable;

The code above is in fact obtained from the code in Listing 15 by considering that

– the foreach’s condition (elem ∈ {(deny)}) implies that elem = (deny), and
– the variable declaration AuthDecisions t = \[\{deny\}\]_{\{r\}} implies that t =
(permit: \{\} deny: \{r\} not-applicable: \{\} indetP: \{\} indetD: \{\} indetDP: \{\}).

Since condition \{r\} is always true while condition r ∈ ∅ is false, the above algorithm returns deny for every request r. Therefore, the authorization decision returned by permit-overrides{(deny)}_R, and hence also by \[(\text{permit-overrides rules : (deny)})\]_R, is:

(permit: \{\} deny: R not-applicable: \{\} indetP: \{\} indetD: \{\} indetDP: \{\})

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

The policy reported in Listing 1 has no attached obligations, thus the authorization decision does not change if we use the semantics with obligations. To be precise, the sets corresponding to decisions permit and deny should be made of pairs of the form \(\langle r, E_O \rangle\). Therefore, when taking obligations into account, in the authorization decision above we replace R by \(\hat{R}\).

**Privacy consent policy.** Let us now consider the policy reported in Listing 2 and rewritten in Section 4.2 with the syntax defined in Table 10. If we temporally ignore obligations, it is as follows:

\[
\text{permit-overrides}
\text{target : equal("medical doctor", subject}/\text{role})
\land \text{equal("TREATMENT", subject}/\text{purposeofuse})
\land \text{equal("34133-9", resource}/\text{resource-id})
\text{rules : ( permit target : equal("Read", action}/\text{action-id})}
\text{condition : subset(}
\text{string-bag("PRD-003","PRD-005","PRD-010","PRD-016"),}
\text{subject}/\text{hl7 : permission}))
\]

In order to define its semantics, we first evaluate its target. In particular, the evaluation of the first match element occurring in the policy’s target is as
follows:

\[
\text{equal}(\text{“medical doctor”}, \text{subject/role}) \mid R = \\
\quad \text{match} : \{ r \in R \mid \exists (\text{subject/role}, \text{value}) \in r : \\
\quad \quad \quad \text{equal}(\text{“medical doctor”}, \text{value}) = \text{true} \} ; \\
\quad \text{no-match} : \{ r \in R \mid \forall (\text{subject/role}, \text{value}) \in r : \\
\quad \quad \quad \text{equal}(\text{“medical doctor”}, \text{value}) = \text{false} \} ; \\
\quad \text{indeterminate} : \{ r \in R \mid \exists (\text{subject/role}, \text{value}) \in r : \\
\quad \quad \quad \text{equal}(\text{“medical doctor”}, \text{value}) = \text{indeterminate} , \\
\quad \quad \quad \not\exists (\text{subject/role}, \text{value}) \in r : \\
\quad \quad \quad \text{equal}(\text{“medical doctor”}, \text{value}) = \text{true} \} 
\]

The above authorization decision can be made more readable by exploiting the fact that function equal never returns indeterminate and returns true if and only if the two argument strings are identical. Thus, we have:

\[
\text{match} : \{ r \in R \mid (\text{subject/role}, \text{“medical doctor”}) \in r \} ; \\
\text{no-match} : \{ r \in R \mid (\text{subject/role}, \text{“medical doctor”}) \not\in r \} ; \\
\text{indeterminate} : \emptyset
\]

By exploiting similar observations, the evaluation of the other match elements occurring in the policy’s target is as follows:

\[
\text{equal}(\text{“TREATMENT”}, \text{subject/purposeofuse}) \mid R = \\
\quad \text{match} : \{ r \in R \mid (\text{subject/purposeofuse}, \text{“TREATMENT”}) \in r \} ; \\
\quad \text{no-match} : \{ r \in R \mid (\text{subject/purposeofuse}, \text{“TREATMENT”}) \not\in r \} ; \\
\quad \text{indeterminate} : \emptyset
\]

\[
\text{equal}(\text{“34133-9”}, \text{resource/resource-id}) \mid R = \\
\quad \text{match} : \{ r \in R \mid (\text{resource/resource-id}, \text{“34133-9”}) \in r \} ; \\
\quad \text{no-match} : \{ r \in R \mid (\text{resource/resource-id}, \text{“34133-9”}) \not\in r \} ; \\
\quad \text{indeterminate} : \emptyset
\]

12 This is due to the fact that all attribute values in the policies are strings and to the assumption that all attribute values in the requests are strings. If such assumption would not hold, non-string values could be retrieved from requests and used as argument of function equal; this would lead to an indeterminate evaluation.
Therefore, the semantics of the target, obtained by combining the above authorization decisions by means of the operator ∧, is as follows:

\[
\text{equal}(\text{"medical doctor"}, \text{subject/role}) \wedge \text{equal}(\text{"TREATMENT"}, \text{subject/purposeofuse}) \wedge \text{equal}(\text{"34133-9"}, \text{resource/resource-id}) = \\
\{ \text{match} : \{ r \in R \mid \text{(subject/role, "medical doctor") \in r} \}
\cap \{ r \in R \mid \text{(subject/purposeofuse, "TREATMENT") \in r} \}
\cap \{ r \in R \mid \text{(resource/resource-id, "34133-9") \in r} \}; \\
\text{no-match} : \{ r \in R \mid \text{(subject/role, "medical doctor") \notin r} \}
\cup \{ r \in R \mid \text{(subject/purposeofuse, "TREATMENT") \notin r} \}
\cup \{ r \in R \mid \text{(resource/resource-id, "34133-9") \notin r} \}; \\
\text{indeterminate} : \emptyset \}
\]

The resulting tuple can be also rewritten as follows:

\[
\{ \text{match} : \{ r \in R_m \mid \text{(subject/role, "medical doctor") \in r}, \\
\text{(subject/purposeofuse, "TREATMENT") \in r}, \\
\text{(resource/resource-id, "34133-9") \in r} \}; \\
\text{no-match} : \{ r \in R_m \mid \text{(subject/role, "medical doctor") \notin r}, \\
\lor \text{(subject/purposeofuse, "TREATMENT") \notin r}, \\
\lor \text{(resource/resource-id, "34133-9") \notin r} \}; \\
\text{indeterminate} : \emptyset \}
\]

In the following we will use \( R_m \) to refer to the first component of the tuple above; hence, the second one is \( R \setminus R_m \).

Now, we evaluate the rule enclosed within the policy with respect to the set \( R_m \) of matching requests. We start from its target and, by exploiting the simplifications discussed above, we have:

\[
\text{equal}(\text{"Read", action/action-id})]_{R_m} = \\
\{ \text{match} : \{ r \in R_m \mid \text{(action/action-id, "Read") \in r} \}; \\
\text{no-match} : \{ r \in R_m \mid \text{(action/action-id, "Read") \notin r} \}; \\
\text{indeterminate} : \emptyset \}
\]

We will use \( R^c_m \) to refer to the first component of the matching tuple above, i.e. the set of matching requests also satisfying the rule’s target; hence, the second component is \( R_m \setminus R^c_m \).

Let \( e \) denote the expression

\[
\text{subset(string-bag("PRD-003", "PRD-005", "PRD-010", "PRD-016"), \\
\text{subject/hl7 : permission})}
\]

within the rule’s condition. It is worth noticing that functions \( \text{subset} \) and \( \text{string-bag} \) never return \( \text{indeterminate} \) even if an empty bag is retrieved from
the request; thus, for any request \( r \), \( r \models e \) can only evaluate to either \( \text{true} \) or \( \text{false} \). Therefore, given \( r \in R_m^e \), the evaluation of the expression \( e \) is as follows:

\[
\begin{align*}
    r \models e &= \begin{cases} 
    \text{true} & \text{if } \{ \text{“PRD-003”, “PRD-005”, “PRD-010”, “PRD-016”} \} \\
    \subseteq \bigcup \{ \text{value}, | (\text{subject}/\text{hl7 : permission}, \text{value}) \in r \} 
    \end{cases} \\
    \text{false} & \text{otherwise}
\end{align*}
\]

Finally, the semantics of the rule is obtained by combining the sets \( R_m^e \) and \( R_m \setminus R_m^e \) of the matching tuple by using the condition above. The resulting tuple, given the set of matching requests \( R_m \) of the enclosing policy, is as follows:

\[
\{(\text{permit target : equal(“Read”, action/action-id) condition : e}) | R_m = \\
(\text{permit : } \{ r \in R_m^e \ | \ \{ \text{“PRD-003”, “PRD-005”, “PRD-010”, “PRD-016”} \} \\
\subseteq \bigcup \{ \text{value}, | (\text{subject}/\text{hl7 : permission}, \text{value}) \in r \} \}) \\
\text{not-applicable : } (R_m \setminus R_m^e) \cup \\
\{ r \in R_m^e \ | \ \{ \text{“PRD-003”, “PRD-005”, “PRD-010”, “PRD-016”} \} \\
\subseteq \bigcup \{ \text{value}, | (\text{subject}/\text{hl7 : permission}, \text{value}) \in r \} \}) \}
\]

We will use \( R_m^{ec} \) to refer to the first component of the tuple above. Notice that the second component is then \( (R_m \setminus R_m^e) \cup (R_m^e \setminus R_m^{ec}) \), that is \( R_m \setminus R_m^{ec} \).

