Journal of Computer and Communications, 2026, 14(11), 104-134
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

Legal Rule for Reasoning and Explainability of Decision

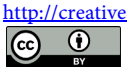
Jeremy Bouché-Pillon[1*], Pascale Zaraté[2], Yannick Chevalier[1], Nathalie Aussenac-Gilles[1]

[1]IRIT, University Toulouse, CNRS, Toulouse INP, Toulouse, France
[2]IRIT, University Toulouse, CNRS, Toulouse INP, University Toulouse Capitole, Toulouse, France
Email: *jeremy.bouche-pillon@irit.fr

How to cite this paper: Bouché-Pillon, J., Zaraté, P., Chevalier, Y. and Aussenac-Gilles, N. (2026) Legal Rule for Reasoning and Explainability of Decision. *Journal of Computer and Communications*, **14**, 104-134.
https://doi.org/10.4236/jcc.2026.1411005

Received: September 24, 2026
Accepted: November 15, 2026
Published: November 18, 2026

Copyright © 2026 by author(s) and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/



Open Access

Abstract

The emergence of the digital transition brought an increasing need to control the processing of digital information, including in Law Enforcement Agencies (LEAs). At the EU level, in recent years, many regulations have emerged to control data processing and exchange. Texts other than the GDPR, such as the "Law Enforcement Directive (LED)", appeared to regulate specifically how LEAs could process data. A formal representation of these regulations can be part of decision systems that support LEAs in processing data in compliance with the regulations. Although many new formalisms have emerged to represent legal norms and rules, few are provided with a reasoning mechanism. Furthermore, systems used in decision-making processes in critical contexts such as medical diagnoses or legal decisions cannot be fully automated, and the explainability of their results is essential to ensure user confidence in decisions. This explainability aspect, while crucial, is lacking in most modern approaches that rely on machine learning. This paper describes a framework to operate formal rules from regulations, by focusing on explainability of the decision. After describing the general architecture of the proposed decision support framework, the paper showcases how symbolic AI and the SPARQL query language can support legal reasoning. It then describes an algorithm to generate justification for the reasoning results and outlines the procedure to be followed when the reasoning does not lead to a satisfactory conclusion. We notably focus on a method based on decision trees to determine what additional information to request from the user.

Keywords

Decision-Support, Rule-Based, Decision Trees, Law Compliance, Explainability

1. Introduction

Digital technologies present enormous growth potential. They promise to provide better and more seamless service and market access to citizens and economic agents by replacing existing services with digital ones that implement best-of-class solutions, are interoperable, and support agents in their tasks. To be accepted, this digital transition[1] requires comprehensive governance to prevent the misuse of the introduced technology. Governance is often domain-specific and is defined through laws establishing guidelines and barriers on what can and cannot be done in Finance, e-Health, administration, etc. For example, while the GDPR[2] applies to all digital processing of EU citizens' personal data, the Digital Service Act (DSA)[3] applies more specifically to personal data processed in marketplaces and social networks.

These regulations force agents willing to transform their activities to prove that their processes comply with laws whose interpretation may vary. A solution is to introduce a Decision Support System (DSS) in which a qualified human agent is responsible for the description of the task at hand, and the system suggests a proper course of action to ensure compliance. To ensure that the human agent understands the legal implications, this advice should be supported with legal reasoning justification based on the existing regulations. Such a DSS requires the formalization of regulations and reasoning on this formalization, as well as appropriate communication with the human agent.

Building such a DSS is challenging, as individual components themselves need to solve complex problems, and the solutions provided need to be compatible across the system. For example, many formalisms have been proposed to represent legal norms and rules, the prime example being LegalRuleML [1] LKIF [2].

However, this formalization needs to be operational, and to that end needs to be accompanied by a working mechanism for reasoning on the rules. For instance, LegalRuleML did not originally provide mechanisms to reason over the rules and supplementary works had to be carried out to allow it [3]. The Carneades argumentation system [2], designed to work with LKIF rules, could not be used successfully when tested and doesn't appear to be maintained, which shows the need for a standard-based solution.

Another major issue, once the rules and the situation are described in a suitable logic, is the inability of pure logic reasoning mechanisms to explain the decision, only stating whether a decision is a logical consequence of its premises. That is, the pure reasoning should be complemented with a module that reflects upon the process that led to the result to provide the user with meaningful information.

Finally, the growing complexity of the relations between stored pieces of information led to represent them using a standard format from Semantic Web tech-

[1]Digital Transition - Reforms and Investments - European Union: https://reforms-investments.ec.europa.eu/technical-support-instrument-0/digital-transition_en
[2]Regulation - 2016/679 - EN - gdpr - EUR-Lex - European Union: https://eur-lex.europa.eu/eli/reg/2016/679/oj/eng
[3]Regulation - 2022/2065 - EN - DSA - EUR-Lex: https://eur-lex.europa.eu/eli/reg/2022/2065/oj/eng

nologies.

Instead of relying *ad hoc* relations stored in relational database tables, this framework promotes the use of well-defined entities and binary relations from standardized ontologies to store information in Knowledge Graphs (KG) as defined in [4] [5], and provides the ability to query these graphs using standard query languages that are broadly supported with tools.

Based on all these observations, we propose to support decision making when one has to check the conformance of a situation to a set of rules describing one or more regulations. In this paper, we will notably focus on establishing a method for generating explanations of the decision suggested. A state of the art on decision support systems presented in previous works [6] led us to consider a symbolic approach, as opposed to learning-based approaches, given the critical nature of legal compliance checking. This approach also requires the development of methods for formally representing regulatory knowledge, legal rules and notions. The aim of this work is to answer the following research questions: (RQ1) How can symbolic AI be used to reason over formalized legal texts to verify compliance and provide justifications? (RQ2) How to integrate semantic web standards in legal reasoning through the formalization of regulations? (RQ3) How to ensure that the results from legal reasoning provide a justified suggestion to the user? (RQ4) How to identify the most relevant information to ask a user when faced with an undecided scenario?

We propose to answer these questions by designing a framework that takes as input a situation description by a human agent using a specific ontology, and that supports reasoning on this situation thanks to formal rules represented using the Semantic Web's standard SPARQL. This part notably raises RQ1 and RQ2.

If a decision can be reached, relevant legal sources are provided to support it. In the event of a failure, which occurs either in case of an inconsistency in the applicable rules or if the situation is insufficiently described, this reasoning module is complemented by an explainability module. This second module can analyze the cause of the failure and then ask the user for any missing information, or report any inconsistency met. Setting up this module requires answering RQ3 and RQ4.

To illustrate the design and use of this framework, we selected a use case about checking the conformance of data sharing and processing in Law Enforcement Agencies (LEAs) to several European regulations. The central piece of legislation is the "Law Enforcement Directive" (LED)[4], which aims at regulating the processing of data related to investigations by LEAs while respecting the right to privacy of EU citizens. This restricted scope makes this case tractable in terms of rules and situation complexity. In spite of this, it is rich enough to capture important concepts such as privacy, agents, processing, and the laws that can be stated over these.

[4]2016/680 - EN - Law Enforcement Directive; LED - EUR-Lex: https://eur-lex.europa.eu/eli/dir/2016/680/oj/eng

Previous works have already covered the creation of the ontology [7] and of the formal rules [8] used in our studied use case. This paper focuses on developing an algorithm for generating an explanation to the rule-based reasoning results.

We first present the overall architecture of the framework and the relations between its components in Section 2. Section 3 introduces the formalism used to represent rules taken from regulations. It is continued in Section 4 in which the different types of results that can be obtained through reasoning over these rules are presented, including the case of an undecided result. This case is handled by the explainability module described in Section 5. In particular, since the undecided cases are caused by a lack of information in the description of the situation, this section presents the procedures employed to identify the supplementary information that should be asked of the user. This paper concludes in Section 6 with a summary of the presented works and a presentation of research perspectives.

2. Decision-Support Framework

Given the lack of explainable reasoning frameworks over formal legal rules, we aim to provide a DSS that interacts with the user to explain its decisions. To that end, we provide an interactive DSS by following the architecture described in [9] to obtain both the operability of formal rules and the explainability of reasoning results.

2.1. General Architecture

The DSS has been fully described in a previous paper [8]. Its main components are depicted in **Figure 1**.

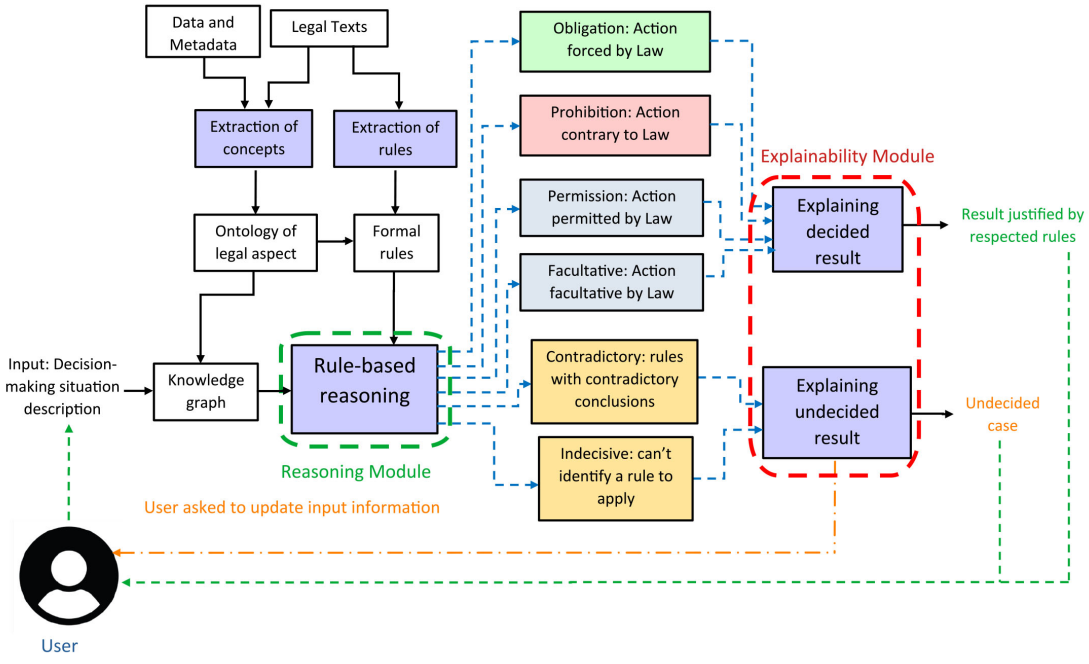


Figure 1. Framework principle.

