

Unification in Privacy Policy Evaluation – Translating EPAL into Prolog*

Michael Backes
IBM Research, Switzerland
mbc@zurich.ibm.com

Markus Dürmuth[†]
University of Karlsruhe, Germany
markus.duermuth@web.de

Günter Karjoth
IBM Research, Switzerland
gka@zurich.ibm.com

Abstract

Privacy policy evaluation engines enable queries whether a specific user is allowed to access specific data for a specific purpose. While tools for authoring, maintaining, and auditing privacy policies already exist, no tool exists yet to deal with unification within such policies, e.g., to enable queries if data might be modified by some user, or how many user entries satisfy a certain constraint. We show how this can be achieved by embedding enterprise privacy policies into Prolog. We show this concretely for IBM's Enterprise Privacy Authorization Language (EPAL). Based on the unification mechanisms of Prolog, our work enables general queries for privacy policies as well as quantitative measurements.

1. Introduction

An increasing number of enterprises make privacy promises to customers or, at least in the US and Canada, fall under new privacy regulations. To ensure adherence to these promises and regulations, enterprise privacy technologies are emerging [3]. An important tool for enterprise privacy enforcement is formalized enterprise privacy policies [4, 6]. Compared with the well-known language P3P [7] intended for privacy promises to customers, languages for the internal privacy practices of enterprises and for technical privacy enforcement must offer more possibilities for fine-grained distinction of data users, purposes, etc., as well as a clearer semantics.

Such languages define the purposes for which collected data can be used, model the consent that data subjects can give, and may impose obligations onto the enterprise. They can formalize privacy statements like “we use data of a minor for marketing purposes only if the parent has given consent” or “medical data can only be read by the patient's primary care physician”. The evaluation mechanisms of enter-

prise privacy policies can then be used to decide if a specific user is allowed to access specific data for a specific purpose. However, the existing evaluation mechanism cannot be used to answer queries that involve unification, e.g., whether a user *exists* that can edit specific data, or, more generally, to give quantitative measurements like the number of user entries that satisfy a certain constraint. In practice however, such queries are highly attractive: Examples range from deciding if employees of the marketing department are allowed to access data that must not be used for marketing purposes, over identifying which data an application is allowed to access, to building up statistics over user entries without violating their individual privacy.

We show how such general queries can be handled by transforming enterprise privacy policies into Prolog. Since unification is a central construct in Prolog, and since Prolog offers very powerful decision procedures, our transformation allows for very general queries for privacy policies as well as quantitative measurements. We show the transformation concretely for IBM's Enterprise Privacy Authorization Language (EPAL) [1], which has recently become an accepted W3C member submission. The transformation has been implemented as an XSLT program.

2. EPAL

In this section we give a brief review of EPAL. Instead of the lengthy XML syntax, we use a corresponding abstract syntax [2] and omit all details that are not necessary for understanding. An EPAL privacy policy consists of a *vocabulary* Voc , a list of *authorization rules* R , a *global condition* gc , a *default ruling* dr , and a *default obligation* \bar{do} .

The *vocabulary* defines the categories of users and data, the actions being performed on the data, the business purposes associated with the access requests, and obligations incurred on access. Note that purposes and obligations are not part of classical access control languages. Users, data, and purposes are captured in *hierarchies*. Formally, a hierarchy is pair $(H, >_H)$ of a finite set H and a transitive, non-reflexive relation $>_H \subseteq H \times H$, where every $h \in H$ has at most one immediate predecessor (parent). We write

*The long version of this paper has been published as IBM Research Report RZ 3541.

[†]This work was done when the author was on internship at the IBM Zurich Research Laboratory.

\geq_H for the reflexive closure of $>_H$. We write $h \geq_H h'$ if $h \geq_H h'$ or $h' \geq_H h$ holds. Technically, a vocabulary is a tuple (UH, DH, PH, A, Var, O) where UH , DH , and PH are hierarchies called the *user*, *data*, and *purpose hierarchy*, respectively, and A is a set of *actions*. Var is a set of *variables* used to build conditions that determine when a rule of the policy is applicable, and O is a set of *obligations*.

