Deontic Logic Reified: a deontic logic for the Semantic Web

Livio Robaldo [a,*]
[a] *HRC School of Law, Swansea University, Wales, UK*
E-mail: livio.robaldo@swansea.ac.uk

Abstract. This paper introduces Deontic Logic Reified, a novel deontic logic implemented in RDF and SPARQL for automated compliance checking over Semantic Web data. The framework is designed to support scalable deontic reasoning by leveraging RDF as a standard knowledge representation layer, thereby enabling seamless interoperability with existing data ecosystems within the Semantic Web. Deontic Logic Reified incorporates and extends research that I have conducted over the last five years, primarily presented in three key publications of which I am the main author: [1], [2], and [3]. This paper integrates the results of these contributions while addressing their complementary limitations. In particular, the paper extends the RDF/SPARQL framework in [3], which provides the core building blocks reused here, with constructs investigated and compared in [2], which surveys the main legal reasoners in the contemporary literature on compliance checking, as well as with constructs for implementing a temporal dimension for legal norms, which has been studied in the literature primarily from a theoretical perspective despite its crucial role in compliance checking. These extensions enable the resulting framework to achieve large-scale legislative coverage in a unified and executable manner. To demonstrate its applicability, a substantial subset of norms from the European General Data Protection Regulation (GDPR), previously formalised in [1], is translated into the proposed formalism and executed over RDF data. The results show improved computational performance compared to the SHACL-based reasoner surveyed in [2], which is the only one among those considered that is capable of natively processing RDF data. All examples presented in this paper are available in https://github.com/liviorobaldo/dlr-swj, together with Java procedures to execute them.

Keywords: Compliance checking, Semantic Web, Deontic logic, Symbolic AI

1. Introduction

LegalTech, i.e., the use of Artificial Intelligence (AI) for the legal domain, is experiencing significant growth, including at the industrial level, due to the ever-increasing amount of compliance filing documents that companies must submit to governmental authorities in order to demonstrate compliance with in-force regulations [4].

Current LegalTech solutions available on the market predominantly rely on Machine Learning (ML) techniques. In particular, in recent years a growing number of approaches have been based on Large Language Models (LLMs), motivated by their strong empirical results across a wide range of language understanding and text processing tasks. These systems are designed to approximate aspects of legal decision-making in order to support and partially automate the work of legal practitioners [5], [6], [7], [8], [9].

However, ML is fundamentally based on statistical reasoning. As a consequence, it grounds outcomes on patterns extracted from historical data. Law, however, is not fixed in time; it evolves and adapts alongside technical, societal, and ethical advancements [10], [11]. Decision-making procedures trained on historical data therefore tend to

*Corresponding author. E-mail: livio.robaldo@swansea.ac.uk.

replicate past reasoning patterns that may no longer be ethically acceptable, and may consequently lead to outcomes considered discriminatory under current ethical and societal standards [12].

More generally, legal reasoning requires complex multi-step processes, in which each intermediate step must have an explicit logical connection to both preceding steps and the relevant legal norms. Any weak or unsupported link can invalidate the entire argument and undermine the legitimacy of the legal conclusion [13]. The opacity of LLMs, in which legal knowledge is embedded in billions of neural network parameters rather than in explicit rules, thus poses a critical challenge for earning trust in a domain where interpretability and correctness are paramount.

To mitigate these limitations, LegalTech applications should integrate *symbolic representations* more systematically. Unlike purely data-driven approaches, symbolic formalisms enable explicit, human-understandable forms of logical reasoning and make intermediate inference steps transparent and verifiable, thereby supporting correctness, accountability, and legal interpretability in compliance-related tasks.

Nevertheless, building and maintaining up-to-date symbolic representations is highly time-consuming, as it requires substantial manual effort. A natural approach to alleviate this burden is to adopt *standardized formats* capable of funneling efforts from multiple contributors, as well as the sharing and reuse of resources.

The main hypothesis of this paper, and of my recent research efforts as a whole, is that the Resource Description Framework (RDF) is the best candidate for this purpose. Almost thirty years after it was first proposed and standardized as a W3C recommendation, a large number of RDF datasets have become publicly available on the Web, and continue to grow in both number and size. In parallel, a rich ecosystem of libraries has been developed to connect RDF standards with mainstream programming languages and efficient database management systems, thereby enabling the use of RDF datasets in industrial contexts.

It is therefore reasonable to claim that RDF is perhaps the most widely used knowledge representation format worldwide, that it is likely to increasingly serve as a foundation for symbolic AI, and that, consequently, it constitutes a natural choice as the underlying framework for LegalTech applications. Since legislation regulates all aspects of everyday life and society, many categories of big data, such as financial or healthcare data, must comply with, and are thus highly dependent on, in-force norms [14]. An RDF-based LegalTech framework will therefore allow legislative information to be directly matched against big data already encoded in RDF, without incurring the computational costs associated with format conversion.

However, to the best of my knowledge, although several frameworks for checking compliance over RDF data have been proposed in the literature (e.g., [15], [16], [17], [18], [19], among others), these approaches have so far been evaluated primarily on constructed examples or restricted use cases. It therefore remains underexplored how they can be scaled to handle legislation more broadly.

The present paper aims to take a significant step towards filling this gap. It introduces a novel framework, based on RDF and SPARQL, that is suitable for encoding a reasonably wide and representative set of norms drawn from existing legislation. The proposed framework builds upon and extends results from three of previous journal publications of which I am the primary author: [1], [2], and [3].

[1] was used to guide the design of the proposed framework, in particular to achieve the aforementioned wide and representative coverage. [1] introduced the DAPRECO knowledge base (D-KB), which formalises the norms of the European General Data Protection Regulation (GDPR)[1] into 966 formulae in reified Input/Output logic [20], a first-order deontic logic specifically designed for modeling natural language semantics. Given that I am not aware of any other publicly available repository of digital legal norms encoded in first-order formulae larger than the D-KB, and given the central role of the GDPR within the European legislative framework, the D-KB can be regarded as a reasonably representative benchmark for existing legislation.

Nevertheless, reified Input/Output logic is incompatible with RDF, and no automated reasoner for it has ever been implemented. As a consequence, the results in [1] remain purely theoretical: they provide an analysis of the expressivity required to represent GDPR norms, but the corresponding formulae are not directly usable in practical applications. This paper will translate the reified Input/Output logic formulae corresponding to the norms from Chapter II of the GDPR into the framework proposed here; the norms in this chapter are representative of the variety identified in the D-KB, as explained in Section 5 below.

[1] https://eur-lex.europa.eu/eli/reg/2016/679/oj

[2] was instead used to guide the design of the proposed framework from a computational perspective. [2] surveys and compares the main legal reasoners recently proposed in the LegalTech literature for compliance checking, among which is a SHACL-based reasoner that I previously proposed in [18], which is the only one capable of natively processing RDF data. The comparison presented in [2] focuses on the trade-off between usability and efficiency of the legal reasoners under investigation; however, both the experiments and the analysis in [2] are based exclusively on constructed examples, whereas the present paper, as explained in the previous paragraph, uses the norms from Chapter II of the GDPR as a benchmark.

The results in [2] show that legal reasoners based on Answer Set Programming (ASP) are the most efficient, followed by the SHACL-based reasoner that I previously proposed in [18]. Among the ASP-based reasoners, it is worth mentioning DLV2 [21], which, although not natively capable of processing RDF data, allows RDF imports via dedicated libraries. DLV2 and my SHACL-based reasoner were specifically compared in [22]. Therefore, DLV2 is the legal reasoner capable of processing RDF data with the best reported performance. However, it also presents some disadvantages with respect to my SHACL-based reasoner. First, when used with SPARQL endpoints, as would be the case in a real-world setting, it requires a dedicated ASP import statement for each RDF class and property in the TBox, which can become rather cumbersome when the number of classes and properties is large. Secondly, DLV2 is distributed under a commercial license, is implemented in C++, and runs only on the Linux operating system, whereas the Java libraries used to run the SHACL-based reasoner from [18], as well as the one presented in this paper, run on all major operating systems and are open source, namely released under the Apache License 2.0.

However, both the ASP- and the SHACL-based reasoners share a disadvantage with respect to other available legal reasoners: they both implement defeasibility, a key component for handling exceptions in legal reasoning, through *negation-as-failure*, which significantly undermines usability, particularly for legal practitioners, who are typically unfamiliar with logical formalizations and related concepts. Moreover, negation-as-failure is non-monotonic and can therefore lead to non-deterministic outcomes: depending on rule ordering, different conclusions may be derived. While ASP addresses this issue through its stable-model semantics, which imposes an *implicit* execution order among rules, it remains an open problem in SHACL, where priorities among rules must be specified *explicitly* to avoid non-determinism, further complicating usability, in particular for large knowledge bases.

As an alternative to negation-as-failure, two of the legal reasoners compared in [2], SPINdle [23] and Proleg [24], propose the use of *superiority relations* among norms. In these approaches, norms are typically labeled, and a separate meta-level is used to specify which norms override others based on their labels. When a general norm is overridden, its conclusion is defeated and excluded from the compliance-checking process. Superiority relations allow rules to remain monotonic, as they avoid the use of negation-as-failure and other non-monotonic operators to implement defeasibility. Another advantage, as shown in [25], is that superiority relations are far more intuitive for legal practitioners, who only need to specify which norms act as context-specific exceptions to more general ones.

The framework proposed in this paper differs from the SHACL-based one from [18] because, like SPINdle and Proleg, it makes use of superiority relations. Since superiority relations do not require the specification of priorities among rules, SHACL will no longer be needed to implement the framework, but only SPARQL.

The previous paper of mine on which the framework proposed here is primarily based is, however, [3]. This paper provides the basic formal machinery that is reused and extended in the present work. It introduces a basic deontic logic, encoded in RDF and SPARQL and grounded in RDF reification[2], which implements the well-known Deontic Traditional Scheme[3]. This scheme constitutes the minimal set of axioms widely recognized as the foundation of any deontic logic. However, [3] only handles *irresolvable conflicts*, i.e., situations in which two or more norms conflict without either overriding the other. In other words, [3] is unable to handle *resolvable conflicts*, where one norm represents a more specific exception to another. The framework from [3] will be illustrated in the next section.

Section 3 then extends the basic framework from [3] by incorporating constructs for defeasibility, specifically mechanisms for handling resolvable conflicts via exception handling. This extension, as explained above, makes use of superiority relations mirroring what is done in SPINdle and Proleg.

Section 4 further extends the framework by incorporating another essential component of legal reasoning, that is specifically needed to translate into SPARQL the reified Input/Output logic from the D-KB: the temporal dimension.

[2] https://www.w3.org/TR/rdf11-mt/#reification

[3] https://plato.stanford.edu/entries/logic-deontic/#TradScheModaAnal

In real-world regulatory contexts, norms are rarely timeless; they typically specify, either explicitly or implicitly, the time frame during which they are in force and against which compliance must be assessed. As discussed in Section 4, the temporal dimension has been somewhat neglected in automated legal reasoners proposed in the literature, despite its crucial role. It was also not considered in the experiments reported in [2], nor in the formalization in [3]. However, a body of prior work proposes *theoretical* logical systems that incorporate temporal aspects into legal reasoning; this literature is taken into account in Section 4 for the design of the constructs to enable time management within norms in the proposed framework.

Section 5 then illustrates the translation of the reified I/O logic formulae from the D-KB corresponding to the norms in Chapter II of the GDPR into the proposed framework. Defeasibility and temporal reasoning were the only missing elements in the framework from [3] required to translate the formulae in the D-KB.

Finally, Section 6 presents a generate-and-test methodology for formally validating the translated formulae. Section 7 demonstrates how the encoded representations can be automatically executed on sample ABoxes. It also compares the performance obtained with that of the SHACL reasoner from [18] executed on the same representations. The results show that the proposed approach processes the data in less computational time.

The overall result of this paper is a new deontic logic, called "Deontic Logic Reified". To the best of my knowledge, Deontic Logic Reified is the only deontic logic for the Semantic Web that provides representative coverage of norms drawn from existing legislation, while achieving satisfactory computational performance.