- A first module, framed in green in Figure 1, implements a rule-based approach to ensure respect of Law and regulations. One of the major drawbacks of rule-based systems is their inability to provide a response in cases where none of the rules in the system rule base is triggered. There is also a risk of triggering several contradictory rules. Based on this observation, it is already possible to envisage 2 different types of results for this first module: 1) Cases where the rules manage to reach a *decided* conclusion, for example by stating that, "*given the situation described in the input, an action is permitted*". 2) In the other cases, the rules do not manage to reach a satisfactory conclusion and end up in an *undecided* state. The handling of the result will differ depending on which of these two situations the first module ends up in.
- A second module, framed in red in Figure 1, is in charge of justifying the output of the reasoning module. While *decided* results are easily justifiable, *undecided* results do not provide a satisfactory answer. Therefore, it is advisable to improve the result before returning it to the users, by asking them to adjust the input information so that the output could become *decided*.

The operation of these modules is based on several other components that are also illustrated in Figure 1 and described in the remainder of this section.

2.2. Legal Sources

The rule-based reasoning module relies on several components that contain information taken from legal sources such as laws and regulations. In most regulations, there are two types of information: 1) Definitions that clearly establish the terms and concepts involved, as well as the limitations of the regulations. These elements are integrated in the ontology and knowledge base, which will enhance the system's inference capabilities during the reasoning phase; 2) The normative articles that use the terms and concepts defined earlier and establish legal rules. These rules are extracted and formalized to then serve as the basis of the reasoning module.

2.3. Ontology for the Reasoning Module

Ontologies are a first-order logic vocabulary of entities and relations aiming at capturing the concepts in a given domain with a set of rules and usually rely on the concepts described in existing ontologies. For example, a hierarchy of types of agents can be defined with the names of the types, a relation giving a type to agents, and rules capturing that agents in a subclass are also agents in the superclass. Their standardised description in RDF or OWL is the basis of the Semantic Web[5]. Our DSS integrates a dedicated ontology to represent legal rules and situations in which a legal decision has to be computed. This ontology covers the representation of two types of legal knowledge:

- The knowledge related to the legal rules and their metadata, like for example

[5]Semantic Web: https://en.wikipedia.org/wiki/Semantic_Web

the deontic modalities they contain, or the legal documents they come from. This part of the ontology is generic and can be used for any legal compliance checking scenario. This part of the ontology can be built by extracting concepts and metadata from legal documents like regulations.

- The knowledge related to the specific type of situations the DSS is applied to. It must capture the characteristics of the data, people, and authorities involved in the decision-making process, as well as contextual information, such as whether the decision is made in the context of an emergency. This part of the ontology is unique for each possible use case of the framework and must be built specifically for the use case prior to using the framework. Building this part of the ontology requires the analysis of the concepts related to the use case.

Accordingly, a situation is a *knowledge graph* (KG), *i.e.* a set of instances of the entities and relations in the ontology complemented with entities describing the situation. For example, a KG for our data-sharing use-case could have instances "*Alice* ***is a*** *LEA agent*" and "*Bob* ***is a*** *subject under investigation*" of the "***is a***" relation with the "*LEA agent*" and "*subject under investigation*" ontology-defined entities, and the situation-specific entities "*Alice*" and "*Bob*". The rule of an ontology allows knowledge inference, that is, to reason over the most exhaustive description of a situation in a given use case. In the preceding example, instances stating that both Alice and Bob are agents are added to the set of instances. The legal rules encoded are also stored in a KG, and the rule-based reasoning module applies the rules stored in it to situations.

The ontology employed was presented in a previous article [7], and the reasoning is described more precisely in the next three sections.

2.4. Input of the Reasoning Module

The input of the system consists of a description of the situation for which users want to check the conformance to legal rules. All the data involved in the decision-making process are filled in by users using a form and then encoded in a knowledge graph (KG), which is where the reasoning takes place. The KG used in our framework fulfills several roles: It stores the ontology to allow semantic reasoning on the input data, but also the metadata about the rules used in the reasoning module. These metadata cover rule characteristics such as their deontic type or the legal document they come from; The KG can also be used to store data that are not part of the user's query, but that are part of the user's organization database and are also available to reason with; Finally, it is used to store the input information when a user makes a decision support request to the system. While the ontology and the metadata about the rules are stored in a "*global*" knowledge graph, two distinct approaches can be considered to store the input information:

1) The input information can also be stored in the global KG, alongside the ontology and the rules metadata. This approach is simple to implement, but it raises the issue of user query independence. Indeed, if several queries are made on the system, there is a risk that some of them define instances of the ontology with

the same identifier while being fundamentally different. This problem could lead to flawed reasoning and, therefore, result in incorrect decision suggestions. One way to avoid this issue would be to implement a system that guarantees the separation of entity identifiers between different queries made to the system.

2) The other approach consists of storing each query in a specific named KG. This solution offers compartmentalization of the queries and prevents the reasoning process from being contaminated by external information unrelated to the query.

Adopting one of these approaches influences the way the rules need to be written. Within the scope of the proposed framework, we selected the approach with named graphs, where compartmentalization of user queries prevents flawed reasoning by design.

2.5. Formal Rules of Reasoning Module

The formal rules are extracted and derived from regulations and other legal sources, and their metadata are expressed using the ontology. The rules themselves are written in SPARQL, a language that allows to query the KG. The metadata are then inserted into the KG used by the DSS. To encode legal reasoning and be able to provide useful feedback to the user, we differentiate several types of rules:

- Applicability rules that determine if legal articles apply to a situation, and allow for detecting when a situation is insufficiently described;
- Compliance rules that evaluate the compliance of a situation with legal articles;
- Exception rules that allow to define exception relationships between multiple rules, and thereby non-monotonic reasoning.

Each compliance rule is characterized by a deontic type (Permission, Facultative, Obligation or Prohibition) that is used in the reasoning process to evaluate the compliance of a situation with the regulations. It is important to note that the regulations from which the rules are extracted must, of course, be adapted to the scenario in which the framework is used.

2.6. Possible Outcomes of the Rule-Based Reasoning Module

The first module, the Rule-Based reasoning module, develops a policy inspired by access control policies using formal rules extracted from regulations as well as concepts and relationships from the ontology. The two types of output "decided" and "*undecided*" can be further divided using six output values. The decided cases are divided in Obligation (green text in **Figure 1**), Prohibition (red text in **Figure 1**), Permission and Facultative (blue text in **Figure 1**). These types of results are obtained when several rules are complied with, and all arrive at the same deontic conclusion. The undecided cases (yellow text in **Figure 1**) can be either indecisive or contradictory. The reasoning is indecisive when no single formal rule from the rule base has been triggered. It is contradictory if several compliance rules are applicable but give contradictory deontic answers. The handling of undecided cases calls for updating either the rule base or the input data, which occurs in the

explainability module.

2.7. Explainability Module

The second main module of the framework generates a justification for the conclusion of the rule-based reasoning module. It is returned to the user alongside the decision suggestion. Explainability is indeed an essential aspect in decision support systems, and particularly if these systems are used in critical contexts, such as for example establishing medical diagnoses or judging legal cases. This module takes the results from the rule-based reasoning and depending on the type of result, generates an appropriate justification.

In its current version, the proposed framework bases its explainability on the concept of "*Enacted Law*", as opposed to "*Case Law*". Rather than relying on the principle of precedence that would require an extensive database of previously determined cases, the justification of the decision comes from statutes such as laws and regulations. Relying on "*Enacted Law*" has two advantages in the scope of our study: First it provides a strict legal framework with which all institutions must comply, while "*Case Law*" can differ between institutions. Second, the language used for "*Enacted Law*" is very precise and defines the terms and their meanings for all parties affected by the law. By contrast, "*Case Law*" generally cannot be captured by a single authoritative and uncontroversial formulation[6].

Thus, in *decided* cases, the conclusion of the reasoning module is presented together with the legal sources of the rules that led to the conclusion as a justification. The undecided cases, however, require special handling. In principle, contradictory scenarios should not occur. While such cases may arise from contradictions in the applicable law, they would most likely be caused by wrongly encoded rules or by missing exceptions in the rule base. When faced with a *contradictory* case, the system registers the conflict with the intention of forwarding it to an expert who will be able to determine how to resolve the conflict and update the rule base accordingly. For *indecisive* cases, the system lacks information to reach a conclusion and will ask the user to update its input, while guiding him by indicating which information would lead to better chances of reaching a conclusion. Section 5 presents heuristics to identify information that, if added to the input, would allow the rule-based reasoning to reach a decided conclusion.

We will now take a look at the different types of rules that intervene in the rule-base reasoning process and the formalism we chose to represent them.

3. Using SPARQL for Non Monotonic Reasoning

The proposed framework is designed to be used in various situations and contexts where it is desirable to determine whether an action is permitted, required, prohibited, or optional. Depending on the context, it is necessary to consider the relevant laws and regulations in order to extract not only the underlying concepts

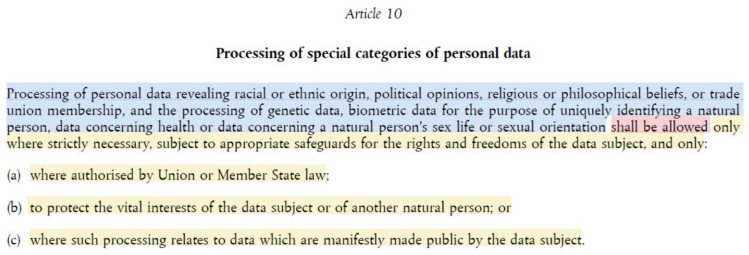
[6]IDENTIFYING APPLICABLE LAW - Law Explorer: https://lawexplores.com/identifying-applicable-law/

but also the rules that govern the decision-support process. The rules obtained that way are "*explicit*" and can be automatically extracted. Other "*implicit*" rules, not explicitly present in regulations, are included to model "default cases" or exception rules defining when some rules should prevail over others. Both the rules and the input contexts are represented as a graph using the concepts and properties of the ontology [7].