Now, the semantics of the policy’s target and rule can be exploited to customize the policy’s combining algorithm \( \text{permit overrides} \) reported in Listing 15. Indeed, by considering that for the policy under consideration we have

\[
\text{Sequence} = \{(\text{permit target : equal(“Read”, action/action-id) condition : e}) \\
\text{t} = \{(\text{permit target : equal(“Read”, action/action-id) condition : e}) \}_{(r)} \\
\text{t }\downarrow\text{permit} = R_m^{ec} \\
\text{t }\downarrow\text{not-applicable} = R_m \setminus R_m^{ec} \\
\text{t }\downarrow\text{deny} = t \downarrow\text{indetP} = t \downarrow\text{indetD} = t \downarrow\text{indetDP} = \emptyset
\]

then the partially evaluated instantiation of the algorithm for a generic request \( r \) is as follows:

```java
1. Boolean atLeastOneErrorD = false;
2. Boolean atLeastOneErrorP = false;
3. Boolean atLeastOneErrorDP = false;
4. Boolean atLeastOneDeny = false;
5. if \((r \in \emptyset)\) {
6.     atLeastOneDeny = true;
7.     continue;
8. }
9. if \((r \in R_m^{ec})\) return permit;
10. if \((r \in \emptyset)\) continue;
11. if \((r \in \emptyset)\) {
12.     atLeastOneErrorD = true;
13.     continue;
14. }
15. if \((r \in \emptyset)\) {
```
Since the condition \( r \in \emptyset \) is false, the authorization decision defining the semantics of the policy is as follows:

\[
(\text{permit} : R^{tc}_m \text{ deny} : \emptyset \text{ not-applicable} : R \setminus R^{tc}_m \text{ indetP} : \emptyset \text{ indetD} : \emptyset \text{ indetDP} : \emptyset)
\]

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

- The set \( R^{tc}_m \) of requests for which the policy evaluates to permit is
  \[
  \{ r \in R \mid (\text{subject}/\text{role}, \text{"medical doctor"}) \in r, \\
  (\text{subject}/\text{purposeofuse}, \text{"TREATMENT"}) \in r, \\
  (\text{resource}/\text{resource-id}, \text{"34133-9"}) \in r, \\
  (\text{action}/\text{action-id}, \text{"Read"}) \in r, \\
  \{\text{PRD}-003", \text{PRD}-005", \text{PRD}-010", \text{PRD}-016"\} \\
  \subseteq \bigcup \{\text{value}, \mid (\text{subject}/\text{hl7} : \text{permission}, \text{value}) \in r\}\}
  \]

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

- The set \( R \setminus R^{tc}_m \) of requests for which the policy evaluates to not-applicable is
  \[
  \{ r \in R \mid (\text{subject}/\text{role}, \text{"medical doctor"}) \notin r, \\
  (\text{subject}/\text{purposeofuse}, \text{"TREATMENT"}) \notin r, \\
  (\text{resource}/\text{resource-id}, \text{"34133-9"}) \notin r, \\
  (\text{action}/\text{action-id}, \text{"Read"}) \notin r, \\
  \{\text{PRD}-003", \text{PRD}-005", \text{PRD}-010", \text{PRD}-016"\} \\
  \notin \bigcup \{\text{value}, \mid (\text{subject}/\text{hl7} : \text{permission}, \text{value}) \in r\}\}
  \]

  This means that the policy is not applicable to a given request if
  1. the request has not been issued by a medical doctor, or
  2. the purpose is not the treatment of a patient, or
  3. the request does not refer to a patient summary, or
  4. the request is not for a read action, or
  5. the doctor does not have the appropriate permissions.
It is worth mentioning that the policy never evaluates to deny or to any type of indeterminate value.

Now, we consider again the policy reported in Listing 2 and rewritten in Section 4.2 with the syntax defined in Table 10. This time we also take obligations into account, thus the policy is as follows:

\[
\langle \text{permit-overrides} \\
\quad \text{target} : \text{equal}(\text{"medical doctor"}, \text{subject}/\text{role}) \\
\quad \quad \wedge \text{equal}(\text{"TREATMENT"}, \text{subject}/\text{purposeofuse}) \\
\quad \quad \wedge \text{equal}(\text{"34133-9"}, \text{resource}/\text{resource-id}) \\
\quad \text{rules} : ( \text{permit target} : \text{equal}(\text{"Read"}, \text{action}/\text{action-id}) \\
\quad \quad \quad \quad \text{condition} : \text{subset}(
\quad \quad \quad \quad \quad \text{string-bag}(\text{"PRD-003"}, \text{"PRD-005"}, \text{"PRD-010"}, \text{"PRD-016"}), \\
\quad \quad \quad \quad \quad \text{subject}/\text{hl7} : \text{permission}) \\
\quad \quad \quad \quad \text{obl} : [\text{permit M log(subject/subject-id, resource/resource-id)}] \\
\rangle
\]

Evaluation of policy’s target and rules is as before, but now evaluation of the policy relies on function \( | \cdot |_r \) for evaluating obligations. The policy has an obligation with applicability effect permit, therefore in case of decision permit the evaluation depends on the application of function \( | \cdot |_r \). The function \( | \cdot |_r \) is applied to a sequence of obligations \( o \), given a generic request \( r \), and returns either a sequence of evaluated obligations or undef in case of errors. Thus, if no error occurs we have the following result

\[
|\left[\text{permit M log(subject/subject-id, resource/resource-id)}\right]|_r = [M \log(\text{value, value}')]
\]

which is obtained under the conditions

\[
\exists \text{value} : (\text{subject/subject-id, value}) \in r, \\
\exists \text{value}' : (\text{resource/resource-id, value}') \in r
\]

Notably, when the policy evaluates to permit, it means that the request \( r \) belongs to set \( R_{tc} \), therefore condition \((\text{resource/resource-id, value}') \in r\) always holds and, moreover, we have \( \text{value}' = \text{"34133-9"} \). Therefore, only the attribute \((\text{subject/subject-id, value})\) can produce an unsuccessfully evaluation.

Obligations, and the conditions for which they succeed or not, must be now taken into account when defining the semantics of the policy. Therefore, given a set \( R \) of requests, it holds that:

- The set \( R_p \) of pairs \( (r, EO) \) for which the policy evaluates to permit is

\[
\{ (r, [M \log(\text{value, \"34133-9\")}] | r \in R_{tc} : (\text{subject/subject-id, value}) \in r \}
\]

Basically, \( R_p \) includes all the requests that were permitted when ignoring obligations which also provide the identifier of the issuing subject.

- The set of requests for which the policy evaluates to deny is still empty.

- The set of requests for which the policy evaluates to not-applicable is the same as before (i.e. the set \( R \setminus R_{tc} \)).
Differently from the case in which obligations are ignored, now the policy can evaluate to an indeterminate result when the obligation cannot be fulfilled. More precisely, the set of requests for which the policy evaluates to \( \text{indetP} \) is \( R_{tc} \setminus R_p \).

Therefore, the authorization decision of the evaluation of the privacy consent policy, in the semantics with obligation, is:

\[
( \text{permit} : R_p \; \text{deny} : \emptyset \; \text{not-applicable} : R \\setminus R_{tc} \; \text{indetP} : R_{tc} \setminus R_p \; \text{indetD} : \emptyset \; \text{indetDP} : \emptyset )
\]

**Privacy consent policy set.** Let us now consider the policy set reported in Listing 3 and rewritten in Section 4.2 with the syntax defined in Table 10 as follows:

\[
\{ \text{permit-overrides} \\
\text{policies :} \\
\langle \text{permit-overrides rules : (deny)} \rangle \\
\langle \text{permit-overrides} \\
\text{target : equal("medical doctor", subject/role)} \\
\text{&& equal("TREATMENT", subject/purposeofuse)} \\
\text{&& equal("34133-9", resource/resource-id)} \\
\text{rules : ( permit target : equal("Read", action/action-id)} \\
\text{condition : subset( \\
\text{string-bag("PRD-003","PRD-005","PRD-010","PRD-016"),} \\
\text{subject/hl7 : permission)});} \\
\text{obl : [ permit M log(subject/subject-id, resource/resource-id)]} \\
\text{obl : [ permit M redirect(epsos/redirect-url)]} \\
\text{obl : [ deny M mail(resource/resource-id.email,} \\
\text{"Your medical record has been requested by epSOS")}]}
\]

Since its target is empty, the policy set applies to all requests. To calculate the final authorization decision, given a set \( R \) of requests, we start from the authorization decision of the deny-all policy

\[
( \text{permit} : \emptyset \; \text{deny} : \emptyset \; \text{not-applicable} : \emptyset \; \text{indetP} : \emptyset \; \text{indetD} : \emptyset \; \text{indetDP} : \emptyset )
\]

and from the authorization decision of the privacy consent policy

\[
( \text{permit} : R_p \; \text{deny} : \emptyset \; \text{not-applicable} : R \\setminus R_{tc} \; \text{indetP} : R_{tc} \setminus R_p \; \text{indetD} : \emptyset \; \text{indetDP} : \emptyset )
\]

and combine the two tuples by means of the policy set’s algorithm, i.e. permit-overrides. In the chosen policy-combining algorithm, the decision permit takes precedence, and each request for which the privacy consent policy returns a decision not-applicable or indeterminate gets a decision deny because of the deny-all policy; hence, the combined decision is still deny. Therefore the resulting tuple returned by the algorithm is:

\[
( \text{permit} : R_p \; \text{deny} : \emptyset \; \text{not-applicable} : \emptyset \; \text{indetP} : \emptyset \; \text{indetD} : \emptyset \; \text{indetDP} : \emptyset )
\]
where \( R_d \) stands for \((R \setminus R_p) \cup (R \setminus R_{tc}^c) \cup (R_{tc}^c \setminus R_p)\).

In forming the final authorization decision, we have of course to also consider the policy set’s obligations, whose evaluation is as follows.

