A Self-Policing Policy Language

Sebastian Speiser and Rudi Studer

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
Institute of Applied Informatics and Formal Description Methods (AIFB)
firstname.lastname@kit.edu

Abstract. Formal policies allow the non-ambiguous definition of situations in which usage of certain entities are allowed, and enable the automatic evaluation whether a situation is compliant. This is useful for example in applications using data provided via standardized interfaces. The low technical barriers of integrating such data sources is in contrast to the manual evaluation of natural language policies as they currently exist. Usage situations can themselves be regulated by policies, which can be restricted by the policy of a used entity. Consider for example the Google Maps API, which requires that applications using the API must be available without a fee, i.e. the application's policy must not require a payment. In this paper we present a policy language that can express such constraints on other policies, i.e. a self-policing policy language. We validate our approach by realizing a use case scenario, using a policy engine developed for our language.

1 Introduction

Policies are declarative descriptions of constraints and conditions that apply to some entity (the policy subject). Formal languages allow non-ambiguous policies, that can be automatically evaluated by computers. Many existing policy languages represent essentially an implicit access control matrix [1]. While this is sufficient for applications such as rights management for local file systems, there are entities that still impose constraints on their use after initial access was granted. This often applies to data representing factual information or creative works. Examples include images that require attribution of their creator, or real-time stock quotes that can only be published for a fee. Generally such policies classify usage situations into compliant or non-compliant. Conditions, required to be fulfilled by compliant situations, may restrict the policy of the situation. Consider for example the Google Maps API, which requires that applications using the API must be made available to the public without a fee. This is basically a constraint in the API's policy, which restricts the application's policy to not grant exclusive access to paying users. There exist approaches to usage restrictions, but our work is to the best of our knowledge the first self-policing policy language, in the sense that it can express restrictions on other policies.

Today, vast amounts of data are published on the Internet with standardized interfaces, e.g. as Web services or as Linked Data[1]. This imposes only low technical

[1] http://linkeddata.org

barriers to the use and reuse of data in new ways and their composition into new applications or data sources. In contrast the policies regulating their allowed uses are either not made explicit at all [2], or published in natural language, in form of terms and conditions. The former case makes it impossible, the latter case a manual and very tedious task to evaluate if a given usage situation is compliant or not. This may lead to frequent violations of usage restrictions, not because of ill will, but convenience. Evidence for this assumption is delivered e.g. by Seneviratne et al. who discovered that around 70%-90% of the reuses of Flickr images with a Creative Commons attribution license actually violate the license [3]. Formal policies are required to build tools that help users to check compliance of their data usages with the same ease as just using the data.

Restrictions on other policies include testing if one policy is contained in another. The resulting query containment problem is undecidable for many policy languages (e.g. in the presence of general negations and disjunctions). This means that these languages cannot simply be extended with self-policing conditions. Another difficulty is that simple query containment may not work, as restrictions have to apply to policies with subjects that are unknown at specification time. Therefore a policy structure is required that separates identifying applicable policy subjects and required compliance conditions. Other restrictions include checking if a partial situation description is sufficient for fulfilling a policy, possibly under further restrictions on aspects not specified in the partial description. Such restrictions need novel algorithms. Another requirement for the policy language is usability for the policy specifiers. Two enabling properties for usability are an intuitive policy structure and the reuse of policy conditions.

The rest of the paper is structured as follows. After introducing a use case in Section 2 for further motivation and evaluation of the approach, and presenting preliminaries in Section 3, the following contributions are presented:

– A policy model with formal semantics based on unions of conjunctive queries and RDFS (see Section 4.1).
– A model for structuring policies to improve usability and enable the reuse of policy parts (comparable to the Creative Commons building blocks, such as (non-)commercial use). The structure is based on RDF and RIF and is provided with rules that map it to our policy model (see Sections 4.2 and 4.3).
– Formal definitions of useful types of policy restrictions and their integration into policy conditions (see Section 5).

We evaluate the approach by implementing a policy engine and applying it to policies realizing the use cases. This is described in Section 6 together with some performance experiments. In Section 7 the policy language is compared to existing work. In Section 8 we conclude and give an outlook to future work.

2 Use Case and Requirements

The policy language presented in this paper is thought to be applicable to different application scenarios. However for further motivating the features of the language and validating how they fulfill concrete requirements, we describe a specific application and a concrete use case in this section. The application area we deal with is the use of services and data in dynamic and composed documents. Another thinkable application would be expressing right restrictions of music pieces that also affect the right restrictions of a musical work that samples the original piece.

Dynamic and composed documents are an approach for integrating data and functionalities that are provided over standardized interfaces, e.g. as Web services or as Linked Data. Dynamic document compositions specify links to resources and how the obtained data is combined to form a final document. An example for such a composition is a dynamic PHP page, that reads stock quotes from a Web service and displays them in a human-friendly way. Both the Web service and the PHP page can be equipped with a policy restricting who can access them. The Web service could also have a clause that requires that Web pages displaying its result, have to have the same access restrictions as the service.