It was decided to use SPARQL to express the formal rules, as it allows to reason over RDF and OWL data. Some works conducted on the semantics of this language showcased its expressivity and showed that, subject to a few minor restrictions, it is equivalent to first-order logic. Notably, it is possible to translate the body of SPARQL queries into First-Order Logic (FOL) formulas using Answer Set Programming (ASP) as an intermediate language [10] [11]. The rules are applied using a SPARQL engine such as GraphDB[7] or Apache Jena Fuseki[8] to reason directly over knowledge graph content. However, for clarity reasons, in this paper, the formalization will be presented in a FOL form following a ***Conditions → Effect*** syntax. The SPARQL equivalents of the FOL expressions can be found on our Github repository[9] in the "*rules*" directory.

3.1. Explicit Rules from Regulations

Explicit rules are those that can be directly extracted from the regulations in natural language. Although extraction is currently manual, rule extraction from regulations can be automated using Large Language Models (LLMs) and semantic analysis [12]-[14], which we plan to do in future works. The formalization principle applied in this framework is adapted from Gandon *et al.*'s work [15], where each rule extracted from the regulation does not indicate directly whether an action is permitted, prohibited or mandatory. Instead, each rule is classified as either permission, obligation or prohibition and the reasoning aims at assessing the compliance of the input situation to each rule. Moreover, as explained in a previous paper [8], a legal rule is always encoded into two SPARQL queries, one indicating the applicability of the rule to the input situation, and the other indicating the respect of the rule by the situation. To illustrate this principle, the analysis of the article in **Figure 2** would decompose as follows:



Article 10

Processing of special categories of personal data

Processing of personal data revealing racial or ethnic origin, political opinions, religious or philosophical beliefs, or trade union membership, and the processing of genetic data, biometric data for the purpose of uniquely identifying a natural person, data concerning health or data concerning a natural person's sex life or sexual orientation shall be allowed only where strictly necessary, subject to appropriate safeguards for the rights and freedoms of the data subject, and only:

(a) where authorised by Union or Member State law;

(b) to protect the vital interests of the data subject or of another natural person; or

(c) where such processing relates to data which are manifestly made public by the data subject.

Figure 2. Example of law article: article 10 from 2016/680/UE directive (LED).

[7]GraphDB homepage: https://graphdb.ontotext.com
[8]Apache Jena Fuseki homepage: https://jena.apache.org/documentation/fuseki2/
[9]Github repository: https://github.com/JeremyBOUCHEPILLON/legalDataProcessing/

1) The deontic class of this article is "Permission", as indicated by the terms "*shall be allowed*". 2) The object of the rule relates to the processing of "*sensitive personal data*". 3) The conditions of this rule are "*the strict necessity*" of the processing, the "*safeguard of rights and freedom of the data subject*" and a disjunction of conditions: "*allowed by Union or Member State*" OR "*protection of vital interests*" OR "*the data are public*". From this analysis, the "permission" aspect of the rule is expressed by the axiom *is Permission* (*LED10*) and the rest can be formalized as the formulas (1) and (2) in FOL:

$$\begin{aligned} &\text{Processing}(\text{action}) \wedge \text{InvolvesDataset}(\text{action}, \text{dataset}) \\ &\wedge \text{ContainsData}(\text{dataset}, \text{data}) \wedge \text{SensitivePersonalData}(\text{data}) \\ &\rightarrow \text{IsApplicable}(\text{LED10}, \text{situation}) \end{aligned} \tag{1}$$

$$\begin{aligned} &\text{isApplicable}(\text{LED10}, \text{situation}) \wedge \text{Action}(\text{action}) \\ &\wedge \text{InvolvesDataset}(\text{action}, \text{dataset}) \\ &\wedge \text{Necessary}(\text{action}) \wedge \text{SafeguardRights}(\text{action}) \\ &\wedge (\text{AuthorizedLaw}(\text{action}) \vee \text{ProtectsVitalInterest}(\text{action}) \\ &\vee (\forall \text{data}, \text{SensitivePersonalData}(\text{data}) \Rightarrow \text{PublicData}(\text{data}))) \\ &\rightarrow \text{HasCompliance}(\text{LED10}, \text{situation}) \end{aligned} \tag{2}$$

To illustrate the presentation of such rules in SPARQL, trying to write the second of these rules in SPARQL would give the query from **Listing 1**.

```
INSERT { graph ?g { nru:PS1 nrv:hasCompliance ?g }}
WHERE {
        {graph ?g {
        ?action a :Processing .
        ?action :involvesData ?dataset .
        ?dataset :containsData ?data .
        ?data a :SensitivePersonalData .
        ?action :isNecessary "true"^^xsd:boolean .
        { ?action :isAuthorizedLaw "true"^^xsd:boolean }
        UNION
        { ?action :protectsVitalInterests "true"^^xsd:boolean }
        UNION
        { FILTER NOT EXISTS {
                ?data a :SensitivePersonalData .
                FILTER NOT EXISTS {
                ?data a :PublicData .
                }
        }}
}}}
```

Listing 1. SPARQL query to express the compliance rule for the 10th article of the LED.

The query can be obtained through the automatic translation from First-Order Logic (FOL) formulas, applying the works conducted in [10] and [16], using Answer-Set Programming (ASP) as an intermediate language. It can be noted that this translation introduces *FILTER NOT EXISTS* to express both the classical logical negation and the existential quantifier's negation. Since the universal quantifier does not exist in SPARQL, we go through the negation of the existential quan-

tifier to express an equivalent formula: A formula of the form $\forall x, p(x)$ will be encoded in SPARQL as $\neg\exists x, \neg p(x)$.

Substitutions. Consider a partial mapping σ from the set of variables χ to the set of terms. We extend it homomorphically to terms with $\sigma(t) = t$ when t is not a variable.

Its *domain* is the minimal set of variables $D \subseteq \chi$ such that $\sigma(x)$ is defined. It is idempotent if for all $x \in \chi$ we have $\sigma(\sigma(x)) = \sigma(x)$. A *substitution* is a homomorphic extension to terms (and then to relations) of an idempotent partial mapping on variables of finite domain. Substitutions are usually denoted in the postfix notation, *i.e.* $x\sigma$ rather than $\sigma(x)$. We denote Σ_D a set of substitutions of domain D, and given $D' \subseteq D$, we denote $\pi_{D'}(\Sigma_D)$ the set of substitutions that are a restriction to D' of the substitutions in Σ_D. The *join* of two sets Σ_D and $\Sigma_{D'}$ of substitutions is denoted $\Sigma_D \bowtie \Sigma_{D'}$ and is the largest set of substitutions of domain $D \cup D'$ such that for all $\sigma \in \Sigma_D \bowtie \Sigma_{D'}$ we have $\pi_D(\sigma) \in \Sigma_D$ and $\pi_{D'}(\sigma) \in \Sigma_{D'}$. The union of two sets of substitutions of common domain D is simply denoted $\Sigma_D^1 \cup \Sigma_D^2$. We note that there exists a unique set $\Sigma_\varnothing$ that contains the substitution of empty domain and that it is a neutral element for the join.

Knowledge Graphs. The information encoded in a FOL model can be encoded as a *Knowledge Graph* (KG) (within a larger domain and more predicates) with vertices representing entities and values, and arcs between 2 vertices labeled with a relation name. To bridge the gap with FOL models, a KG can also be seen as a unique relation between the head, the relation name, and the target of each arc.

Definition 1. (Knowledge Graph) A *Knowledge Graph* is a set of predicate instances *PRED* (const, const, const).

SPARQLTree. SPARQL rules are of the form ***body → effect*** where the *body* designates a graph pattern formula. When applied on a knowledge graph K, the effect is applied once for each substitution σ such that $K \vDash \varphi\sigma$. To avoid the introduction of variables in K, we assume that the set of variables occurring in the effect is a subset of X.

Additional quantifiers can be used in defining $\varphi^{X,W}$ through the *FILTER* construct. In the usual SPARQL syntax, a group $\{\varphi^{X,W} \text{ FILTER } \psi^{X\cup Y,W'}\}$ has, as its set of solutions, the set of substitutions that are the restrictions on X of the substitutions of domain $X \cup Y$ that satisfy both $\varphi^{X,W}$ and $\psi^{X\cup Y,W'}$. This definition is not structural, which led us to the introduction of a structural version of SPARQL where the same construct is denoted $\text{FILTER}(\varphi^{X,W}, \psi^{X\cup Y,W'})$. This change allows for easily defining what $K \vDash \varphi$ means, and for the definition of mappings on SPARQL graph pattern formulas that will be developed *infra*. We have omitted some constructions, such as equality or typing constraints, to focus on those used in this article. The grammar for this SPARQLTree query body is, denoting t the terms that are either variables or (literal) constants and decorations thereof:

$$\text{pred} ::= \text{PRED}(t, t, t)$$

$$\text{body} ::= \text{AND}(\text{body}, \text{body}) \vee \text{UNION}(\text{body}, \text{body}) \vee \text{FILTER}(\text{body}, \text{filterConstraint})$$

$$\text{filterConstraint} ::= \text{NOT}(\text{filterConstraint}) \vee \text{EXISTS}(\text{body})$$

Definition 2. The set of *free variables* of a SPARQLTree expression φ is denoted $Var(\varphi)$ and is defined inductively as $Var(\text{PRED}(h,p,t)) = \{h,p,t\} \cap \chi$, $Var(\text{AND}(\text{body}_1, \text{body}_2)) = Var(\text{body}_1) \cup Var(\text{body}_2)$, $Var(\text{UNION}(\text{body}_1, \text{body}_2)) = Var(\text{body}_1) \cap Var(\text{body}_2)$, and $Var(\text{FILTER}(\text{body}, \text{fc})) = Var(\text{body})$.

The SPARQL language definition is intricate, and we have chosen here to present a simplified version to preserve clarity. For example, the definition of *UNION* means some variables may be unbound. For that reason, we have chosen to include in the free variables only those that must be bound in a solution. Beyond the implicit existential quantification on the variables of a query, nested universal and existential quantifiers are encoded using the *FILTER* construct, and thus variables introduced in the *EXISTS* part of the filter constraint are removed when computing the solutions.

Given a formula $\text{PRED}(h,p,t)$ and a knowledge graph G we denote $s(\text{PRED}(h,p,t), G)$ the largest set of substitutions of domain $Var(\text{PRED}(h,p,t))$ such that for every $\sigma \in s(\text{PRED}(h,p,t), G)$ we have $\text{PRED}(h,p,t)\sigma \in G$.