- When the policy evaluates to permit, the function \( \{ \cdot \} \), once applied to the obligation with applicability effect permit, returns the evaluated obligation \([M \ \text{redirect}(\text{value}')]\), if \((\text{epsos}/\text{redirect-url}, \text{value}') \in r\), for some value \text{value}'. Otherwise, the obligation does not succeed.
- When the policy evaluates to deny, the function \( \{ \cdot \} \), once applied to the obligation with applicability effect deny, returns the evaluated obligation \([M \ \text{mail}(\text{value}, "Your medical record has been requested by epSOS")]\) if \((\text{resource}/\text{resource-id.email}, \text{value}) \in r\), for some value \text{value}. Otherwise, the obligation does not succeed.

By taking the obligations into account when determining the authorization decisions for the policy set, we have that:

- The set \( R_{Sp} \) of pairs \((r, \text{EO})\) for which the policy set evaluates to permit is

\[
\{(r, [M \ \text{log}(\text{value}, "34133-9")]) [M \ \text{redirect}(\text{value}')]) | (r, [M \ \text{log}(\text{value}, "34133-9")]) \in R_p, (\text{epsos}/\text{redirect-url}, \text{value}') \in r\}
\]

Basically, it includes all the requests evaluated as permit when combining the policies which also provide an url for the obligation redirect.

- The set \( R_{Sd} \) of pairs \((r, \text{EO})\) for which the policy set evaluates to deny is

\[
\{(r, [M \ \text{mail}(\text{value}, "Your medical record has been requested by epSOS")])] | r \in R_d, (\text{resource}/\text{resource-id.email}, \text{value}) \in r\}
\]

It includes all the requests evaluated as deny when combining the policies which also provide the patient’s mail address.

- The set of requests for which the policy set evaluates to not-applicable is empty, because of the empty target and of the policy deny-all that can assign a decision (i.e. deny) to all requests.

- Because in the policy set the privacy consent policy is paired with the deny-all policy, it cannot return an indeterminate value any longer. Anyway, the policy set can still return decisions indetP and indetD, and this happens when at least one obligation cannot be fulfilled. More precisely,

  - the set of requests for which the policy set evaluates to indetP is \( R_p \setminus R_{Sp} \);
  - the set of requests for which the policy set evaluates to indetD is \( R_d \setminus R_{Sd} \).

Therefore the final authorization decision \( AD \) for the privacy consent policy set, given the set of requests \( R \), is

\[
(\text{permit} : R_{Sp}, \text{deny} : R_{Sd}, \text{not-applicable} : \emptyset, \text{indetP} : R_p \setminus R_{Sp}, \text{indetD} : R_d \setminus R_{Sd}, \text{indetDP} : \emptyset)
\]
6.2 The enforcement process

We know from the formal semantics that the authorization decision $AD$ calculated by the PDP decision process is passed as an input to the function $[PEP]_{AD}$, which enforces the decision by following the chosen enforcement algorithm and returns the enforceable decision $ED$.

Now, in the authorization decision coming from the deny-all policy there are no obligations. Hence, whatever is the used enforcement algorithm, the PEP does not need to execute any additional action for these decisions.

Instead, in the authorization decision of the privacy consent policy, the requests evaluated as permit have attached the obligation log as an additional action. Similarly, in the authorization decision returned by the enclosing policy set, the requests evaluated as permit or deny have also attached obligation redirect or mail, respectively. All these obligations are evaluated by the PEP according to the chosen enforcement algorithm. For the considered policies, the most suitable enforcing algorithms are deny-biased and base, whose behaviour is described below.

- **deny-biased** grants access only if the decision is permit and all obligations are successfully discharged, otherwise it forbids access. Therefore, the requests evaluated as deny, indetP or indetD are all enforced as deny. Instead, the enforceable decision for the requests evaluated as permit is subjected to the result of discharging obligations. Now, let $(R_{ps}, R_{pu})$ be a partition of $R^{S}$ such that

  - $R_{ps}$ is the set of requests for which both the obligations log and redirect are successfully discharged;
  - $R_{pu}$ is the set of requests for which at least one obligation between log and redirect cannot be successfully discharged.

Hence, the enforceable decision returned by the PEP is:

$$(\text{permit} : R_{ps} \ \text{deny} : R^{S}_{d} \cup (R_{p} \setminus R^{S}_{d}) \cup (R_{d} \setminus R^{S}_{d}) \cup R_{pu} \not= \emptyset \ \text{indetP} : \emptyset \ \text{indetD} : \emptyset \ \text{indetDP} : \emptyset)$$

- **base** grants (resp. forbids) access only if the decision is permit (resp. deny) and all attached obligations are successfully discharged, otherwise it returns indetDP. The enforceable decision is then defined as follows:

  - the set of requests with decision permit is the set $R_{ps}$;
  - the set of requests with decision deny is the set $R_{ds} \subseteq R^{S}_{d}$ of requests for which the obligation mail is successfully discharged;
  - the set of requests with decision not-applicable is empty since the corresponding set in the PDP’s authorization decision is empty and the algorithm base never generates a decision not-applicable;
  - the set of requests with decision indeterminate contains the set of requests with decision indeterminate from the authorization decision and the set of requests evaluated as permit or deny for which at least one obligation cannot be successfully discharged. Formally, it is defined as

$$(R^{S}_{p} \setminus R_{ps}) \cup (R^{S}_{d} \setminus R_{ds}) \cup (R_{p} \setminus R^{S}_{p}) \cup (R_{d} \setminus R^{S}_{d})$$
that is
\[(R_p \setminus R_{ps}) \cup (R_d \setminus R_{ds})\].

Hence, the enforceable decision returned by the PEP is:

\[
\text{permit} : R_{ps} \quad \text{deny} : R_{ds} \quad \text{not-applicable} : \emptyset \\
\text{indetP} : \emptyset \quad \text{indetD} : \emptyset \quad \text{indetDP} : (R_p \setminus R_{ps}) \cup (R_d \setminus R_{ds})
\]

Notice that all the sets involved in the enforceable decisions do not contain any obligation.

7 FACPL Supporting Tools

In the previous section, we have seen how the FACPL language can be used to define policies that are intuitive and easy to read, write and understand. In order to promote the use of FACPL in real systems, we have also developed some software tools supporting policy development and a software architecture for policy evaluation and enforcement.

In Figure 3 we show the toolchain supporting the use of FACPL. The FACPL Integrated Development Environment (IDE) allows the policy designer to specify the system policies in FACPL. In addition to policies, the IDE permits also specifying user requests in order to test and validate the policies. The specification task is facilitated both by the high abstraction level of FACPL and by the graphical interface provided by our IDE. By exploiting some translation rules, written using the Xtend language (http://www.eclipse.org/xtend/), which provides facilities for defining code generators, the IDE generates the corresponding low-level policies both in Java and in XML. The latter format obeys the XACML 3.0 syntax and can be used to connect our toolchain to external XACML tools (as, e.g., the test cases generator X-CREATE [21]). The former format relies on a Java library specifically designed for compile- and run-time supporting FACPL code. Once these Java classes are compiled, they can be used by the enclosing main application (e.g., the implementation of our eHealth scenario) for evaluating many requests, simply by means of method invocation.

The FACPL’s supporting tools are developed by using Java-based technologies and are integrated into the Eclipse environment. Their source and binary files, as well as their documentation [22], can be found at the FACPL’s website http://rap.dsi.unifi.it/facpl/ In the rest of this section, we first overview the main features of our supporting tools and the technologies exploited to implement them, then we show our tools at work on the eHealth scenario.

7.1 The FACPL library

The FACPL code can be executed through a Java library that implements all the semantic tasks of the decision and the enforcement process described in Section 5. In order to achieve a flexible and extendible framework, the library
has been designed by exploiting the reflection features provided by Java and best-practice software engineering techniques. In fact, the framework can be easily extended to incorporate custom matching functions and combining algorithms defined by the user to deal with, e.g., new value types or specific decisions’ combinations different from those reported in Appendix A. We first overview the library design and then show how a FACPL policy is transformed into a Java class in order to be involved in the PDP’s evaluation process.

The Library Design. The FACPL library is composed of three different components: one for the specification of rules, policies and policy sets, one for the specification of access requests and decision responses, and one for the implementations of the matching functions, of the combining algorithms, and of the decision and enforcing processes. For each of them, there are specific Java interfaces defining the signatures of the methods that must be implemented.

Rules, policies and policy sets are the elements evaluated by the PDP during the decision process. To specify these elements we have designed a class hierarchy, graphically depicted in Figure 4, implementing the interface EvaluableElement, which defines methods for calculating target matching and authorisation decision. All the evaluable elements share some common features, i.e. an identifier, a target, and a set of obligation expressions. Therefore, these features are factorized into the abstract class PAFEvalElement, which is the root of the class hierarchy.

The elements from which the decision process starts are policies and policy sets. Therefore, policies and policy sets are implemented via a common class PAFElement. The class for rules, instead, is a branch of the hierarchy. The class PAFElement contains the combining algorithm, while the class Rule contains the effect and the condition. The sub-classes Policy and PolicySet of PAFElement contain, respectively, the enclosed list of rules and of policies and/or policy sets. Each leaf class of the hierarchy, i.e. Rule, Policy and PolicySet, provides the method evaluate to calculate the element’s DecisionResult value for the request.
Fig. 4. Policy, policy set and rule hierarchy

passed as argument. The result contains the decision (i.e. permit, deny, indetP, etc.) and, possibly, a set of obligations.

For an easy and automatic creation of the Java classes from the FACPL code, the classes described above provide specific setting methods for each element’s attribute, such as addTarget and addCondition. Notably, the language implementation permits using policy and policy set references, which are useful for modularising the code.

