GUCON: A Generic Graph Pattern based Policy Framework for Usage Control Enforcement

Ines Akaichi[1][0000−0002−6020−5572], Giorgos Flouris[2][0000−0002−8937−4118], Irini Fundulaki[2][0000−0002−4812−9896], and Sabrina Kirrane[1][0000−0002−6955−7718]

[1] Institute for Information Systems and New Media, WU, Vienna, Austria
name.lastname@wu.ac.at
[2] Institute of Computer Science, FORTH, Heraklion, Greece
{fgeo,fundul}@ics.forth.gr

Abstract. Robust Usage Control (UC) mechanisms are necessary to protect sensitive data and resources, especially when these are distributed across multiple nodes or users. Existing solutions have limitations in expressing and enforcing usage control policies due to difficulties in capturing complex requirements and the lack of formal semantics necessary for automated compliance checking. To address these challenges, we propose GUCON, a generic policy framework that allows for the expression of and reasoning over granular UC policies. This is achieved by leveraging the expressiveness and semantics of graph pattern expressions, as well as the flexibility of deontic concepts. Additionally, GUCON incorporates algorithms for conflict detection, resolution, compliance and requirements checking, ensuring active policy enforcement. We demonstrate the effectiveness of our framework by proposing instantiations using SHACL, OWL and ODRL. We show how instantiations provide a bridge between abstract formalism and concrete implementations, thus allowing existing reasoners and implementations to be leveraged.

Keywords: Usage Control · Policy · Deontic Rules · Reasoning · Enforcement.

1 Introduction

In emerging decentralized environments, such as data spaces, or social linked data[1], the distribution of data across multiple nodes and its accessibility to numerous users raises various concerns. These concerns encompass issues related to privacy, especially in relation to the sharing of personal data; questions surrounding ownership and control of data concerning creative digital work; and notably, the risk of data misuse for purposes that deviate from the original intent. To address these concerns, policy-based UC emerges as an extension of access control, providing a technical instrument to manage not only resource access in terms of permissions and prohibitions but also future data usage defined by obligations and dispensations. UC serves as an umbrella term encompassing access control, data privacy protection, copyright, and various legislative and institutional policies (e.g., the General Data Protection Regulation (GDPR)[12] and the new copyright legislation [13]).

In order to specify UC policies, several policy languages and frameworks have been proposed, focusing on access control, privacy, or trust management, such as Rei [17], Protune [3], KAoS [26], and the SPECIAL policy language [5]. These languages are designed to meet the specific requirements of their respective domain areas. However, most standard and well-known languages, such as the Open Digital Rights Language (ODRL)[27] lack formal semantics for specifying or configuring enforcement mechanisms or ensuring policy adherence. Although there are languages with formal semantics that address reasoning over specific deontic concepts, they may not cover the complete range of required concepts in UC [7, 21, 16]. Furthermore, in general, there is a lack of full support for policy-specific tasks such as compliance checking,

[1] Social Linked Data: https://solidproject.org/

consistency checking, or requirement checking [18]. Notably, the presence of diverse policy languages can also lead to interoperability issues in distributed environments.

In this paper, we propose GUCON, a Generic Graph Pattern based Policy Framework for Usage Control enforcement. GUCON introduces an abstract structure with formal and implementable semantics for policy specification, and describes algorithms for policy-specific reasoning tasks. Policies are specified using conditional deontic rules based on graph patterns and deontic concepts (permissions, prohibitions, obligations, and dispensations), offering flexibility in expressing general UC restrictions, while the formalization of policy rules is based on the formal semantics of graph patterns [22]. In GUCON, we also introduce the concept of state of affairs, which captures domain knowledge and events, and serves as the basis for reasoning about, and enforcing UC policies. Using GUCON, diverse types of usage control policies can be explicitly defined and effectively enforced, owing to the adaptable nature of graph patterns and the formal implementable semantics that underlie the framework. To assess its usefulness and effectiveness, we demonstrate how to instantiate GUCON using recognized recommendations from the World Wide Web Consortium (W3C), namely the Shape Constraint Language (SHACL)[2], the Web Ontology Language 2 (OWL 2)[3], and ODRL[4]. This enables us to leverage their existing implementations and bridge the gap between GUCON's abstract formalism and concrete implementations.

The remainder of the paper is structured as follows: Section 2 discusses related work. Section 3 presents the necessary background. Section 4 introduces the building components of our framework specification, covering policies and the state of affairs and their semantics, while Section 5 describes algorithms for policy-specific tasks, namely compliance, consistency, and requirements checking. In Section 6, we demonstrate the usefulness and effectiveness of our framework by instantiating it using SHACL, OWL 2, and ODRL. Finally, Section 7 summarizes the paper and discusses future work.

2 Related Work

Several policy languages/models have been proposed to address UC. UCON [21] is an abstract model that extends access control with the concepts of decision continuity and attribute mutability. Although several formalisms have been suggested for UCON, and attempts have been made to include it in standard representation languages [19, 15, 9], there is currently no established reference or standard policy specification and implementation for UCON. As a result, UCON has not gained widespread adoption as a UC model in the industry. Another language, the Obligation Specification Language (OSL) formalized in Z [16], is utilized to express mainly conditional prohibitions and obligations but lacks support for dispensations. While OSL offers technical models for policy enforcement of obligations, it lacks provisions for consistency checking or policy querying. DUPO [7] is a policy language that employs defeasible logic to express permissions, prohibitions, and obligations. The language facilitates policy comparison by matching user requests for data access against DUPO polices, resulting in either permitting or prohibiting access. Additionally, the language supports consistency checking. However, reasoning over obligations is not explicitly defined in the current understanding of DUPO.

In the realm of the semantic web community, researchers have put forth various general policy languages and frameworks, including Rei [17], Protune [3], and KAoS [26]. These languages, which are grounded in knowledge representation languages, primarily concentrate on specifying and reasoning about access control and trust management, rather than UC. Specifically, the primary focus of these works is on

[2] SHACL, `https://www.w3.org/TR/shac`
[3] OWL 2, https://www.w3.org/TR/owl2-prim
[4] ODRL, `https://www.w3.org/TR/odrl-model`

permission checking, while aspects such as compliance checking with regard to obligations and dispensations are not addressed. Furthermore, as noted in [6]'s analysis, these languages may encounter challenges related to undecidability in certain policy-related tasks. Among the existing proposals, the most closely related to our work from a specification point of view is AIR [18], a language designed to facilitate accountable privacy protection in web-based information systems. AIR employs rules and graph patterns represented in N3. However, it should be noted that AIR was not specifically devised for UC, lacks deontic concepts, and does not encompass consistency checking or policy querying as integral components.