The final section preceding the Conclusions, i.e., Section 8, outlines several directions for future research aimed at extending both the expressiveness and the scalability of Deontic Logic Reified. In particular, it discusses the introduction of generalized quantifiers, as well as richer structures for representing conjunctions and disjunctions of eventualities. These extensions are necessary to capture more complex forms of legislative norms beyond those considered in this paper. Moreover, Section 8 emphasizes that a key limitation of the present approach is the substantial manual effort required to construct and maintain large knowledge bases of symbolic representations. To address this issue, a central research direction is the investigation of Large Language Models (LLMs) for generating *draft* Deontic Logic Reified representations from legislative texts, which can then be validated and refined within human-in-the-loop and semi-automated workflows.

2. The basic framework from [3]

The basic formulae employed in the framework proposed in this paper follow this pattern, from [3]:

$$(1)\quad (\neg)\,\mathtt{M(e)} \wedge \mathtt{ET(e)} \wedge \bigwedge_{i=1}^{n} \mathtt{t_i(e, v_i)}$$

In (1), the term `e` is referred to as an "eventuality", following [26][4] and [27]. In general, an eventuality represents the *reification* of an abstract entity. However, in the pattern shown in (1) and throughout the paper, eventualities such as `e` can denote only actions or states, as specified by the predicate `ET`, which stands for "Eventuality Type".

The symbols `ti` represent thematic roles, while `vi` denote their corresponding values. Two thematic roles frequently used below are `has-agent` and `has-object`. Another thematic role, discussed in detail in Section 4, is `has-time`, which is used to represent the temporal dimension of eventualities. For simplicity, it is assumed that each formula presented below includes the `has-agent` thematic role[5].

Finally, the predicate `M` expresses the modality of the eventuality. In general, an eventuality may occur in reality, exist only in someone's imagination, or be merely possible. However, not all such modalities are relevant here. The only values considered for `M` in this work are `Rexist`, which indicates the "real existence" of an eventuality[6], and

[4] See also https://www.isi.edu/people-hobbs/research-areas/encoding-commonsense-knowledge.

[5] This assumption is justified by the focus of the paper on modeling normative rules, which always have a bearer, typically the agent. Indeed, there is extensive literature on the interaction between deontic modalities and agency; see https://plato.stanford.edu/entries/logic-deontic/#DeonLogiAgen for a general overview of the topic.

[6] [3] also defines another modality, `Subsist`, to indicate the existence of an eventuality. The difference between `Rexist` and `Subsist` lies in the type of eventuality they apply to: the former refers to concrete eventualities, while the latter refers to abstract ones. Since the distinction between concrete and abstract eventualities is not relevant in this paper, only `Rexist` is used to assert the existence of eventualities.

the deontic modalities `Obligatory` and `Permitted`. These deontic modalities indicate, respectively, that the agent is obliged or permitted to perform a given action or be in a certain state. Modalities can also be negated; in other words, the pattern in (1) can also be used to specify that an eventuality `e` does not really exist ($\neg$`Rexist(e)`), is not obligatory ($\neg$`Obligatory(e)`), or it it not permitted, i.e., it is *prohibited* ($\neg$`Permitted(e)`).

Two examples of formulae following the pattern in (1) are shown in (2). Example (2.a) states that `John` pays `rj` at `3pm`. This is expressed by specifying that the eventuality `ej`, which represents an action of "paying", has `John` as its agent, `rj` as its object, and `3pm` as its time, really exists in the state of affairs. Conversely, in (2.b), `eb` represents an action of "leaving" with `Bill` as the agent; (2.b) asserts that Bill is not obliged to leave.

(2) a. $\texttt{Rexist(ej)} \wedge \texttt{Pay(ej)} \wedge \texttt{has-agent(ej, John)} \wedge \texttt{has-object(ej, rj)} \wedge \texttt{has-time(ej, 3pm)}$

b. $\neg\texttt{Obligatory(eb)} \wedge \texttt{Leave(eb)} \wedge \texttt{has-agent(eb, Bill)}$

The pattern in (1) represents only the *basic* structure. In the proposed framework, additional predications can also be conjoined to the formulae. For example, `Person(John)` and `rent-of(rj, John)` might be added to (2.a) to explicitly indicate that `John` is a person and `rj` is `John`'s rent. The formula in (2.a) is then expanded as follows:

(3) $\texttt{Rexist(ej)} \wedge \ldots \wedge \texttt{has-time(ej, 3pm)} \wedge \texttt{Person(John)} \wedge \texttt{rent-of(rj, John)}$

An important predicate in Hobbs's logic is the binary predicate `not`, which connects an eventuality with its (what Hobbs calls) "negated eventualities". The predication `not(e, en)` states that `e` and `en` are mutually negating eventualities. For example, if `enb` denotes the fact that Bill stays, we can write `not(eb, enb)`, where `eb` denotes the fact that Bill leaves in (2.b), to express that the two eventualities "negate" each other.

The `not` predicate is axiomatized with respect to the defined modalities. For instance, with respect to the three modalities introduced above, the following axioms hold for `not`:

(4) a. $\forall_{\texttt{e, en}}\,[\,\texttt{not(en, e)} \rightarrow (\texttt{Rexist(e)} \leftrightarrow \neg\texttt{Rexist(en)})\,]$

b. $\forall_{\texttt{e, en}}\,[\,\texttt{not(en, e)} \rightarrow (\texttt{Permitted(e)} \leftrightarrow \neg\texttt{Obligatory(en)})\,]$

Therefore, if `eb` really exists, i.e., if Bill really leaves, then it is false that `enb` really exists, meaning that it is not true that Bill stays. Conversely, if `eb` is not obligatory, as in (2.b), (4.b) allows us to derive that Bill is permitted to stay. If, instead, `eb` were obligatory, (4.b) would imply that Bill is *not* permitted, i.e., prohibited, to stay.

Hobbs also defines two similar predicates, `and` and `or`, to represent the conjunction and disjunction of eventualities. While Hobbs axiomatizes them with respect to the `Rexist` modality[7], [3] axiomatize them with respect to the two deontic modalities `Obligatory` and `Permitted`. However, since the axiomatization of `and` and `or` is quite complex and the D-KB contains only a few simple cases involving these two predicates, the present paper omit them, while dealing with the simple cases in the D-KB via some ad-hoc solution (see Section 5 below).

Similarly, this paper omits the SPARQL rules presented in [3] for inferring when two eventualities contradict each other (for example, when one really exists and the other does not) or when they are in conflict (for example, when one is permitted and the other is not). This paper does not address the detection of conflicts, contradictions, or other abnormalities; the corresponding inference rules from [3] are therefore omitted.

This paper, instead, focuses on advocating LegalTech applications capable of automatically checking compliance against the D-KB or similar normative repositories. To this end, it introduces rules for compliance checking, for example, to infer when a really existing eventuality complies with an obligation or violates a prohibition. Some of these rules were already defined in [3] and will be presented in the next subsection. This paper extends and generalizes them, in particular to account for defeasible rules and temporal constraints.

[7] See https://www.isi.edu/people-hobbs/wp-content/uploads/sites/58/2023/10/bgt-logic.txt

2.1. Inferring compliance, violations, and more

This subsection presents several formulae for enforcing compliance checking within a normative system. These formulae will be implemented in RDF and SPARQL in subsection 2.3, making them executable.

[3] introduce rules to infer compliance with obligations and violations of prohibitions (i.e., negated permissions). These rules allow one to determine whether agents have actually performed the actions they were obliged to carry out and those they were prohibited from performing. In Hobbs's logic, these could be formalized as in (5).

(5) $\forall_{\texttt{eo,e}}[(\texttt{Obligatory(eo)} \wedge \texttt{Rexist(e)} \wedge \texttt{ET(eo)} \wedge \texttt{ET(e)} \wedge$
$\neg\exists_{\texttt{tr,vo}}[\texttt{tr(eo, vo)} \wedge \neg\exists_{\texttt{v}}[\texttt{tr(e, v)}]] \wedge$
$\neg\exists_{\texttt{tr,vo,v}}[\texttt{tr(eo, vo)} \wedge \texttt{tr(e, v)} \wedge (\texttt{vo} \neq \texttt{v})] \rightarrow \texttt{complies-with(e, eo)}]$

$\forall_{\texttt{enp,e}}[(\neg\texttt{Permitted(enp)} \wedge \texttt{Rexist(e)} \wedge \texttt{ET(enp)} \wedge \texttt{ET(e)} \wedge$
$\neg\exists_{\texttt{tr,vnp}}[\texttt{tr(enp, vnp)} \wedge \neg\exists_{\texttt{v}}[\texttt{tr(e, v)}]] \wedge$
$\neg\exists_{\texttt{tr,vnp,v}}[\texttt{tr(enp, vnp)} \wedge \texttt{tr(e, v)} \wedge (\texttt{vnp} \neq \texttt{v})] \rightarrow \texttt{violates(e, enp)}]$

In (5), the two "$\neg\exists$" conditions in the antecedent of the implications specify that there must be no thematic role `tr` assigned to the obligation `eo` (or the prohibition `enp`) that is not also assigned to the really existing eventuality `e`, and no thematic role `tr` assigned to both but with differing values, `vo` and `v`. In other words, these two clauses ensure that the rule in (5) links obligations (or prohibitions) to really existing eventualities representing valid ways of complying with (or violating) them, that is, to possible *specializations* of the generic eventuality described in the obligation (or prohibition). For example, if John is obliged to pay the rent before the end of the month, he may do so in cash, by card, or via bank transfer at any time before the deadline. Each of these variants constitutes a compliant instantiation of the obligatory payment.

Two additional rules are introduced for permissions and non-obligations in (6). Symmetrically to the rules in (5), the first rule in (6) infers that whenever a really existing eventuality specializes a permitted one, the former conforms to the latter, meaning that the agent of the former performs an action that is indeed permitted. This relation is represented by the predicate `conforms-to`. Regarding non-obligations, to the best of my knowledge there is no single verb that succinctly captures the case in which an agent performs an action without being obliged to do so. Therefore, the second rule in (6) infers that the eventuality occurs even though it is not obligatory, using the (admittedly verbose) predicate `rexists-although-not-obligatory`.

(6) $\forall_{\texttt{ep,e}}[(\texttt{Permitted(ep)} \wedge \texttt{Rexist(e)} \wedge \texttt{ET(ep)} \wedge \texttt{ET(e)} \wedge$
$\neg\exists_{\texttt{tr,vp}}[\texttt{tr(ep, vp)} \wedge \neg\exists_{\texttt{v}}[\texttt{tr(e, v)}]] \wedge$
$\neg\exists_{\texttt{tr,vp,v}}[\texttt{tr(ep, vp)} \wedge \texttt{tr(e, v)} \wedge (\texttt{vp} \neq \texttt{v})] \rightarrow \texttt{conforms-to(e, ep)}]$

$\forall_{\texttt{eno,e}}[(\neg\texttt{Obligatory(eno)} \wedge \texttt{Rexist(e)} \wedge \texttt{ET(eno)} \wedge \texttt{ET(e)} \wedge$
$\neg\exists_{\texttt{tr,vno}}[\texttt{tr(eno, vno)} \wedge \neg\exists_{\texttt{v}}[\texttt{tr(e, v)}]] \wedge$
$\neg\exists_{\texttt{tr,vno,v}}[\texttt{tr(eno, vno)} \wedge \texttt{tr(e, v)} \wedge (\texttt{vno} \neq \texttt{v})] \rightarrow$
$\texttt{rexists-although-not-obligatory(e, eno)}]$

2.2. Representing norms

The previous subsection illustrated how simple sentences are formalized within the proposed framework. However, all examples involved single individuals, such as `John`, `Bill`, `3pm` or `John`'s rent.

On the other hand, legislative norms typically do not apply to *single* individuals such as `John` and `Bill`, the so-called "bearers" of the norm, but rather to *sets* of bearers, namely all individuals who belong to a given category, perform specific actions, or satisfy particular conditions. For this reason, standard legal theory generally represents norms as *if-then rules*, in which the antecedent specifies the conditions an entity must satisfy in order to qualify as a bearer of the norm (see [28], among others).