The *authorization rules*, short *rule list*, is a list containing elements of the form $(u, d, p, a, r, c, \bar{o})$ called *rules*, where $(u, d, p, a) \in U \times D \times P \times A$,¹ c – the *condition* of the rule – is a well-typed formula of a subset of standard logic based on the set Var of variables, $r \in \{+, -\}$ is the *ruling* of the rule, and \bar{o} is a set of obligations. Conditions are used to check context, consent, and other data subject properties. Formally, a rule list for a vocabulary Voc is a list containing elements of $U \times D \times P \times A \times \{+, -\} \times C(Var) \times \mathfrak{P}(O)$, where $C(Var)$ is the set of well-typed formulas of the underlying logic over Var , and $\mathfrak{P}(O)$ denotes the powerset of O . In EPAL, the precedences of the rules are contained implicitly by the textual order of the rules. The rulings $+$ and $-$ mean ‘allow’ and ‘deny’. We say that a rule is *negative* if it has a ‘deny’ ruling, otherwise it is *positive*. For the ease of writing, we augment each rule *rule* with a natural number $n(rule)$ called the rule’s *precedence*, such that rules that come first in the rule list have higher precedence. Thus, we write $(u, d, p, a, r, c, \bar{o}, n(rule))$ instead of $(u, d, p, a, r, c, \bar{o})$.

Finally, the *global condition* gc , the *default ruling* dr , and the *default obligation* \bar{do} are arbitrary elements from $C(Var)$, $\{+, \circ, -\}$, and $\mathfrak{P}(O)$, respectively, where \circ corresponds to a ‘don’t care’ decision.

A *query* to an EPAL policy is a tuple (u, d, p, a) . Typically, queries belong to the set $U \times D \times P \times A$ for the given vocabulary; queries that do not fulfill this will yield a special *scope error*. EPAL queries are not restricted to “ground terms”, i.e., minimal elements in the hierarchies, to handle policy extension, refinement, and composition [2].

Whether a rule with a satisfied condition matches a given query crucially depends on its ruling: First, every rule matches the query which it is defined for. Positive rules additionally match for all children of the defined query (component-wise in the hierarchies), i.e., positive rules are inherited downward the hierarchies. Negative rules are additionally inherited upwards the hierarchies, i.e., they apply also for parents of the defined query.

The semantics of an EPAL privacy policy is a function $eval$ that evaluates a query based on a given assignment

¹In the XML syntax of EPAL, rules may have subsets of each of these four sets instead of single elements which allow a more convenient way of specifying a policy. In the abstract syntax, rules are then flattened under adherence of the relative order, i.e., such a rule $(\bar{u}, \bar{d}, \bar{p}, \bar{a}, r, c, \bar{o})$ with $\bar{u} \subseteq U$, $\bar{d} \subseteq D$, $\bar{p} \subseteq P$, and $\bar{a} \subseteq A$ corresponds to the list of rules $((u, d, p, a, r, c, \bar{o}) \mid u \in \bar{u}, d \in \bar{d}, p \in \bar{p}, a \in \bar{a})$, where the order of the rules can be chosen arbitrarily.

χ of the variables in Var and returns as result a *decision* and a *set of associated obligations*. For a compact representation of the function $eval$, as depicted in Algorithm 1, we introduce the following parent-child notation on quadruples. Let $(u, d, p, a) \square (u', d', p', a')$ for $\square \in \{\geq, \succ\}$ iff $u \square u' \wedge d \square d' \wedge p \square p' \wedge a = a'$.

```

if  $gc \rightarrow_\chi$  false then return  $(dr, \emptyset)$ 
if  $(u_R, d_R, p_R, a_R) \notin U \times D \times P \times A$  then return
(scope_error,  $\emptyset$ )
foreach  $(u, d, p, a, r, c, \bar{o}) \in R$  (in the given order) do
  if  $c \rightarrow_\chi$  true then
    if  $r = + \wedge (u, d, p, a) \geq (u_R, d_R, p_R, a_R)$  then
      return  $(r, \bar{o})$ 
    if  $r = - \wedge (u, d, p, a) \succ (u_R, d_R, p_R, a_R)$  then
      return  $(r, \bar{o})$ 
return  $(dr, \bar{do})$ 