Definition 3. (Solutions of a SPARQLTree body) The application of a SPARQLTree formula φ on a KG G yields a set of substitutions $Sol_G(\varphi, \Sigma_\varnothing)$ of domain $Var(\varphi)$ defined inductively on φ as follows:

$$Sol_G(\text{PRED}(h,p,t), \Sigma_D) = \Sigma_D \bowtie s(\text{PRED}(h,p,t), G)$$

$$Sol_G(\text{AND}(\varphi_1, \varphi_2), \Sigma_D) = Sol_G(\varphi_2, Sol_G(\varphi_1, \Sigma_D))$$

$$Sol_G(\text{UNION}(\varphi_1, \varphi_2), \Sigma_D) = \pi_{(Var(\varphi_1) \cap Var(\varphi_2)) \cup D}(Sol_G(\varphi_1, \Sigma_D) \cup Sol_G(\varphi_2, \Sigma_D))$$

$Sol_G(\text{FILTER}(\varphi, fc), \Sigma_D) = \pi_{D'}(Sol^c_G(fc, Sol_G(\varphi, \Sigma_D)))$ with D' domain of $Sol_G(\varphi, \Sigma_D)$

The computation of solutions with constraints $Sol_G(\varphi, \Sigma_D)$ is defined with:

$$Sol^c_G(\text{NOT}(\varphi), \Sigma_D) = \Sigma_D \setminus \pi_D(Sol^c_G(\varphi, \Sigma_D))$$

$$Sol^c_G(\text{EXISTS}(\varphi), \Sigma_D) = \pi_D(Sol_G(\varphi, \Sigma_D))$$

A complete and more general computation of solutions is given in [10], but **Definition 3** is the basis of the algorithms provided *infra* aiming at explaining the lack of solutions of a query on a KG G.

In addition to the explicit rules extracted from regulations, we also need to integrate in our rule base potential default cases, expressed implicitly in legal texts. We also have to integrate conflict-solving rules to implement the defeasible aspect of legal reasoning.

3.2. Identifying Implicit Rules and Conflict-Solving Rules

In addition to the explicit rules, it is necessary to make explicit, with the help of legal experts, rules that are either implicit or that allow to solve conflicts between other rules. Those conflicts can have several causes. For example, one rule may be an exception to another, or a legal doctrine defining priority between two rules may have not been applied (lex superior, lex posterior). This can be illustrated with the article in **Figure 2**. Once analyzed, this article states that "the processing of sensitive personal data is permitted ONLY where some conditions are met". Intuitively, this suggests that there is an implicit "default" rule stating that "the processing of sensitive personal data is forbidden" and the rule from the LED is an exception to this default rule. The formalization of "default" rule is based on the axiom isProhibition ($\text{LED10}_{\text{default}}$) and the FOL expression (3):

$$\begin{aligned} &\text{Processing}(\text{action}) \wedge \text{InvolvesDataset}(\text{action}, \text{dataset}) \\ &\wedge \text{ContainsData}(\text{dataset}, \text{data}) \wedge \text{SensitivePersonalData}(\text{data}) \\ &\rightarrow \text{HasCompliance}(\text{LED10}_{\text{default}}, \text{situation}) \end{aligned} \tag{3}$$

and the rule (4) that resolves the conflict states that if both rules are complied with, only the exception has to be considered (represented by negating the compliance with the default rule):

$$\begin{aligned} &\text{HasCompliance}(\text{LED10}_{\text{default}}, \text{situation}) \wedge \text{HasCompliance}(\text{LED10}, \text{situation}) \\ &\rightarrow \neg\text{HasCompliance}(\text{LED10}_{\text{default}}, \text{situation}) \end{aligned} \tag{4}$$

We therefore have a representation of legal rules based on the Semantic Web, thereby addressing our research question RQ2. The explicit rules form the core of our rule base and are direct formalizations of normative legal articles, while the implicit rules define default cases and the conflict-solving rules implement the defeasible aspect of legal reasoning. To justify the use of SPARQL, we have formally defined the elements of a tree-based SPARQL syntax, SPARQLTree, noting that translations between FOL and SPARQL are possible via ASP.

4. Reasoning and Explainability

The extraction of explicit rules and the identification of implicit, exceptions and priority solving rules generate a full rule based on which to reason. The reasoning itself is decomposed into two steps:

1) All the explicit and implicit rules are applied in the situation KG, and if the premises of a rule are satisfied, its conclusion and an indication that it was satisfied are added to the situation KG;

2) All the exception and priority rules are applied on the obtained KG to solve potential conflicts by removing the decisions of the rules that are superseded by other rules.

The reasoning aims not only at checking the compliance of a case with the regulation and at suggesting a decision to the end user, but it will also produce an explanation of this decision. This module of the framework deals with RQ3.

At this point, the framework possesses a list of legal rules that apply to the input situation. The legal texts to be formalized within the framework are laws and regulations that specify what can and cannot be done in a given context. Numerous studies have examined the use of deontic logic to formally express this type of text [17] [18]. Although some works have explored complex relationships between different deontic modalities [19], we base our study on a standard "deontic square" comprising four modalities that can be linked by a negation relation: The Permission (*P*) and its opposition the Prohibition/Interdiction (*I*), the Obligation (*O*) and its opposition the Facultative (*F*). Each normative rule formalized in the framework can be classified into one of these four categories.

Thus, the list of satisfied rules generated by the reasoning module also provides the deontic modalities of those rules. In order to reach an overall conclusion regarding the situation entered by the user, it is necessary to understand how the various modalities interact, which will be detailed in the next section.

4.1. Evaluating the Resulting Deontic Conclusion

Establishing the resulting deontic conclusion from a set of respected rules requires to analyze all the possible combinations of deontic types. To do that, we built a deontic truth table (cf. **Table 1**), using the principles of the deontic square of oppositions [19]. It summarizes the relations between each combination of deontic types and the resulting global deontic conclusion, and its content reads as follows:

Each column represents a possible combination of deontic types, where *1* denotes the presence of rules of this type in the list of respected rules, and *0* the absence of rules of this type in the list. The letters *F*, *O*, *I* and *P* stand respectively for *Facultative* (the right to not do), *Obligation*, *Interdiction* and *Permission* (the right to do). In the result column, an ∅ represents a situation where no single rule is respected, and × highlights all the cases in which the respected rules lead to an inconsistent decision in deontic terms.

Table 1. Truth table for the global deontic conclusion of a set of deontic rules.

Case	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Facultative	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Obligation	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
Interdiction	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
Permission	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Result	∅	*P*	*I*	×	*O*	*O*	×	×	*F*	*P* and *F*	*I*	×	×	×	×	×

For example, Case 5 corresponds to a situation where some remaining decisions are Obligations, and others are Permissions (there is a 1 in the "*O*" and the "*P*" columns, 0 in the others). In this situation, the deontic notion of obligation prevails over the permission, resulting in an Obligation. In Case 3, the remaining decisions are of the Permission and Interdiction types (there is a 1 in the "*I*" and

the "P" columns, 0 in the others). In this situation, interdiction and permission are conflicting deontic conclusions, and the computed result is a contradiction, denoted with the × symbol.

It can be noted that one specific combination in this truth table that combines only Permission and Facultative rules results in a double deontic conclusion, stating that the action is neither obligated nor prohibited (Case 9).

Another point of note is that Table 1 distinguishes between Facultative and Permitted by making the former compatible with an interdiction and the latter compatible with an obligation.

The results from Table 1 can be organized more succinctly in a Karnaugh-like map as in Table 2. This representation allowed us to find a minimal simplified combination of deontic types to compute the conclusion of the rule-based reasoning module (cf. logical formulas ((5)-(10))).

Table 2. Deontic karnaugh table.

		FO			
		00	01	11	10
IP	00	$\varnothing$	O	×	F
	01	P	O	×	P and F
	11	×	×	×	×
	10	I	×	×	I

$$P = P \cdot \overline{I} \cdot \overline{O} \tag{5}$$

$$F = F \cdot \overline{I} \cdot \overline{O} \tag{6}$$

$$O = O \cdot \overline{I} \cdot \overline{F} \tag{7}$$

$$I = I \cdot \overline{P} \cdot \overline{O} \tag{8}$$

$$\times = I \cdot P + F \cdot O + O \cdot I \tag{9}$$

$$\varnothing = \overline{P} \cdot \overline{I} \cdot \overline{O} \cdot \overline{F} \tag{10}$$

The formula (5) indicates that the resulting deontic conclusion is a permission if the list of respected rules contains permission rules and no interdiction and no obligation rules. Formulas (6)-(8) are the analogous formulas for respectively facultatives, obligations and interdictions. The formula (9) indicates that there is conflict in three configurations: 1) There are Interdiction and Permission rules in the list; 2) There are Facultative and Obligation rules in the list; 3) There are rules classified as Obligation and Interdiction in the list. Finally, the last formula (10) clearly expresses the fact that no rule is being respected, which forms the undecided case that is addressed infra.

In conclusion, this deontic reasoning can thus generate 7 different output values that derive from 3 more general types of results. First, cases where at least one rule is satisfied by the input situation, with all the satisfied rules giving consistent decisions such that it is possible to draw a clear overall deontic conclusion ("F",

"*O*", "*I*", "*P*", as well as "*P and F*"). Secondly, cases where at least 2 rules are respected but their conclusions are inconsistent, creating a contradiction ("×"). And lastly, undecided cases in which no compliance rule is satisfied by the input ("∅"). Each of these cases is handled in a different way, allowing us to write an algorithm on how to handle the output of the reasoning.

4.2. General Algorithm for Explainability

The explanation given to the user will depend on the type of results from the reasoning phase, following the logic illustrated in **Algorithm 1** that allows us to answer research question RQ3.

Algorithm 1: Explanation of the rule-based reasoning results

```
Data: (listApplicableRules, listRespectedRules) ;        /* Result from reasoning */
Result: (result, explanation) ;                  /* The justified decision suggestion */
if listApplicableRules is empty then                    // There are no applicable rules
    return "The situation is outside the scope of applicability of the verified regulations" ;
else if listRespectedRules is empty then            // None of the applicable rules are
 respected
    result ← undecided ;
    ask the user for potentially missing information to try to improve the result ;
else
    if all rules in listRespectedRules are Permission then
        return (Permission, listRespectedRules) ;
    else if all rules in listRespectedRules are Facultative then
        return (Facultative, listRespectedRules) ;
    else if all rules in listRespectedRules are either Permission or Facultative then
        return (Permission & Facultative, listRespectedRules) ;
    else if all rules in listRespectedRules are either Obligation or Permission then
        return (Obligation, listRespectedRules) ;
    else if all rules in listRespectedRules are either Prohibition or Facultative then
        return (Prohibition, listRespectedRules) ;
    else                                        // There are contradictory respected rules
        return (Contradictory, listRespectedRules) ;
    end
end
```

When all the respected rules are coherent, the explanation is straightforward and consists of informing the user of the regulation parts that support the result. However, two problematic cases may arise. The first is when the rules produce contradictory decisions after the second conflict-solving step. This indicates that these rules are missing from the rule base. The result returned to the user highlights the contradiction, indicates the conflicting rules and asks him which rule should prevail. The answer is added as a new, temporary conflict-solving rule in the system, and is put forward for validation by an expert. The second problematic case arises when no compliance rule from the rule base has been respected. If no applicability rule is satisfied, the system indicates that the input situation is not in the scope of the verified regulations. This general algorithm allows the system to distinguish a condition that is false from one for which the knowledge graph lacks information. The next section describes the cases in which some applicability rules are satisfied but no compliance rule.