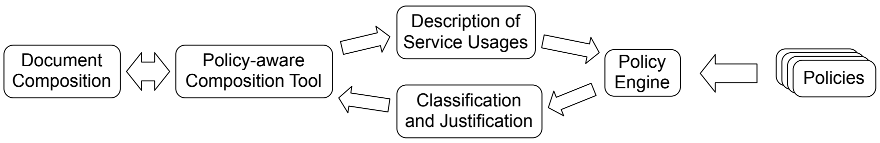
Document Composition
Policy-aware Composition Tool
Description of Service Usages
Classification and Justification
Policy Engine
Policies

Fig. 1. Use case scenario

For realizing the policies we use an abstract model of service and data usages that is the base for policy conditions. The policy-aware composition tool, as visualized in Figure 1, mediates between the concrete document composition (e.g. the PHP page) and its abstract description in terms of the usage model. The policy engine classifies the composition according to the policies of the used services and returns the result to the composition tool. In future work the

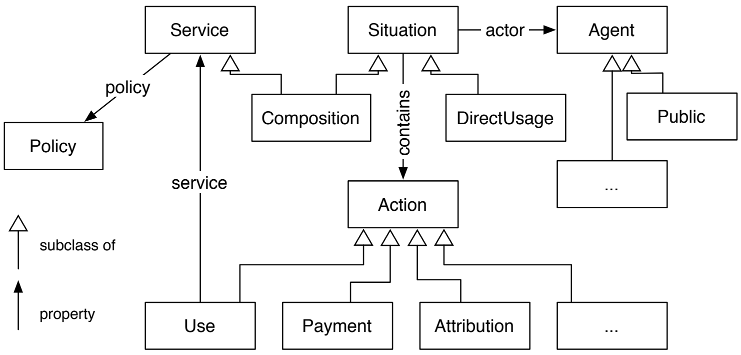

Fig. 2. Conceptual Model of Use Case for Policy Conditions

classification will be accompanied with a justification that helps to fix problems, if a situation is non-compliant.

The abstract conceptual model for compositions and service usages is visualized in Figure 2. Situations contain actions, that can be for example uses of services, payments or attributions. A situation can either be a direct usage, meaning that the actions are executed and the result directly used, or a composition, meaning that the result of the situation is again provided as a service. Situations are conducted by an agent, which can be optionally classified in subclasses. Services (including compositions) have a policy regulating their allowed uses.

In Section 6 we will show how our policy language can be used to model the following representative examples:

– The terms and conditions of the Google Maps API[2], which require (besides other clauses) that "Your Maps API Implementation must be generally accessible to users without charge."
– A service in a company internal scenario delivers confidential information, thus it can only be accessed by managers; the same must hold for compositions using the service.
– A service provider offers two stock quote services: one with real-time quotes that requires a payment, and one with delayed quotes that only requires an attribution. A service user is searching for stock quote services that can be used without payments.

3 Preliminaries

We choose RDF Schema (RDFS [4]) as data model for situation descriptions, as it provides desirable modeling features, but still has decidable algorithms for conjunctive query answering and containment. Modeling features of RDFS that are useful for describing usage situations include: (i) the use of URIs for individuals and classes, allowing heterogeneous actors and extensibility of situation models, (ii) class memberships and subclasses, e.g. an action belonging to a credit card payment class, fulfills the requirement of a general payment action, and (iii) subproperties, e.g. two actions in an application that always occur together (subproperty) are also related by a property describing actions that can possibly occur together (superproperty).

Let I, B, L, and V be disjoint infinite sets of IRIs, blank nodes, literals and variables. In the following $P(S)$ denotes the powerset of S.

Definition 1. *An RDF graph is a finite set of triples, defined as* $r \in P((I \cup B) \times I \times (I \cup B \cup L))$.

In Section 4, we introduce our policy model, which is based on conjunctive queries (CQs), as defined in the following.

[2] `http://code.google.com/apis/maps/terms.html`

Definition 2. *A conjunctive query $cq = (x, t)$ is a pair of head variables $x \subset V$ and a finite set of triple patterns $t \in P((I \cup V) \times I \times (I \cup V \cup L))$. We denote as $V_t = \{v \in V \mid \exists p, o\ (v, p, o) \in t \vee \exists s, p\ (s, p, v) \in t\}$ the set of all variables in a set of triple patterns t.*

Let M be the set of all function $\mu : I \cup L \cup V \to I \cup L$, s.t. $\forall a : (a \in I \cup L \to \mu(a) = a)$. As an abbreviation we also apply a function $\mu \in M$ to a set S ($\mu(S) = \{\mu(s) \mid s \in S\}$), to a triple or triple pattern $t = (s, p, o)$ ($\mu(t) = (\mu(s), \mu(p), \mu(o))$) or to sets of triples or triple patterns.

Definition 3. *The result set for a conjunctive query $cq = (x, t)$ applied to a RDF graph r is defined as $Q_{cq}(r) = \{x' \in (I \cup L)^{|x|} | \exists \mu \in M\ \mu(x) = x' \wedge \mu(t) \subseteq r\}$.*

Definition 4. *A union of conjunctive queries (UCQ) is a set CQ of conjunctive queries with the same head predicate. We define $Q_{CQ}(r) = \bigcup_{cq \in CQ} Q_{cq}(r)$.*

We assume that the we can evaluate queries on a RDF graph that is the fixpoint according to RDFS semantics for the properties and classes used in the queries, i.e. all implicit properties and class memberships are materialized.

In Section 5, we discuss restrictions on policies, which are partially defined using query containment. Query containment of a query CQ_1 in a query CQ_2, denoted as $CQ_1 \sqsubseteq CQ_2$, means that for every possible RDF graph r, every result of CQ_1 is also a result of CQ_2, i.e. $CQ_1(r) \subseteq CQ_2(r)$.