The signature of PDP’s methods is defined by the interface PDP shown in Figure 5. Specifically, it provides an entry method evalRequest for triggering the request evaluation, which takes as input a list of PAFElement objects. The class PAFElement has been indeed introduced in the class hierarchy in order to simplify the definition of the PDP implementation.
PDP’s combing algorithm and a request to evaluate. The method also requires a boolean argument indicating if the evaluation process has to use extended indeterminate values or not. Moreover, the interface defines the method combineDecision for combining multiple request decisions into one. The decision returned is, if exists, the common decision value, otherwise will be indetDP. Moreover, if there is any obligation attached to at least one decision of the set, the combined decision is indetDP. This is needed because an obligation is related to a specific decision, then in a common decision it could provide an erroneous behaviour. The FACPL library provides a standard implementation of PDP, faithful with the evaluation process formalised in Section 5.1 through the class PDP.Standard.

The decision process is coordinated by the combining algorithm’s implementation class passed as argument to the method evalRequest provided by PDP. The list of PAFEElements and the access request to evaluate are then passed as arguments to the combining algorithm, which starts the decision process following its specific implementation. At the first call of the process, each element of the list is initialised and, hence, subsequent invocations of the decision process over the same policies do not require further initialisations. The outcome result of the PDP is an authorisation decision, which is passed to the PEP implementation class for enforcing it. Internally, the decision process supports the extended indeterminate values, but the final authorisation decision only contains the four authorisation values, i.e. the three indeterminate values are merged under the more general indeterminate decision. This avoids providing additional information about error causes to a malicious attacker that could exploit it for violating the access control system; the details about extended indeterminate decisions are however reported in a log file.

The functionalities supporting the enforcement process are defined by the PEP interface shown in Figure 5. This interface defines the method doEnforcement to enforce a PDP decision by discharging the additional actions possibly associated to the decision. The interface also specifies the method initializePepAction for loading enforcement actions from external classes at run-time. This permits adding new enforcement actions to the PEP definition without affecting the code of the class implementing it. The FACPL library provides

---

**Fig. 5.** PDP and PEP interfaces
a standard implementation of this interface, namely class \texttt{PEP\_Standard}, which follows the formalisation presented in Section 5.2. Specifically, it supports the algorithms \texttt{base}, \texttt{deny-biased} and \texttt{permit-biased}, and provides two basic enforcement actions: sending of emails and insertion of entries into log files. Anyway, as mentioned above, it is possible to define new enforcement actions, by exploiting the class \texttt{PEPAction}, and to add them to the current PEP implementation by means of method \texttt{initializePepAction}. In particular, for each additional action to add, we have to report in the \texttt{PEPAction} class the name and the reference to the class that implements it. This latter class is auto-generated by a Java code generator.

Access requests and responses of PDP and PEP are designed according to the syntax reported in Table 13. Moreover, each request is linked to its \textit{context} through which it retrieves values from sources outside the request, i.e. we have an \textit{explicit} context. In our eHealth scenario, for example, the context is used to retrieve the redirecting URL requested by the evaluation of the obligation within the policy set. This feature can also be used to integrate our library with other services.

Both the decision and the enforcement processes are well-documented by log information, available in the standard output or in a dedicated file. The logging tool exploited by our library is the Simple Logging Facade for Java (SLF4J\footnote{http://www.slf4j.org/}) that permits to abstract from a single way of logging and supports multiple well-known logging libraries, e.g. Apache Log4j\footnote{http://logging.apache.org/log4j/1.2/} or CommonsLogging\footnote{http://commons.apache.org/proper/commons-logging/}.

The main advantage of our Java-based implementation, with respect to the usual XML-based ones, is to avoid parsing and surfing the XML tree structure of policies, which require an additional computational load.

The \textit{eHealth} Java Policy. From the description above, it is worth noticing that each FACPL element in the library corresponds to an abstract class, which provides a method for evaluating requests with respect to such element. Therefore, a FACPL policy is rendered as a Java class that extends the corresponding abstract class \texttt{Policy}. Policy elements, i.e. combining algorithm, target, rules and obligations, are added to a \texttt{Policy} object at creation time by the class constructor, by means of specific methods, e.g. \texttt{addTarget}. Policy sets are translated similarly. For example, an excerpt of the Java code corresponding to the \textit{Patient Summary} policy is reported below.

\begin{verbatim}
public class Policy_PatientSummary extends Policy{
    public Policy_PatientSummary(){
        addId("PatientSummary");
        //Algorithm Combining
        addCombiningAlg(it.unifi.facpl.lib.algorithm.PermitOverrides.class);
        //Target
        addTarget(new TargetTree(Connector.AND,

\end{verbatim}
new TargetTree(
    new TargetExpression(it.unifi.facpl.lib.function.Equal.class,
        "TREATMENT", new StructName("subject.access","purposeofuse")),
    new TargetTree(new TargetExpression(it.unifi.facpl.lib.function.
        Equal.class, "medical doctor", 
        new StructName("subject","role"))),
    new TargetTree(new TargetExpression(it.unifi.facpl.lib.function. 
        Equal.class, "34133-9", 
        new StructName("resource","resource-id"))));
// Rule
addRule(new Rule1());
// Obligation
addObligation(new ObligationExpression("log", Effect.PERMIT, TypeObl.M,
    new StructName("subject","doctor-id"),
    new StructName("resource","resource-id")));

private class Rule1 extends Rule{
    Rule1 (){ 
    addId("rule1");
    // Effect
    addEffect(Effect.PERMIT);
    // Target
    addTarget(...);
    // Condition
    addConditionExpression(new ConditionExpression(it.unifi.
        facpl.lib.function.StringSubset.class,
        new Bag("PRD-003","PRD-005","PRD-010","PRD-016"),
        new StructName("subject","hl7.permission")));
    }
}

The policy evaluation is coordinated by the class implementing the combing algorithm (i.e. PermitOverrides.class). The expression corresponding to the policy target is structured as nested expressions organised according to the structure of the original FACPL target (possibly defined by brackets). Since rules are only used inside their enclosing policy, for each of them the policy class contains an inner class (in the code above, the enclosed rule rule1 is implemented by the inner class Rule1). As for policies, both rule and policy set classes have to extend the corresponding abstract class from the library.

For each FACPL request, instead, it is generated a class containing a list of attributes and a reference to a stub class corresponding to the context handler. By implementing this stub, it is possible to retrieve external information needed for evaluating requests. This can be exploited in those scenarios where the policy decision depends on the usage of system resources, e.g. condition expressions are used to check the amount of available resources in the system.

When all policies and requests have been translated into Java classes, it is possible to start the evaluation process by invoking the method evalRequest of the PDP.Standard class for calculating the authorization decision, and then the method doEnforcement of PEP.Standard for enforcing this decision. This workflow is provided by the FACPL IDE through the main method of the generated PEP class.
7.2 The FACPL development environment

Designing an access control system is a task of the security engineers, that usually use graphical formalizations of the system life-cycle. Starting from an UML formalisation or other graphics models, an engineer needs a way to implement and tests his specification. The FACPL language is a tool that can be used for designing and verifying access control systems, however writing directly in XML or similar style can be an hard and error-prone task for a designer. Therefore, we have implemented a specific IDE with a set of facilities, such as a graphic interface and an automatic generator of Java and XML code. Before describing the main features of the IDE, we briefly present Xtext [20], a framework that we have exploited to develop the IDE.

The Xtext Framework. Xtext provides a developing framework to design and implement domain specific languages. Starting from an EBNF formalisation of the language syntax, such as the one for FACPL reported in Table 10, we can obtain an attribute grammar that defines the structure of the abstract syntax tree (AST) of the language. The AST is created by the ANTLR [23] parser and is merged into an Eclipse plugin by a specific workflow called MWE. Then, the AST is specified as an Ecore Model [24], which is translated into a set of Java classes. Using these classes we can define syntax and semantic checks and a code generator for the language. Specifically, the generator can be easily implemented with the Xtend language [25], which defines a specific generation rule for each syntactic category of the FACPL grammar. All these features are available as an Eclipse plugin that contains a multi-page editor for the language and the generation features developed with Xtend. The plugin is created as an Eclipse RCP project, allowing additional customisations as any other plugin development project. However, the parser and the generator provided by Xtext and Xtend can be also called from other applications external to the deployed plugin.

The Features of the IDE. The IDE is developed as an Eclipse plugin using the Xtext framework previously presented. The plugin is available on-line at address [http://rap.dsi.unifi.it/facpl/](http://rap.dsi.unifi.it/facpl/) and can be installed by resorting to the standard procedure for installing new software into Eclipse. For the installation is required Java 1.6 or higher and the Xtext dependencies reachable from the standard Eclipse repository site.

The IDE provides a multi-page editor where the code writing activity is supported by syntactical controls, auto-completion and code suggestion. Besides these features, the IDE implements static checks for expression typing and for achieving uniqueness of identifiers. Moreover, for facilitating code organisation, it is possible to split the code into multiple files.

The policy editor is organised as shown in the screenshot in Figure 6: the view on the left shows the projects’ structure which permits accessing to FACPL files, the multi-page editor in the center highlights the language keywords with different colours, and the view on the right shows the navigational outline. Toolbar and pop-up menus provide commands for generating Java and XACML code.
Fig. 6. The FACPL IDE

A new FACPL project can be created from the available wizard menu Facpl Project. The wizard creates a customised Java project where the evaluation library and the Xtext features are already included. Then, for adding a new FACPL file to the project, which will have extension \textit{.fpl}, it can be used the FACPL file wizard, which also provides a basic template for policies and requests. A FACPL file is composed of a set of access policies and requests, following the syntax of Table 10.