```

Algorithm 1: Query Evaluation

If the query is not contained in the considered vocabulary or the global condition is not satisfied under the given assignment then the result is $(scope_error, \emptyset)$ respectively (dr, \emptyset) . Otherwise, the decision is determined by the first matching rule whose condition is satisfied. If no such rule exists, the decision equals the default ruling.

3. Translation

Our transformation imitates the inheritance mechanisms of EPAL in Prolog and transforms EPAL rules more or less one-to-one to Prolog rules. This approach is usually less efficient than a specialized translation, where the decisions of the policy are determined for all possible queries and assignments, allowing to implement query evaluation very efficiently by simple table look-up. However, it allows to answer more general queries.

3.1. A Brief Review of Prolog

In Prolog, a *clause* is an expression of the form $B :- A_1, \dots, A_n$. ($n \geq 0$) where B is a *goal* and A_1, \dots, A_n are *conditions* (or sub-goals) of the clause. In the above clause, the symbol $:-$ reads *if* to indicate logical implication and the symbol $,$ reads *and*:

B is true if $(A_1$ is true and \dots and A_n is true).

If a clause does not have a condition, it is called a *fact*, and a *rule* otherwise. *Functors* are names that begin with a lower case letter or that are enclosed in single quotes, and numbers. *Variables* are names that either begin with an upper case letter or an underscore. A *term* is either a functor or

a variable. Any variable is implicitly considered to be universally quantified with the scope being the entire clause. A *predicate* is a collection of clauses whose goals have the same functor and arity.

If a predicate offers multiple clauses to solve a goal, they are tried one-by-one until one succeeds. In particular, Prolog’s backtracking mechanism for non-determinism can return more than one answer. Unification is used to assign values to variables.

3.2. Transformation and Semantical Encoding

Hierarchies. The hierarchies of an EPAL privacy policy are mapped directly to Prolog by one fact for each edge of the hierarchy. We show this exemplarily for the user hierarchy. A user relation $\text{user}_A \geq_U \text{user}_B$ is transformed to the Prolog fact $\text{uh}(\text{user}_A, \text{user}_B)$, where uh abbreviates “user hierarchy”. The transitive closure deruh of the rules $\text{uh}(\cdot, \cdot)$ is defined by

```
deruh(X,X).
deruh(X,Y) :- uh(X,Y).
deruh(X,Y) :- uh(X,Z), deruh(Z,Y),
```

which we call the *derived user hierarchy*. By definition, we have $u_1 \geq_U u_2$ if and only if $\text{deruh}(u_1, u_2)$ for all users u_1, u_2 . The corresponding derived hierarchies derdh and derph for data and purposes are defined similarly.

Conditions. EPAL provides a set of predefined functions and predicates to build conditions. Most Prolog systems also have a rich set of built-in predicates, which can be used to implement EPAL conditions. Note that EPAL predicates and their corresponding Prolog predicate might differ slightly. We identify different degrees of converting conditions into Prolog. In the basic version, we assume that conditions are coded as Prolog facts. A more elaborated version deals with the individual attributes (variables) of the conditions. According to their type, we assign values as above, stating them as facts, and then evaluate the condition expressions accordingly.

Rules. Prolog clause $\text{rule}(u, d, p, a, r, o, n) :- c$ captures rule (u, d, p, a, r, c, o, n) from the rule list R , where rulings are defined as allow for $+$ and deny for $-$. For a rule list R , we refer to the set $P(R)$ of these clauses as the *Prolog rules*.