More recent studies proposed policy languages tailored to privacy, such as the SPECIAL policy language, which was specifically designed to facilitate privacy policies by utilizing decidable fragments of OWL [5]. However, this language primarily focuses on expressing authorizations and is constrained by the requirements imposed by GDPR. Another notable language is ODRL [27], which is based on a rich RDF model and vocabulary but lacks formal semantics.

3 Preliminaries

In order to specify the main components of our framework, we rely on the syntax and semantics of graph patterns expressions presented in [22]. Throughout the paper, we assume two pairwise disjoint and infinite sets I and L to denote respectively Internationalized Resource Identifiers (IRIs) and literals. We denote by T the union of $I \cup L$. We introduce the concepts of subject `s`, property `p`, and object `o` to form `subject-property-object` expressions, called Resource Description Framework (RDF) triples. A triple is defined as $(s, p, o) \in (I) \times (I) \times (I \cup L)$. Note that blank nodes are omitted for simplicity. A set of RDF triples form an *RDF graph.* We assume additionally the existence of an infinite set V of variables disjoint from the above sets. As a notational convention, we will prefix variables with"?"(e.g., ?x, ?y).

Syntax of Graph Patterns. The definition of graph pattern expressions is based on triple patterns. A triple pattern is defined as $(sp, pp, op) \in (I \cup V) \times (I \cup V) \times (I \cup L \cup V)$. The variables occurring in a graph pattern G are denoted as $\text{var}(G)$.

Definition 1 (Graph Pattern). *A graph pattern is defined recursively as follows:*

- *A triple pattern is a graph pattern.*
- *If $G1$ and $G2$ are graph patterns, then (G1 AND G2), (G1 OPT G2), (G1 UNION G1), (G1 MINUS G2) are graph patterns.*
- *If G is a graph pattern and R is a filter expression, then (G FILTER R) is a graph pattern. A Filter expression is constructed using elements of the sets $I \cup L \cup V$, logical connectives $(\neg, \wedge, \vee)$, inequality symbols $(<, \leq, \geq, >)$, equality symbol $(=)$, plus other features (see [23] for a complete list).*

Semantics of Graph Patterns. The semantics of graph pattern expressions are defined based on a mapping function μ, such as $\mu : V \to T$. For a triple pattern t, we denote by $\mu(t)$ the triple obtained by replacing the variables in t according to μ. The domain of μ, $dom(\mu)$, is the subset of V where μ is defined.

Definition 2 (Evaluation of a Graph Pattern). *Let D be an RDF graph over T. Mapping a graph pattern against D is defined using the function $[[.]]_D$, which takes a graph pattern expression and returns a set of mappings Ω.*

Two mappings μ_1 and μ_2 are said to be compatible, i.e., $\mu_1 \sim \mu_2$ when, for all $x \in dom(\mu_1) \cap dom(\mu_2)$, it is the case that $\mu_1(x) = \mu_2(x)$. The evaluation of a compound graph pattern $G_1 \times G_2$ is defined as follows:
$[[G_1 \text{ AND } G_2]] = \Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \,|\, \mu_1 \in \Omega_1, \mu_2 \in \Omega_2,\, \mu_1 \sim \mu_2\}$
$[[G_1 \text{ UNION } G_2]] = \Omega_1 \cup \Omega_2 = \{\mu_1 \cup \mu_2 \,|\, \mu_1 \in \Omega_1 \cup \mu_2 \in \Omega_2,\, \mu_1 \sim \mu_2\}$
$[[G_1 \text{ OPT } G_2]] = \Omega_1 ⟕ \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 / \Omega_2)$, where
$\Omega_1/\Omega_2 = \{\mu | \mu \in \Omega_1 \wedge \nexists \mu' \in \Omega_2, \mu \sim \mu'\}$

4 Usage Control Framework Specification

In this section, we provide an in-depth analysis of the fundamental constituents of our framework, namely knowledge bases (KBs) and usage control policies (UCPs). We also present a motivating use case scenario that guides our analysis and describes the address registration process in Austria.

4.1 Motivating Use Case Scenario

Assume the address registration process that exists in Austria, which is a legal requirement for all those that change their normal residence to Austria. The address registration process results in the issuing of a registration confirmation, which serves as official proof of residence, necessary for banking, voting, etc. Specifically, residents in Austria must register their address at a local registration office within three days when permanently changing residence or moving from a foreign country. This process requires completing a registration form with personal information and obtaining the signature of the property owner. In the event that a person permanently (or temporarily) stays in a hotel, they may request a signature from the hotel as proof of their stay. Temporary visitors for tourism purposes are exempt from this requirement. Failure to provide a registration confirmation may result in ineligibility to open a bank account or vote, among other things.

4.2 Specification of KBs and UCPs

In our framework, KBs are used to store domain events that have already occurred and general knowledge about the domain and related entities, while UCPs are used to specify imposed restrictions on resource usage. Below, we present a detailed syntactic specification of these components.

Knowledge Bases. We assume the existence of a certain set of triples that represent the current state of affairs or knowledge about the world, called the KB.

Definition 3 (Knowledge Base). *A KB, denoted as K, is an RDF graph describing the set of actual knowledge.*

A KB stores facts related to subjects, which can be either in the form of general knowledge (e.g., *Alice is a person*) or events that have occurred (e.g., *Alice registered her address*), which refer to the execution of an action.

Usage Control Policies. We assume three sets, N (entity names), C (action names), and R (resource names), such that N, C, $R \subseteq I$. We consider a special type of RDF triples, which are generated by the three sets N, C, and R. An RDF triple $(n, c, r) \in (N) \times (C) \times (R)$ is called an *action*. Let us further suppose the presence of disjoint sets V_n, V_c, V_r representing variables from the sets N, C, R respectively, such that V_n, V_c, $V_r \subseteq V$. A UCP consists of a set of rules that govern the behavior of entities with regard to resource usage, specifying what actions are permitted, prohibited, required, or optional. Deontic logic, which deals with permissions, obligations, and related concepts, provides a suitable framework for representing and reasoning about UCPs [10]. As part of our framework, the specification of usage control rules (UCRs) is based on the following deontic operators: permissions or allowance $\mathbf{A}$, prohibitions $\mathbf{P}$, obligations $\mathbf{O}$, and dispensations $\mathbf{D}$ that denote optionality [11]. In the context of an action (n, c, r), these operators have the following meanings:

- $\mathbf{A}(n, c, r)$: indicates that an entity n is permitted (allowed) to perform an action c over a resource r.
- $\mathbf{P}(n, c, r)$: indicates that an entity n is prohibited from performing an action c over a resource r.