To facilitate the process of evaluating new policies we introduce policy references, therefore a policy set can also be created by referring to policies defined in external files. The language supported by the IDE permits also specifying some project parameters for: (i) choosing if the decisions corresponding to multiple requests should be combined into a single decision, (ii) choosing if extended indeterminate values should be used in the decision process, (iii) setting the destination package name for the Java classes, and (iv) setting the names of the requests to evaluate.

From FACPL code, the corresponding executable Java code can be generated by using the command \textit{Generate FACPL code} available in the menus. The produced Java classes are placed into the package specified as parameter, if present, or in the default package, otherwise. To execute the Java code, one has to run the class implementing the PEP interface. For running the classes outside the FACPL project, i.e. without all the dependencies to evaluation and logging libraries, we refer the interested reader to [22].

By means of menu commands it is also possible to generate a FACPL policy or request starting from the corresponding XACML 3.0 code and vice versa. The translation from FACPL code to XACML is straightforward for most of the code elements, except for targets’ expressions that are reorganised into the three level
syntax of XACML. The translation from XACML to FACPL is straightforward as well, except for some XACML elements, i.e. attribute selector, variable declaration and combining parameters, that are not directly supported by FACPL. However, these features do not significantly affect the expressiveness of the language and the evaluation of policies.

The FACPL features can be also tested without installing the Eclipse plugin, by means of a web application available at [http://rap.dsi.unifi.it/facpl/](http://rap.dsi.unifi.it/facpl/). The web interface permits to easily create a set of FACPL policies and a set of access requests, and then to obtain the decision computed by the remote engine.

### 7.3 IaaS case study implementation

The IaaS scenario described in Section 3.2 is implemented as a web application providing both a front-end for the administrator, from where he can manage the policies governing the hypervisors, and a front-end for the clients, from where they can submit requests for the creation or the shutdown of VMs.

The server-side implementation is a Tomcat server that, by integrating FACPL and Xtext libraries, is able to accept FACPL polices, to parse and compile these policies, and finally to enforce the corresponding decisions for adapting hypervisors' state to client requests.

The administrator panel is shown in Figure 7. The policies chosen by the administrator can be either those previously outlined or other ones written by using the on-line editor. For analysing the current state of the Cloud system, a graphical representation of the load of each hypervisor is provided.

When the administrator submits a new FACPL policy, the Cloud provider translates it into Java classes by applying the Xtext parsing rules. Then, if all classes are successfully created, the provider compiles the classes by relying on
If no error occurs during any stage of this workflow, these policies became the new policies in force in the platform; otherwise, the process stops and no system update is made, hence the system remains into a safe state.

To submit a request, a client must provide, by choosing them on the front-end panel, shown in Figure 8, the information needed for the wanted request: the profile identifier, i.e. $P_1$ or $P_2$, and the VM type, for creating a VM, and the profile identifier and the VM’s id, for shutting down a VM. While profile $P_1$ only permits to create $TYPe_1$ VMs, $P_2$ permits to create both types of VMs. For the sake of simplicity, profile identifier and VM type are the only authentication information we use to determine if client requests should be granted or denied. Of course, in a real application, access control should rely on more trustful authentication information, which could be provided, e.g., through such a mechanism as SAML [7].

When the system load is high and it is needed to freeze some VMs, the owners of such VMs get some credits as reward. On the server side, the frozen VMs are added to a queue, which is periodically checked with the aim of trying to reactivate suspended VMs, according to the incoming order in the queue. This event is reported to the client through the front-end panel, where the possibly received credits and the status of the client’s VMs are shown.

### 8 Related Work

The work closest to ours is [26], where a preliminary version of FACPL is introduced to formalize the semantics of XACML. However, the language considered here is more expressive as it is equipped with features such as, e.g., obligations. Most of all, the contribution of the two works are different: in [26] it is the formalization of XACML, here it is the presentation of FACPL, the development of related supporting tools and the illustration of their effectiveness by means of a significative case study from the eHealth domain. The preliminary version of FACPL is used in [27] to formalize UML-based models of access control policies for Web applications, while [28] sketches an early description of the development methodology for FACPL policies.

Recently, many policy-based languages and models have been developed for managing different aspects of programs’ behaviour as, e.g., adaptation and autonomic computing. For example, [29] introduces PobSAM, a policy-based formalism that combines an actor-based model, for specifying the computational aspects of system elements, and a configuration algebra, for expressing autonomic managers that, in response to changes, lead the adaptation of the sys-
tem configuration according to given adaptation policies. This formalism relies on a predefined notion of policies expressed as Event-Condition-Action (ECA) rules. Adaptation policies are specific ECA rules that change the manager configurations. Our notion of policies, being defined for a broader application, is instead more flexible and expressive. To approach autonomic computing issues, IBM has developed a simplified policy language, named Autonomic Computing Policy Language (ACPL) [30]. Such language, however, comes without any precise syntax and semantics. Another policy language is Ponder [31], for which a number of toolkits have been developed and applied to various autonomous and pervasive systems. The language borrows the idea introduced in [32] of using two separate types of policies for authorisation and obligation. Policies of the former type have the aim of establishing if an operation can be performed, while those of the latter type basically are ECA rules. Differently from Ponder, and similarly to more recent languages (e.g., XACML), obligations in FACPL are expressed as part of authorization policies, thus providing a more uniform specification approach.

As a result of the widespread use of policy-based languages in service-oriented systems and international projects, many attempts of formalisation have been made. A largely followed approach is based on 'transformational' semantics that translates from a policy-based language, e.g. XACML, into terms of some formalism. For example, [33] uses description logic expressions as target formalism, [34] exploits the process algebra CSP [35], and [36] the model-oriented specification language VDM++ [37]. The main advantage of these approaches is the possibility of analysing policies by means of off-the-shelf reasoning tools that may be already available for the target formalisms. From the semantics point of view, this approach provides some alternative high-level representations of policies, which have their own semantics, but not all of them are suitable to drive implementations. Moreover, our semantics can be conveniently exploited by software engineers to drive XACML implementations. At the same time, its mathematical foundations enable the development of reasoning tools (as we briefly discuss in Section 9).

There are also approaches using XACML as a target language. For example, in [38], the policies are first specified by means of the description language RW [39], then analysed through a model checking technique, and finally translated in XACML. Advantages and disadvantages with respect to our approach are as before.

Other formalisation approaches, more similar to ours, define the semantics of XACML policies in a more direct way, i.e. without introducing intermediate formalisms. For example, [40] proposes a semantics based on (multi-terminal) binary decision diagrams, which permits efficiently carrying out the proposed analysis techniques (i.e. property verification and change-impact analysis), but are not suitable as an implementation guide. Instead, [41] 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 with respect to 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.

Concerning policy evaluation tools, there are by now many un-official implementations of policy-based standards, especially of XACML. In software production, the most used tools are SUN XACML [42] and HERASAP [43], which manage XACML policies written in XML. To evaluate a request, they parse the XML policies and then visit parts of the generated DOM trees for calculating the authorisation decision. This differs from our enforcement tool, where we evaluate (sets of) requests by executing Java classes implementing the semantic representations of policies, written in the more intuitive syntax of FACPL.

Another notable implementation of XACML is XEngine [44], but it aims at highly efficient request processing, achieved by converting XACML policies into numerical representations. This differs from our main goal that is the development of a policy language driven by a formal semantics. Another implementation of an access control mechanism is PERMIS [45], a modular infrastructure specifically devised for Grid systems and integrated in modern toolkits (like, e.g., [46,47]). However, PERMIS relies on an ad-hoc, non-standard policy language which is less expressive than XACML [48] and, hence, than FACPL.

A tool closer to our approach is the ALFA Eclipse plugin [49] developed by Axiomatics. It provides a domain specific language to represent and automatically build XACML policies. The Eclipse environment provides code completion and reuse of identifiers. However, differently from our IDE, this plugin does not freely provide a request evaluation feature, since the Axiomatics’s evaluation engine is a proprietary software.

To sum up, differently from related works, our formalisation and software tools have a twofold aim: it serves as a guide for policy developers and, at the same time, paves the way for the design and implementation of analysis tools.

9 Concluding remarks

We have described a user-friendly, uniform, and comprehensive approach to the development and enforcement of access control policies which is based on FACPL and its software tools. The policy language FACPL has a compact and intuitive syntax and is endowed with a formal semantics, which lays the basis for developing tools and methodologies for analysing policies. Moreover, FACPL is equipped with easy-to-use software tools, i.e. a powerful IDE and a Java library, supporting policy designers and system administrators in the policy development and enforcement tasks. We have illustrated potentialities and effectiveness of our approach through a significative case study from the eHealth domain.

As a future work, we plan to continue the validation of FACPL and its tools, on the one hand, by applying them to even more realistic case studies from different domains (e.g., smart grids) and, on the other hand, by experimentally evaluating the performance of our enforcement tool. We will also extend our IDE with new features as, e.g., a dedicated view to summarize and analyse policy evaluation results.
Another research line we intend to pursue is the development of methods and techniques for analysing FACPL policies. In particular, they will be first theoretically defined and, then, integrated in our software tools in order to achieve a complete framework for developing trustworthy 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 authorization decisions coincide or, simply, have the same \textit{permit} set (indeed, sometimes it does not matter the reason why the access is not permitted, as e.g. with a \textit{deny-biased} PEP \cite[Section 7.1.2]{NIST2009} that grants the access if the decision taken by the PDP is \textit{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 \textit{least-privilege} concept \cite{OASIS2012}, 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 \textit{weights} (indicating the access privilege level \cite{Bradner1997}) are associated to request data and are used to identify, within the \textit{permit} set of the authorization decision associated to the considered policy, the requests with minimum total weight. We will also exploit our semantics as a basis for studying \textit{separation of duty} aspects of XACML policies.