To capture the semantics of EPAL within Prolog, i.e., the evaluation of queries based on the function eval , we first define that an EPAL rule is applicable for a given query $q = (u', d', p', a')$ if and only if there is a rule (u, d, p, a, r, c, o, n) such that $a = a'$ and $x \geq x' \vee (x' \geq x \wedge r = -)$ for every $x \in \{u, d, p\}$. Next, we define Prolog clauses as

```
ask(U,D,P,A,R,O,N)
:- deruh(U,U1), ask1(U1,D,P,A,R,O,N).
ask(U,D,P,A,R,O,N)
:- deruh(U1,U), ask1(U1,D,P,A,deny,O,N).
ask1(U,D,P,A,R,O,N)
:- derdh(D,D1), ask2(U,D1,P,A,R,O,N).
ask1(U,D,P,A,R,O,N)
:- derdh(D1,D), ask2(U,D1,P,A,deny,O,N).
ask2(U,D,P,A,R,O,N)
:- derph(P,P1), rule(U,D,P1,A,R,O,N).
ask2(U,D,P,A,R,O,N)
:- derph(P1,P), rule(U,D,P1,A,deny,O,N).
```

and we show that the Prolog goal ask corresponds to the individual rules in a natural way. Therefore, we introduce the scope of a Prolog rule.

Definition 1 *The scope of a Prolog rule is defined as*

$$\begin{aligned} \text{scope}(\text{rule}(u, d, p, a, +, o, n) : -c) \\ := \{ (u', d', p', a') \mid u' \leq u, d' \leq d, p' \leq p, a' = a \}, \\ \text{scope}(\text{rule}(u, d, p, a, -, o, n) : -c) \\ := \{ (u', d', p', a') \mid u \geq u', d \geq d', p \geq p', a' = a \}. \end{aligned}$$

If a query q is in the scope of a Prolog rule, we say that this rule is applicable for q .

We can now state the following simple lemma.

Lemma 1 *Let an EPAL policy with rule list R be given and let $\text{rule}(u, d, p, a, r, o, n) :- c \in P(R)$. Then the Prolog goal $\text{ask}(u', d', p', a, r, o, n)$ is derivable if and only if this rule is applicable for $q = (u', d', p', a)$ and the condition c evaluates to true under the considered assignment.*

Because $?- \text{ask}(u, d, p, a, -, -, -)$ returns all applicable rules independent of their precedences, we sort them based on their precedences:

```
query(U,D,P,A,R,O,N)
:- ask(U,D,P,A,R,O,N),
not(ask(U,D,P,A,_,_,Y), N > Y).
```

The goal query searches for applicable rules and succeeds if there is no other applicable rule with lower precedence. It uses negation, i.e., if the goal $\text{ask}(U, D, P, A, -, -, Y), N > Y$ has a solution, it fails; otherwise it succeeds. This algorithm has the advantage that it also permits queries with free variables for users, data, purposes, and actions, which is not possible for other algorithms.

The following theorem finally captures that this Prolog goal corresponds to the EPAL semantics.

Theorem 1 *Let P be an EPAL policy, let $P(P)$ denote the corresponding Prolog program obtained by applying above transformations, and let χ be an arbitrary assignment for Var . Then the goal $\text{query}(u, d, p, a, r, O, -)$ is derivable in $P(P)$ and the Prolog variable O is bound to obligation o if and only if $\text{eval}_{P, \chi}(u, d, p, a) = (r, o)$.*

4. Benefits of the Embedding

Our transformation shows that all the information contained in EPAL can easily be stored in a Prolog database and inheritance can be canonically expressed within Prolog. A formal mapping ensures that no errors are introduced in the translation. Thus, we can use Prolog to perform consistency checks for privacy policies and to evaluate access queries.