Furthermore, standard legal theory classifies norms into two main types based on the nature of their consequents: *regulative* norms and *constitutive* norms. Regulative norms establish deontic statements, i.e., they specify a deontic

modality in their consequent. In contrast, constitutive norms specify the criteria by which entities or actions acquire particular legal statuses, thereby determining what "counts as" the concepts used in regulative norms [29].

Examples of regulative and constitutive rules are provided in (7.a-b). Unlike the previous examples, which apply exclusively to single individuals such as John and Bill, (7.a) is an if-then rule stipulating that *every man* is obliged to pay his rent. To determine whether this obligation has been fulfilled for some individuals, the knowledge base could include additional rules that specify sufficient conditions for deriving when some of the predicates in the consequent of (7.a) are true. In other words, these extra rules define what "counts as" the payment of the rent. For example, as specified in (7.b), if someone gives to his landlord an amount of money equal to the value of his rent, it can be inferred that he has paid it. Similar constitutive rules can be added; for instance, if some tenants do not have money, some landlords might accept material services in exchange for rent (e.g., mowing the landlord's lawn or similar). In such cases, these services also "count as" the payment of the rent.

(7) a. $\forall_{\mathtt{m,r}}\,[\,(\mathtt{Man(m)} \wedge \mathtt{rentOf(m,r)}) \rightarrow$
$\exists_{\mathtt{e}}\,[\mathtt{Obligatory(e)} \wedge \mathtt{Pay(e)} \wedge \mathtt{has\text{-}agent(e,m)} \wedge \mathtt{has\text{-}object(e,r)}\,]\,]$

b. $\forall_{\mathtt{eg,m,v,l,r}}[(\mathtt{Rexist(eg)} \wedge \mathtt{Give(eg)} \wedge \mathtt{has\text{-}agent(eg,m)} \wedge \mathtt{has\text{-}object(eg,v)} \wedge$
$\mathtt{has\text{-}recipient(eg,l)} \wedge \mathtt{valueOf(v,r)} \wedge \mathtt{rentOf(r,m)} \wedge \mathtt{landlordOf(l,m)}) \rightarrow$
$\exists_{\mathtt{ep}}\,[\mathtt{Rexist(ep)} \wedge \mathtt{Pay(ep)} \wedge \mathtt{has\text{-}agent(ep,m)} \wedge \mathtt{has\text{-}object(ep,r)}\,]\,]$

The syntax of regulative and constitutive rules is formally defined as follows:

(8) a. Regulative rules:
$\forall_{\mathtt{x1,\ldots,xn}}\,[\mathtt{Condition(x1\ldots,xn)} \rightarrow \exists_{\mathtt{y1,\ldots,ym}}\,[\mathtt{Regulative((x1,\ldots,xn,y1,\ldots,ym)}\,]\,]$

b. Constitutive rules:
$\forall_{\mathtt{x1,\ldots,xn}}\,[\mathtt{Condition(x1\ldots,xn)} \rightarrow \exists_{\mathtt{y1,\ldots,ym}}\,[\mathtt{Constitutive((x1,\ldots,xn,y1,\ldots,ym)}\,]\,]$

Where `Condition`, `Regulative`, and `Constitutive` are conjunctions of atomic predicates. These conjunctions are further constrained as follows:

(9) a. `Condition`: it *might* include one or more formulae in the pattern shown above in (1).

b. `Regulative`: it *must* include one or more formulae in the pattern shown above in (1), but only with the `Obligatory` or `Permitted` modalities (or their negations). On the other hand, it *cannot* include formulae in the pattern shown above in (1) with the `Rexist` modality.

c. `Constitutive`: it *might* include one or more formulae in the pattern shown above in (1), but only with the `Rexist` modality. On the other hand, it *cannot* include formulae in the pattern shown above in (1) with the `Obligatory` or `Permitted` modalities (or their negations).

Therefore, `Condition` may consist of a single predication, such as `Man(m)` in (8.a), or it may include one or more basic formulae following the pattern shown in (1), with any[8] modality. On the other hand, the consequent of regulative and constitutive rules always entails at least one formula following the pattern in (1), involving either the deontic modalities or the `Rexist` modality, as specified in (9.b-c).

[8] It may also include basic formulae following the pattern in (1) that involve one of the two deontic modalities or their negations. In such cases, the rule incorporates what [1], §4.1, terms "nested obligations/permissions". The D-KB contains a few norms featuring nested obligations/permissions; see Subsection 5.6 below.

2.3. Implementing the basic framework in RDF and SPARQL

As mentioned earlier, this paper is part of my broader research objective of developing a deontic logic for the Semantic Web, capable of directly checking compliance over RDF data. For this reason, the formulae presented above are implemented in RDF and SPARQL. This section presents these implementations.

The predicates introduced in the previous subsection can be easily implemented in RDF. In fact, the formulae presented above use only unary predicates, such as `Pay`, `Rexist`, and `Obligatory`, and binary predicates, such as `complies-with` or the thematic roles. These predicates can be directly mapped to corresponding RDF classes and properties, respectively.

On the other hand, standard negation (operator "¬") is not directly supported in RDF, which instead follows the Open World Assumption; therefore, its implementation is not straightforward. To represent standard negation, [3] propose introducing two special classes, `true` and `false`, and asserting as instances of the latter the *reifications* of all triples that do *not* hold in the knowledge graph. Since standard RDF reification is rather verbose, [3], as well as the present paper, use the compact operator "<<...>>" defined in RDF*[9], which will also be included in the forthcoming RDF 1.2 recommendation[10]. Thus, ¬`Rexist(e)`, for instance, is represented as:

```
<<:e rdf:type :Rexist>> rdf:type :false.
```

This indicates that the fact that `e` really exists is false.

Triples that hold true in the knowledge graph could similarly be represented using the class `true`; however, since this is redundant in most cases, it is done only in exceptional situations. None of these exceptional cases are relevant here, so the class `true` will not be used below. [3] also define RDF classes and properties to implement standard disjunction (operator ∨) as well as possibility and necessity (operators ◊ and □). Since these are not relevant here either, these RDF classes and properties are omitted from this paper.

Regarding the entailments enforced by the formulae, since these are standard first-order logic entailments between conjunctions of atomic formulas, they can be implemented as SPARQL rules using the `CONSTRUCT-WHERE` form. When the knowledge graph contains a set of triples matching the pattern in the `WHERE` clause, these rules infer and add the corresponding triples specified in the `CONSTRUCT` clause.

As in [3], the SPARQL rules are processed by a simple Java application built on Apache Jena[11]. This application repeatedly executes the SPARQL rules until no new triples are asserted. However, as discussed below in (13), repeated execution can pose a major problem when the `CONSTRUCT` clause introduces new anonymous individuals, as the application may enter an infinite loop. To prevent this, special `NOT EXISTS` clauses will be added.

The first formula shown above in (5) is implemented as a SPARQL rule as follows, with the clause `NOT EXISTS` paralleling the operators "¬∃":

```
(10) CONSTRUCT{?e :complies-with ?eo}
     WHERE{?eo a :Obligatory,?ET. ?e a :Rexist,?ET. ?ET a :EventualityType.
       NOT EXISTS{?tr a :ThematicRole. ?eo ?tr ?vo. NOT EXISTS{?e ?tr ?v}}
       NOT EXISTS{?tr a :ThematicRole. ?eo ?tr ?vo. ?e ?tr ?v. FILTER(?vo!=?v)}}
```

The second formula in (5) is implemented in a similar way, with the only difference being that the predicate `Permitted` is negated. Consequently, the SPARQL translation requires the use of the <<...>> operator:

```
(11) CONSTRUCT{?e :violates ?eo}
     WHERE{<<?enp a :Permitted>> a :false. ?enp a ?ET.
           ?e a :Rexist,?ET. ?ET a :EventualityType.
           NOT EXISTS{?tr a :ThematicRole. ?enp ?tr ?vnp. NOT EXISTS{?e ?tr ?v}}
           NOT EXISTS{?tr a :ThematicRole. ?enp ?tr ?vnp. ?e ?tr ?v.
                                                   FILTER(?vnp!=?v)}}
```

[9] https://www.w3.org/groups/wg/rdf-star
[10] https://www.w3.org/TR/rdf12-primer
[11] https://jena.apache.org

The other two entailments in (6), which derive the conformity of permissions and the real existence of eventualities that are nonetheless not obligatory, are implemented in a similar manner and their SPARQL implementation is therefore omitted here; readers can find it directly in the GitHub repository associated with this paper.

As an illustration of the rule's execution, consider the following brief example. Suppose that John pays his rent by card at 3:15 p.m., represented in standard logic as in (12.b) and in RDF as in (12.c). The rule in (10) infers the triple "`:epj :complies-with :epjo`" from (12.c) and the RDF representation of (2.b): the fact that John pays his rent by card at 3.15pm complies with his obligation to pay his rent.

(12) a. John pays his rent by card at 3.15pm.

b. $\texttt{Rexist(epj)} \wedge \texttt{Pay(epj)} \wedge \texttt{has-agent(epj, John)} \wedge \texttt{has-object(epj, rent)} \wedge \texttt{has-instrument(elj, card)} \wedge \texttt{has-time(elj, 3.15pm)}$

c.
```
:epj a :Rexist, :Pay; :has-agent :John; :has-object :rent;
:has-instrument :card; :has-time :3.15pm
```

Let me now illustrate how to represent norms, such as the one seen in subsection 2.2 above, in SPARQL. Norms correspond to if-then rules that apply *universally* to all tuples of individuals quantified in the antecedent. For each of these tuples, the consequent is inferred. However, the consequent may also include additional variables that were not bound in the antecedent. These additional variables are *existentially* quantified in the consequent.

SPARQL rules in the form `CONSTRUCT-WHERE` allow for the representation of existential quantifications by introducing new anonymous individuals into the knowledge graph. More precisely, these new anonymous individuals correspond to *Skolemizations* of the existential quantifiers.

To improve readability, the proposed framework creates anonymous individuals in the `WHERE` clause, that is, the antecedent of the rule, using the SPARQL function `BNode()`. These newly created anonymous individuals are bound to variables that are then used in the `CONSTRUCT` clause. However, as mentioned above, to prevent infinite loops, every rule that introduces new anonymous individuals includes a corresponding `NOT EXISTS` clause to ensure that the rule is applied only if such an anonymous individual does not already exist in the knowledge graph.

For instance, the formula in (7.a), which states that every man is obliged to pay the rent, is represented in SPARQL as shown in (13). As encoded in the `NOT EXISTS` clause, this formula infers the obligation only if the knowledge graph does not already state that John has an obligation to pay the rent.

(13)
```
CONSTRUCT{?e a :Obligatory, :Pay; :has-agent ?m; :has-object :rent}
WHERE{?m a :Man. ?r :rent-of ?m. BIND(BNode() AS ?e)
    NOT EXISTS{?er a :Obligatory, :Pay; :has-agent ?m; :has-object :rent}}
```

In (13), `BNode()` creates a *new* RDF resource. However, since this resource does not have an explicit name, the SPARQL engine automatically assigns it an identifier to distinguish it from other resources. In this sense, the newly created individual is *anonymous*. This anonymous individual is bound to the variable `?e` using the SPARQL function `BIND`. The variable `?e` is then used in the `CONSTRUCT` clause to infer the new triples. A different variable, `?er` (where "`r`" stands for repeated"), is instead used in the `NOT EXISTS` clause; this avoids clashes with `?e`, which could otherwise result in unexpected behavior in the SPARQL engine. Since the content of the `NOT EXISTS` clause is identical to that of the `CONSTRUCT` clause, apart from the variable name, these additional clauses can be easily generated programmatically, as further explained in Section 5 below.

The formula in (7.b) is instead represented as follows:

(14)
```
CONSTRUCT{?e a :Rexist, :Pay; :has-agent ?m; :has-object ?r}
WHERE{?eg a :Rexist,:Give; :has-agent ?m; :has-object ?v; :has-recipient ?l.
  ?v :value-of ?r. ?r :rent-of ?m. ?l :landlord-of ?m. BIND(BNode() AS ?e)
  NOT EXISTS{?er a :Rexist, :Pay; :has-agent ?m; :has-object ?r}}
```

Now, if the initial knowledge graph contains the triples in (15.b), which represents the sentence in (15.a):

(15) a. Bill gives £500, the value of his rent, to his landlord, Jack.

b. `:egb a :Rexist, :Give; :has-agent :Bill; :has-object :500pounds;`
`:has-recipient :Jack. :Bill :a :Man. :500pounds :value-of :r.`
`:r :rent-of :Bill. :Jack :landlord-of :Bill.`

The rule in (13) infers that Bill is obliged to pay his rent, while the rule in (14) infers that Bill has indeed paid it. Then, as in the previous example, the rule in (10) infers that Bill has complied with his obligation.