- $\mathbf{O}(n, c, r)$: indicates that an entity n is obliged to perform an action c over a resource r.
- $\mathbf{D}(n, c, r)$: indicates that an entity n is exempt from performing an action c over a resource r.

In practice, we will allow variables to be present in the n, c, r positions (e.g., *?entity :request ?resource*), to allow generally applicable restrictions to be expressed, and we refer to such a tuple as an *action pattern* defined as a triple $(np, cp, rp) \in (N \cup V_n) \times (C \cup V_c) \times (R \cup V_r)$. We denote by $\mathcal{AP}$ the set of all action patterns.

A deontic pattern can be defined as follows:

Definition 4 (Deontic Pattern). *Let* $\mathcal{D} = \{\mathbf{A}, \mathbf{P}, \mathbf{O}, \mathbf{D}\}$ *denote the* deontic operators *of* Permission*,* Prohibition*,* Obligation*, and* Dispensation*. A* deontic pattern *is a statement of the form da, where* $d \in \mathcal{D}$ *and* $a \in \mathcal{AP}$*.*

Example 1 (Deontic Pattern). The tuple $\mathbf{A}(?x, :request, ?y)$ states that an "entity" $?x$ is allowed to request a "resource" $?y$.

Some deontic patterns apply under specific conditions, giving rise to conditional deontic rules [2] (e.g, *to be "allowed" to "request" a resource "signature", one needs to be a "person" staying permanently in a hotel, etc.*). These rules not only prescribe the permissible or impermissible actions that entities may undertake with resources, but also the corresponding obligations or dispensations, under specific conditions. A *condition* is modeled based on a graph pattern expression. Following Definition 1, AND (often abbreviated using ".") and UNION are used to express conjunctive and disjunctive (respectively) conditions. The operator OPT behaves similarly to the outer join operator in SQL, whereas MINUS is used to express conditions involving negation (e.g., to identify persons without a registration confirmation one could write $(?x, :type, :Person)$ `MINUS` $(?x, :hasA, :registrationConfirmation)$). Lastly, the filter operator is used to express different conditions pertaining to specific sub-elements of a triple. Using a deontic pattern, graph pattern, and the operator $\rightsquigarrow$ in-between, a conditional deontic rule, simply called a *UCR*, can be defined as follows:

Definition 5 (Usage Control Rule). *A UCR is of the form: cond* $\rightsquigarrow$ *da, where cond is a graph pattern, and da is a deontic pattern. We denote by* $\mathcal{R}$ *the set of all UCRs.*

A UCR can be read as follows: If the condition (*cond*) is satisifed by the KB, then the deontic pattern (*da*) may/must not/must/need not be satisfied. Furthermore, it is assumed that for a given rule, the condition $var(a) \subseteq var(cond)$ holds. Violating this restriction would create an infinite number of deontic requirements, making the model unusable in practice.

Example 2 (Usage Control Rule). The following requirement from our motivating use case scenario: *"In the event that a person permanently stays in a hotel, they may request a signature from the hotel as proof of their stay"*, can be expressed using the following UCR:

```
(?x, :type, :Person).                 (?x, :stayIn, ?l).
(?x, :hasStayDuration, :permanent).   (?l, :type, :Hotel).
(?l, :hasManagementUnit, ?m).         (?m, :type, :HotelManagement).
(?y, :hasSignatory, ?m).              (?y, :type, :Signature)
```
$\rightsquigarrow\mathbf{A}$(?x, :request, ?y)

The requirements described in our scenario can be expressed in the same general form in Definition 5. The various requirements expressed as UCRs form a UCP.

Definition 6 (Usage Control Policy). *A set of UCRs* $R \subseteq \mathcal{R}$ *is called a UCP.*

4.3 Semantics of UCPs

To effectively perform the reasoning tasks in our framework, it is imperative that we have the ability to reason about the rules governing UCPs. This is primarily accomplished through the process of evaluating these rules against the KB, resulting in the identification of *active rules*. Active rules are characterized by having a *satisfied condition*, ensuring their applicability in the given KB.

Definition 7 (Satisfied condition). *Let K be a KB over T, P a UCP, and a rule $r \in P$, such that $r = cond \rightsquigarrow da$. A condition cond is satisfied for μ, denoted by $K \rhd cond$, if and only if there exists a mapping μ such that $\mu \in [[cond]]_K$.*

Note that multiple mappings may be used to satisfy a given rule.

Definition 8 (Active Rule). *A rule $r \in P$, such that $r = cond \rightsquigarrow da$, is active for a mapping μ, if and only if cond is satisfied for μ.*

Note that a rule may be active for multiple mappings. Based on the definition of an active rule, a UCP P is called *active* if any of its rules are active. Otherwise, It is called *inactive*. Furthermore, it is often important to identify the entity/entities for which a rule applies:

Definition 9 (Applicable Rule). *Consider an active rule $r \in P$ such that $r = cond \rightsquigarrow da$, a denotes (n, c, r), $n \in V_n \cup N$, and there is some $n_o \in N$. We say that r is applicable for n_0 with respect to μ if and only if $\mu(n) = n_0$.*

Based on this definition, we call a UCP P *applicable* for n_0 if any of its rules are applicable. Otherwise, It is called *non-applicable*.

5 Reasoning Tasks

Our UC framework leverages KBs and UCPs to support three primary tasks for managing and monitoring UCPs: *consistency checking*, *compliance checking*, and *requirements checking*. Consistency checking ensures that policies do not contain conflicting or contradictory rules and are logically consistent. Compliance checking aims to verify that the actions of a system and its users, which are stored in KBs, conform to a predefined set of rules and policies, and to identify any instances of non-compliance. Finally, requirements checking helps to ensure that users of systems are informed of their up-to-date rights and obligations specified in policies. In the following, we provide a detailed overview of each reasoning task, related algorithms[5], and concepts.

5.1 Consistency Checking

Consistency checking aims to identify and resolve conflicts among policy rules, with a focus on detecting and resolving *conflicting rules* involving *deontic dilemmas*, which are considered to be application-independent [20]. Deontic dilemmas occur when positive operators (permissions and obligations) in an active rule and negative operators (prohibitions and dispensations) in another active rule refer to the same action triple a. Abstracting from the interplay between positive and negative operators, we mainly focus on the following types of deontic dilemmas defined in [20]: $\mathbf{O}a$ and $\mathbf{P}a$; $\mathbf{A}a$ and $\mathbf{P}a$; $\mathbf{O}a$ and $\mathbf{D}a$.