Unification. First, we want to stress that after transforming a privacy policy into Prolog, much more general queries are possible compared to plain EPAL. Our Prolog representation enables queries of the form “Who is allowed to read personal data of category credit card number” by

```
| ?- query(U,creditCardNumber,anyPurpose,
          read,allow,_,_).

U = marketingDep
```

One can also ask Prolog to return other successful unifications which forms the basis to obtain quantitative measurements for queries. As an example, consider a query “Are employees of the marketing department allowed to access data that must not be used for marketing purposes?”, which can be represented by

```
| ?- query(_,D,marketing,deny,_,_),
     query(marketingDep,D,_,_,allow,_,_).
```

Moreover, such queries may support the development of new business processes, for example by determining what personal data is accessible by department `salesDep` under purpose `statistics`. The following query returns the accessible data categories in variable `L`:

```
| ?- setOf(D,query(salesDep,D,statistics,
                  allow,_,_), L).

L = [age,gender,citizenship]
```

Building up statistics. In EPAL, consent choices of the data subject are “stored” in (auditable) attributes (i.e., whether a certain data subject opted-in or opted-out to third-party marketing may be determined by an attribute called `consentToMarketing`). Having such a consent repository, for example expressed as Prolog facts, one can ask how many percent of the customer population could be reached by a marketing campaign.

Conflicts between stated policy and actual practice. The Platform for Privacy Preferences (P3P) specification [7] enables Web sites to describe their data collection practices, which can then be read and displayed by P3P-enabled browser software or other user agents. However, whether

or not the data inside the organization is used as promised depends on the enterprise’s actual privacy practices as expressed by a fine-grained privacy policy like EPAL.

There are two ways to assure that the EPAL policy correctly implements the privacy “promises” of the P3P policy. To achieve correctness by construction, the P3P policy is translated into an equivalent EPAL policy or vice versa [5]. However, these transformations are not trivial and must be verified. To achieve correctness by validation, it is checked whether each query granted by the EPAL policy is also granted by the P3P policy. If there is a P3P into Prolog mapping as well, Prolog offers the capability to verify the above property.

Assuming that goal `queryP3P(u,d,p)` returns a positive answer if the P3P specification grants access for user u on data d for purpose p , the below clause checks for implementation errors:

```
conflict(U,D,P) :- query(U,D,P,_,allow,_,_),
                  not(queryP3P(U,D,P)).
```

If the evaluation of goal `conflict` fails then the EPAL policy correctly implements the P3P policy; otherwise it outputs a triple for which access is granted by the EPAL policy but not by the P3P policy.

References

- [1] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter. Enterprise Privacy Authorization Language (EPAL 1.2), 10 Nov 2003. www.w3.org/Submission/SUBM-EPAL-20031110/.
- [2] M. Backes, B. Pfitzmann, and M. Schunter. A toolkit for managing enterprise privacy policies. In *8th European Symposium on Research in Computer Security (ESORICS)*, Lecture Notes in Computer Science 2808, pages 162–180. Springer, 2003.
- [3] A. Cavoukian and T. J. Hamilton. *The Privacy Payoff: How successful businesses build customer trust*. McGraw-Hill/Ryerson, 2002.
- [4] S. Fischer-Hübner. *IT-security and privacy: Design and use of privacy-enhancing security mechanisms*, Lecture Notes in Computer Science 1958. Springer, 2002.
- [5] G. Karjoth, M. Schunter, and E. Van Herreweghen. Translating privacy practices into privacy promises — how to promise what you can keep. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks (Policy’03)*, pages 135–146. IEEE Computer Society, 2003.
- [6] G. Karjoth, M. Schunter, and M. Waidner. The platform for enterprise privacy practices – privacy-enabled management of customer data. In *Privacy Enhancing Technologies*, Lecture Notes in Computer Science 2482, pg. 69–84. Springer, 2002.
- [7] Platform for Privacy Preferences (P3P). W3C Recommendation, Apr. 2002. <http://www.w3.org/TR/2002/REC-P3P-20020416/>.