This section has presented the subset of the framework from [3] necessary to formalize the norms in the D-KB. While [3] focuses on modeling irresolvable conflicts, that is, cases in which one norm imposes an obligation and another permits or prescribes the opposite, it does not address two key aspects required to model a broader range of norms in existing legislation, and in particular those in the D-KB: (1) defeasibility, i.e., the conditions under which one norm overrides another in case of conflict, and (2) the temporal structure of norms.

The next two sections extend the initial framework in [3] by introducing constructs for (1) and (2), respectively. The result is a new framework, which I call "Deontic Logic Reified", inspired by Hobbs's first-order logic for natural language semantics[12] [26]. Section 5 then applies the extended framework to the D-KB.

3. Incorporating defeasibility

As explained above, this paper extends and integrates insights from three of my previous research works, the most recent of which is [3]. That work defines a framework based on RDF and SPARQL, a portion of which was presented in the previous section, and is designed to represent and perform inferences on *irresolvable* conflicts, i.e., situations in which one norm imposes an obligation while another permits or prescribes the opposite.

While identifying and reasoning with irresolvable conflicts is important from the perspective of a LegalTech application, *only* addressing such conflicts is clearly insufficient for a platform intended to perform compliance checking on real data. Irresolvable conflicts represent "flaws" in the normative code, which is why they should be detected and *resolved* by the authority responsible for amending the code. To support such amendments, the logical framework must therefore provide constructs that enable their resolution, along with inference rules that take into account these constructs during the reasoning process.

Indeed, when norms are found to be in conflict, perhaps only in specific contexts, a common legislative response is not to rewrite the entire act, but rather to amend the existing provisions by introducing *exceptions*. These exceptions typically specify the conditions under which the original deontic statement does not apply. This approach is practical: rewriting complex legal acts from scratch would be both politically and administratively cumbersome. Instead, the legal text is *incrementally* extended with exceptions tailored to the identified contexts. Over time, as further conflicts or edge cases are discovered, the process becomes *recursive*: exceptions are added to exceptions, sometimes even leading to multiple levels of nested exceptions. This recursive exception-handling is one of the main reasons why amended legal texts often appear as intricate networks of conditional norms and carve-outs.

A simple constructed example illustrating this phenomenon is shown in (16). Suppose the law states (16.a): individuals prescribed a medicine by a GP must pay the cost of the medicine. Later on, it might be decided that both disabled individuals and those facing high medicine costs (>£50) should be exempted from paying, thereby adding the exceptions in (16.b) and (16.c). However, it might then be decided that (16.c) applies only to UK nationals, since the cost of medicines exceeding £50 is covered by public funds collected through taxation in the UK. (16.d) is therefore added as an exception to the exception. Finally, a further decision might be made to treat refugees as UK nationals, thus restoring the exception in (16.c) for this specific subgroup: (16.e) is consequently added as an exception to the exception to the exception.

[12] Although Hobbs does not assign a specific name to his framework, the fourth chapter of his book [26] is titled "Logic Reified". Since my work primarily focuses on integrating deontic modalities into that framework, the name "Deontic Logic Reified" seems both fitting and appropriate.

(16) a. Anyone who is prescribed a medicine by a GP is obliged to pay the cost of the medicine.

b. Disabled individuals are exempt from paying the cost of the medicine.

c. The cost of the medicine is waived if it exceeds £50.

i. However, this waiver applies only to UK nationals; non-UK nationals must pay the full amount.

1. However, if an individual is classified as a refugee, they are treated as a UK national and thus qualify for the exemption in (16.c).

As the structure shows, the exceptions in (16.b-c) form a *tree* of nested conditions. To perform accurate compliance checking, a LegalTech system must be able to model this hierarchical logic and apply inference rules that correctly determine whether the norms apply or not within the identified contexts (and sub-contexts, sub-sub-contexts, etc.).

The next subsection briefly reviews the state of the art in representing and reasoning with exceptions in automated compliance checking. After that, the solution implemented in the proposed framework, informed by the reviewed approaches from the literature, will be presented.

3.1. Background: state-of-the-art approaches for compliance checking to represent and reason with exceptions

The literature in automated compliance checking features two main approaches to handle defeasible reasoning through exceptions: *negation-as-failure* and *superiority relations*.

Approaches based on negation-as-failure, such as the SHACL-based reasoner in [18], the only system among those compared in [2] that can directly process RDF data, handle norms with exceptions by requiring the failure to prove the exceptions. Specifically, if the conditions under which an exception applies are not *explicitly* asserted as true, meaning that they are either false or *unknown*, the general norm is applied *by default*. In this way, a general rule applies only when none of its exceptions can be inferred.

In forward-chaining production-rule systems, such as SHACL, this may require rules to be *prioritized*: exception rules must fire *first* in order to assert the conditions that block the general rule via negation-as-failure. Conversely, if general rules are executed before their corresponding exceptions, the inference engine may incorrectly apply them, failing to account for those exceptions. This is the case for the SHACL-based reasoner in [18], which uses the SHACL operator `sh:order`[13] to ensure that rules associated with exceptions are executed before general rules. However, not all monotonic forward-chaining production-rule systems require explicit prioritization. Some systems, such as reasoners based on Answer Set Programming, evaluate rules containing negation-as-failure according to a specific semantics, known as the "stable model semantics", which ensures that rules depending on negation are evaluated consistently, so that exceptions are properly taken into account. Another widely used technique for avoiding the explicit specification of priorities is *stratification*, which implicitly orders rules according to their dependencies, ensuring that those involving negation are evaluated only after the predicates they depend on. Stratification will be adopted in the forthcoming SHACL 1.2 recommendation[14].

An alternative to negation-as-failure is to introduce dedicated meta-constructs that explicitly encode exceptions. These constructs are commonly referred to as "superiority relations". In this approach, a mechanism takes both a general norm and its exception as input and ensures that, if the exception holds in the current state of affairs, the inferences triggered by the general norm are *defeated*, where "defeated" is a technical term indicating that such inferences are no longer considered valid by the reasoner.

This approach is exemplified, among others, by the Proleg reasoner [24], one of the most well-known implementations for compliance checking. Proleg extends Prolog by introducing the meta-predicate `exception`:

(17) `exception(norm(`x_1`),exp(`x_1`))`

In which `norm(`x_1`)` and `exp(`x_1`)` are Prolog predicates representing a norm and an exception to that norm, respectively. For simplicity, we assume that `norm` and `exp` are unary predicates, although the meta-predicate `exception`

[13] https://w3c.github.io/data-shapes/shacl-af/#rules-order

[14] See https://w3c.github.io/data-shapes/shacl12-rules/#stratification

can take predicates of any arity as arguments. In (17), `exp(x1)` is evaluated first, and if it succeeds, `norm(x1)` is not executed, thus preventing any inferences it would otherwise trigger. See [24] for further details.

Prof. Ken Satoh, the lead author of Proleg, strongly advocates using superiority relations instead of negation-as-failure for handling exceptions, particularly in contexts aimed at legal practitioners who may not be familiar with the underlying logical semantics (see [25] or §6.2.3 in [2]). Insights gathered through discussions with domain experts, as reported in [25], indicate that, from a usability perspective, specifying `exception(norm(x1),exp(x1))` is significantly more intuitive than expressing the same logic through negation-as-failure, especially when dealing with recursively nested exceptions, as in the complex structure illustrated in (16). Meta-predicates such as `exception` help maintain a clearer *isomorphism* between formal rules and the corresponding provisions in legal texts [30].

On the other hand, the use of the meta-predicate `exception` is closely tied to the *backward-chaining* evaluation model of the Prolog language. In Prolog, rules are evaluated *top-down* in response to queries. When the goal `exception(norm(x1),exp(x1))` is evaluated, the system first attempts to satisfy the exception; if it succeeds, the norm is not evaluated, effectively blocking any inferences it would trigger.

In contrast, many other reasoners, including the one adopted in the proposed framework, rely on a *forward-chaining*, *bottom-up* inference strategy. These systems function as standard production-rule engines that repeatedly apply inference rules until no further facts can be derived. In such systems, exception handling must be explicitly simulated, as the engine does not inherently support top-down goal resolution.

To address this, another widely adopted solution, used, for example, in SPINdle [23], another well-known automated legal reasoner mainly develop by Prof. Guido Governatori, is to assign each rule a label and then define meta-rules over these labels to express which rules have priority over others. These priority relations are then used to determine which rules "defeat" others in case of conflict. Once conflict resolution has been performed and the undefeated rules identified, compliance is assessed solely against this active set.

The SPINdle counterpart to the Proleg formula in (17) is shown in (18). Note that, in (18), `normx1` and `expx1` are *propositional* symbols. Unlike Proleg, the SPINdle reasoner does not support first-order predicates; therefore, all first-order formulae must be *grounded* before they can be executed in SPINdle.

(18)
```
ln1: normx1
le1: expx1
le1>ln1
```

In the simple SPINdle formulas in (18), `ln1` and `le1` are the labels of the propositional symbols `normx1` and `expx1`, while `le1>ln1` is a superiority relation explicitly encoding that the formula with label `le1` has priority over the one with label `ln1`: if the exception holds, the latter is defeated, meaning that the SPINdle reasoner will not consider it when checking for compliance.

In light of the literature on automated legal reasoning discussed above, superiority relations have been selected as the preferred method to implement defeasibility in the proposed framework. Compared to negation-as-failure, superiority relations provide more transparent control over rule conflicts, are easier to edit and debug, and are generally more accessible to legal professionals.

The next paragraph presents the proposed solution, which can be seen as intermediate between the approaches of Proleg and SPINdle. The framework is based on SPARQL and is therefore inherently forward-chaining, like SPINdle; accordingly, the implemented method parallels SPINdle's approach by defining inference rules that determine when certain rules override others. At the same time, whereas SPINdle's input format is *propositional*, SPARQL, like Proleg, operate on *first-order* logic. In the approach presented below, exceptions are implemented as specialized SPARQL rules that correspond to Proleg's meta-predicate `exception` and SPINdle's system of ordered labels.

3.2. Representing and reasoning with exceptions in the proposed framework

As discussed above, to handle defeasibility this paper integrates into the proposed framework a mechanism inspired by the one implemented in SPINdle, but adapted to operate at the first-order level, as in Proleg.

In the proposed framework, norms are encoded as SPARQL rules in the `CONSTRUCT-WHERE` form. The `CONSTRUCT` clause specifies the statements to be inferred, such as (not-)obligations and (not-)permissions, while

the WHERE clause defines the conditions for those statements to hold, i.e., the triples that must already exist in the knowledge graph for the inferred statements to be added. Examples are provided in (13) and (14) above.

In cases where rules have exceptions, the CONSTRUCT clauses of these rules include additional triples, one for each exception, that effectively *activate* the rules responsible for handling them.

To support this, a new property, :checks-exception, is introduced into the vocabulary of the proposed framework. Defeasible rules define sub-properties of :checks-exception to link the inferred statements to the first-order arguments of their conditions, which must be checked to determine whether an exception applies.

The rules implementing the exceptions are activated if the knowledge graph includes these sub-properties of :checks-exception. They evaluate the conditions under which the exception applies to the first-order arguments and, if these conditions are met, infer that the exception *defeats* the general rule.

An example is provided in (19). This SPARQL rule implements the obligation described in (16.a), which stipulates that patients prescribed a medicine by their GP are required to pay the corresponding cost. However, this regulative norm is defeasible, as it is subject to the exceptions in (16.b) and (16.c). To account for this, the CONSTRUCT clause of the rule infers the properties :checks-exception-in-16-b and :checks-exception-in-16-c, both sub-properties of :checks-exception. These properties link the inferred obligation ?eo to the RDF list of relevant individual(s) referenced in the WHERE clauses of the rule or of one of its exceptions.