5. Handling Undecided Cases Using Decision Trees

In the case where no rule from the ruleset is respected, the system will ask the user

for complementary information regarding the input situation. In order to determine which information to request from the user, the idea is to check what part of the rules that were "the closest to being complied with" were not respected. To do this, the following procedure is applied: 1) Consider only the rules "applicable" to the situation. 2) For each of these rules, generate a binary decision tree where each node is one of the conditions of the rule, the left edge of a node corresponds to "the condition of the node is respected", and the right edge to "the condition of the node is not respected". 3) Confront the input situation to each tree and keep track of the non-respected conditions in them. 4) Sort the rules depending on how deep in their decision tree the verification went. 5) Use the non-respected conditions in each rule to ask the user complementary information regarding the situation in input.

If the user is able to give supplementary information, the system processes the updated input from the beginning of the framework, hoping to obtain a "*decided*" decision. After this new processing, if the reasoning fails again to find respected rules or if the user cannot give supplementary information, the case is definitely classified as "*undecided*".

The structure of the decision trees proposed in this section contributes to identifying relevant information to ask the end user, answering RQ4. In addition, the reasoning about undecided cases is an answer to RQ3 as it participates in providing better explanations to the end user.

In the following section, we will detail the principle behind the tree construction (step 2 in the procedure), and then we will explore two approaches for confronting the input situation to the trees (step 3 of the procedure).

5.1. Architecture of the Decision Trees

The idea behind the constructed trees is to use their traversal to check the conditions of each applicable rule individually and establish which specific conditions were not fulfilled during the reasoning phase. Each tree is a binary tree and implements an applicable rule. Each node corresponds to one of the rule conditions, with the root node representing the first condition. Left edges correspond to a positive evaluation of a condition, while the right edges correspond to its negative evaluation. Each leaf on the tree may represent one of two types of results: 1) A positive output would indicate that all the rule conditions have been met for it to be respected. This type of leaf will never be reached during tree traversal because the construction of these trees is precisely motivated by the fact that no rule is being respected. 2) A negative output contains the condition or the set of conditions that have not been met, explaining why the associated rule, although applicable, was assessed as not being respected during the reasoning process.

Analysis of failure. Definition 3 provides a total order of evaluation of the parts of a query. let us analyze the first failure according to the different cases with the notations of that definition (unrolling the cases of filter constraints):

- ***PRED***: Assuming $\Sigma_D = \varnothing$ (first failure), the result is an empty set only if no

instance of the predicate in G is joinable with a substitution in Σ_D. In that case the explanation is the predicate;

- ***AND*** and ***UNION***: A failure cannot occur at this level; it has to occur during the computation of either subformulas of ***AND*** or both subformulas of ***UNION***;
- ***FILTER EXISTS***: This case behaves as an ***AND*** node with an added projection that cannot make the set of solutions empty;
- ***FILTER NOT EXISTS***: This case results in an empty set of solutions if $\Sigma_D \subseteq \pi_D\left(Sol_G\left(\varphi, \Sigma_D\right)\right)$, *i.e.* if all substitutions in Σ_D can be extended to satisfy φ.

The last case often occurs in the translation of universal subformulas, and in that case φ is itself of the form $\mathrm{FILTER}\left(\varphi_1, \mathrm{NOT}\left(\mathrm{EXISTS}\left(\varphi_2\right)\right)\right)$. To provide a better explanation in that case, we introduce a new node "***UNIVERSAL***" of arity 2. Noting that a first-order logic formula $\varphi_1 \wedge \left(\forall x, \varphi_2 \Rightarrow \varphi_3\right)$ is translated into $\mathrm{FILTER}\left(\varphi_1, \mathrm{NOT}\left(\mathrm{EXISTS}\left(\mathrm{FILTER}\left(\varphi_2, \mathrm{NOT}\left(\mathrm{EXISTS}\left(\varphi_3\right)\right)\right)\right)\right)\right)$, all such patterns are rewritten from the bottom-up into $\mathrm{AND}\left(\varphi_1, \mathrm{UNIVERSAL}\left(\varphi_2, \varphi_3\right)\right)$. By unrolling **Definition 3**, we find that:

$$Sol_G\left(\mathrm{UNIVERSAL}\left(\varphi_1, \varphi_2\right), \Sigma_D\right)$$

$$Sol_G\left(\mathrm{FILTER}\left(\top, \mathrm{NOT}\left(\mathrm{EXISTS}\left(\mathrm{FILTER}\left(\varphi_1, NOT\left(\mathrm{EXISTS}\left(\varphi_2\right)\right)\right)\right)\right)\right), \Sigma_D\right)$$

$$\Sigma_D \setminus \pi_D\left(Sol_G\left(\mathrm{FILTER}\left(\varphi_1, \mathrm{NOT}\left(\mathrm{EXISTS}\left(\varphi_2\right)\right)\right), \Sigma_D\right)\right)$$

$$\Sigma_D \setminus \pi_D\left(Sol_G\left(\varphi_1, \Sigma_D\right) \setminus \pi_{D'}\left(Sol_G\left(\varphi_2, Sol_G\left(\varphi_1, \Sigma_D\right)\right)\right)\right)$$

$$\Sigma_D \setminus \left(\pi_D\left(Sol_G\left(\varphi_1, \Sigma_D\right)\right) \setminus \pi_D\left(Sol_G\left(\varphi_2, Sol_G\left(\varphi_1, \Sigma_D\right)\right)\right)\right)$$

Consider a substitution $\sigma \in \Sigma_D$. It does not occur in the solution if it is present in $\pi_D\left(Sol_G\left(\varphi_1, \Sigma_D\right)\right)$—meaning extra variables can be instantiated to satisfy the condition of the filter—but not in $\pi_D\left(Sol_G\left(\varphi_2, Sol_G\left(\varphi_1, \Sigma_D\right)\right)\right)$—meaning the conclusion of the universal cannot be satisfied with these additional instances. We note that failing to satisfy the condition φ_1 of the universal is not a cause of failure. *i.e.*, the explanation of the failure of a ***UNIVERSAL*** node is that of the failure of φ_2.

Let us now consider the origin of these rules. We assume that, as in our example, universal quantification is employed to constrain (using φ_2) the items described by φ_1. These constraints are naturally a conjunction of properties that have to be satisfied. Instead of exploring *why* a given property failed, we believe it is sufficient to handle the property as a whole. The same principle holds for the ***FILTER NOT EXISTS*** construct. We propose to handle this structure of explanation by constructing an *explanation forest* to structure the search for the explanation of failure by adding sequence points in the sequential evaluation of the query.

Definition 4. (Explanation tree, explanation forest) A *explanation tree* is either:

- A **NORMAL_block** with a list of predicates;
- A **UNION_block** with a list of explanation trees whose root is not a UNION_block;
- A **FILTER_NOT_EXISTS_block** with two lists, one of explanation trees called its *condition*, and one of SPARQLTree formulas called its *consequence*;
- A **UNIVERSAL_block** with a SPARQLTree formula called its *condition*, and a list of SPARQLTree formulas called its *consequence*;

An *explanation forest* is a list of explanation trees.

A SPARQLTree query body is translated into an explanation forest. Its construction from a SPARQLTree formula is given in the **Appendix 1**.

Definition 5. (Decision tree) A *decision tree* is either a ***OK*** node, a ***KO*** node, or a ***F(ormula)*** node labeled with a SPARQLTree formula with two sons labeled one with ***YES*** and the other with a text providing an explanation.

Each explanation forest is translated into one decision tree with exactly one ***OK*** and one ***KO*** node. This transformation algorithm performs operations on nodes in trees, and we believe it is not particularly illuminating. We have presented a full translation example in **Figure 3**, and described the translation patterns for each kind of explanation tree nodes.

To illustrate what such a tree looks like, we will use simple arbitrary letters a, b, c, etc. to represent conditions or groups of conditions in the nodes. Starting from the root, recursively, the construction of the tree follows these principles:

- When the condition a in the current node and the next condition b are connected by a logical $\wedge$ operator (structure $a \wedge b$), the node containing b is connected to the left branch leaving from the node a. In this case, it is because, if condition a is not verified, then $a \wedge b$ as a whole is not verified. On the other hand, if a is verified, in order to verify $a \wedge b$, it is still necessary to test whether b is verified.
- When the condition a in the current node and the next condition b are connected by a logical $\vee$ operator (structure $a \vee b$), the node containing b is connected to the right branch leaving from the node a. In this case, it is because if condition a is already verified, then $a \vee b$ as a whole is also verified. On the other hand, if a is not verified, in order to verify $a \vee b$, it is still necessary to test whether b is verified.
- The branch leaving from a that is not yet connected is then connected either to a leaf or, if there are still other conditions connected with $\vee$ operators later in the rule, to the first node of the set of conditions that follows the block currently being processed.

Let's consider an arbitrary example of a simplified SPARQL query representing a rule with conditions expressed as simple predicates. Since the head is irrelevant for the construction of the tree, we will only consider the body of the query. In First-Order Logic, the rule would be written as $a \wedge b \wedge \left(c \vee \left(d \wedge \neg e\right) \vee \left(\forall x.\left(f\left(x\right) \wedge g\left(x\right)\right) \Rightarrow \left(h\left(x\right) \wedge i\left(x\right)\right)\right)\right) \wedge j$, which in SPARQL would be expressed as shown in **Listing 2**.