References

3. OASIS XACML TC: eXtensible Access Control Markup Language (XACML) version 3.0 - Candidate OASIS Standard (September 2012)
4. The epSOS project: A european ehealth project \url{http://www.epsos.eu}
7. OASIS Security Services TC: Assertions and protocols for the OASIS security assertion markup language (SAML) v2.02 (2005)
10. Bradner, S.: Key words for use in rfc\$ to indicate requirement levels (1997)

\footnote{For example, the privilege level of “head physician” would be higher than the level of “nurse”, which would be higher than that of “anonymous”.}


15. The Regenstrief Institute: Logical observation identifiers names and codes (LOINC) [http://www.loinc.org].


43. The Herasaf consortium: HERAS AF http://www.herasaf.org
45. ISSRG: The Modular PERMIS Project http://sec.cs.kent.ac.uk/permis/
A Combining algorithms

We report in this appendix the definitions of the combining algorithms not described in the paper. Specifically, we present only one code for both rule- and policy-combining algorithms; therefore, from time to time, Sequence stands for a sequence of rules, policies or policy sets, and elem for one of its elements. Notice that the instruction ‘foreach (elem ∈ Sequence)’ traverses the elements within Sequence in the order in which they are listed in the policy or policy set.

Listing 18. The combining algorithm deny-overrides

```
1  Boolean atLeastOneErrorD = false;
2  Boolean atLeastOneErrorP = false;
3  Boolean atLeastOneErrorDP = false;
4  Boolean atLeastOnePermit = false;
5  foreach (elem ∈ Sequence) {
6      AuthDecision t = [ elem ](r);
7      if (r ∈ t↓deny) return deny;
8      if (r ∈ t↓permit) {
9          atLeastOnePermit = true;
10         continue;
11      }
12      if (r ∈ t↓not-applicable) continue;
13      if (r ∈ t↓indetD) {
14          atLeastOneErrorD = true;
15          continue;
16      }
17      if (r ∈ t↓indetP) {
18          atLeastOneErrorP = true;
19          continue;
20      }
21      if (r ∈ t↓indetDP) {
22          atLeastOneErrorDP = true;
23          continue;
24      }
25  }
26  if (atLeastOneErrorDP) return indetDP;
27  if (atLeastOneErrorP & & (atLeastOneErrorD || atLeastOnePermit)) return indetDP;
28  if (atLeastOneErrorD) return indetD;
29  if (atLeastOnePermit) return permit;
30  if (atLeastOneErrorP) return indetP;
31  return not-applicable;
```

Listing 19. Combining algorithm deny-unless-permit

```
1  foreach (elem ∈ Sequence) {
2      AuthDecision t = [ elem ](r);
3      if (r ∈ t↓permit) return permit;
4  }
```
\textbf{Listing 20. Combining algorithm permit-unless-deny}

\begin{verbatim}
foreach (elem ∈ Sequence) {
    AuthDecision t = [ elem ](r);
    if (r ∈ t↓deny) return deny;
}
return permit;
\end{verbatim}

\textbf{Listing 21. Combining algorithm first-applicable}

\begin{verbatim}
foreach (elem ∈ Sequence) {
    AuthDecision t = [ elem ](r);
    if (r ∈ t↓deny) return deny;
    if (r ∈ t↓permit) return permit;
    if (r ∈ t↓not-applicable) continue;
    if (r ∈ t↓indetP) return indetP;
    if (r ∈ t↓indetD) return indetD;
    if (r ∈ t↓indetDP) return indetDP;
} return not-applicable;
\end{verbatim}

\textbf{Listing 22. Policy-Combining algorithm only-one-applicable}

\begin{verbatim}
Boolean atLeastOne = false;
Policy selectedPolicy = null;
MatchingTuple matchTuple;
foreach (policy ∈ Policies) {
    matchTuple = [ policy . target ](r);
    if (r ∈ matchTuple↓indeterminate) return indeterminate;
    if (r ∈ matchTuple↓match) {
        if (atLeastOne) {
            return indeterminate;
        } else {
            atLeastOne = true;
            selectedPolicy = policy;
        }
    } else {
        if (r ∈ matchTuple↓no-match) continue;
    }
    if (atLeastOne) {
        return [ selectedPolicy ](r);
    } else {
        return not-applicable;
    }
}
\end{verbatim}
A.1 A possible custom algorithm

We report here a custom variant of the permit-overrides algorithm that combines the authorization decisions of rules evaluation in a compositional way.

\[
\text{permit-overrides-custom}(Rule^+)_R = \\
(\text{permit} : \bigcup_j R^j_p ; \\
\text{deny} : \bigcap_j R^j_d ; \\
\text{not-applicable} : \bigcap_j R^j_n ; \\
\text{indetP} : \bigcup_j R^j_{ip} \setminus \bigcup_j R^j_p ; \\
\text{indetD} : \bigcup_j R^j_{id} \setminus \bigcup_j R^j_d ; \\
\text{indetDP} : \ldots \simile \text{a indetP} \ldots )
\]

where, for all rule\(_j\) in Rule\(_+\), we have

\[
[\text{rule}_j]_R = (\text{permit} : R^j_p ; \text{deny} : R^j_d ; \text{not-applicable} : R^j_n ; \\
\text{indetP} : R^j_{ip} ; \text{indetD} : R^j_{id} ; \text{indetDP} : R^j_{idp})
\]

B Cloud Case Study Policies

Listing 23. Energy saver policy

```java
{ pep:
  deny-biased
  permit-overrides
  { permit-overrides
    target:
      equal("CREATE", action/action-id)
    policies:
      < deny-unless-permit
        target:
          (equal("P_1", subject/profile-id)) || \
          (equal("P_2", subject/profile-id)) \
          & & \
          equal("TYPE_1", resource/vm-type)
        rules:
          (permit
            target:
              less-than-or-equal(1, system/hyper1.availableResources) \
              obl:
                [permit M create("HYPER_1", system/vm-id, "TYPE_1")]
          ) \
          (permit
            target:
              less-than-or-equal(1, system/hyper2.availableResources) \
              obl:
                [permit M create("HYPER_2", system/vm-id, "TYPE_1")]
          ) \
          obl:
            [deny O warning("Not enough available resources for TYPE_1 VMs")]
      >
      < deny-unless-permit
        target:
          equal("P_2", subject/profile-id) & & \
          equal("TYPE_2", resource/vm-type)
        rules:
  }
```
Listing 24. Load balancer policy

```plaintext
{ permit
    target:
        less-than-or-equal(2, system/hyper1.availableResources)
    obl: [permit M create("HYPER_1", system/vm-id, "TYPE_2")]
}
{ permit
    target:
        less-than-or-equal(2, system/hyper2.availableResources)
    obl: [permit M create("HYPER_2", system/vm-id, "TYPE_2")]
}
{ permit
    condition:
        or( and (equal(0, system/hyper1.availableResources), less-than-or-equal(2, system/hyper1.vm1-counter)),
            and (equal(1, system/hyper1.availableResources), less-than-or-equal(1, system/hyper1.vm1-counter)))
    obl: [permit M freeze("HYPER_1", subtract(2, system/hyper1.availableResources), "TYPE_1")]
        [permit M create("HYPER_1", system/vm-id, "TYPE_2")]
}
{ permit
    condition:
        or( and (equal(0, system/hyper2.availableResources), less-than-or-equal(2, system/hyper2.vm1-counter)),
            and (equal(1, system/hyper2.availableResources), less-than-or-equal(1, system/hyper2.vm1-counter)))
    obl: [permit M freeze("HYPER_2", subtract(2, system/hyper2.availableResources), "TYPE_1")]
        [permit M create("HYPER_2", system/vm-id, "TYPE_2")]
}
obl: [deny O warning("Not enough available resources for TYPE_2 VMs")]
>
{ permit-overrides
    target: (equal("CREATE", action/action-id) & &
        (equal("P_1", subject/profile-id)) | equal("P_2", subject/profile-id))
    rules:
        { permit
            condition: at-least-one-member-of(resource/vm-id, system/hyper1.vm-ids)
            obl: [permit M release("HYPER_1", resource/vm-id)]
        }
        { permit
            condition: at-least-one-member-of(resource/vm-id, system/hyper2.vm-ids)
            obl: [permit M release("HYPER_2", resource/vm-id)]
        }
>
{ pep:
    deny-biased
pdp:
    permit-overrides
        { permit-overrides
            target:
                equal("CREATE", action/action-id)
            policies:
                < deny-unless-permit
```
target:
(equal("P_1", subject/profile-id))
| equal("P_2", subject/profile-id))
&
& equal("TYPE_1", resource/vm-type)

rules:

{permit
  target:
  less-than-or-equal(1, system/hyper1.availableResources)
  condition:
  less-than-or-equal(system/hyper2.availableResources, system/hyper1.availableResources)
  obl:
  [permit M create(“HYPER_1”, system/vm-id, “TYPE_1”)]
}

{permit
  target:
  less-than-or-equal(1, system/hyper2.availableResources)
  condition:
  less-than-or-equal(system/hyper1.availableResources, system/hyper2.availableResources)
  obl:
  [permit M create(“HYPER_2”, system/vm-id, “TYPE_1”)]
}

obl:
[deny O warning(“Not enough available resources for TYPE_1 VMs”)]

< deny-unless-permit
target:
equal("P_2", subject/profile-id) &
equal("TYPE_2", resource/vm-type)

rules:

{permit
  target:
  less-than-or-equal(2, system/hyper1.availableResources)
  condition:
  less-than-or-equal(system/hyper2.availableResources, system/hyper1.availableResources)
  obl:
  [permit M create(“HYPER_1”, system/vm-id, “TYPE_2”)]
}