(19)
```
:checks-exception-in-16-b rdfs:subPropertyOf :checks-exception.
:checks-exception-in-16-c rdfs:subPropertyOf :checks-exception.

CONSTRUCT{?eo a :Obligatory,:Pay; :has-agent ?pa; :has-object ?cm;
          :checks-exception-in-16-b (?pa);
          :checks-exception-in-16-c (?cm ?pa)}
WHERE{?eg a :Rexist,:Prescribe; :has-agent ?g; :has-object ?m;
      :has-recipient ?pa. ?g a :GP. ?m a :Medicine..
      ?m :has-cost ?cm. BIND(BNode() AS ?eo)
      NOT EXISTS{?eor a :Obligatory,:Pay; :has-agent ?pa; :has-object ?cm}}
```

The properties checks-exception-in-16-b and checks-exception-in-16-c correspond to the meta-predicate exception in Proleg and the labels in SPINdle. Their function is to associate the newly inferred obligation, which is linked to the variable ?eo and created as an anonymous individual using the SPARQL function BNode, with the list of individuals quantified in the condition and evaluated in the exceptions.

The two exceptions are handled by two additional SPARQL rules, each of which infers a new eventuality that defeats the general obligation when its conditions are satisfied. To support this, a new RDF property defeats is introduced into the vocabulary of the proposed framework; furthermore, to maintain alignment with the name of the checks-exception sub-property that activated the SPARQL rule checking for the exception, a corresponding sub-property of defeats is also introduced.

The two SPARQL rules that check the exceptions in (16.b) and (16.c) are shown in (20). The first rule checks whether the patient ?pa is disabled. If so, it creates a new anonymous individual, bound to the variable ?ed, and infers that this individual defeats the obligation on the basis of this specific exception. Similarly, the second rule checks whether the value of the charges ?c exceeds £50. In this case, the rule not only creates a new anonymous individual and infers that it defeats the obligation, but also infers that another exception must be checked *recursively*: the one specified in (16.c.i) above.

(20)
```
:defeats-on-16-b rdfs:subPropertyOf :defeats.
:defeats-on-16-c rdfs:subPropertyOf :defeats.
:checks-exception-in-16-c-i rdfs:subPropertyOf :checks-exception.

CONSTRUCT{?ed :defeats-on-16-b ?eo}
WHERE{?eo :checks-exception-in-16-b (?pa). ?pa a :Disabled.
      BIND(BNode()AS ?ed) NOT EXISTS{?edr :defeats-on-16-b ?eo}}
```

```
CONSTRUCT{?ed :defeats-on-16-c ?eo;
              :checks-exception-in-16-c-i (?pa)}
WHERE{?eo :checks-exception-in-16-c (?cm ?pa). ?cm :value-of ?v.
      FILTER(?v>50) BIND(BNode()AS ?ed)
      NOT EXISTS{?edr :defeats-on-16-c ?eo}}
```

The defeasibility mechanism in the proposed framework should now be clear. Nested exceptions can be defined in the same manner, enabling the encoding of "trees" of default rules and their exceptions, while preserving a one-to-one correspondence with the structure of the normative code, thereby ensuring *isomorphism*. The final two SPARQL rules, corresponding to (16.c.i) and (16.c.i.1), are shown below:

(21)
```
:defeats-on-16-c-i rdfs:subPropertyOf :defeats.
:checks-exception-in-16-c-i-1 rdfs:subPropertyOf :checks-exception.
:defeats-on-16-c-i-1 rdfs:subPropertyOf :defeats.

CONSTRUCT{?ed :defeats-on-16-c-i ?e;
              :checks-exception-in-16-c-i-1 (?pa)}
WHERE{?e :checks-exception-in-16-c-i (?pa). ?pa a :NonUKNational.
      BIND(BNode()AS ?ed) NOT EXISTS{?edr :defeats-on-16-c-i ?e}}

CONSTRUCT{?ed :defeats-on-16-c-i-1 ?e}
WHERE{?e :checks-exception-in-16-b (?pa). ?pa a :Refugee.
      BIND(BNode()AS ?ed) NOT EXISTS{?edr :defeats-on-16-c-i-1 ?e}}
```

The rules presented so far in this subsection determine when the obligation in (16.a) is defeated by an applicable exception in the modeled context, and when those exceptions are themselves defeated by further (nested) exceptions. Once all relevant `defeats` properties have been inferred, compliance can be assessed.

As explained earlier, exceptions and their nested exceptions form a *tree* of eventualities, where the (abstract, primary) eventuality is the root. It should be clear that the inference of `defeats` properties must be performed *top-down*, i.e., from the root to the leaves, whereas determining which rules apply to the input knowledge graph must be carried out *bottom-up*, i.e., from the leaves back to the root. In other words, in this second phase we start from the leaves of the tree and recursively determine which exceptions remain in force and which are defeated, propagating the results upward to the eventuality at the root.

In the proposed framework, this bottom-up phase is implemented by introducing two new classes, `Active` and `Inactive`, together with the two SPARQL rules shown in (22).

The first rule marks as `Active` an eventuality, holding under a deontic modality or the `Rexist` modality, if it is not defeated by another exception, or if it is defeated only by exceptions that are `Inactive`. This behavior is enforced by the two nested `NOT EXISTS` clauses in the first rule of (22). Conversely, the second rule infers that any eventuality defeated by at least one `Active` exception is classified as `Inactive`.

In summary, the two rules operate iteratively as follows: they initially classify the leaves of the exception tree as `Active`, as these are not defeated by any other eventuality. Then, they mark as `Inactive` any eventuality defeated by the leaves, followed by marking as `Active` any eventuality defeated only by `Inactive` exceptions, and so on, until the root eventuality is ultimately inferred to be either `Active` or `Inactive`.

(22)
```
:Active a rdfs:Class. :Inactive a rdfs:Class.

CONSTRUCT{?e a :Active}
WHERE{{{?e a ?M}UNION{<<?e a ?M>> a :false}
       ?M rdfs:subClassOf*/a :Modality}UNION{?e :defeats ?ea}
       NOT EXISTS{?se :defeats ?e. NOT EXISTS{?se a :Inactive}}}

CONSTRUCT{?se a :Inactive}
WHERE{?e a :Active. ?e :defeats ?se}
```

Let me now show how the rules presented above are applied to a sample ABox. Suppose a GP `gp` prescribes the same medicine `m`, costing £100, to three patients: John, Pedro, and Irina. John is from the UK, Pedro is from Spain, and Irina is a UK refugee from Ukraine. This information can be encoded in RDF as follows:

(23)
```
[:Rexist,:Prescribe; :has-agent :gp; :has-object :m; :has-recipient :John].
[:Rexist,:Prescribe; :has-agent :gp; :has-object :m; :has-recipient :Pedro].
[:Rexist,:Prescribe; :has-agent :gp; :has-object :m; :has-recipient :Irina].
:gp a :GP. :m a :Medicine; :has-cost :cm. :cm :has-value-in-pounds 100.
:Pedro a :NonUKNational. :Irina a :NonUKNational,:Refugee.
```

From this ABox, the SPARQL rule in (19) infers that all three patients are obliged to pay the cost of the medicine:

(24)
```
_:b0 a :Pay,:Obligatory; :has-agent :John; :has-object :cm;
:checks-exception-in-16-b (:John);
:checks-exception-in-16-c (:cm :John).

_:b1 a :Pay,:Obligatory; :has-agent :Pedro; :has-object :cm;
:checks-exception-in-16-b (:Pedro);
:checks-exception-in-16-c (:cm :Pedro).

_:b2 a :Pay,:Obligatory; :has-agent :Irina; :has-object :cm;
:checks-exception-in-16-b (:Irina);
:checks-exception-in-16-c (:cm :Irina).
```

Since `checks-exception-in-16-c` is inferred for all three patients, the `WHERE` clause of the second rule in (20) is satisfied, as the cost of the medicine exceeds £50. Consequently, the second rule in (20) infers that all three obligations are defeated by the exception from (16.c); however, this rule also infers `checks-exception-in-16-c-i`, i.e., that the exception on the exception from (16.c.i) must be checked:

(25)
```
_:b3 :defeats-on-16-c _:b0; :checks-exception-in-16-c-i (:John).
_:b4 :defeats-on-16-c _:b1; :checks-exception-in-16-c-i (:Pedro).
_:b5 :defeats-on-16-c _:b2; :checks-exception-in-16-c-i (:Irina).
```

Now, although the three exceptions all occur as subject of the property `checks-exception-in-16-c-i`, the first SPARQL rule in (21) triggers only for Pedro's and Irina's, since John does not belong to the class `NonUKNational`. Consequently, the rule infers that the exceptions `_:b4` and `_:b5` in (25) are defeated (but not `_:b3`); however, the rule also infers that the property `checks-exception-in-16-c-i-1` must be checked for the two new exceptions of exceptions:

(26)
```
_:b6 :defeats-on-16-c-i _:b4; :checks-exception-in-16-c-i-1 (:Pedro).
_:b7 :defeats-on-16-c-i _:b5; :checks-exception-in-16-c-i-1 (:Irina).
```

By recursion, although both exceptions of exceptions occur as subject of `checks-exception-in-16-c-i-1`, the second SPARQL rule in (21) triggers only for Irina's, since Pedro is not a `Refugee`. Consequently, the rule infers that `_:b7` is defeated:

(27)
```
_:b8 :defeats-on-16-c-i-1 _:b7.
```

It should now be clear that the rules in (20) and (21) "unfold" the trees of exceptions *at the first-order level*, i.e., for each individual agent. After the rules have been (re)applied until no further triples can be inferred, it becomes possible to determine what each agent is ultimately obliged or not obliged to do.

This is achieved through the two rules in (22), which, as explained above, are applied *bottom-up*. First, they infer that the exceptions at the leaves of the three trees are `Active`:

(28) `_:b8 a :Active. _:b6 a :Active. _:b3 a :Active.`

Next, they infer that the eventualities defeated by the exceptions at the leaves are `Inactive`. They then infer that all eventualities defeated only by `Inactive` exceptions are `Active`, and so on, propagating this process up to the roots of the trees. We eventually obtain:

(29)
```
_:b7 a :Inactive. _:b4 a :Inactive. _:b0 a :Inactive.
_:b5 a :Active. _:b1 a :Active. _:b2 a :Inactive.
```

From these inferences, it is derived that only Pedro is obliged to pay the cost of the medicine, because, among the three obligations in (24), only Pedro's, i.e., `_:b1`, is `Active`.

The final minor adjustment in the proposed framework is a slight modification of the rules used to infer compliance, violations, and related properties. These rules are now applied only to *active* eventualities. For example, the rule for inferring compliance of obligations, shown above in (10), is modified as follows:

(30)
```
CONSTRUCT{?e :complies-with ?eo}
WHERE{?eo a :Obligatory,?ET,:Active.
  ?e a :Rexist,?ET,:Active. ?ET a :EventualityType.
  NOT EXISTS{?tr a :ThematicRole. ?eo ?tr ?vo. NOT EXISTS{?e ?tr ?v}}
  NOT EXISTS{?tr a :ThematicRole. ?eo ?tr ?vo. ?e ?tr ?v. FILTER(?vo!=?v)}}
```

Note that the above mechanisms only work in the absence of *cycles* among exceptions, i.e., when exceptions can be hierarchically ordered in a tree or, in the most general case, in a directed acyclic graph. If cycles are present, one or more eventualities could be inferred as both `Active` and `Inactive`, which is of course inconsistent. SHACL shapes could be introduced to check whether the rules are hierarchically structured as a directed acyclic graph; however, this level of detail is beyond the scope of the present paper, and the specific implementation of such validation constraints is therefore omitted.

4. Incorporating temporal management

The temporal dimension is a crucial component in the formal representation of normative systems. In real-world regulatory contexts, norms are rarely timeless; they almost always specify, either explicitly or implicitly, a time frame during which they are in force and against which compliance can be assessed. For example, an obligation to submit a report or a prohibition from entering a park is meaningless unless anchored to specific deadlines or time intervals during which the norms are in force. Therefore, any practical and expressive normative framework must include mechanisms to model and reason about the temporal aspects of norms.