Definition 5. *A function $h : (I \cup L \cup B \cup V) \to (I \cup L \cup B \cup V)$ is a containment mapping from $cq_2 = (x_2, t_2)$ to $cq_1 = (x_1, t_1)$, if the following conditions hold:*
- $\forall x \in (I \cup L) : h(x) = x$
- $\forall x \in x_2 : h(x) \in x_1$
- $\forall (s, p, o) \in t_2 : (p = \texttt{rdf:type} \to$
  $\exists (s', p', o') \in t_1 : h(s) = s' \wedge p' = \texttt{rdf:type} \wedge o'\ \texttt{rdfs:subClassOf}\ o)$
- $\forall (s, p, o) \in t_2 : (p \neq \texttt{rdf:type} \to$
  $\exists (s', p', o') \in t_1 : h(s) = s' \wedge h(o) = o' \wedge p'\ \texttt{rdf:subPropertyOf}\ p)$

Note that `rdfs:subClassOf` *and* `rdfs:subPropertyOf` *are both reflexive.*

Definition 6. *A CQ cq_1 is contained in a CQ cq_2, if and only if there exists a containment mapping h from cq_2 to cq_1 (see[5, p. 882]).*

For showing query containment of a UCQ CQ_1 in another UCQ CQ_2, it is sufficient to show containment on the component CQs, i.e. $CQ_1 \sqsubseteq CQ_2 \leftarrow \forall cq_1 \in CQ_1\ \exists cq_2 \in CQ_2 : cq_1 \sqsubseteq cq_2$ (see [5, p. 904]).

4 Policy Model

As mentioned in the introduction, we want a policy to describe the circumstances in which it is allowed to use the entity that is the subject of the policy. We distinguish between policy applicability and compliance. Applicability describes the situations, which are regulated by a policy, i.e. considered a use of the policy subject. If a situation is not applicable it is trivially compliant, otherwise only if the situation fulfills the corresponding conditions.

This corresponds to a goal-based policy as defined by Kephart and Walsh in [6], as only the desired states are specified. Such policies are on a higher conceptual level than action-based policies, which specify for every situation what has to be done next. The notions are based on the classification of agents according to Russel and Norvig [7]. To arrive at a compliant state based on a goal policy, algorithms are needed that help to determine the needed actions, respectively situation modifications. In Section 4.2 we further elaborate on this aspect, after we describe in Section 4.1 the used formalisms for modeling descriptions and policy conditions.

4.1 Formal Policy Model

The sets of situations that are applicable, respectively compliant for a given policy, are described by conjunctive queries. CQs allow the declarative specification of properties that a situation must fulfill, using predicates (i.e. RDF properties and classes) on variables and constants which are connected by conjunctions.

Consider for example a policy that requires either a payment by credit card or if the usage is for scientific purposes, then an attribution of the service provider is sufficient. In order to avoid having two different policies, we define policy compliance conditions to be UCQs. Formally we define: a policy $P = (id, cq_a, CQ_c)$, where $id \in I$ is the IRI representing the policy entity, cq_a is a CQ defining the applicable policy subjects, and CQ_c is a UCQ defining the compliant policy subjects.

We define the two properties `applicable` and `compliant` with domain of policy subjects and range of policies. The extensions of these properties are defined in the following way for all policies $P = (id, cq_a, CQ_c)$ and all potential policy subjects s in an RDF graph r:

$$\texttt{s applicable id} \leftrightarrow (s) \in Q_{cq_a}(r), \text{ and}$$
$$\texttt{s compliant id} \leftrightarrow (s) \in Q_{cq_a}(r) \land \exists cq \in CQ_c : (s) \in Q_{cq}(r).$$

For the representation of such policies we employ the RIF-Core Dialect [8], to define a policy as a group of conjunctive rules, using RIF's annotation to link it to the policy entity. The RIF documents specify in [9] how RIF frame formulas of the form `s[p->o]` correspond to RDF triple (patterns) of the form `s' p' o'`.

Note that the policies do not support negation. This means that for example it is not possible to check that there is no activity with commercial purpose, instead such an absence has to be stated and required explicitly. Approaches like scoped negation (cf. [10]) make it possible to combine negation as failure with RDF's open world assumption. However, negation together with hierarchical predicates as introduced in Section 4.2 generally leads to undecidability of query containment.

4.2 Policy Structure

Unions of conjunctive queries (UCQs) provide a nice formal model of policies that is suitable for evaluation. However specifying them can introduce redundancy in the likely case that several alternatives of a union share common conditions. Furthermore UCQs lack an hierarchical structure which eases the specification and maintainability of policies. Therefore we allow not only the use of frame formulas in conditions that can be directly mapped to triple patterns but also the use of predicates with arbitrary arity that are themselves again defined as UCQs. This essentially means that policies can be specified as non-recursive datalog programs, which can always be expanded to UCQs using only base predicates (i.e. RDF class memberships and properties).

Note that such predicates can also be defined externally, which enables reuse of conditions across policies from different specifiers. This is comparable to the Creative Commons approach, where certain standard terms are defined that can be used to define custom policies (cf. [11]). As rules are identified by IRIs, they can be described not only by their formal definition as RIF documents but also by a legal or layman description, if the IRI is resolved by a Web browser (recognized by the `Accept` header of the HTTP request).