Definition 10 (Conflicting Rules). *Let $r_1, r_2 \in P$, such that $r_1 = cond_1 \rightsquigarrow d_1a_1$, $r_2 = cond_2 \rightsquigarrow d_2a_2$. Let μ_1 be a mapping in $[[cond_1]]_K$, μ_2 be a mapping in $[[cond_2]]_K$, and $\mu_1 \sim \mu_2$. We say that a pair of rules (r_1, r_2) are conflicting with respect to μ, such that $\mu = \mu_1 \bowtie \mu_2$, if and only if the following conditiofns hold:*

[5] The functions invoked in the algorithms defined below are available here: `https://github.com/Ines-Akaichi/GUCON-Instantiation/blob/main/GUCON-Appendix.pdf`

```
Input: Policy $P$, Knowledge Base $K$, Meta-policy $MP$
Output: Policy $\hat{P}$
begin
    /* $n$ is the size of P */
    for $i$=1 to $n$ do
        $\Omega$ = GetMappings($P[i].cond$, $K$)
        if $\Omega$ is not empty then
            foreach $\mu$ in $\Omega$ do
                $\hat{P}$.insert ($\mu(P[i])$)
    if $\hat{P}$ is not empty then
        for $i$=1 to $n$-1 do
            for $j$=$i$+1 to $n$ do
                /* If two rules are conflicting */
                if ($\hat{P}[i].a == \hat{P}[j].a$) and (IsOpposite($\hat{P}[i].d$, $\hat{P}[j].d$) then
                    temp = Compare($\hat{P}$[i], $\hat{P}$[j], MP)
                    if $temp == 1$ then
                        /* $\hat{P}$[j] $\preceq$ $\hat{P}$[i] */
                        /* $\hat{P}$[i] does not change */
                        $\hat{P}$[j].cond = Minus($\hat{P}$[j].cond, $\hat{P}$[i].cond)
                    else if $temp == 2$ then
                        /* $\hat{P}$[i] $\preceq$ $\hat{P}$[j] */
                        /* $\hat{P}$[j] does not change */
                        $\hat{P}$[i].cond = Minus($\hat{P}$[i].cond, $\hat{P}$[j].cond)
                    else if $temp==0$ then
                        /* $\hat{P}$[i] $\not\preceq$ $\hat{P}$[j] and $\hat{P}$[j] $\not\preceq$ $\hat{P}$[i] */
                        quit ();
    return $\hat{P}$
```

Algorithm 1: Dynamic Conflict Detection and Resolution

1. d_1 and d_2 present a deontic dilemma
2. $\mu(a_1) = \mu(a_2)$

We say that the pair (r_1, r_2) are conflicting if they are conflicting for some μ. We denote by $\mathcal{CR}$ the set of all conflicting rules in P.

When conflicts are detected in a policy, it is said to be *inconsistent*. In this case, a decision must be made on how to restore its consistency. A UCP P is called *consistent*, if and only if, there is no $r_1, r_2 \in P$ such that r_1 conflicts with r_2. Otherwise, It is called *inconsistent*.

Using a conflict resolution strategy automatically addresses policy conflicts and restores consistency. One prevalent strategy for resolving modality conflicts is establishing a precedence relationship between rules. Principles for establishing precedence include giving negative policies priority over positive ones, prioritizing specific policies over general ones, and prioritizing new laws over old ones. The specificities of each strategy are outlined in [20]. Our framework incorporates a meta-policy that defines the conflict strategy and the corresponding conflicting rules. Precedence is expressed using the binary operator $\preceq$. That is, for each conflicting pair of rules (r_1, r_2), if $r_1 \preceq r_2$, then r_2 (stronger rule) takes precedence over r_1 (weaker rule).

Definition 11 (Usage Control Meta-policy). *A usage control meta-policy is a tuple $\langle \mathcal{CR}, \preceq \rangle$, such that $\preceq \subseteq \mathcal{CR}^2$ is a partial pre-order over pairs of rules in $\mathcal{CR}$. For any pair of rules (r_1, r_2) and (r_2, r_3) in $\mathcal{CR}$, the relation $\preceq$ satisfies: a) reflexivity: $r_1 \preceq r_1$, i.e. every element is related to itself; b) transitivity: if $r_1 \preceq r_2$ and $r_2 \preceq r_3$ then $r_1 \preceq r_3$.*

Note also the use of $\preceq$ instead of $\prec$: as usual, we will write $r_1 \prec r_2$, as a shorthand for $r_1 \preceq r_2$ and $r_2 \not\preceq r_1$. An example of negative policies override positive ones can be expressed as follows:

Example 3 (Prohibition overrides Permission). This strategy can be formally expressed as follows: for any two rules $r_1 = cond_1 \rightsquigarrow \mathbf{P}a_1$, $r_2 = cond_2 \rightsquigarrow \mathbf{A}a_2$, such that $a_1 = a_2$, it holds that $r_2 \preceq r_1$.

```
Input: Policy $P$, Knowledge Base $K$, IRI $iri$
Output: Boolean $compliant$
begin
    $compliant = true$
    $i = 1$
    /* $n$ is the size of P */
    while $i \leq n$ do
        $\mu$ = GetMapping ($P[i].cond$, $K$, $iri$)
        if $\mu$ is not empty then
            if $P[i].d == \mathbf{O}$ then
                if Not Exists($\mu(P[i].a)$, $K$) then
                    $compliant = false$
                break
            if $P[i].d == \mathbf{P}$ then
                if Exists($\mu(P[i].a)$, $K$) then
                    $compliant = false$
                break
    return $compliant$
```

Algorithm 2: Ex-post Compliance checking

Generally speaking, repairing the consistency of policies involve detecting conflicting rules and then resolving them based on the precedence strategy specified in the corresponding meta-policy. To resolve these conflicts, changes are made to the weaker rule defined based on the precedence strategy specified in the corresponding meta-policy. Specifically, the resolution involves subtracting (using MINUS) the condition in the stronger rules from the condition in the weaker rule. This operation yields a modified version of the weaker rule that no longer conflicts with the stronger rule, allowing its application in the current context. In cases where no precedence is defined, conflicts remain unresolved. This means that conflicting rules without a designated precedence relationship would persist, necessitating manual intervention for resolution [20].
To automate the process of consistency checking and repairing in policies, we propose Algorithm 1, which is a systematic approach to detecting and resolving conflicting rules. The algorithm involves four main steps: (1) It determines an active policy based on a given KB by evoking the function `ReturnMappings` that evaluates each rule in the policy against a given KB and returns a set of mappings. The resulting mappings are used to populate the inactive rules. (2) Next, each pair of active rules in the returned policy is examined to identify any conflicts. This is done by checking whether the pair has equal actions (triples) and features a deontic dilemma, using the `IsOpposite` function. (3) If the algorithm detects any conflicting pairs, the function `Compare` is evoked to decide which rule holds precedence over which rule. (4) Finally, if a precedence is defined between two rules, the meta-policy is applied by invoking the function `Minus`, which apply necessary changes to the weaker rule. In general, the verification of active rules may be carried out each time the KB is updated.