In the previous sections, the thematic role `has-time` was occasionally used in examples; however, no details were provided regarding its intended use or the types of entities that can serve as its object.

In accordance with the principles of reusability and interoperability promoted by the Semantic Web, this work employs the Time Ontology to represent temporal information. The Time Ontology is widely regarded as the de facto standard for representing temporal information on the Semantic Web. While the W3C Candidate Recommendation Draft of the Time Ontology[15] is defined in OWL, in [31] I have co-introduced a new SHACL-based version of the Time Ontology, which is considerably more expressive than the official OWL-based one.

The Time Ontology distinguishes between two fundamental categories of temporal entities: `Interval` and `Instant`, the latter being conceived as an `Interval` with zero duration, in line with the seminal work of [32]. Accordingly, the `rdfs:range` of the thematic role `has-time` is set to the Time Ontology class `Interval`.

However, the rules defined in [31] are intended solely for validation purposes, i.e., for identifying ill-formed knowledge graphs that represent temporal information. Conversely, the development of additional inference rules for deriving temporal information tailored to specific use cases is mentioned in [31] as a direction for future work.

[15] Available online at https://www.w3.org/TR/owl-time

This paper contributes to that direction. The next subsection outlines the specific resources from the Time Ontology employed in the proposed framework in combination with the `has-time` thematic role. Subsequently, Subsection 4.2 illustrates how the temporal dimension is integrated into the framework.

4.1. Representing the temporal dimension of norms

As mentioned above, the core classes in the Time Ontology are `Interval` and `Instant`. As their names imply, they represent, respectively, a span of time delimited by two endpoints and a single point in time, conceptualized as an interval of zero duration in which the endpoints coincide. The properties `hasBeginning` and `hasEnd` are used to specify the starting and ending instants of intervals, while `inXSDDateTime` associates an instant with its corresponding `xsd:dateTime` value. Both [31] and this work use `xsd:dateTime` exclusively for representing time values, although the Time Ontology also supports other datatypes, such as `xsd:dateTimeStamp`. Furthermore, for the sake of simplicity, this paper discretizes `xsd:dateTime` values to the level of seconds. That is, although `xsd:dateTime` supports the representation of fractional seconds, these will not be considered in this work.

An example of an `xsd:dateTime` value used in the paper is `2025-01-01T00:00:00Z`, which refers to midnight on January 1st, 2025. The `Z` indicates the UTC timezone; all examples below will use this timezone.

It is important to note that `Instant` and `Interval` are *not* disjoint classes. As mentioned earlier, in line with [32], an instant is considered a type of interval, specifically one that begins and ends at the same point in time[16].

In the proposed framework, the thematic role `has-time` is defined with `Interval` as its `rdfs:range`. As a result, an eventuality can occur either at a specific instant (i.e., an interval with zero duration), in which case it is termed "punctual", or over a (proper) interval delimited by two instants, in which case it is termed "durative".

The distinction between punctual and durative events is well-established in the literature. It was originally introduced by [33] in his seminal work on verbal aspect, where he classified events based on their internal temporal structure. This distinction has since been extended and refined by subsequent scholars in natural language semantics, e.g., [34], among others. However, exploring and formalizing these approaches goes beyond the scope of the present paper, which is limited to the basic foundational two-way categorization.

It is stipulated that obligations may also hold for a single instant; in such cases, they are typically referred to as "punctual obligations" [35]. The same applies to permissions, which are then termed "punctual permissions". The instant during which a punctual obligation is in force is the only moment in which it can be fulfilled. While it may be difficult to imagine a human fulfilling an obligation at an *exact* instant in time, one can consider obligations whose bearer is a computer or a robot[17], as illustrated in the following example:

(31) The computer must reset itself at 00:15:00.

The obligation in (31) must be fulfilled at that exact moment, i.e., 15 minutes and 0 seconds after midnight. However, this is entirely feasible for a computer.

Finally, the proposed framework reuses the Time Ontology properties `hasBeginning` and `hasEnd`, which link an interval to its two endpoints, i.e., the instants that mark the beginning and end of the interval, respectively.

Let me now show an example. (32) illustrates how to represent the statement from (2.a), reformulated as in (32.a), using the resources of the Time Ontology. Note that in (2.a), "3pm" is implicitly understood as the time on the day the sentence is uttered. Since this date is unspecified but `xsd:dateTime` requires a concrete value, (32) uses 1st January 2025 as an illustrative example. In (32), as well as in all subsequent examples, the prefix "`time:`" denotes RDF resources belonging to the Time Ontology.

(32) a. John leaves at 3pm (of 1/1/2025, UTC time zone).

[16]The Time Ontology also defines the class `ProperInterval`, a subclass of `Interval` that is disjoint with `Instant`; a `ProperInterval` must span a non-zero duration, with its end strictly following its beginning.

[17]This example illustrates that the bearers of obligations are not limited to humans; they may also include computers or robots. This raises important questions about liability: if a computer or robot fails to fulfill the obligation, who is held responsible? There is indeed considerable ongoing debate on this issue, such as the liability of autonomous vehicles (see, e.g., [36]). However, this paper does not examine this literature further, as it lies well outside the scope of the current work.

b. `:elj a :Rexist, :Leave; :has-agent :John;`
`:has-time [time:inXSDDateTime "2025-01-01T15:00:00Z"^^xsd:dateTime].`

In addition to the Time Ontology resources listed above, the proposed framework introduces some additional resources and rules. These will be needed in the next subsection to evaluate the compliance of obligations with respect to their temporal dimension.

First, the two special classes `minusInfinity` and `plusInfinity` are added to the proposed framework to tag instances of the class `Instant` that correspond to the values $-\infty$ and $+\infty$, respectively. These two classes were already proposed in [31], §7, as a possible extension of the Time Ontology vocabulary for defining infinite intervals. [31], §7, also introduces SHACL shapes to validate intervals involving these two instants (for example, ensuring that no interval begins at $+\infty$ or ends at $-\infty$). These shapes will not be repeated here; the interested reader is referred to [31], §7. In what follows, all triples involving `minusInfinity` or `plusInfinity` are assumed to be already asserted or inferred as valid.

In this paper, `minusInfinity` and `plusInfinity` are used to model intervals referring to any time before or after a given time point. For example, in (33) the individual `I` represents the interval referring to any time *before* midnight on 1st January 2025, i.e., `[`$-\infty$`, 2025-01-01T00:00:00Z]`:

(33)
```
:I a time:ProperInterval;
   time:hasBeginning [a :minusInfinity];
   time:hasEnd [time:inXSDDateTime "2025-01-01T00:00:00Z"^^xsd:dateTime].
```

As illustrated below, these intervals will be assigned to the `has-time` thematic role to model legal constraints that require certain actions to be performed before or after a specific instant, such as a deadline.

Secondly, two new RDF properties are introduced to relate pairs of intervals: `includes` and `intersects`. As their names suggest, `includes` relates two intervals such that all instants within the interval in the object also belong to the interval in the subject, while `intersects` relates two intervals that share at least one instant.

The SPARQL rule in (34) allows the inference of when an interval `?I` includes another interval `?sI`.

(34)
```
CONSTRUCT{?I :includes ?sI} WHERE{
    {?I time:hasBeginning/rdf:type :minusInfinity}UNION
    {?I time:hasBeginning*/time:inXSDDateTime ?bI.
        ?sI time:hasBeginning*/time:inXSDDateTime ?bsI. FILTER(?bI<=?bsI)}
    {?I time:hasEnd/rdf:type :plusInfinity}UNION
    {?I time:hasEnd*/time:inXSDDateTime ?eI.
        ?sI time:hasEnd*/time:inXSDDateTime ?esI. FILTER(?esI<=?eI)}}
```

As illustrated in Figure 1, if `?I` begins at $-\infty$, there is no need to check the beginning of `?sI`: whether `?sI` also begins at $-\infty$ or at a specific point in time, `?I` begins before or simultaneously with `?sI`. Conversely, if `?I` begins at a specific point in time, it must be verified that `?sI` begins at a point in time greater than or equal to that of `?I`. Symmetrically, it must be ensured that `?I` ends either at $+\infty$ or at a specific point in time greater than or equal to the end of `?sI`. These two conditions are enforced by the two subclauses "`{...}UNION{...}`" in (34).

Two additional SPARQL rules, omitted from the paper but available in the GitHub repository, infer that every interval includes itself and its beginning and end points.

Fig. 1. Rationale of the SPARQL rule in (34).

The SPARQL rule for inferring whether two intervals intersect is simpler, as it reuses the `includes` property and, by extension, the rule in (34):

(35)
```
CONSTRUCT{?I1 :intersects ?I2. ?I2 :intersects ?I1}
WHERE{{?I1 :includes ?I2}UNION
      {?I1 :includes ?i. ?I2 time:hasBeginning|time:hasEnd ?i}}
```

Two intervals intersect if one of them, or at least its start or end instant, is included within the other one.

4.2. Compliance checking of time-indexed eventualities: achievement vs maintenance obligations and permissions

Building on the basic constructs provided by the Time Ontology, this subsection shows how to incorporate them into the SPARQL rules introduced in the previous section so as to account for the temporal dimension.

In some cases, the validity of an obligation, understood as the period during which it is in force and must therefore be fulfilled, is explicitly defined in the normative code. For example, if John is obliged to pay a fine, he will typically be given a deadline by which the payment must be made (e.g., one month from the date the fine was issued). In other cases, however, the duration is implicit or even unspecified, implying a *reasonable* timeframe. For instance, if John is required to pay for parking after leaving his car, the obligation is not violated the moment he switches off the engine; rather, he is granted a reasonable amount of time to obtain a parking ticket and place it on the dashboard. This period should be regarded as reasonable in the sense that it takes into account contextual factors such as the distance to the parking meter or the presence of a queue.

The formalization of what constitutes a "reasonable" duration requires modeling legal interpretations, which lies beyond the scope of this paper. In the parking ticket example, for instance, in the event of a dispute, it would be up to a judge or another competent authority to determine, based on all relevant circumstances, whether John complied with the obligation within a reasonable timeframe. Given these considerations, the present formalization adopts a simplifying assumption: if an obligation does not specify the `has-time` thematic role, any time is considered acceptable for the eventuality that fulfills the obligation.

Second, and more importantly, this paper adopts the well-known and widely accepted distinction in the deontic logic literature concerning the ways in which real-world actions can fulfil obligations. Specifically, obligations are typically divided into two main categories: *achievement* obligations and *maintenance* obligations. This distinction has been formally captured in numerous deontic logic frameworks, e.g., [37], [38], [39], [40], [41].

Achievement obligations refer to duties that are satisfied when the time interval of the action intersects the period during which the obligation is in force, after which they are no longer in effect. For example, if John is obliged to submit a report by 5:00pm, the obligation is fulfilled if John submits the report at any time before the deadline, after which he is no longer required to submit it. In this sense, the obligation is considered "achieved".

In contrast, maintenance obligations involve ongoing actions that must be sustained over time. These obligations do not focus on a single point of completion, but rather on the continuous fulfillment of a requirement. For instance, if John is obligated to keep his phone on silent during the meeting, the obligation is fulfilled as long as the phone remains on silent throughout a timeframe that includes the entire duration of the meeting, not just part of it.

The same twofold sub-categorization also applies to not-obligations and (not-)permissions, with the main difference that the RDF properties `rexists-although-not-obligatory`, `conforms-to`, and `violates` must be used in place of `complies-with`. Thus, for example, achievement not-permissions are violated when the prohibited action occurs during any interval that intersects the time frame associated with the prohibition. For instance, if John is prohibited from entering the park between 3:00pm and 6:00pm, he violates the prohibition if he is in the park at any time that intersects this period, such as from 2:00pm to 4:00pm or from 5:00pm to 7:00pm. By contrast, maintenance not-permissions are violated if the time frame of the action fully encompasses the time

frame associated with the prohibition. For example, if it is prohibited to park in the city center for more than one hour (as in time-limited parking zones where drivers must leave after a fixed duration), then John would violate the prohibition if, for instance, he parks there from 3:00pm to 5:00pm.