The query in **Listing 2** contains all the structures that can be found in a legal article: conjunctions and disjunctions of conditions, negations, universal quantification, some of them nested in each other. In the corresponding SPARQL query, conjunction and disjunction are written using respectively the dot and the *UNION* keyword. Universal quantification is encoded using the *FILTER NOT EXIST* construction. For example, the second *UNION* of the query represents a condition "*All the elements that verify f and g, are such that they also verify h and i*" that can

```
a .
b .
{c .}
UNION
{
    d .
    FILTER NOT EXISTS{e .}
}
UNION
{
    FILTER NOT EXISTS{
        f .
        g .
        FILTER NOT EXISTS{
            h .
            i .
        }
    }
}
j .
```

Listing 2. Body of a simplified SPARQL rule to illustrate the construction of the tree.

be found in a legal rule "*All the data that are personal and sensitive, are such that they also are public and disclosed by the person concerned*". In SPARQL, this universal quantification is expressed with two negated existential operators, literally saying that "*There is no f and g without also having h and i*" or, in the example, it can be rephrased as "*There is no data that is personal and sensitive, that is not also public and disclosed by the person concerned*". A side-effect of rephrasing the universal quantification is to allow a more precise identification of the rule parts that could be violated. The tree constructed from this rule and following these principles is shown in **Figure 3**.

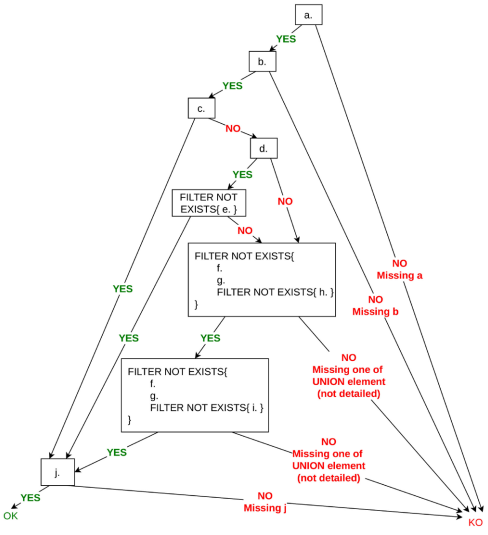


Figure 3. Part of tree to determine unfulfilled condition of the arbitary rule from listing.

If we apply the same principle to our running example of article 10 from the LED, we obtain the tree in Figure 4.

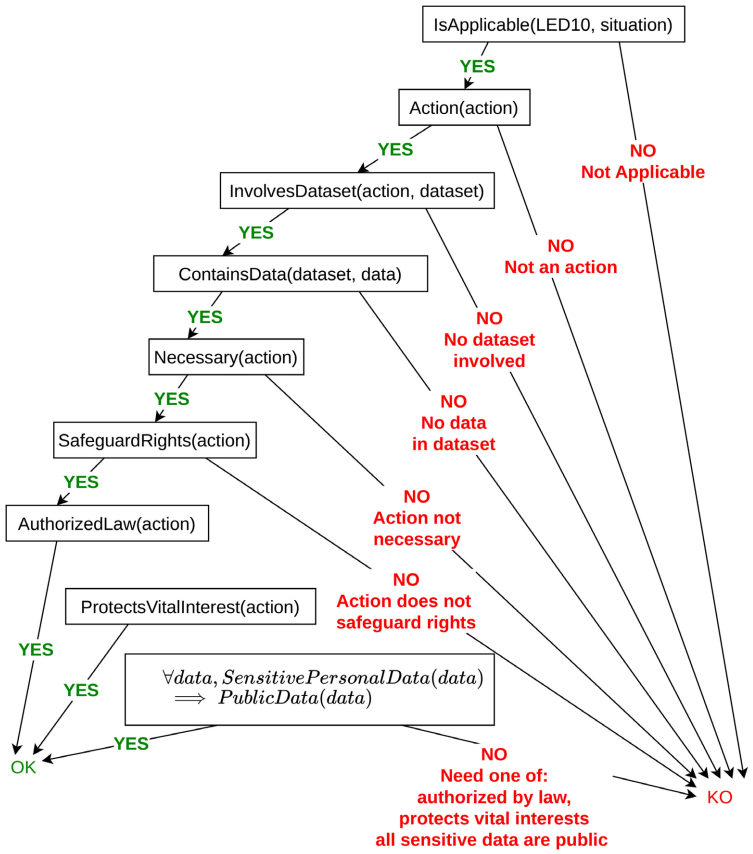


Figure 4. Tree to determine the unfulfilled condition of the arbitrary rule from listing.

5.2. Constructing the Trees from the SPARQL Queries

Now that we have established the principles behind the architecture of the decision trees, we need a procedure that allows us to transform SPARQL rules into this tree form. First, the construction of the decision tree for each applicable rule requires parsing the related SPARQL rules about compliance. This parsing is based on breaking down the SPARQL query into blocks of different types that can either be sequential or nested within each other. The type of blocks may be ***NORMAL_Block***, ***NEGATION_Block***, ***UNIVERSAL_Block*** and ***UNION_Block***, each of which is defined by different attributes:

- The ***NORMAL_Block*** is the most basic element to be parsed. It is defined by a list of strings, each of these strings being one of the triples of the query.
- The ***NEGATION_Block*** is used when a part of the query is inside a simple *FILTER NOT EXISTS* structure. It is defined by the list of the blocks inside the *FILTER NOT EXISTS*.
- The ***UNIVERSAL_Block*** allows to parse universal quantification in queries, that are expressed with two nested *FILTER NOT EXISTS* in SPARQL. This block is defined by two lists of blocks. The first list corresponds to the part

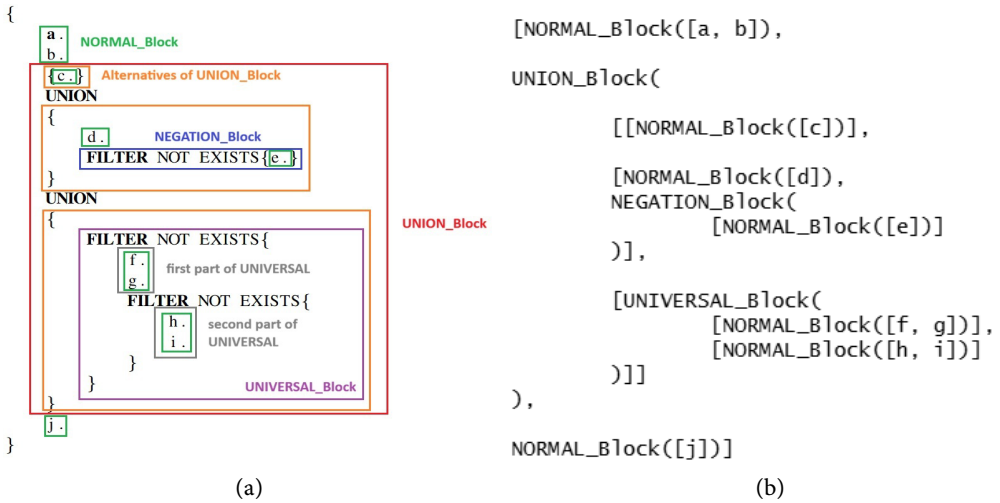
inside the first *FILTER NOT EXISTS* of the structure and the second list to the elements inside the second *FILTER NOT EXISTS* structure.

- The ***UNION_Block*** represents all the alternatives of a *UNION* structure in SPARQL. The datatype used to characterize it is a list of Blocks. Indeed, a *UNION* structure is composed of several alternatives, each of which is a sequence of blocks.

Finally, the query itself is parsed as a list of blocks. To illustrate the parsing, let us consider the SPARQL structure from **Listing 2**. Parsing this example results in the following rule decomposition:

The query itself is composed of a succession of 3 main blocks, a NORMAL_Block followed by a UNION_Block and finally another NORMAL_Block:

- The first NORMAL_Block contains the two conditions *a* and *b*.
- The UNION_Block is composed of three alternatives.
- The NORMAL_Block with the condition *c*.
- The second alternative contains a NORMAL_Block followed by a NEGATION_Block.
- The NORMAL_Block contains only the condition *d*.
- The NEGATION_Block contains a single NORMAL_Block whose content is *e*.
- The third alternative contains a UNIVERSAL_Block.
- The first part of the UNIVERSAL_Block is composed of a single NORMAL_Block that contains *f* and *g*.
- The second part of the UNIVERSAL_Block contains also a single NORMAL_Block, composed of the conditions *h* and *i*.
- The last NORMAL_Block contains the last condition *j*.



(a) (b)

Figure 5. Parsing of the blocks that compose the query used as example. (a) Blocks that compose the rule in the studied example; (b) Textual translation of the parsing.

Figure 5(a) illustrates the different blocks that compose this example. Block identification enables the construction of a decision tree based on specific patterns

that keep a one-to-one correspondence between the tree nodes and the rule structure and conditions. The different blocks are then translated into text see Figure 5(b).

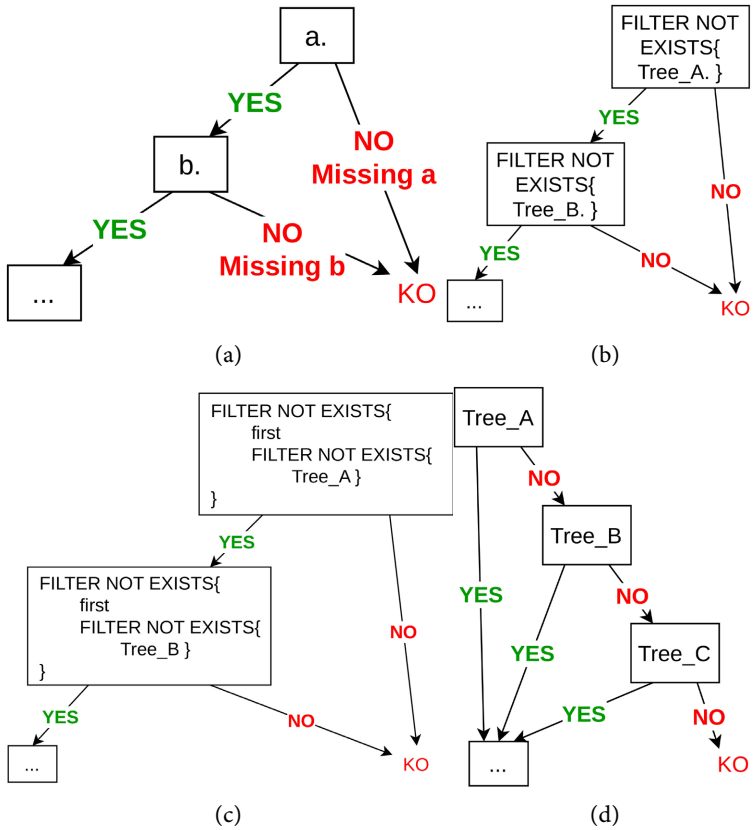


Figure 6. Patterns of tree building depending on the type of block parsed; (a) Pattern of tree buidling for a NORMAL_Block; (b) Pattern of tree buidling for a NEGATION_Block; (c) Pattern of tree buidling for a UNIVERSAL_Block; (d) Pattern of tree buidling for a UNION_Block.