5.2 Compliance Checking

We propose ex-post compliance of a given KB against a policy or a set of policies. Compliance checking is capable of identifying any breaches or non-compliant behavior of an entity, which is identified by an IRI, thereby ensuring the proper and secure operation of the system. The criteria for determining whether a KB is compliant with a given policy can vary depending on the specific rule being considered, as well as the KB and mappings used to interpret that rule. Note that we do not define compliance for inconsistent policies, however, if we are given any UCP, Algorithm 1 needs to be initially performed, and then compliance checking can be carried out.

Definition 12 (KB Compliance Against a Rule). *Given a KB K and a rule $r \in P$, such that P is consistent, we say that K complies with r, denoted by $K \rhd r$, if and only if any of the following is true:*

- *If r is of the form $cond \rightarrow \mathbf{A}a$, then $K \rhd r$*

```
Input: Policy $P$, Knowledge Base $K$, IRI $iri$, Deontic $d$
Output: Policy $P_{req}$
begin
    /* $n$ is the size of P */
    for $i = 1$ to $n$ do
        $\mu =$ GetMapping ($P[i].cond$, $K$, $iri$)
        if $\mu$ is not empty then
            if $P[i].d == d$ then
                if $d == \mathbf{O}$ then
                    if Exists($\mu(P[i].a)$, $K$) then
                        $i = i + 1$
                    else
                        $P_{req}$.insert($\mu(P[i])$)
                else
                    $P_{req}$.insert($\mu(P[i])$)
    return $P_{req}$
```

Algorithm 3: Requirements Checking

– *If r is of the form cond $\rightarrow \mathbf{P}a$ and for all μ such that r is applicable for μ, it holds that $\mu(a) \notin K$*
– *If r is of the form cond $\rightarrow \mathbf{O}a$ and for all μ such that r is applicable for μ, it holds that $\mu(a) \in K$*
– *If r is of the form cond $\rightarrow \mathbf{D}a$, then $K \rhd r$*

Based on this definition, a KB K is said to be *compliant* with a UCP P, denoted by $K \rhd P$, if and only for each applicable rule $r \in P$, $K \rhd r$. Otherwise, It is called *non-compliant.*

Automatically checking for compliance between a KB and an applicable policy, with respect to a given IRI, is performed using Algorithm 2, which employs a two-step approach: (1) The first step of the algorithm involves verifying the applicability of each rule for a given IRI by invoking the `ReturnMapping` function. (2) The second step involves evaluating the KB for compliance with each applicable rule, utilizing the `Exists` function as per the criteria defined in Definition 12. Notably, the algorithm halts as soon as a non-compliant rule is identified.

5.3 Requirements Checking

Requirement checking is a task that enables an entity to query a policy and receive information regarding their applicable deontic rules. This task ensures that entities are continuously aware of the applicable permissions, prohibitions, obligations, and dispensations relevant to their activities.

Algorithm 3 facilitates the retrieval of specific requirements, in accordance with an applicable policy, for a given IRI and a KB. The algorithm operates through a two-step process: (1) In the first step, It verifies the applicability of each rule for a given IRI by invoking the `ReturnMapping` function. This step ensures that only applicable rules are considered in the subsequent requirement checking process. (2) In the second step, It checks each applicable rule to determine whether it is in accordance with the requested deontic, adding it to the list of requested requirements as appropriate. For requested obligations, the algorithm first checks whether a given obligation has already been executed by invoking the `Exists` function. If the obligation has not been executed, the respective rule will be added to the list of requirements.

6 Assessment

In order to demonstrate the effectiveness and suitability of our abstract framework in expressing and reasoning over UCPs in practical contexts, we provide instantiations of our framework using three widely recognized recommendations from the W3C

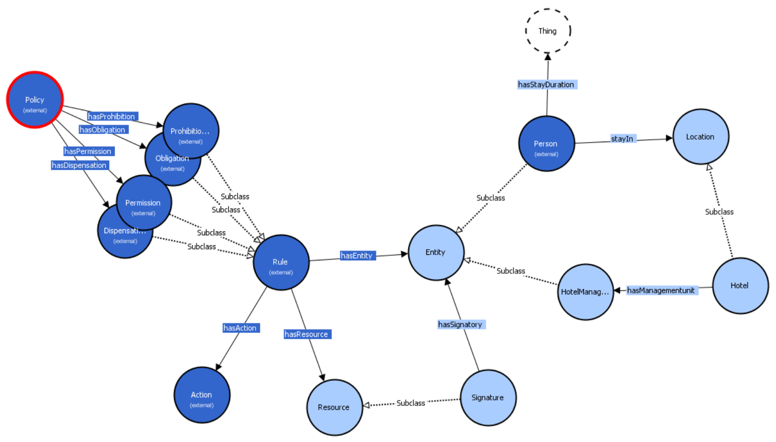

Fig. 1. The UCP Core Ontology and Profile. Dark blue: core model; Light blue: profile

organization, namely, SHACL, OWL 2, and ODRL. These instantiations allow us to showcase how can we effectively map the abstract concepts in our framework into concrete implementations.

In what follows, we first introduce the Usage Control Policy (UCP) ontology and profile, which respectively represent our framework specification and motivating use case scenario, and which serve as the basis for the instantiations performed below. Both the UCP core ontology and the UCP profile are employed in the instantiation process using SHACL and OWL. However, for the ODRL language, we express our motivating scenario by constructing an ODRL profile in accordance with the ODRL specification, which is described in detail below. Finally, we assess the adequacy of instantiations and reasoning capabilities of each of the languages. All the details describing the various instantiations are provided on our GitHub repository[6,7].