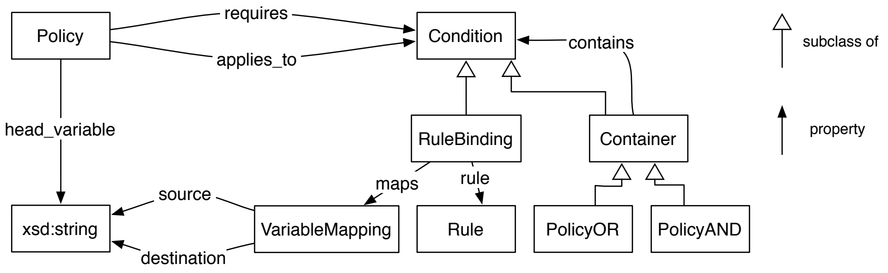

Fig. 3. Visualization of Policy Structures

In the following we define a conceptual model of combining policies from predicates defined by UCQs. It is based on an RDF model that refers by IRIs to rules defined in RIF. The model is visualized in Figure 3. A `Policy applies_to` subjects that are answers to the applicability query, which is defined by a `Condition`, which is either a `RuleBinding`, a conjunction of other `Conditions` (i.e. a `PolicyAND` container), or a disjunction of `Conditions` (i.e. a `PolicyOR` container). Furthermore a policy `requires` a condition, which represents the validity test, and has a `head_variable` which defines the policy subject in the conditions. Both `PolicyAND` and `PolicyOR` `contain` a number of conditions. `RuleBindings` refer by the `rule` property to an IRI which is the id of a group in a RIF document that defines the corresponding predicate. Note that by resolving the IRI we expect a RIF representation containing this group. The metadata of the group specifies via `defines_predicate` the head predicate of the rules. Furthermore a `RuleBinding` `maps` a number of `VariableMapping`s each with a `source` variable name of the defined predicate that is mapped to the `destination`, which is either a variable in the policy condition or a string representation of an IRI.

Such an hierarchical policy definition with simple boolean operators to combine basic conditions is a more user friendly way to specify policies, which is already familiar from filter creation in many email programs. Furthermore the structuring allows users to group conditions in sensible blocks, which can be exploited for giving justifications of (mainly negative) policy decisions. Due to the use of IRIs and metadata, the rules and policy parts can be annotated with further useful and human-readable information. See for example the work by Kagal et al. [12] for a policy engine that exploits policy structures for human-friendly justifications.

4.3 Mapping the Policy Structure to the Formal Model

The mapping from the proposed structural model to a policy's normal form (i.e. its UCQ as defined in Section 4) is defined in a bottom-up way. The most basic part is a rule defining a predicate based only on RDF properties. Using RIF presentation syntax (cf. [13]) it is expressed in the following way (`p:` is used in the following for the namespace of the policy vocabulary):

```
(* "RULEID"^^rif:iri
   "RULEID"^^rif:iri[p:defines_predicate->"PREDICATE"^^rif:iri] *)
Group (
   Forall ?h1 ... ?hn (
      "PREDICATE"^^rif:iri(?h1 ... ?hn) :-
         Exists ?e1 ... ?em (
            And( s1[p1->o1]
                    ...
                 sk[pk->ok])))
   Forall ?h1 ... ?hn (
      "PREDICATE"^^rif:iri(?h1 ... ?hn) :-
         Exists ?e1 ... ?em (
            And( s'1[p'1->o'1]
                    ...
                 s'l[p'l->o'l]))))
```

This maps to a union of conjunctions of the following form:

$$CQ_{RULEID} = \Big\{ \Big((h_1, \dots, h_n), \big\{(s_1, p_1, o_1), \dots, (s_k, p_k, o_k)\big\}\Big), \dots, \\ \Big((h_1, \dots, h_n), \big\{(s'_1, p'_1, o'_1), \dots, (s'_l, p'_l, o'_l)\big\}\Big)\Big\},$$

where $(s_1, p_1, o_1), \dots, (s'_l, p'_l, o'_l) \in I \cup V \times I \times I \cup V \cup L$. Note that it is also possible to use other (non-recursive) RIF predicates instead of only RDF properties. In this case, we assume that the IRI of the used predicate resolves to a RIF document that defines the corresponding UCQ. In this way a rule definition can always be expanded to a union of conjunctions in terms of simple RDF properties.

The rules are used in our policy model by `RuleBinding`s, which has the general form:

```
RB a p:RuleBinding;
   p:rule RULEID;
   p:maps MAP1;  p:maps ...;  p:maps MAPN.
```

We define a function $f_{\texttt{MAP}} : V \cup I \cup L \to V \cup I \cup L$ for each variable mapping `MAP = (source, destination)` in the following way:

$$f_{\texttt{MAP}}(x) = \begin{cases} \texttt{destination}, & \text{if } x = \texttt{source} \\ x, & \text{otherwise.} \end{cases}$$

We also use these functions when applied to UCQs with the meaning that it is applied to all variables, IRIs and literals in the UCQ. Thus, we can define the UCQ of the rule binding `RB` in the following way:

$$CQ_{\texttt{RB}} = f_{\texttt{MAP1}}(f_{\dots}(f_{\texttt{MAPN}}(CQ_{\texttt{RULEID}}))).$$

The mapping for both AND and OR containers are defined by treating them as binary operators. Due to the associativity of these operators, the mapping naturally applies also to containers with more components.