{permit
  target:
  less-than-or-equal(2, system/hyper2.availableResources)
  condition:
  less-than-or-equal(system/hyper1.availableResources, system/hyper2.availableResources)
  obl:
  [permit M create(“HYPER_2”, system/vm-id, “TYPE_2”)]
}

{permit
  condition:
  or(and(equal(0, system/hyper1.availableResources),
         less-than-or-equal(2, system/hyper1.vm1-counter)),
  and(equal(1, system/hyper1.availableResources),
      less-than-or-equal(1, system/hyper1.vm1-counter)))
  obl:
  [permit M freeze("HYPER_1", subtract(2, system/hyper1.availableResources), “TYPE_1”)]
  [permit M create("HYPER_1", system/vm-id, “TYPE_1”)]
}

{permit
  condition:
  or(and(equal(0, system/hyper2.availableResources),
         less-than-or-equal(2, system/hyper2.vm1-counter)),
  and(equal(1, system/hyper2.availableResources),
      less-than-or-equal(1, system/hyper2.vm1-counter)))
  obl:
>> <permit overrides>
  target:
    equal("RELEASE", action/action-id) & &
    (equal("P_1", subject/profile-id)||equal("P_2", subject/profile-id))
  rules:
    (permit
      condition: at-least-one-member-of(resource/vm-id, system/hyper1.vm-ids)
      obl: [permit M release("HYPER_1", resource/vm-id)])
    (permit
      condition: at-least-one-member-of(resource/vm-id, system/hyper2.vm-ids)
      obl: [permit M release("HYPER_2", resource/vm-id)])
  >
</permit>

Listing 25. Load balancer creation policy

```xml
<PolicySet xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17
  http://docs.oasis-open.org/xacml/3.0/xacml-core-v3-schema-wd-17.xsd"
  PolicySetId="Create_Policies" Version="1.0"
  PolicyCombiningAlgId="urn:oasis:names:tc:xacml:3.0:policy-combining-algorithm:permit-overrides"
  MaxDelegationDepth="1">
  <Target>
    <AnyOf>
      <AllOf>
        <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
            CREATE
          </AttributeValue>
          <AttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#anyURI"
            MustBePresent="false"
            Category="urn:oasis:names:tc:xacml:1.0:attribute-category:action"
            AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id" />
        </Match>
      </AllOf>
    </AnyOf>
  </Target>
  <Policy PolicyId="SLA_Type1" Version="1.0"
    RuleCombiningAlgId="urn:oasis:names:tc:xacml:3.0:rule-combining-algorithm:deny-unless-permit">
    <Target>
      <AnyOf>
        <AllOf>
          <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
              P_1
            </AttributeValue>
          </Match>
        </AllOf>
      </AnyOf>
    </Target>
  </Policy>
</PolicySet>
```
<Condition>
  <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-less-than-or-equal">
    <AttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#anyURI">
      <AttributeId>urn:oasis:names:tc:xacml:3.0:attribute-category:system:hyper2_availableResources</AttributeId>
    </AttributeDesignator>
    <AttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#anyURI">
      <AttributeId>urn:oasis:names:tc:xacml:3.0:attribute-category:system:hyper1_availableResources</AttributeId>
    </AttributeDesignator>
  </Apply>
</Condition>

<ObligationExpressions>
  <ObligationExpression ObligationId="create">
    <Rule RuleId="hyper2" Effect="Permit">
      <Target>
        <AnyOf>
          <Match>
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#integer">
              <AttributeValue>
                1
              </AttributeValue>
            </Match>
            <AttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#anyURI">
              <AttributeId>urn:oasis:names:tc:xacml:3.0:attribute-category:system:vm-id</AttributeId>
            </AttributeDesignator>
          </AnyOf>
        </Target>
      </Rule>
    </ObligationExpression>
  </ObligationExpression>
</ObligationExpressions>
A formal sw. eng. approach to Policy-Based Access Control

<Match>
</Match>

<AllOf>
</AllOf>

<AnyOf>
</AnyOf>

<Condition>
<Apply>
  <FunctionId>urn:oasis:names:tc:xacml:1.0:function:integer
  less-than-or-equal</FunctionId>
  <AttributeValue Data-Type="http://www.w3.org/2001/XMLSchema#anyURI"
  MustBePresent="false"
  Category="urn:oasis:names:tc:xacml:3.0:attribute-category:system"
  AttributeId="urn:oasis:names:tc:xacml:3.0:system:hyper1.availableResources"/>
  <AttributeValue Data-Type="http://www.w3.org/2001/XMLSchema#anyURI"
  MustBePresent="false"
  Category="urn:oasis:names:tc:xacml:3.0:attribute-category:system"
  AttributeId="urn:oasis:names:tc:xacml:3.0:system:hyper2.availableResources"/>
</Apply>
</Condition>

<ObligationExpressions>
  <ObligationExpression ObligationId="create"
    FulfillOn="Permit">
    <AttributeAssignmentExpression
      AttributeId="urn:oasis:names:tc:xacml:3.0:argument">
      <AttributeValue Data-Type="http://www.w3.org/2001/XMLSchema#string">HYPER2</AttributeValue>
    </AttributeAssignmentExpression>
  </ObligationExpression>
</ObligationExpressions>

</Rule>

<AdviceExpressions>
  <AdviceExpression AdviceId="warning" AppliesTo="Deny"
    AttributeAssignmentExpression
      AttributeId="urn:oasis:names:tc:xacml:3.0:argument">
      <AttributeValue Data-Type="http://www.w3.org/2001/XMLSchema#string">Not enough available resources for TYPE1 VMs</AttributeValue>
  </AttributeAssignmentExpression>
  <AdviceExpression>
    <AdviceAssignments>
      <AttributeValue Data-Type="http://www.w3.org/2001/XMLSchema#string">Not enough available resources for TYPE1 VMs</AttributeValue>
    </AdviceAssignments>
  </AdviceExpression>
</AdviceExpressions>

</AdviceExpressions>

</Policy>

<Policy PolicyId="SLA_Type2" Version="1.0"/>


```
RuleCombiningAlgId="urn:oasis:names:tc:xacml:3.0:rule-combining-algorithm:deny-unless-permit">
  <Target>
    <AnyOf>
      <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
          P_2
        </AttributeValue>
        <AttributeDesignator DataUri="http://www.w3.org/2001/XMLSchemaAnyURI"
          MustBePresent="false"
          Category="urn:oasis:names:tc:xacml:3.0:attribute-category:subject"
          AttributeId="urn:oasis:names:tc:xacml:1.0:subject:profile-id"/>
      </Match>
    </AnyOf>
    <AnyOf>
      <AllOf>
        <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
            TYPE_2
          </AttributeValue>
          <AttributeDesignator DataUri="http://www.w3.org/2001/XMLSchemaAnyURI"
            MustBePresent="false"
            Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
            AttributeId="urn:oasis:names:tc:xacml:1.0:resource:vm-type"/>
        </Match>
      </AllOf>
    </AnyOf>
  </Target>
  <Rule RuleId="hyper_1_create" Effect="Permit">
    <Target>
      <AnyOf>
        <Match>
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#integer">
            2
          </AttributeValue>
          <AttributeDesignator DataUri="http://www.w3.org/2001/XMLSchemaAnyURI"
            MustBePresent="false"
            Category="urn:oasis:names:tc:xacml:3.0:attribute-category:system"
            AttributeId="urn:oasis:names:tc:xacml:3.0:system:hyper1.availableResources"/>
        </Match>
      </AnyOf>
    </Target>
    <Condition>
      <Apply
        FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-less-than-or-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#integer">
          2
        </AttributeValue>
      </Apply>
    </Condition>
  </Rule>
</Target>
```

A formal sw. eng. approach to Policy-Based Access Control 87

```xml
<Model>
  <Apply>
    <Condition>
      <ObligationExpressions>
        <ObligationExpression ObligationId="create" FulfillOn="Permit">
          <AttributeAssignmentExpression AttributeId="urn:oasis:names:tc:xacml:3.0:attribute-category:system:hyper2:availableResources" />
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#anyURI" MustBePresent="false">
          </AttributeValue>
        </AttributeAssignmentExpression>
        <AttributeAssignmentExpression AttributeId="urn:oasis:names:tc:xacml:3.0:attribute-category:system:hyper1:availableResources" />
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#anyURI" MustBePresent="false">
        </AttributeValue>
      </ObligationExpressions>
      <Rule RuleId="hyper2.create" Effect="Permit">
        <Target>
          <AnyOf>
            <AllOf>
              <Match>
                <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#integer" MustBePresent="false">
                </AttributeValue>
              </Match>
            </AllOf>
          </AnyOf>
        </Target>
      </Rule>
    </Condition>
  </Apply>
</Model>
```
FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-less-than-or-equal">
  <AttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#anyURI"
    MustBePresent="false"
    Category="urn:oasis:names:tc:xacml:3.0:attribute-category:system"
    AttributeId="urn:oasis:names:tc:xacml:3.0:system:hyperl.availableResources" />
  <AttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#anyURI"
    MustBePresent="false"
    Category="urn:oasis:names:tc:xacml:3.0:attribute-category:system"
    AttributeId="urn:oasis:names:tc:xacml:3.0:system:hyper2.availableResources" />
</Apply>
</Condition>
<AttributeAssignmentExpression>
  <AttributeValue DataTypes="http://www.w3.org/2001/XMLSchema#string">
    HYPER2
  </AttributeValue>
</AttributeAssignmentExpression>
</AttributeAssignmentExpression>
<AttributeAssignmentExpression>
  <AttributeValue DataTypes="http://www.w3.org/2001/XMLSchema#string">
    TYPE2
  </AttributeValue>
</AttributeAssignmentExpression>
</ObligationExpression>
</Rule>
<Rule RuleId="hyperl_freeze" Effect="Permit">
  <Condition>
    <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-equal">
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:or">
        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
          <AttributeValue DataTypes="http://www.w3.org/2001/XMLSchema#integer">
            0
          </AttributeValue>
        </Apply>
      </Apply>
    </Apply>
  </Condition>
  <AttributeDesignator DataTypes="http://www.w3.org/2001/XMLSchema#anyURI"
    MustBePresent="false"
    Category="urn:oasis:names:tc:xacml:3.0:attribute-category:system"
    AttributeId="urn:oasis:names:tc:xacml:3.0:system:hyperl.availableResources" />
</Rule>
</Apply>
</Apply>
A formal sw. eng. approach to Policy-Based Access Control

FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-less-than-or-equal">
   <AttributeValue Data-Type="http://www.w3.org/2001/XMLSchema#integer">2</AttributeValue>
</Function>

<AttributeDesignator Data-Type="http://www.w3.org/2001/XMLSchema#anyURI" MustBePresent="false"
   Category="urn:oasis:names:tc:xacml:3.0:attribute-category:system"
   AttributeId="urn:oasis:names:tc:xacml:3.0:system:hyper1.vm1-counter">
   <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-equal">
         <AttributeValue Data-Type="http://www.w3.org/2001/XMLSchema#integer">1</AttributeValue>
      </Apply>
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-less-than-or-equal">
         <AttributeValue Data-Type="http://www.w3.org/2001/XMLSchema#integer">1</AttributeValue>
      </Apply>
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-less-than-or-equal">
         <AttributeValue Data-Type="http://www.w3.org/2001/XMLSchema#integer">2</AttributeValue>
      </Apply>
   </Apply>
</AttributeDesignator>

<AttributeDesignator Data-Type="http://www.w3.org/2001/XMLSchema#anyURI" MustBePresent="false"
   Category="urn:oasis:names:tc:xacml:3.0:attribute-category:system"
   AttributeId="urn:oasis:names:tc:xacml:3.0:system:hyper1.availableResources">
   <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-equal">
         <AttributeValue Data-Type="http://www.w3.org/2001/XMLSchema#integer">1</AttributeValue>
      </Apply>
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-less-than-or-equal">
         <AttributeValue Data-Type="http://www.w3.org/2001/XMLSchema#integer">2</AttributeValue>
      </Apply>
   </Apply>
</AttributeDesignator>

<AttributeAssignmentExpression AttributeId="urn:oasis:names:tc:xacml:3.0:argument">
   <AttributeAssignmentExpression AttributeId="urn:oasis:names:tc:xacml:3.0:argument">
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:subtract">
         <AttributeValue Data-Type="http://www.w3.org/2001/XMLSchema#integer">2</AttributeValue>
      </Apply>
   </AttributeAssignmentExpression>
</AttributeAssignmentExpression>

<AttributeAssignmentExpression AttributeId="urn:oasis:names:tc:xacml:3.0:argument">
   <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-less-than-or-equal">
      <AttributeValue Data-Type="http://www.w3.org/2001/XMLSchema#integer">2</AttributeValue>
   </Apply>
</AttributeAssignmentExpression>

<AttributeAssignmentExpression AttributeId="urn:oasis:names:tc:xacml:3.0:argument">
   <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-equal">
      <AttributeValue Data-Type="http://www.w3.org/2001/XMLSchema#integer">1</AttributeValue>
   </Apply>
</AttributeAssignmentExpression>

<AttributeAssignmentExpression AttributeId="urn:oasis:names:tc:xacml:3.0:argument">
   <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-less-than-or-equal">
      <AttributeValue Data-Type="http://www.w3.org/2001/XMLSchema#integer">1</AttributeValue>
   </Apply>
</AttributeAssignmentExpression>

<AttributeAssignmentExpression AttributeId="urn:oasis:names:tc:xacml:3.0:argument">
   <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer-less-than-or-equal">
      <AttributeValue Data-Type="http://www.w3.org/2001/XMLSchema#integer">2</AttributeValue>
   </Apply>
</AttributeAssignmentExpression>
<AttributeDesignator DataTypes="http://www.w3.org/2001/XMLSchema#anyURI"
MustBePresent="false"
Category="urn:oasis:names:tc:xacml:3.0:attribute−category:system"
AttributeId="urn:oasis:names:tc:xacml:3.0:system:hyper1.availableResources"/>
</Apply>
</AttributeAssignmentExpression>
<AttributeAssignmentExpression
AttributeId="urn:oasis:names:tc:xacml:3.0:argument">
<AttributeValue DataTypes="http://www.w3.org/2001/XMLSchema#string">
HYPER1
</AttributeValue>
</AttributeAssignmentExpression>
<AttributeAssignmentExpression
AttributeId="urn:oasis:names:tc:xacml:3.0:argument">
<AttributeValue DataTypes="http://www.w3.org/2001/XMLSchema#string">
HYPER2
</AttributeValue>
</AttributeAssignmentExpression>
</AttributeAssignmentExpression>
</ObligationExpression>
</ObligationExpression>
</ObligationExpression>
</ObligationExpression>
</ObligationExpression>
</Rule>
</Rule RuleId="hyper_2_freeze" Effect="Permit">
<Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:or">
<Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
<Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer−equal">
<AttributeValue DataTypes="http://www.w3.org/2001/XMLSchema#integer">
0
</AttributeValue>
</ApplyDesignator DataTypes="http://www.w3.org/2001/XMLSchema#anyURI"
MustBePresent="false"
Category="urn:oasis:names:tc:xacml:3.0:attribute−category:system"
AttributeId="urn:oasis:names:tc:xacml:3.0:system:hyper2.availableResources"/>
</Apply>
</Apply>
<Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:integer−less−than−or−equal">

Listing 26. Release policy

```xml
<Policy xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17
   http://docs.oasis-open.org/xacml/3.0/xacml-core-v3-schema-wd-17.xsd"
   PolicyId="Release_Policies" Version="1.0"
   RuleCombiningAlgId="urn:oasis:names:tc:xacml:3.0:rule-combining-algorithm:permitoverrides"
   MaxDelegationDepth="1">
  ...
</Policy>
```
<Target>
  <AnyOf>
    <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
      <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
        RELEASE
      </AttributeValue>
      <AttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#anyURI" MustBePresent="false">
        Category="urn:oasis:names:tc:xacml:3.0:attribute-category:action"
        AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id" />
    </Match>
  </AnyOf>
</Target>

<Rule RuleId="hyper.1_release" Effect="Permit">
  <Condition>
    <Apply>
      <FunctionId="urn:oasis:names:tc:xacml:1.0:function:anyURI-at-least-one-member-of">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#anyURI" MustBePresent="false">
          Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
          AttributeId="urn:oasis:names:tc:xacml:1.0:resource:vm-id" />
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#anyURI" MustBePresent="false">
          Category="urn:oasis:names:tc:xacml:3.0:attribute-category:subject"
          AttributeId="urn:oasis:names:tc:xacml:1.0:subject:profile-id" />
        </AttributeValue>
      </FunctionId>
      <AttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#anyURI" MustBePresent="false">
        Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
        AttributeId="urn:oasis:names:tc:xacml:1.0:resource:vm-id" />
      </AttributeDesignator>
    </Apply>
  </Condition>
</Rule>
<Apply>
</Apply>

</Condition>

<AttributeAssignmentExpression>
  <AttributeValue>
    <AttributeDesigner DataType="http://www.w3.org/2001/XMLSchema#anyURI"
      MustBePresent="false"
      Category="urn:oasis:names:tc:xacml:3.0:attribute-category:system"
      AttributeId="urn:oasis:names:tc:xacml:3.0:system:hyper1-vm-ids"/>
  </AttributeValue>
</AttributeAssignmentExpression>

<AttributeAssignmentExpression>
  <AttributeValue>
    <AttributeDesigner DataType="http://www.w3.org/2001/XMLSchema#anyURI"
      MustBePresent="false"
      Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
      AttributeId="urn:oasis:names:tc:xacml:1.0:resource:vm-id"/>
  </AttributeValue>
</AttributeAssignmentExpression>

</Condition>

<AttributeAssignmentExpression>
  <AttributeValue>
    <AttributeDesigner DataType="http://www.w3.org/2001/XMLSchema#anyURI"
      MustBePresent="false"
      Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
      AttributeId="urn:oasis:names:tc:xacml:1.0:resource:vm-id"/>
  </AttributeValue>
</AttributeAssignmentExpression>

<AttributeAssignmentExpression>
  <AttributeValue>
    <AttributeDesigner DataType="http://www.w3.org/2001/XMLSchema#anyURI"
      MustBePresent="false"
      Category="urn:oasis:names:tc:xacml:3.0:attribute-category:system"
      AttributeId="urn:oasis:names:tc:xacml:3.0:system:hyper2-vm-ids"/>
  </AttributeValue>
</AttributeAssignmentExpression>

</Condition>

<AttributeAssignmentExpression>
  <AttributeValue>
    <AttributeDesigner DataType="http://www.w3.org/2001/XMLSchema#anyURI"
      MustBePresent="false"
      Category="urn:oasis:names:tc:xacml:3.0:attribute-category:resource"
      AttributeId="urn:oasis:names:tc:xacml:3.0:resource:vm-id"/>
  </AttributeValue>
</AttributeAssignmentExpression>
A formal sw. eng. approach to Policy-Based Access Control

```
<AttributeAssignmentExpression
  <ObligationExpression
    <ObligationExpressions
      <Rule
        <Policy
          AttributeId="urn:oasis:names:tc:xacml:1.0:resource:vm-id" />
        </ObligationExpression
        </ObligationExpressions
      </Rule>
    </Policy>
```