6.1 The Usage Control Policy Ontology and Profile

Herein, we present the UCP core ontology that defines essential concepts for modeling a UCP based on our framework specification. The core ontology is shown in dark blue in Figure 1. We use the `ucp` prefix to identify our `<http://example.org/ucp/>` ontology. The UCP core ontology includes the following main classes: `ucp:Policy`, `ucp:Rule`, `ucp:Action`, `ucp:Resource`, and `ucp:Entity`. A `ucp:Rule` can be a `ucp:Permission`, `ucp:Prohibition`, `ucp:Obligation`, or `ucp:Dispensation`. Connections between a `ucp:Rule` and an `ucp:Action`, `ucp:Resource`, and `ucp:Entity` are established using corresponding OWL properties. A policy can consist of one or more rules and is linked to the `ucp:Rule` class based on the rule type, using properties like `ucp:hasPermission`, ect. The ontology incorporates additional constraints and restrictions, but they are not discussed in details in this paper. For instance, `ucp:hasPermission` has a domain of `ucp:Policy` and a range of `ucp:Permission`. The deontic classes are also `owl:disjointWith` each other. We also introduce the UCP profile `<http://example.org/ucp:profile:01/>` described with the `ucpr` prefix, which includes the minimal concepts needed for modeling Example 2 from our motivating scenario, as shown in Figure 1 with an overview in light blue. It defines a `foaf:Person` of type `ucp:Entity` staying at a particular `ucpr:Location` for a specific period. `ucpr:Hotel` is of type `ucpr:Location` and has a `ucpr:HotelManagement` unit. The profile includes a `ucpr:request` as an instance of the `ucp:Action` class, and a `ucpr:Signature` as a `ucp:Resource` involving a signatory of type `ucp:Entity`.

[6] GitHub, https://github.com/Ines-Akaichi/GUCON-Instantiation

[7] The following prefixes are used throughout Section 6: rdf:`<http://www.w3.org/1999/02/22-rdf-syntax-ns#>`; rdfs:`<http://www.w3.org/2000/01/rdf-schema#>`; owl:`<http://www.w3.org/2002/07/owl#>`; foaf:`<http://xmlns.com/foaf/0.1/>`; ex:`<http://example.org/>`

6.2 Instantiation

Initially, SHACL was introduced to validate RDF graphs using specific conditions called *shapes*. However, it has evolved to include advanced features like *SHACL rules*, namely Triple and SPARQL rules. These rules allow for the derivation of inferred triples, expanding SHACL's capabilities beyond validation into a logical programming language [24]. On the other hand, OWL is an ontology language designed for the Semantic Web, with formally defined semantics. OWL 2 is the latest version, offering various options that balance expressive modeling capabilities with efficient computational reasoning [6]. In contrast, ODRL was developed to express policies for digital content and services. It includes a core model and vocabulary that enables the expression of profiles for different use cases [27].

Policy representation using SHACL. To express a UCR using SHACL-SPARQL, the rule is defined as a `sh:SPARQLRule` identifier, bound to the special term `$this`. The UCR is expressed through the `sh:construct` property using CONSTRUCT-WHERE assertions in SPARQL. The deontic pattern is rewritten using the CONSTRUCT operator as $this rdfs:subClassOf *DeonticClass*, where *DeonticClass* can be one of the following `ucp:Permission`, `ucp:Prohibition`, `ucp:Obligation`, or `ucp:Dispensation`. The entity, resource, and action are represented as a conjunction of triples in the WHERE clause, using the identifier $this as the subject; `ucp:hasEntity`, `ucp:hasAction`, `ucp:hasResource` as properties; and the corresponding values as objects. Graph pattern conditions can be added to the WHERE clause alongside the other triples, using an AND connector. Using SHACL-SPARQL rules preserves the expressivity of graph pattern conditions, allowing for more flexible and detailed rule specifications. Example 2 formalized in SHACL is shown in Listing 1.1.

```
ex:PermissionRequestSignature
  a sh:NodeShape ;
  sh:rule [
    a sh:SPARQLRule ;
    sh:prefixes ex: , rdf: , ucp: , :ucpr;
    sh:construct """
      CONSTRUCT {
            $this rdfs:subClassOf ucp:Permission . }
      WHERE {
            $this ucp:hasAction ?x .
            $this  ucp:hasEntity ?y .
            $this ucp:hasResource ?z .
            ?x rdf:type  ucp:Action .
            ?x rdfs:label "request"@en .
            ?y rdf:type  foaf:Person .
            ?y ucpr:stayIn ?l.
            ?y ucpr:hasStayDuration ex:permanent.
            ?l rdf:type ucpr:Hotel .
            ?l ucpr:hasMangemetUnit ?m .
            ?m rdf:type   ucpr:HotelManagement .
            ?z rdf:type  ex:Signature .
            ?z ucpr:hasSignatory ?m . } """ ; ] .
```

Listing 1.1. Policy Representation using SHACL

Policy representation using OWL. To represent a UCR using OWL, the rule can be described as follows: the deontic pattern of the rule is represented by an identifier (denoted by *id*) of type `owl:Class`, which is generated in a way that guarantees uniqueness. The deontic operator is conveyed through the RDF triple representation as *id*, rdfs:subClassOf, *DeonticClass*. *id* describes the entity, resource, and action of the deontic pattern using `owl:equivalentClass` and `owl:intersectionOf` of restrictions on the properties `ucp:hasEntity`, `ucp:hasAction`, and `ucp:hasResource`. These properties can be recursively described using `owl:hasValue` or `owl:allValuesFrom`. Mapping specific instances of type `ucp:Entity`, `ucp:Resource`, or `ucp:Action` described using the graph pattern operators AND, UNION, OPT, FILTER, or MINUS to the OWL representation can be performed as follows: the `owl:intersectionOf` and `owl:unionOf` can be used to express respectively the operators AND and UNION. The `owl:complementOf` property serves as a workaround to express OPT, while also enabling the expression of soft negation to describe MINUS operations. It is important to note that OWL does not explicitly represent the FILTER operator. Nevertheless, constraints on properties and values can be expressed through OWL restrictions and property assertions. Example 2 formalized in OWL is shown in Listing 1.2.

```
ex:PermissionRequestSignature rdf:type owl:Class ;
rdfs:subClassOf  ucp:Permission ;
owl:equivantClass
[
rdf:type     owl:Class;
owl:intersectionOf
( [
rdf:type  owl:Restriction ;
owl:onProperty    ucp:hasAction ;
owl:hasValue ucpr:request  ]
[
rdf:type owl:Restriction;
owl:onProperty ucp:hasResource;
owl:allValuesFrom  [
  rdf:type ucpr:Signature ;
  ucpr:signatory  ucpr:HotelManagement ] ]
[
rdf:type owl:Restricton;
owl:onProperty ucp:hasEntity;
owl:allValuesFrom  [
  rdf:type  foaf:Person ;
  ucpr:stayIn [
    rdf:type ucpr:Hotel ;
    ucpr:hasManagementUnit  ucpr:HotelMangement; ] ;
        ucpr:stayDuration ex:permanent ; ] ] ) ] .
```

Listing 1.2. Policy Representation using OWL

Policy representation using ODRL. To address the requirements specific to our motivating use case scenario, the first step involves extending the ODRL policy model by creating a profile that represents our particular example. The following prefixes are used to describe respectively the original model odrl:`<http://www.w3.org/ns/odrl/2/>` and our profile odrlp:`<http://example.org/odrl:profile:01/>`. The profile follows the design guidlines of the ODRL model, particularly, we employ an instance `odrlp:persons` of type `odrl:PartyCollection`, while the class `foaf:Person` is part of `odrlp:persons`. `odrlp:Signature` is a subclass of `odrl:Asset`, the action `odrlp:request` is an instance of `odrl:Action`. The properties `odrlp:stayIn`, `odrlp:hasManagementUnit`, `odrlp:hasStayduration`, and `odrlp:hasSignatory` are defined similarly to the UCP profile. A more comprehensive description of the ODRL profile can be found in our GitHub repository.