Conditions (e.g. rule bindings) are used in `PolicyAND` containers of the following form:

```
AND a p:PolicyAND;
    p:contains C1;
    p:contains C2.
```

The corresponding UCQ is obtained by creating the union of the conjunctions for each pair of alternatives of the two components. More formally:

$$CQ_{\texttt{AND}} = \bigcup_{(x_1,t_1)\in CQ_{\texttt{C1}}} \bigcup_{(x_2,t_2)\in CQ_{\texttt{C2}}} \{(x_1 \cup x_2, t_1 \cup t_2)\}.$$

For a `PolicyOR` container of the following form

```
OR a p:PolicyOR;
   p:contains C1;
   p:contains C2.
```

we define the UCQ as the union of the two components: $CQ_{\texttt{OR}} = CQ_{\texttt{C1}} \cup CQ_{\texttt{C2}}$. Finally we define the mapping for a `Policy` object to the formal model. Given the following representation

```
POL a p:Policy;
  p:head_variable HV;
  p:applies_to CA;
  p:requires CR.
```

, we define a policy $P_{\texttt{POL}} = (\texttt{POL}, CQ_{\texttt{CA}}, CQ_{\texttt{CR}})$.

5 Restrictions on Policies

Policies classify policy subjects into compliant, non-compliant, and inapplicable categories. If we want to ensure that certain kinds of policy subjects are always, respectively never, compliant with a policy, we have to restrict the policy with regard to a specification of the policy subject. Specifying restrictions on policies is useful for several tasks, as outlined in the following:

- Searching for an entity with a policy that allows certain situations, e.g. searching for a service that can be used without a payment.
- Validation of a policy, i.e. ensuring that it fulfills test restrictions.
- Comparison to other policies, e.g. to a previous version, in order to see, if the policy is stricter or more lax.
- Policing other policies, if the compliance of a policy subject depends on restrictions of the subject's policy.

Independent of their application, we found the three types of restrictions particularly useful, which are listed in the following, including examples of their use:

1. **Required for compliance:** is it necessary that a policy subject fulfills certain conditions in order to be compliant. If we specify the required conditions themselves as a policy, this restriction is equivalent to asking if all compliant subjects of the restricted policy are also compliant with the restricting policy. This can be solved by checking query containment of the policies.

   Examples for such restrictions are: (i) a policy must always require a payment, or (ii) a policy must restrict data access to a certain class of users.
2. **Not required for compliance:** is it possible that a policy subject is compliant without necessarily fulfilling certain conditions. This restriction is basically just the negation of the previous one, and thus can also be checked by query containment.

   Examples are: (i) can a subject be compliant without having a payment, (ii) can a service be used without being a registered user?
3. **Sufficient for compliance:** can a partially described subject be compliant by adding only further restrictions that do not affect the given description?

   An example for this restriction is: can a situation, where data is provided to the general public, be compliant? This is true if the policy does not further restrict the data recipient, but it may for example require a payment.

The presented restrictions rely on query containment, i.e. comparison of policies. As we want to compare policies that generally can apply to different subjects (i.e. the subjects of the restricted and the restricting policy), we define the comparisons in terms of the compliance conditions of policies, and dismiss the applicability conditions. This is one of the reasons for separating applicability and compliance, besides avoiding redundancy as applicability is part of every policy alternative (i.e. conjunction in the policy's UCQ).

In order to use the above defined restrictions in policy conditions, we introduce three RDF properties and formally define their extensions:

– `req_for_comp` (see 1. “required for compliance”),
– `not_req_for_comp` (see 2. “not required for compliance”, needed as we do not support negation), and
– `sufficient_for_comp` (see 3., “sufficient for compliance”).

As discussed above we can reduce the first two restrictions to query containment in the following way:

$$\texttt{P2 req\_for\_comp P1} \leftrightarrow P_{\texttt{P1}} = (\texttt{P1}, cq_a^1, CQ_c^1) \wedge P_{\texttt{P2}} = (\texttt{P2}, cq_a^2, CQ_c^2) \wedge CQ_c^1 \sqsubseteq CQ_c^2$$
$$\texttt{P2 not\_req\_for\_comp P1} \leftrightarrow \neg(\texttt{P2 req\_for\_comp P1})$$

The `sufficient_for_comp` property is defined between policies specifying the sufficient condition and a target policy. The sufficient condition policy $P_s = (id^s, cq_a^s, CQ_c^s)$ should only consist of a single acyclic conjunctive query ([14], also called tree queries [15]) $CQ_c^s = \{cq_s(x_s, t_s)\}$, whereas the target policy $P_t = (id^t, cq_a^t, CQ_c^t)$ can be a union of CQs. P_s is sufficient for P_t if there exists one policy alternative $cq_t \in CQ_c^t$ for which it is sufficient. Finding out, if $cq_s = (x_s = \{hv_s\}, t_s)$ is sufficient for $cq_t = (x_t = \{hv_t\}, t_t)$ can be done by doing a tree traversal of cq_s according to the following recursive condition:
$suff(cq_s, cq_t) \leftrightarrow is\_suff(hv_s, cq_s, cq_t, \{(hv_s, hv_t)\})$, where:

$$\begin{aligned}
&is\_suff(n, cq_s, cq_t, \mu)\\
&= \big(\forall c \in \{c \mid (n, \texttt{rdf:type}, c) \in t_s\} :\\
&\qquad \forall c' \in \{c' \mid (\mu(n), \texttt{rdf:type}, c') \in t_t\} : c \ \texttt{rdfs:subClassOf}\ c'\big) \wedge\\
&\big(\forall p \in \{p \mid \exists o : (n, p, o) \in t_s\} :\\
&\qquad \forall p' \in \{p' \mid \exists o' : (\mu(n), p', o') \in t_t\} :\\
&\qquad\quad \big((p' \ \texttt{rdfs:subPropertyOf}\ p) \rightarrow (p \ \texttt{rdfs:subPropertyOf}\ p')\big)\big) \wedge\\
&\big(\forall (p, o) \in \{(p, o) \mid (n, p, o) \in t_s\} :\\
&\qquad \forall (p', o') \in \{(p', o') \mid (\mu(n), p', o') \in t_t\} :\\
&\qquad\quad \big((p \ \texttt{rdfs:subPropertyOf}\ p') \wedge \{(x, x') \in \mu \mid x = o\} = \emptyset \rightarrow\\
&\qquad\qquad is\_suff(o, cq_s, cq_t, \mu \cup \{(o, o')\})\big)\big) \wedge\\
&\big(\forall (s, p) \in \{(s, p) \mid (s, p, n) \in t_s\} :\\
&\qquad \forall (s', p') \in \{(s', p') \mid (s', p', \mu(n)) \in t_t\} :\\
&\qquad\quad \big((p \ \texttt{rdfs:subPropertyOf}\ p') \wedge \{(x, x') \in \mu \mid x = s\} = \emptyset \rightarrow\\
&\qquad\qquad is\_suff(s, cq_s, cq_t, \mu \cup \{(s, s')\})\big)\big).
\end{aligned}$$

The definition of is_suff is divided into four conditions. The first condition checks for a node mapping, that there are no stricter class requirements in the target policy than in the sufficiency condition. The second condition checks that no stricter property requirements occur (i.e. every mapping to a subproperty must be an equivalent property). The third and fourth conditions follow the patterns connected to a node (depending on its position as a subject or object) and recursively apply is_suff to the newly mapped variables. As we required the sufficiency condition to be an acyclic conjunctive query and only patterns are followed that map previously unmapped variables, the recursion will always come to an end.

The proposed policy restriction properties are defined to have special interpretations, as defined in this section. As the definitions for query containment and sufficiency rely on normal RDFS interpretations of properties, the restriction properties cannot be freely used in policies occuring in restriction conditions. Specifically the current definitions do not support restriction properties in sufficiency conditions (i.e. `S` in `S sufficient_for_comp P`) and containing policies (i.e. `P2` in `P2 (not_)req_for_comp P1`). Note that the properties can occur in contained policies, as they just reduce the set of compliant subjects and thus can be ignored.

6 Evaluation

In Section 2 we presented a use case and three concrete examples. We modeled the use case using our policy language and tested it with a prototypical implementation of a policy engine, that we developed. In the following we present and discuss interesting aspects of the example policies. The full examples in RDF and RIF, as well as the policy engine and its source code are available online[3]. At the end of the section, we elaborate on the performance of the policy engine.

Policy for Google Maps API. In subsequent descriptions we use N3-syntax for RDF and the abstract syntax for RIF. The URI prefix `p:` stands for the policy vocabulary, and `m:` points to the conceptual model for compositions and service usages. In the following we show the description of the maps policy, the `policy:` prefix points to a RIF file containing maps policy rules, and `generalrules:` refers to a RIF file describing general rules that can be reused by different policy specifiers.

```
@prefix gm: <http://example.org/googlemapsapi#> .
@prefix policy: <http://example.org/gmpolicy#> .

gm:policy a p:Policy;
  p:head_variable "situation";
  p:applies_to [a p:RuleBinding;
              p:rule policy:apprule];
  p:requires [a p:PolicyAND;
    p:contains :RegisteredUser;
    p:contains [a p:PolicyOR;
      p:contains [a p:PolicyAND;
       p:contains <http://example.org/nopaymentreq#NoPaymentReqCondition>;
       p:contains :AvailForPublic];
      p:contains [a p:RuleBinding; p:rule generalrules:DirectUse]]].

:RegisteredUser a p:RuleBinding;
  p:rule policy:GMRegisteredUser.
:AvailForPublic a p:RuleBinding;
  p:rule policy:AvailForPublicRule.
```

[3] `http://code.google.com/p/seppl/`

The policy defines that applicability is determined by the `apprule` and requires for compliance that (i) the actor of an applicable situation is a registered user (rule `GMRegisteredUser`), and (ii) that the situation is either a direct use (rule `generalrules:DirectUse`), or a composition that has a policy which makes it available to the public (rule `AvailForPublicRule`) and does not require a payment (link to external rule binding, reusing this common condition).