In light of the above, four additional classes are introduced in the proposed framework: `A-Obligatory` and `M-Obligatory`, both subclasses of `Obligatory`, and `A-Permitted` and `M-Permitted`, both subclasses of `Permitted`. While the remainder of this subsection focuses only on inference rules for compliance with achievement and maintenance obligations, the rules for achievement and maintenance non-obligations and (not-)permissions are available in the GitHub repository, together with examples.

As noted above, an individual belonging to the class `A-Obligatory` is considered complied with by a really existing eventuality only if the duration of the obligation *intersects* the duration of that eventuality. In contrast, an individual of the class `M-Obligatory` is complied with only if the duration of the really existing eventuality *includes* the duration of the obligation. Intersection and inclusion of intervals are represented through the RDF properties `intersects` and `includes`, introduced in the previous subsection; these properties are inferred by the SPARQL rules shown in (34) and (35).

Thanks to these properties, the rule in (30) above can be replaced with *two* distinct rules, each corresponding to a different subcategory of obligations: `A-Obligatory` and `M-Obligatory`. In both rules, the thematic role `has-time` is handled separately.

(36) shows the SPARQL rule for checking compliance with achievement obligations. The two `NOT EXISTS` clauses are identical to those in (30), except that the second one does not apply to the thematic role `has-time`; this role is instead checked separately in the `FILTER` clause following the two `NOT EXISTS` clauses.

If at least one of the two eventualities does not specify the `has-time` thematic role, the final `FILTER` clause evaluates to true. However, if the obligation `?eo` specifies `has-time` and the really existing eventuality `?e` does not, the first `NOT EXISTS` clause evaluates to false. As a result, the rule does not infer that `?e` complies with the obligation. This is correct: `?e` must comply with `?eo` within the timeframe specified by the latter. If `?e`'s timeframe is instead missing, it is not possible to determine whether `?e` complied with `?eo`. Conversely, if it is `?eo` that does not specify `has-time`, then any value for that thematic role is accepted for `?e`. Nonetheless, as explained above, future work may consider refining the rule to accept only "reasonable" values of `has-time` for `?e` in such cases.

Finally, the second `IF` condition embedded in the final `FILTER` clause handles the case in which both eventualities specify the `has-time` thematic role. In these instances, `?e` is considered compliant with `?eo` only if the interval during which `?eo` is in force `intersects` the interval in which `?e` really exists.

(36)
```
CONSTRUCT{?e :complies-with ?eo}
WHERE{?eo a :A-Obligatory,?ET,:Active.
  ?e a :Rexist,?ET,:Active. ?ET a :EventualityType.
  NOT EXISTS{?tr a :ThematicRole. ?eo ?tr ?vo. NOT EXISTS{?e ?tr ?v}}
  NOT EXISTS{?tr a :ThematicRole. ?eo ?tr ?vo. ?e ?tr ?v.
                                  FILTER((?tr!=:has-time)&&(?vo!=?v))}
  FILTER(IF(NOT EXISTS{?eo :has-time ?vo. ?e :has-time ?v}, true,
         IF(EXISTS{?eo :has-time ?vo. ?e :has-time ?v. ?vo :intersects ?v},
            true, false)))}
```

The rule for checking compliance with maintenance obligations is symmetrical and is shown in (37). The only difference from (36) is that, when the `has-time` thematic role is specified for both eventualities, the really existing eventuality `?e` complies with the obligation `?eo` only if the time frame of the former `includes` that of the latter.

```
(37) CONSTRUCT{?e :complies-with ?eo}
     WHERE{?eo a :M-Obligatory,?ET,:Active.
       ?e a :Rexist,?ET,:Active. ?ET a :EventualityType.
       NOT EXISTS{?tr a :ThematicRole. ?eo ?tr ?vo. NOT EXISTS{?e ?tr ?v}}
       NOT EXISTS{?tr a :ThematicRole. ?eo ?tr ?vo. ?e ?tr ?v.
                                        FILTER((?tr!=:has-time)&&(?vo!=?v))}
       FILTER(IF(NOT EXISTS{?eo :has-time ?vo. ?e :has-time ?v},true,
              IF(EXISTS{?eo :has-time ?vo. ?e :has-time ?v. ?v :includes ?vo},
                 true, false)))}
```

Let me now model two of the examples described at the beginning of this subsection. The first one is shown in (38.a) and formalized in the proposed framework as in (38.b).

(38) a. John is obliged to submit the report by 5:00pm (of 1/1/2025, UTC time zone).

b.
```
:esjo a :A-Obligatory, :Submit; :has-agent :John; :has-object :report;
  :has-time [a time:Interval;time:hasBeginning [a :minusInfinity];
  time:hasEnd [time:inXSDDateTime "2025-01-01T17:00:00Z"^^xsd:dateTime]].
```

(38.a) conveys an achievement obligation; therefore, the eventuality it denotes, i.e., `esjo`, belongs to the class `A-Obligatory`. The interval specified by the `has-time` thematic role is `[-∞, 5:00pm]`, meaning any time *before* 5:00pm on the day (38.a) is uttered, which is assumed to be January 1st, 2025. Naturally, the obligation does not literally begin at $-\infty$, but rather at the moment when John became committed to submitting the report. However, since that moment may be unknown, using $-\infty$ as the starting point of the interval is the only fully reliable way to represent "anytime before 5:00pm" across all possible scenarios.

One of the really existing submissions that complies with `esjo` is the one represented in (39), which refers to the fact that John actually submitted the report at 3:34pm. Note that this really existing eventuality is instantaneous; in other words, the submission of the report is represented here as a punctual action.

(39)
```
:esj a :Rexist, :Submit; :has-agent :John; :has-object :report;
  :has-time [time:inXSDDateTime "2025-01-01T15:34:00Z"^^xsd:dateTime].
```

From the RDF triples in (38.b) and (39), the rule in (36) infers the triple `:esj :complies-with :esjo`: the fact that John submitted his report at 3:34pm fulfills the obligation requiring him to submit it before 5:00pm.

Let me now present an example of a maintenance obligation, shown in (40). In this example, it is assumed that the meeting takes place from 11:00am to 2:00pm on January 1st, 2025. To represent the obligation that John's phone must remain silent throughout the meeting, a new thematic role is introduced in (40.b): `has-object-state`, which specifies the state that the individual in `has-object` must maintain for the entire duration of the meeting.

(40) a. John is obligated to keep his phone on silent during the meeting.

b.
```
:ekjo a :M-Obligatory, :Keep; :has-agent :John; :has-object :phone;
  :has-object-state :silent; :has-time [a time:Interval;
       time:hasBeginning
           [time:inXSDDateTime "2025-01-01T11:00:00Z"^^xsd:dateTime];
       time:hasEnd
           [time:inXSDDateTime "2025-01-01T14:00:00Z"^^xsd:dateTime]].
```

Now, if John keeps his phone on silent from the moment he wakes up, i.e., 9:00am, until he goes to sleep, i.e., 9:00pm, as shown in the following RDF representation:

```
(41) :ekj a :Rexist, :Keep; :has-agent :John; :has-object :phone;
     :has-object-state :silent; :has-time [a time:Interval;
          time:hasBeginning
              [time:inXSDDateTime "2025-01-01T9:00:00Z"^^xsd:dateTime];
          time:hasEnd
              [time:inXSDDateTime "2025-01-01T21:00:00Z"^^xsd:dateTime]].
```

The rule shown above in (37) infers that `ekj` complies with `ekjo`: the fact that John kept his phone on silent all day fulfilled his obligation to keep it silent during the meeting.

5. Translating norms from the D-KB into Deontic Logic Reified

The previous sections extended the framework from [3] by introducing constructs necessary for handling defeasibility and temporal management. This extended framework is called "Deontic Logic Reified".

The present section illustrates how some selected norms in the DAPRECO knowledge base (D-KB) [1] have been translated into Deontic Logic Reified, making them executable for compliance checking purposes.

The D-KB is a repository of formulae in reified I/O logic [20] that I developed several years ago as part of a research project bearing the same name[18]. Each formula represents an if-then rule, which can be either a regulative rule, a constitutive rule, or an entailment that implements an exception to another rule (see examples below).

The D-KB contains 966 formulae, each representing a provision from the European General Data Protection Regulation (GDPR). These formulae were edited manually throughout the duration of the project and encoded in LegalRuleML[19], the XML standard for representing the semantic and logical content of legal texts. To the best of my knowledge, the D-KB remains the largest knowledge base in LegalRuleML freely available online. The D-KB was developed on top of the Privacy Ontology (PrOnto) [42], an ontology in the Privacy and Data Protection domain. Both PrOnto and the D-KB are widely cited in the literature[20], with the D-KB in particular serving as a benchmark for training and evaluating Machine Learning systems in legal classification and reasoning (e.g., [43], [5]).

Nevertheless, as explained in the Introduction, there is currently no automated reasoner available for reified I/O logic. As a result, the D-KB cannot (yet) be used for automated compliance checking. Rather than developing a dedicated reasoner for reified I/O logic, this paper proposes an alternative approach: translating reified I/O logic formulae into Deontic Logic Reified rules. Since rules in Deontic Logic Reified are encoded in RDF and SPARQL, they can be executed using Apache Jena.

The translation into Deontic Logic Reified, like the construction of the original D-KB, was carried out manually. However, I did not translate all 966 formulae in the D-KB. Instead, I selected a subset corresponding specifically to the norms in Chapter II of the GDPR, which outlines the regulation's core principles, along with all exceptions to those norms, some of which originate in other chapters. A complete translation would require substantial manual effort, which is beyond the scope of this paper. That said, the chosen subset is fully representative of the variety present in the D-KB; [2] discusses this variety, and the norms in Chapter II of the GDPR include at least one example of each category identified therein. In other words, the subset of translated reified I/O logic formulae is sufficient to demonstrate that Deontic Logic Reified includes all constructs necessary to represent the entire D-KB, since the formulae implementing the norms in the remaining chapters of the GDPR do not introduce additional variations in expressive requirements. As explained in the Introduction, the extension of the framework in [3] with constructs for defeasibility and temporal management was motivated precisely by the need to cover the full variety of norms identified in the D-KB, which these additions complete.

A second important clarification is that the formulae in the D-KB are used in this paper as a benchmark to evaluate the expressiveness, computability, and scalability of Deontic Logic Reified. It is therefore assumed that the encoding

[18] https://www.liviorobaldo.com/dapreco.html

[19] https://docs.oasis-open.org/legalruleml/legalruleml-core-spec/v1.0/os/legalruleml-core-spec-v1.0-os.html

[20] See https://www.scopus.com/record/display.uri?eid=2-s2.0-85052876661&origin=recordpage and https://www.scopus.com/record/display.uri?eid=2-s2.0-85076004898&origin=recordpage

of GDPR norms in the D-KB faithfully reflects their intended legal meaning. In other words, this paper is concerned solely with making the formulae in the D-KB computable, not with assessing their legal validity. Naturally, different legal scholars may hold differing views and propose alternative encodings of the norms; however, this paper does not engage with such interpretative legal debates, as they fall outside its scope. It is worth emphasizing, nonetheless, that the D-KB underwent rigorous validation by experts in both law and deontic logic, as documented in [44]. The methodology used in [44] was later generalised and, by the same authors, termed the "DAPRECO methodology", which constitutes another important outcome of the DAPRECO project.

The next subsection offers a brief overview of the general architecture of the D-KB and reified I/O logic. The following subsections will then discuss examples of the translated Deontic Logic Reified representations.