Within the ODRL, a UCR can be described using an identifier *id* of type `odrl:Set`, which is generated in a way that guarantees uniqueness. The deontic pattern of a UCR is described as follows: the deontic operator is specified through an ODRL deontic property (e.g., `odrl:permission`), although it is important to note that ODRL mainly covers permissions, prohibitions, and obligations. That means, It is not possible to represent a UCR expressing a dispensation in ODRL without creating a new profile. The deontic property is linked to other concepts through properties like `odrl:assignee` (corresponding to ucp:hasEntity), `odrl:action` (corresponding to ucp:hasAction), and `odrl:target` (corresponding to ucp:hasResource). Describing the entity, action, and resource can be done using operators defined in ODRL (e.g., `odrl:refinement`, `odrl:leftOperand`, `odrl:rightOperand`, `odrl:operator`). Mapping specific instances of type `ucp:Entity`, `ucp:Resource`, or `ucp:Action` described using the graph pattern operators AND, UNION, OPT, FILTER, or MINUS to the ODRL representation can be performed as follows: the intersection or union of patterns can be expressed using the operators `odrl:and` and `odrl:or`, respectively. However, defining the OPT operator within the ODRL framework does not have a clear workaround. For the FILTER operator, the properties `odrl:leftOperand`, `odrl:rightOperand`, and `odrl:operator` can be used as a workaround. Negation can be achieved using operators that belong to the class `odrl:Operator`, such as `odrl:neq` (not equal) or `odrl:isNoneOf` (is none of). An ODRL formalization of Example 2 is found in Listing 1.3.

```
ex:PermissionRequestSignature
a odrl:Set;
odrl:profile    <http://example.org/odrl:profile:01/>;
odrl:permission
[
odrl:assignee
    [
    a  odrl:PartyCollection ;
    odrl:source odrlp:persons;
    odrl:refinement
    [
    odrl:and
    [ odrl:leftOperand    odrlp:stayIn    ;
```

```
        odrl:operator    odrl:eq    ;
        odrl:rightOperand
          [a odrlp:Hotel;
          odrlp:hasManagementUnit odrlp:HotelManagement; ]; ]   ,
      [ odrl:leftOperand    odrlp:hasStayDuration   ;
        odrl:operator   odrl:eq    ;
        odrl:rightOperand   ex:permanent   ; ] ; ]   ];
odrl:action odrlp:request;
odrl:target
      [ a odrlp:Signature;
      odrlp:hasSignatory odrlp:HotelManagement; ] ].
```

Listing 1.3. Policy Representation using ODRL

6.3 Usage Control Requirements Assessment

The implementation of our framework has demonstrated its versatility and adaptability by successfully mapping it to different languages. This is achieved through the strategic utilization of the inherent expressive capabilities present in each language. In the following, we specifically assess two key aspects: (1) the adequacy of mapping our framework to the different instantiations and (2) the reasoning over the given policies by leveraging existing implementations of the defined languages.

Instantiation: *Expressiveness.* UCRs can be mapped directly to SHACL-SPARQL. Additionally, SHACL-SPARQL leverages SPARQL operators and built-in functions, making it expressive for policy specification. OWL is expressive and offers features like conjunction, disjunction, and filtering through property restrictions. It also supports OPT and MINUS using the `owl:complementOf` construct. SHACL-SPARQL and OWL require an ontology to express policy elements. Whereas ODRL is built specifically to express policies by supporting mainly permissions, prohibitions, and obligations, but lacks support for dispensations. ODRL provides conjunction, disjunction, and refinement operators that act as filters for conditions. A form of “Negation” can be achieved using `odrl:neq` or `odrl:isNoneOf`. Whereas, It is not clear how ODRL can support the OPT operator. *Flexibility & extensibility.* SHACL-SPARQL and OWL offer flexibility and extensibility through ontologies. They can accommodate various requirements by defining new concepts and relationships in the ontology. Whereas, ODRL’s flexibility and extensibility are achieved through profiles, enabling customization of the language for specific application domains. *Unambiguous.* SHACL-SPARQL and OWL are declarative in syntax, promoting unambiguous policy specifications, while ODRL’s syntax is designed to be intuitive, which aids in reducing ambiguity to some extent. *Formal semantics.* SHACL-SPARQL and particularly SPARQL have clear formal semantics, making it well-defined and suitable for formal reasoning. OWL has formal semantics defined by W3C, enabling reasoning and inference, whereas ODRL lacks explicitly defined formal semantics. Nonetheless, there is an active w3c community group dedicated to defining formal semantics for ODRL[8]. Finally and most importantly, the adoption of graph patterns formal semantics into the specification of our policy rules provides a rigorous foundation for our framework, which enables precise reasoning about policy rule interactions, conflicts, and compliance, when combined with other representation languages.

Reasoning: One advantage of using representation languages like SHACL, OWL 2, and ODRL is their ability to leverage (and extend) existing engines for implementing our framework reasoning tasks. In the domains of regulatory and privacy compliance research, SHACL is used for representing privacy policies, i.e. permissions, prohibitions, and obligations [1, 24], with the TopBraid[9] engine being used to assess compliance of user access request or user data processing against the SHACL policies. Furthermore, and very recently, SHACL-ACL [25] has been introduced as an extension of SHACL, focusing on access control in RDF knowledge graphs. The validation process for access control involves checking whether SPARQL queries are compliant with access control policies expressed in SHACL-ACL. This validation is

[8] ODRL Formal Semantics, https://w3c.github.io/odrl/formal-semantics/

[9] TopBraid SHACL, https://github.com/TopQuadrant/shacl

carried out using a SHACL validator known as Trav-SHACL[10]. Various implementations of SHACL and its advanced features are available, and a full list of engines can be found here[11]. Similarly, recent works [5, 4, 14] have utilized OWL 2 to express privacy policies and employed off-the-shelf reasoners like Hermit, Pellet, and Racer for compliance checking. As noted in [5], OWL 2 exhibits the advantage that all major policy-reasoning tasks are decidable, and if policies adhere to OWL 2 profiles, they are also tractable. Although ODRL lacks a standard enforcement engine, workarounds have been proposed, such as translating ODRL policies into InstAL and perform compliance verification using an answer set solver [10]. Furthermore, the current efforts within a W3C working group to propose formal semantics for ODRL would represent a significant advancement towards enabling the establishment of a standard implementation for the language [8].