- The pattern to build a node from a *NORMAL_Block* is the simplest. It consists of chaining the conditions in the block through left branches (the branches for a positive checking of the condition). An example for the block *NORMAL_Block* ([*a*, *b*]) is given in Figure 6(a).
- Building a node from a *NEGATION_Block* only consists of prefixing the content of all the nodes constructed from the blocks inside the negation, as illustrated in Figure 6(b) for the example *NEGATION_Block* ([*Block_A*, *Block_B*]).
- Building a node from a *UNIVERSAL_Block* is also straightforward and consists of chaining the subtrees created from the blocks in the second part of the universal structure. The result of such construction from the example *UNIVERSAL_Block* (*first*, [*Block_A*, *Block_B*]) is given in Figure 6(c).
- The pattern to build a node from a *UNION_Block* is essentially the mirrored principle compared to a *NORMAL_Block*. Each subtree created from the al-

ternatives of the *UNION* are chained through the right branches (the branches for a negative checking of the condition). For example, the tree created from example *UNION_Block* ([[*Block_A*], [*Block_B*], [*Block_C*]]) is given in Figure 6(d).

Next, when going through the trees, it is necessary to establish how each node will be evaluated in order to take the correct branch at each step. The evaluation of a node is actually carried out by executing a SPARQL query designed to check specifically the condition inside the node. These queries are not *INSERT* queries like the ones used in the first module of the framework but *ASK* queries. ***ASK*** is a SPARQL keyword that allows to test whether or not a query pattern has a solution, without returning information about the possible solutions. The binary response of these queries are then directly used as a way to determine which tree branch to take from each node. However, building the query pattern is challenging, as it requires to keep track of the conditions that have already been checked as satisfied.

Figure 7 gives an example of the queries constructed in each node of the tree from Figure 3.

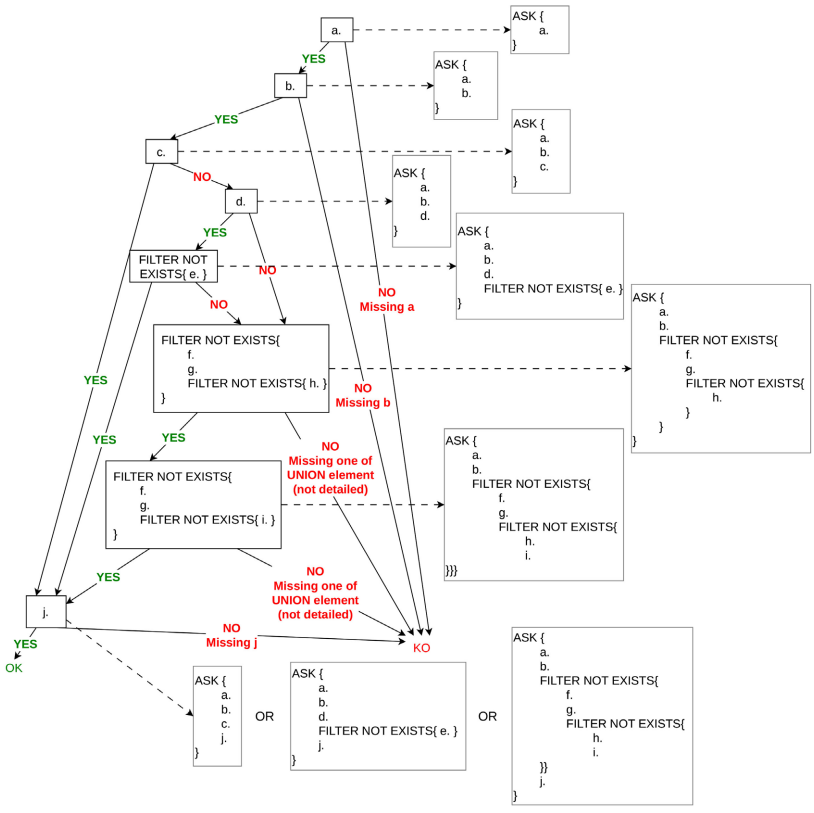


Figure 7. SPARQL queries executed to check each node of the tree.

We can note that three possible SPARQL queries can be executed in the last node. This is because prior to this node was a *UNION* with three alternatives. These alternatives are checked in order until one of them is verified. If one of the

alternatives is verified, verification of others is skipped, and the conditions that are part of the verified alternative are registered in order to construct the SPARQL query for the following nodes.

When an *undecided* conclusion is reached, a tree is constructed from each applicable rule in our system following the patterns presented. An algorithm then explores this forest to identify the conditions that are not actually met by the input situation.

5.3. Exploring the Forest to Identify Unfulfilled Conditions

Several approaches are possible for traversing the trees, in order to identify both the rules that were the closest to being respected and the unfulfilled conditions within these rules. We will explore two of them: 1) A first, naive approach, that looks only for the first set of decisive conditions that are not met in each applicable rule. 2) A second, more optimized approach, which consists of searching for the largest subset of conditions that are satisfied by each applicable rule.

5.3.1. A Naive Approach: Seeking the First Unfulfilled Decisive Conditions

On a concrete example, **Figure 4** illustrates the tree that would be created from the compliance rule obtained from article 10 of the LED (cf. **Figure 2**). For clarity and intelligibility, the nodes of this tree contain the conditions written in a more compact syntax, equivalent to what is written in SPARQL.

The first approach aims at identifying the first unfulfilled decisive conditions in each applicable rule to the input situation. It can be noted that when the law articles are formalized, the conditions specified in the article, once translated, are listed in the same order in the SPARQL query than in natural language. This order tends to follow an order of increasing precision, where the further one goes into a article of law, the more requirements there are to verify, and the more detailed those requirements become. It is therefore relevant to consider that the further down the tree we go, the closer we are to fully complying with the rule.

We call *decisive conditions* a subset of the conditions of the rules such that failure to meet all the conditions in this subset is necessary and sufficient for the rule to be violated. All leaves of the tree that are not a positive outcome (leaves that are not labeled "*OK*" in our examples) contain one such subset. In other words, finding the first unfulfilled decisive conditions simply consists of finding the leaf reached when confronting a situation with the tree.

Given a tree created from a rule and a system input situation, the general principle is to go down the tree starting from the root node. For each node, take the branch that corresponds to the result of the binary check of compliance with the condition specified in that node. We stop when a leaf is reached, which gives us the first set of unfulfilled conditions of the rules.

Repeating the process for each tree created from the applicable rules gives us, for each of these rules, their set of unmet conditions as well as the depth of the leaf that was reached. From the depth reached and the maximum depth of the tree, we

can calculate the relative depth reached, which can then be used as a metric to sort the rules in descending order of its value. An algorithm that describes the implementation of this naive approach is provided in **Appendix 2**.

However, this naive approach has two major flaws:

1) It stops the validation of the conditions at first unfulfilled decisive conditions identified and doesn't check any further. By doing this, there is no guarantee that the information that the user is asked to provide will be sufficient to validate the rules. For instance, in our example, the procedure stopped after finding the missing information regarding the necessity of the action. But the end of the rule, with a disjunction of conditions, is not met either. With the naive approach, this lack will only be identified after the user has provided the required information about *necessity* and the system has been run again.

2) Using the relative depth reached is not an optimal metric either for ranking rules in order of their closeness to being satisfied. For example, a rule can start with a part establishing a certain context, *e.g.* that the user must be a police officer. That context is irrelevant in the evaluation of the rest of the rule, and a failure can just be the result of an unintended omission. However, with the naive approach, this rule would be put aside in terms of "*closeness to being fulfilled*". To formally express the issue, let us consider a situation that must be compared against two rules, both of which have the same number of conditions to check. The trees of these rules would then have the same maximum depth. For the first rule, let us assume that the situation meets all conditions except one, and this condition is very high up in the tree, among the first conditions checked.

For the second rule, let us suppose that the situation meets almost all the conditions, except 3, but all these conditions are at the end of the tree, and are the very last conditions checked. According to our metric, the rule with the higher relative depth reached would be the second one, and yet, the first rule, with only 1 unfulfilled condition, should be considered *the closest of being respected*.

Although the naive approach proves to be sufficient in simple cases, the two flaws mentioned above make it ineffective in more complex cases, with rules involving numerous conditions to be checked to establish a proper context for the application of the rule.

This observation leads to the establishment of a more optimized strategy presented in the next section.

5.3.2. An Optimized Approach: Seeking All the Unfulfilled Conditions

This second approach aims at identifying not just the first, but all the unfulfilled decisive conditions in each applicable rule to the input situation. To do this, we need to check all the conditions of the rule, and note which ones are unfulfilled. We achieve this by applying the following procedure to each rule:

1) Construct the tree associated with the rule.

2) Find and take note of the first unfulfilled conditions by applying the naive approach.

3) Create a new version of the rule where the unfulfilled conditions have been

removed. Determining which part of the rule to remove is, however, not trivial. For example, if an unfulfilled condition is that a user is a member of a LEA, we should remove in the simplified version the dependent conditions that the LEA has jurisdiction over the place where the situation occurs, without removing the other conditions on that place. This step thus requires a dependency analysis among the conditions of the rule. The algorithm that carries out this analysis is described in detail below.

4) Create a new tree from the updated version of the rule.

5) Check again for the first unfulfilled conditions.

This procedure is repeated until the leaf reached in the tree is a leaf that signifies the rule is respected (a leaf labeled "*OK*" in our examples). By doing this, the initial rule gets split into two. On the one hand, a rule that is respected, composed of a subset of the conditions of the initial rule. We call it the "*Biggest respected sub-rule*". On the other hand, there is the set of all the unfulfilled conditions of the rule, that can then be requested from the user all at once.