The `apprule` specifies that situations are applicable to this policy, if it uses the Google Maps API (defined as `gm:service a m:Service`):

```
(* policy:apprule
   policy:apprule[p:defines_predicate -> policy:apprulepred *)
Group (
  Forall ?situation (
    policy:apprulepred(?situation) :- Exists ?usage (
            And ( ?situation[rdf:type -> m:Situation]
                  ?situation[m:contains -> ?usage]
                  ?usage[rdf:type -> m:Usage]
                  ?usage[m:service -> gm:service] ) ) ) )
```

The `AvailForPublicRule` has the following rule body:

```
And ( ?situation[rdf:type -> m:Composition]
      ?situation[m:policy -> ?policy]
      gm:AvailableForPublicPolicy[p:sufficient_for_comp -> ?policy] )
```

This means that another policy is described (`gm:AvailableForPublicPolicy`) that defines a partial situation description which must be sufficient for fulfilling the policy of a composition which is using the API. The partial situation is defined by a binding of a rule with the following body:

```
And ( ?situation[m:actor -> ?actor]
      ?actor[rdf:type -> m:Public] )
```

The partial situation is thus only sufficient if the composition's policy allows access by actors without requiring them to belong to any other class than `m:Public`.

Confidential company internal service. The policy of the confidential service requires two rules: (i) one checking if the actor of the using situation is a manager, and (ii) one that checks if the policy of a using composition is contained in a policy restricting access to managers. The second rule ensures that if the service is used in a composition, then the composition inherits the access restrictions. For the realization of this rule, the `p:req_for_comp` property was used.

Stock quotes service. The policies of the stock quote services are rather straightforward, one checking for a payment and the other one for an attribution. The search process is realized in the following way: (i) the user creates a policy that requires a situation that contains a payment, (ii) he asks the policy engine to check for both stock quote services if their policy is not contained in his policy. The engine answers the request by using the `p:not_req_for_comp` property, which only holds for the delayed stock quote service.

Performance. Compliance checking using our policy language corresponds to answering unions of conjunctive queries. Conjunctive query answering is known to be NP-complete [16] for relational databases. This result can be transferred to RDFS knowledge bases with a materialized fixpoint, where the properties can be treated as relations. However, in our approach special properties exist that check restrictions on policies. The evaluation, if two instances are related by such a property involves checking query containment, which for positive conjunctive queries is equivalent to query answering and thus also NP-complete. Thus in combination this means a complexity of up to $\Sigma_2 P$ (i.e. NP with an NP oracle) for policy evaluation. The theoretical complexity relates to the size of the queries defining the policies.

For testing what the theoretical complexity means for practical purposes we conducted some performance measurements using our (non-optimized) policy engine. We created for both the maps API policy and the confidential service policy each three situation descriptions: one that is compliant, one that is non-compliant and one that is not applicable. We measured the classification time on a laptop with an Intel Core2Duo 2.4GHz processor and 4 GB of main memory. Furthermore we measured the search time for determining for the real-time and the delayed stock quote services if they do not require payments. The results are shown in Table 1.

Table 1. Results of the Performance Experiments

Task/Policy	time non-compliant	time compliant	time not-applicable
Maps API	0.69 s	0.68 s	0.60 s
Confidential Service	0.53 s	0.54 s	0.38 s
Policy search	0.35 s	0.34 s	n/a

Even with our prototypical policy engine the time required for performing policy checks are all well below 1 second. With further optimizations (e.g. caching formal representations of policies instead of parsing them again for every policy action), it seems feasible to integrate real-time compliance checking in a policy-aware composition tool.

7 Related Work

XACML is a widely-used industry standard for policies [17], but lacks a formal, declaratively defined semantics for its very extensive condition model, which includes XPath queries, string and date comparisons, arithmetic functions, logical negation and regular expressions besides others. Especially negation in combination with arbitrary XPath queries leads to undecidability of query containment. Another difference to our work is that XACML focuses specifically on access control policies, whereas our proposed policy language is suitable for usage control, which does not only check if initial access to data or services is allowed, but also restricts the ongoing usage afterwards.

WS-Policy provides a standard that can be used to specify policies that express requirements and capabilities in systems based on Web services [18]. The policy language itself is not especially targeted at Web services and can be extended by custom policy assertions, which are basic conditions that can be combined to form policies (similar to using our containers). The standard is based on XML and syntactic matching and is thus, in contrast to our approach, not suitable for heterogeneous environments where different vocabularies are mixed. There exist however several extensions to WS-Policy, which link assertions to OWL concepts (e.g. [19,20]). Such policies are thus based on description logics and therefore restricted to conditions with tree structures. The same restriction applies to KAoS, an early semantic policy framework [21]. Our approach uses conjunctive queries and thus can express non-tree conditions.

Accountability in RDF (AIR) is a policy language that comes with an engine that supports RDFS models, and an extensive justification framework [12]. It is based on N3 syntax, and supports quantified variables, as well as if-then-else statements. The "else" path is followed if the condition does not hold, which means that the language supports negation on non-atomic conditions. Therefore query containment on AIR policies is not decidable and thus the policy restrictions presented in this paper cannot be easily integrated into AIR. For future work it is certainly interesting to see which features of AIR and our policy language can be fruitfully combined. Especially interesting is an adaption of AIR's justification framework, for which we already laid out the foundation by associating policy rules with RIF metadata.

Another semantic policy language is Protune [22]. It is based on logic programming rules, including negation. Its main focus is not on the classification of situations, but on trust negotiation, which includes the execution of actions. It includes the explanation facility ProtuneX [23], which supports decision justifications and different kind of policy queries, such as how-to queries that tell a user what is needed to fulfill a policy. None of these queries can however be integrated into the conditions of other policies, which is a key feature of our policy language. Bonatti and Mogavero present a restricted version of Protune (e.g. no negation) for which they show decidability of policy comparison, i.e. query containment [24]. Their work does not support integration of the comparisons into policy conditions, and does not treat the "sufficient for compliance" restriction introduced in this paper.