7 Conclusion & Future Work

In this paper, we presented GUCON, a comprehensive framework that defines the specifications of a KB designed to store factual knowledge, as well as UCPs used to define UCRs. Our framework leverages the flexibility of deontic concepts and the expressiveness and semantics of graph pattern expressions to capture general UCRs. In addition, we have introduced algorithms for policy-based reasoning tasks, mainly, consistency checking, compliance checking and requirements checking, which can be accomplished by leveraging the semantics of UCPs and the KB. To demonstrate the effectiveness of our framework, we showed how to instantiate our framework using three different well-known languages. In doing so, we demonstrated not only the expressive power of our framework, but also its adaptability to other relevant languages. This enables us to draw on existing literature and industry solutions supporting these languages as a means of implementing our reasoning tasks.

Our paper leaves room for future work. While the majority of existing literature and industrial solutions focus primarily on compliance checking as the main reasoning task, a potential avenue for future research is to use one of these many implementations to study the integration of the remaining reasoning tasks and bring in our formal semantics to these implementations, in particular consistency checking and requirement checking. In addition, We also plan to investigate how can we facilitate the mapping between our framework and the representation languages introduced herein by implementing and evaluating automated translation algorithms. Finally, we plan to evaluate our framework following its implementation based on performance and security criteria, among other aspects.

Acknowledgements This work is funded by the European Union Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 860801. Sabrina Kirrane is funded by the FWF Austrian Science Fund and the Internet Foundation Austria under the FWF Elise Richter and netidee SCIENCE programmes as project number V 759-N.

References

1. Al Bassit, A., Krasnashchok, K., Skhiri, S., Mustapha, M.: Policy-based automated compliance checking. In: Rules and Reasoning: 5th International Joint Conference, RuleML+RR 2021, Leuven, Belgium, September 13–15, 2021, Proceedings (2021)
2. Beller, S.: Deontic norms, deontic reasoning, and deontic conditionals. Thinking & Reasoning **14**(4) (2008)
3. Bonatti, P., De Coi, J.L., Olmedilla, D., Sauro, L.: A rule-based trust negotiation system. IEEE Transactions on Knowledge and Data Engineering **22** (2010)

[10] Trav-SHACL, https://github.com/SDM-TIB/Trav-SHACL

[11] https://book.validatingrdf.com/bookHtml011.html

4. Bonatti, P., Ioffredo, L., Petrova, I., Sauro, L., Siahaan, I.: Real-time reasoning in owl2 for gdpr compliance. Artificial Intelligence **289** (2020)
5. Bonatti, P., Kirrane, S., Petrova, I., Sauro, L.: Machine understandable policies and gdpr compliance checking. KI - Künstliche Intelligenz **34** (2020)
6. Bonatti, P.A.: Fast compliance checking in an owl2 fragment. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence (2018)
7. Cao, Q.H., Giyyarpuram, M., Farahbakhsh, R., Crespi, N.: Policy-based usage control for a trustworthy data sharing platform in smart cities. Future Generation Computer Systems **107** (2020)
8. Cimmino, A., Cano-Benito, J., García-Castro, R.: Practical challenges of odrl and potential courses of action. In: Companion Proceedings of the ACM Web Conference (2023)
9. Colombo, M., Lazouski, A., Martinelli, F., Mori, P.: A proposal on enhancing xacml with continuous usage control features. In: Grids, P2P and Services Computing (2010)
10. De Vos, M., Kirrane, S., Padget, J., Satoh, K.: Odrl policy modelling and compliance checking. In: Rules and Reasoning: Third International Joint Conference, RuleML+RR 2019, Bolzano, Italy, September 16–19, 2019, Proceedings (2019)
11. Dimishkovska, A.: Deontic logic and legal rules. Encyclopedia of the Philosophy of Law and Social Philosophy (2017)
12. European Commission: 2018 reform of eu data protection rules (2018), `https://ec.europa.eu/commission/sites/beta-political/files/data-protection-factsheet-changes_en.pdf`
13. European Commission: 2021 reform of eu copyright protection rules (2021), `https://ec.europa.eu/commission/presscorner/detail/en/IP_21_1807`
14. Francesconi, E., Governatori, G.: Patterns for legal compliance checking in a decidable framework of linked open data. Artificial Intelligence and Law **31**(3) (07 2022)
15. e Ghazia, U., Masood, R., Shibli, M.A., Bilal, M.: Usage control model specification in xacml policy language. In: Computer Information Systems and Industrial Management (2012)
16. Hilty, M., Pretschner, A., Basin, D., Schaefer, C., Walter, T.: A policy language for distributed usage control. In: Computer Security – ESORICS (2007)
17. Kagal, L.: Rei : A Policy Language for the Me-Centric Project. Tech. rep., HP Labs (September 2002), http://www.hpl.hp.com/techreports/2002/HPL-2002-270.html
18. Khandelwal, A., Bao, J., Kagal, L., Jacobi, I., Ding, L., Hendler, J.: Analyzing the air language: A semantic web (production) rule language. In: Web Reasoning and Rule Systems (2010)
19. Lazouski, A., Martinelli, F., Mori, P.: Usage control in computer security: A survey. Computer Science Review **4**(2) (2010)
20. Lupu, E., Sloman, M.: Conflicts in policy-based distributed systems management. IEEE Transactions on Software Engineering **25**(6) (1999)
21. Park, J., Sandhu, R.: The uconabc usage control model. ACM Transactions on Information and System Security **7** (2004)
22. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of sparql. In: The Semantic Web - ISWC 2006 (2006)
23. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. `https://www.w3.org/TR/rdf-sparql-query/` (2008), w3C Recommendation 15 January 2008
24. Robaldo, L., Batsakis, S., Calegari, R., et al.: Compliance checking on first-order knowledge with conflicting and compensatory norms: a comparison among currently available technologies. Artificial Intelligence and Law (2023)
25. Rohde, P.D., Iglesias, E., Vidal, M.E.: Shacl-acl: Access control with shacl. In: European Semantic Web Conference (2023)
26. 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: Proceedings POLICY 2003. IEEE 4th International Workshop on Policies for Distributed Systems and Networks (2003)
27. W3C Working Group: The open digital rights language (odrl). `https://www.w3.org/TR/odrl-model/` (2018)