However, one aspect of this procedure requires special treatment. As mentioned in the third point of the procedure, when creating the new version of the rule where the identified unfulfilled conditions are removed, a problem of dependency between variables can occur. The conditions expressed in SPARQL are in the form of RDF triples that link a subject with an object through a specific property, with some of the elements being part of an ontology, and the others being variables. If, during the checking of the individual conditions of a rule, an unfulfilled condition is found and that this condition introduces a new variable in the SPARQL query, getting rid of the condition also suppresses the introduction of the new variable, and further conditions that depend on this variable need to be removed from the rule as well before generating the tree. The difficulty lies in determining whether variables present in unfulfilled conditions are introduced into these conditions only.

To do so, we generalize our sequentially assumption into a sequence-parallel assumption that the ordering on conditions is partial, but that along each path from the root new variables are introduced in relation with already determined variables. We can then build a dependency graph, where nodes are the variables of the rule, and edges represent links that exist between these variables in the conditions of the rule. Generally, let us separate the conditions of a rule in three groups. 1) The conditions already validated, 2) the conditions that have just been evaluated as violated, and 3) the conditions that have not been verified yet.

When creating the updated version of the rule, all the conditions in the second group will be removed. Then we need to determine whether conditions in the third group should be removed too. To do this, we consider each condition in the third group in turn. For each condition, we note the variables that appear in it. Using our dependency graph, we can determine whether all the variables in the current condition depend on variables that have been removed. If that is the case,

the current unverified condition must be removed too. Otherwise, this means that there exists a link between the variables in this condition and the variables in validated conditions. In this case, we can keep it.

6. Conclusions

This paper presented reasoning and justification methods for a decision support framework. We notably focused on answering four research questions regarding the use of symbolic AI and semantic web for formalizing regulations, and the generation of justified suggestions, requiring, in cases where the reasoning fails to reach a conclusion, to identify the most relevant information to ask the user. After describing the architecture of the framework and its components, we presented our approach to formalizing legal rules. We then described the logic and algorithms behind the generation of a justified decision in our framework. Finally, to handle cases where not a single rule was satisfied, an innovative approach was presented to identify unfulfilled conditions in legal rules using decision tree structures. The use of the proposed framework has been illustrated in a use case of data sharing among European Law Enforcement Agencies (LEAs).

Several issues are still to be addressed in future work. First, the number of formal rules used to test our framework shall be extended, in order to see how it scales up to bigger use cases. This could be done by extracting rules from new directives that have come into force in 2023 regarding the data processing by LEAs[10] and the gathering of electronic evidence in criminal proceedings[11]. We would also like to automate the extraction process of new formal rules, using state-of-the-art Natural Language Processing methods [12]-[14]. While the formal language used in this study is SPARQL, other formal standards like LegalRuleML [1] could be tested to compare the results. Looking further ahead, we are also aware that the tests conducted on our framework are currently limited to proof-of-concepts, and we would like to establish a full benchmark to conduct large-scale experiments using real-world data. Finally, we plan to eventually expand our framework with a case-based reasoning module. This would complement rule-based reasoning by leveraging learning derived from past decision traces.

Acknowledgements

This work is partially funded by the H2020 project STARLIGHT ("Sustainable Autonomy and Resilience for LEAs using AI against High-priority Threats") that received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 101021797. We would also like to thank Ronan PONS, PhD student in Law, who assisted this work by providing his insight as a legal expert.

[10]Directive - 2023/977 - EN - EUR-Lex - European Union: https://eur-lex.europa.eu/legal-content/EN/HIS/?uri=oj:JOL_2023_134_R_0001

[11]Directive - 2023/1544 - EN - EUR-Lex - European Union: https://eur-lex.europa.eu/legal-content/EN/NIM/?uri=CELEX:32023L1544

Author Contributions

All authors have read and agreed to the published version of the manuscript.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Palmirani, M., Governatori, G., Rotolo, A., Tabet, S., Boley, H. and Paschke, A. (2011) LegalRuleML: XML-Based Rules and Norms. In: Olken, F., *et al.*, Eds., *Rule-Based Modeling and Computing on the Semantic Web*, Springer, 298-312. https://doi.org/10.1007/978-3-642-24908-2_30

[2] Gordon, T.F. (2008) Constructing Legal Arguments with Rules in the Legal Knowledge Interchange Format (LKIF). In: Casanovas, P., *et al.*, Eds., *Computable Models of the Law*: *Languages*, *Dialogues*, *Games*, *Ontologies*, Springer, 162-184. https://doi.org/10.1007/978-3-540-85569-9_11

[3] Lam, H. and Hashmi, M. (2019) Enabling Reasoning with LegalRuleML. *Theory and Practice of Logic Programming*, **19**, 1-26. https://doi.org/10.1017/s1471068418000339

[4] Lazarska, M. and Siedlecka-Lamch, O. (2019) Comparative Study of Relational and Graph Databases. 2019 *IEEE* 15*th International Scientific Conference on Informatics*, Poprad, 20-22 November 2019, 363-370. https://doi.org/10.1109/informatics47936.2019.9119303

[5] Medhi, S. and Baruah, H. (2017) Relational Database and Graph Database: A Comparative Analysis. *Journal of Process Management. New Technologies*, **5**, 1-9. https://doi.org/10.5937/jouproman5-13553

[6] Bouché-Pillon, J. (2025) A Decision Support System for Digital Law: An Approach Based on Ontologies, Deontic Logic and Access Control. Ph.D. Thesis, Université de Toulouse.

[7] Bouché-Pillon, J., Aussenac-Gilles, N., Chevalier, Y. and Zaraté, P. (2024) An Ontology for Legal Reasoning on Data Sharing and Processing between Law Enforcement Agencies. 3*rd International Workshop on Knowledge Management and Process Mining for Law* (*KM4LAW*2024).

[8] Bouche-Pillon, J., Aussenac-Gilles, N., Chevalier, Y. and Zarate, P. (2024) Decision Support in Law: From Formalizing Rules to Reasoning with Justification. *JURIX* 2024: *The Thirty-Seventh Annual Conference*, Brno, 11-13 December 2024, 25-36. https://doi.org/10.3233/faia241231

[9] Marakas, G.M. (2003) Decision Support Systems in the 21st Century (Vol. 134). Prentice Hall.

[10] Polleres, A. and Wallner, J.P. (2013) On the Relation between SPARQL1.1 and Answer Set Programming. *Journal of Applied Non-Classical Logics*, **23**, 159-212. https://doi.org/10.1080/11663081.2013.798992

[11] Lee, J. and Palla, R. (2009) System F2LP-Computing Answer Sets of First-Order Formulas. In: Erdem, E., *et al.*, Eds., *International Conference on Logic Programming and Nonmonotonic Reasoning*, Springer, 515-521. https://doi.org/10.1007/978-3-642-04238-6_51

[12] Fawei, B. (2024) NLP-Based Rule Learning from Legal Text for Question Answering. *Asian Journal of Research in Computer Science*, **17**, 31-40.

https://doi.org/10.9734/ajrcos/2024/v17i7475

[13] Ferraro, G., Lam, H., Tosatto, S.C., Olivieri, F., Islam, M.B., van Beest, N., *et al.* (2020) Automatic Extraction of Legal Norms: Evaluation of Natural Language Processing Tools. In: Sakamoto, M., *et al.*, Eds., *New Frontiers in Artificial Intelligence*, Springer International Publishing, 64-81. https://doi.org/10.1007/978-3-030-58790-1_5

[14] Recski, G., Lellmann, B., Kovacs, A. and Hanbury, A. (2021) Explainable Rule Extraction via Semantic Graphs. ASAIL/LegalAIIA@ ICAIL, 24-35.

[15] Gandon, F., Governatori, G. and Villata, S. (2017) Normative Requirements as Linked Data. *JURIX* 2017—*The* 30*th International Conference on Legal Knowledge and Information Systems*, Luxembourg, 13-15 December 2017, 1-10.

[16] Pérez, J., Arenas, M. and Gutierrez, C. (2009) Semantics and Complexity of SPARQL. *ACM Transactions on Database Systems*, **34**, 1-45. https://doi.org/10.1145/1567274.1567278

[17] Jones, A.J.I. and Sergot, M. (1992) Deontic Logic in the Representation of Law: Towards a Methodology. *Artificial Intelligence and Law*, **1**, 45-64. https://doi.org/10.1007/bf00118478

[18] Navarro, P.E. and Rodríguez, J.L. (2014) Deontic Logic and Legal Systems. Cambridge University Press. https://doi.org/10.1017/cbo9781139032711

[19] Moretti, A. (2009) The Geometry of Standard Deontic Logic. *Logica Universalis*, **3**, 19-57. https://doi.org/10.1007/s11787-009-0003-4

Appendix 1

Algorithm 2: Tree traversal in the naive approach

```
Data: query, tree, endpoint ;      /* Body of the SPARQL query, root of associated
      tree, endpoint to execute SPARQL queries */
Result: (node, unrespected) ;              /* Node reached by traversing the tree,
        unrespected conditions */
curr_query ← "ASK{"                              // ASK query currently verified
node ← tree                                    // node initialized with the root
unrespected ← []
temp ← ""                                          // temporary part of ASK query
while node is not a leaf do
    if node is the beginning of union sub-block then       // content of UNION temporary
        temp ← ""
        temp_query ← curr_query
        while node not the end of UNION sub-block do
            temp ← temp + node.content   // adding condition in the node to temp
            temp_query ← curr_query + temp            // adding temp to the query
            result = ask_query_execution(temp_query, endpoint)      // boolean result
            (node, unrespected) ← next_node(result, node, unrespected)   // see Alg. 3
        end
                                                                  // End of UNION sub-block
        temp ← temp + node.content
        temp_query ← curr_query + temp
        result ← ask_query_execution(temp_query, endpoint)
        (node, unrespected) ← next_node(result, node, unrespected)
        if node is not the beginning of a new UNION sub-block then          // We finished
         traversing the whole UNION Block
            curr_query ← temp_query          // the temp query becomes definitive
    else                                                  // Not dealing with UNION
        curr_query ← curr_query + node.content
        result ← ask_query_execution(curr_query, endpoint)
        (node, unrespected) ← next_node(result, node, unrespected)
    end
end
return (node, unrespected)
```

Appendix 2

Algorithm 3: next_node: Function for advancing in the tree depending on the result of the ASK query

```
Data: result, node, unrespected ;   /* boolean result of ASK query, current node,
      list of unrespected conditions */
Result: (node, unrespected) ;            /* New node, updated list unrespected */
if result then                      // ASK query returned true: condition verified
    node ← node.left                                  // condition OK: left branch
else
    add node content to unrespected
    node ← node.right                              // condition NOK : right branch
end
return (node, unrespected)
```