8 Conclusions and Future Work

We presented a policy language with the novel capability to express restrictions on other policies given in the same language. The policy language has formal semantics defined in terms of conjunctive queries over RDFS data. Furthermore we described a concrete representation format being based on the W3C standards RDF and RIF. We motivated the need for our self-policing policy language by a use case about composed documents, including a real-world service, namely the Google Maps API.

We implemented a policy engine for our language and used it to model the use case. We conducted first performance measurements. The results show that the language and engine can effectively represent the required policies. As next steps we plan to develop a justification framework for the language and based thereon build a policy-aware composition tool.

Furthermore we plan to extend the expressivity of the policy language in one of the following possible directions:

– a more expressive data model, i.e. using one of the OWL 2 profiles, instead of RDFS,
– allow some limited negation (e.g. only on basic patterns),
– allow viral policies, in the meaning that restricting policies can also include conditions using the special policy restriction properties.

We currently evaluate, which of these extensions are most desirable in terms of required expressivity and preservation of decidability.

Acknowledgments. The authors wish to thank Markus Krötzsch and Andreas Harth for the useful discussions. This work was supported by the European project SOA4All.

References

1. Lampson, B.W.: Protection. In: Proc. Fifth Princeton Symposium on Information Sciences and Systems, pp. 437–443. Princeton University, Princeton (March 1971); reprinted in Operating Systems Review 8 (1), 18 – 24 (January 1974)
2. Dodds, L.: Rights Statements on the Web of Data. Nodalities Magazine (9) (2010), `http://www.talis.com/nodalities/pdf/nodalities_issue9.pdf`
3. Seneviratne, O., Kagal, L., Berners-Lee, T.: Policy Aware Content Reuse on the Web. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 553–568. Springer, Heidelberg (2009)
4. W3C: RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation (2004), `http://www.w3.org/TR/rdf-schema/`
5. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, vol. II. Computer Science Press, Rockville (1989)
6. Kephart, J.O., Walsh, W.E.: An Artificial Intelligence Perspective on Autonomic Computing Policies. In: IEEE Workshop on Policies for Distributed Systems and Networks, POLICY (2004)
7. Russel, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd edn. Prentice Hall, Englewood Cliffs (2003)
8. W3C: RIF Core Dialect. W3C Recommendation (2010), `http://www.w3.org/TR/rif-core/`
9. W3C: RIF RDF and OWL Compatibility. W3C Recommendation (2010), `http://www.w3.org/TR/rif-rdf-owl/`
10. Polleres, A., Feier, C., Harth, A.: Rules with Contextually Scoped Negation. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 332–347. Springer, Heidelberg (2006)

11. Abelson, H., Adida, B., Linksvayer, M., Yergler, N.: ccREL: The Creative Commons Rights Expression Language. W3C Submission (2008)
12. Kagal, L., Hanson, C., Weitzner, D.: Using Dependency Tracking to Provide Explanations for Policy Management. In: IEEE Workshop on Policies for Distributed Systems and Networks, POLICY (2008)
13. W3C: RIF Basic Logic Dialect. W3C Recommendation (2010), `http://www.w3.org/TR/rif-bld/`
14. Gottlob, G., Leone, N., Scarcello, F.: The complexity of acyclic conjunctive queries. Journal of the ACM 48(3), 431–498 (2001)
15. Goodman, N., Shmueli, O.: Tree queries: a simple class of relational queries. ACM Trans. Database Syst. 7(4), 653–677 (1982)
16. Chandra, A.K., Merlin, P.M.: Optimal implementation of conjunctive queries in relational data bases. In: Annual ACM Symposium on Theory of Computing (1977)
17. OASIS: eXtensible Access Control Markup Language (XACML) Version 2.0. OASIS Standard (2005), `http://docs.oasis-open.org/xacml/2.0/`
18. W3C: Web Services Policy 1.5 - Framework. W3C Recommendation (2007), `http://www.w3.org/TR/ws-policy/`
19. Verma, K., Akkiraju, R., Goodwin, R.: Semantic matching of web service policies. In: Semantic and Dynamic Web Processes (SDWP) In Conjunction with the Third International Conference on Web Services, ICWS 2005 (2005)
20. Kolovski, V., Parsia, B.: WS-Policy and Beyond: Application of OWL Defaults to Web Service Policies. In: Semantic Web Policy Workshop (SWPW) at 5th International Semantic Web Conference, ISWC (2006)
21. Uszok, A., Bradshaw, J., Jeffers, R., Suri, N., Hayes, P., Breedy, M., Bunch, L., Johnson, M., Kulkarni, S., Lott, J.: KAoS Policy and Domain Services: Toward a Description-Logic Approach to Policy Representation, Deconfliction, and Enforcement. In: IEEE Workshop on Policies for Distributed Systems and Networks, POLICY (2003)
22. Bonatti, P.A., De Coi, J.L., Olmedilla, D., Sauro, L.: A Rule-based Trust Negotiation System. IEEE Transactions on Knowledge and Data Engineering (2010)
23. Bonatti, P.A., Olmedilla, D., Peer, J.: Advanced Policy Explanations on the Web. In: European Conference on Artificial Intelligence, ECAI (2006)
24. Bonatti, P.A., Mogavero, F.: Comparing Rule-Based Policies. In: IEEE Workshop on Policies for Distributed Systems and Networks, POLICY (2008)