5.1. The D-KB and reified I/O logic

As stated earlier, the D-KB contains 966 reified I/O logic formulae, each representing an if-then rule. These rules are encoded in LegalRuleML, which allows each rule to be linked to the specific paragraphs of the GDPR from which the corresponding norm is derived[21]. Each if-then rule is in the form:

(42) $((\texttt{pa}_1\ \texttt{args}_{\texttt{pa}_1}) \ \&\ (\texttt{pa}_2\ \texttt{args}_{\texttt{pa}_2}) \ \&\ \ldots\ \&\ (\texttt{pa}_n\ \texttt{args}_{\texttt{pa}_n}),$
$(\texttt{pc}_1\ \texttt{args}_{\texttt{pc}_1}) \ \&\ (\texttt{pc}_2\ \texttt{args}_{\texttt{pc}_2}) \ \&\ \ldots\ \&\ (\texttt{pc}_n\ \texttt{args}_{\texttt{pc}_n}))$

Where $(\texttt{pa}_1\ \texttt{args}_{\texttt{pa}_1}) \ldots (\texttt{pa}_n\ \texttt{args}_{\texttt{pa}_n})$ and $(\texttt{pc}_1\ \texttt{args}_{\texttt{pc}_1}) \ldots (\texttt{pc}_n\ \texttt{args}_{\texttt{pc}_n})$ are predications in the antecedent and the consequent respectively, with $\texttt{pa}_1 \ldots \texttt{pa}_n, \texttt{pc}_1 \ldots \texttt{pc}_n$ being first-order predicates and $\texttt{args}_{\texttt{pa}_1} \ldots \texttt{args}_{\texttt{pa}_n}, \texttt{args}_{\texttt{pc}_1} \ldots \texttt{args}_{\texttt{pc}_n}$ their respective vectors of first-order terms as arguments. Most predicates in the D-KB have arity greater than two. This is not compatible with the patterns in (8.a-b), which support only unary and binary predicates to enable direct translation into RDF. To convert reified I/O logic formulae into Deontic Logic Reified representations, as will be illustrated below, predicates with arity greater than two have been decomposed into conjunctions of unary or binary predicates, following the patterns in (8.a-b).

Another major challenge in translating reified I/O logic formulae into Deontic Logic Reified formulae stems from the fundamental differences between their underlying frameworks. Reified I/O logic formulae follow the axiomatization of standard Input/Output logic [45], which employs a single type of rule, while organizing rules into different sets. Accordingly, the if-then rule pattern in (42) uses a single modality, encoded via the predicate `RexistAtTime`, which combines the predicates `Rexist` and `has-time` introduced earlier. To represent obligations, permissions, and constitutive rules, the rules are then partitioned into distinct sets: `O` (obligations), `P` (permissions), and `C` (constitutive rules). Prohibitions are represented as obligations of the contrary, a standard approach in deontic logic, and are therefore included within the `O` set. In the D-KB, the LegalRuleML tag `Context` specifies the category to which each rule belongs, namely `O`, `P`, or `C`. Each category is governed by different axioms (see [46], [47]).

However, as explained in [3], this axiomatization does not distinguish between contradictions and conflicts; accordingly, [3] re-axiomatise the deontic modalities to capture this distinction by introducing new explicit predicates: `Obligatory`, `Permitted`, and their respective subclasses. Consequently, when translating reified I/O logic formulae into Deontic Logic Reified rules, the appropriate subclasses of `Obligatory` and `Permitted` must first be identified in order to replace `RexistAtTime` in the consequent.

Regarding the constitutive rules, the modality of the consequent instead remains `Rexist`. However, it is important to note that several constitutive rules in the D-KB correspond to standard logical entailments required either by the axiomatization of reified I/O logic or for the mapping of specific predicates to those used in the PrOnto ontology. These rules are not relevant to this paper, for two reasons: on the one hand, as explained above, the axiomatization of Deontic Logic Reified differs from that of reified I/O logic; on the other hand, linking predicates used in the Deontic Logic Reified formulae with those defined in the PrOnto ontology falls outside the scope of this work. Consequently, such rules are simply omitted in the translation from the D-KB to Deontic Logic Reified.

The following subsections present examples of the translation of D-KB formulae into their corresponding Deontic Logic Reified counterparts. While it is not feasible to include or discuss all translated formulae within this paper,

[21] LegalRuleML provides two XML tags for this purpose: `lrml:LegalReferences` and `lrml:Associations`, where `lrml` is the LegalRuleML namespace prefix.

they are all available on the GitHub repository, with each formula described in plain text. For similar reasons, the following subsections do not explore the technical details of (reified) I/O logic, its method of representing deontic statements, or Jerry R. Hobbs's logical framework for natural language semantics incorporated therein. Interested readers are referred to [20], [26], [48], [49], and other related works by the same authors[22].

5.2. The general translation methodology and some initial examples

This subsection illustrates the general methodology I used to manually translate the formulae in the D-KB into the Deontic Logic Reified rules conforming to the patterns in (8.a-b), using some initial examples.

Let me begin with the reified I/O logic if-then rule in (43), which corresponds to Article 5(1)(a) of the GDPR. The full text of this and the subsequent GDPR provisions referenced in the following examples is omitted from the paper, as it is publicly available online[23].

(43) `((RexistAtTime a1 t1) & (and' a1 ep edp) & (DataSubject w) &`
`(PersonalData z w) & (Controller y z) & (Processor x) &`
`(nominates' edp y x) & (PersonalDataProcessing' ep x z),`
`(RexistAtTime a2 t1) & (and' a2 el ef et) & (lawfulness' el ep) &`
`(fairness' ef ep) & (transparency' et ep)) ∈ O`

In (43), as well as in Hobbs's logic[24], two types of predicates are used: primed (e.g., `lawfulness'`) and unprimed (e.g., `DataSubject`). The first argument of a primed predicate corresponds to the reification of that predicate. For example, `el` is a first-order term denoting the lawfulness of the personal data processing `ep`, i.e., the fact that `ep` is lawful. Any predicate can, in principle, be reified in this way; for instance, the predicate `(DataSubject w)` could be reified as `(DataSubject' eds w)`, where `eds` would denote the fact that `w` is a data subject. However, predicates are typically reified only when necessary, and there is no need to reify `DataSubject` in (43).

The formula in (43) constitutes an obligation, as it belongs to the set `O`. It states that if the state of affairs includes a personal data processing `ep` carried out in the interval `t1` by a processor `x`, nominated by a controller `y` of personal data `z` belonging to the data subject `w`, then it is obligatory that the lawfulness (along with fairness and transparency) of `ep` really exists in the state of affairs throughout `t1`.

(43) has been manually translated in the following *partial* SPARQL rule in Deontic Logic Reified:

(44)
```
CONSTRUCT{?el a :M-Obligatory,:Keep; :has-time ?t1; :has-agent ?x;
  :has-object ?ep; :has-object-state :lawful. ?ef a :M-Obligatory,:Keep;
  :has-time ?t1; :has-agent ?x; :has-object ?ep; :has-object-state :fair.
  ?et a :M-Obligatory,:Keep; :has-time ?t1; :has-agent ?x;
  :has-object ?ep; :has-object-state :transparent}
WHERE{?w a :DataSubject. ?z :personaldata-of ?w. ?y :controller-of ?z.
  ?x a :Processor. ?edp a :Rexist,:Nominates; :has-time ?t1;
  :has-agent ?y; :has-object ?x. ?ep a :Rexist, :ProcessPersonalData;
  :has-time ?t1; :has-agent ?x; :has-object ?z}
```

The following technical choices were made in translating (43) into (44):

- The predicate `and'` in (43), which distributes the modality applied to its first argument to all other arguments, has been removed; instead, the modality is explicitly repeated in (44) for each corresponding eventuality.

[22] A brief overview of the formalization proposed in [26] is also available online at https://www.isi.edu/people-hobbs/research-areas/encoding-commonsense-knowledge.

[23] https://eur-lex.europa.eu/eli/reg/2016/679/oj

[24] See https://www.isi.edu/people-hobbs/wp-content/uploads/sites/58/2023/10/bgt-evstruct.txt

- The obligations in this example have been represented as maintenance obligations. Accordingly, occurrences of `RexistAtTime` in (43) were replaced with `M-Obligatory` (in the consequent) and `Rexist` (in the antecedent), together with the thematic role `has-time` applied to the same interval `?t1`.

- The rule in (44) formalizes the maintenance obligations using the same predicates introduced in the example (40) above: `Keep` and its thematic roles. Thus, `?x` is obliged to keep the personal data processing lawful, fair, and transparent throughout the entire interval `?t1`.

On the other hand, as stated above, the SPARQL rule in (44) is only *partial*, as the variables `?el`, `?ef`, and `?et` in the consequent are *not* bound in the antecedent.

To complete the (partial) rule in (44), it is also necessary to bind the three variables to anonymous individuals created in the antecedent using the `BNode` function, as well as to add a `NOT EXISTS` clause to prevent infinite loops, as was done for all the sample norms presented earlier in this paper. However, this can be performed programmatically. In particular, a Java procedure[25] has been implemented to detect all unbound variables in the consequents of partial SPARQL rules such as (44) and automatically add the corresponding `BNode` and `NOT EXISTS` clauses to the antecedent. Thanks to this procedure, only the partial rules need to be manually edited, which are considerably less verbose. Running this Java procedure on (44) yields the following final, fully executable rule:

(45)
```
CONSTRUCT{?el a :M-Obligatory,:Keep; :has-time ?t1; :has-agent ?x;
  :has-object ?ep; :has-object-state :lawful. ?ef a :M-Obligatory,:Keep;
  :has-time ?t1; :has-agent ?x; :has-object ?ep; :has-object-state :fair.
  ?et a :M-Obligatory,:Keep; :has-time ?t1; :has-agent ?x;
  :has-object ?ep; :has-object-state :transparent}
WHERE{?w a :DataSubject. ?z :personaldata-of ?w. ?y :controller-of ?z.
  ?x a :Processor. ?edp a :Rexist,:Nominates; :has-time ?t1;
  :has-agent ?y; :has-object ?x. ?ep a :Rexist, :ProcessPersonalData;
  :has-time ?t1; :has-agent ?x; :has-object ?z.
  BIND(BNode() AS ?ef) BIND(BNode() AS ?el) BIND(BNode() AS ?et)
  NOT EXISTS{?elr a :M-Obligatory,:Keep; :has-time ?t1; :has-agent ?x;
  :has-object ?ep; :has-object-state :lawful. ?efr a :M-Obligatory,:Keep;
  :has-time ?t1; :has-agent ?x; :has-object ?ep; :has-object-state :fair.
  ?etr a :M-Obligatory,:Keep; :has-time ?t1; :has-agent ?x; :has-object
  ?ep; :has-object-state :transparent}}
```

The remainder of the paper presents only examples of partial SPARQL if-then rules, in order to keep the exposition concise. All SPARQL rules, both partial and complete, as well as the Java procedure used to expand the former into the latter and to execute them, can be found in the GitHub repository.

An example of an achievement obligation in the GDPR, instead, can be found in Article 5(1)(d), which states that the data processor is obliged to delete or rectify without undue delay any personal data that are found to be inaccurate. The D-KB represents this obligation using the following reified I/O logic formula, where `enaw` refers to the fact that the personal data `z` is inaccurate.

[25] https://github.com/liviorobaldo/dlr-swj/blob/main/DLRsuite/DKBmanager/DRLrulesBuilder.java

(46) `((RexistAtTime a1 t1) & (and' a1 ep edp enaw) & (DataSubject w) &`
`(PersonalData z w) & (Controller y z) & (Processor x) &`
`(nominates' edp y x) & (PersonalDataProcessing' ep x z) &`
`(Purpose epu) & (isBasedOn ep epu) & (inaccurate' enaw z) &`
`(RelatedTo enaw epu),`
`(RexistAtTime o1 t2) & (or' o1 eca eco) & (Delete' eca x z) &`
`(Rectify' eco x z) & (nonDelayed o1)) ∈ O`

(46) represents an achievement obligation because the actions of deleting or rectifying the personal data (reified as the variables `eca` and `eco`, respectively) are *punctual* actions that occur at the instant `t2`. This instant must take place without undue delay relative to the time `t1`, which is quantified in the antecedent and represents the moment when the processing of inaccurate personal data occurs.

Note that `eca` and `eco` must be connected by an *disjunction*: the obligation is fulfilled if either action is performed. For this reason, unlike (43), the consequent in (46) uses Hobbs's predicate `or'` rather than `and'`[26].

To simplify the SPARQL if-then rules, I avoid introducing predicates that directly correspond to Hobbs's predicates `and'` and `or'`. Instead, I introduce alternative predicates and rules, and employ modeling strategies that achieve the same semantic effect. As illustrated in the previous example, `and'` can be eliminated by distributing the modality applied to the first argument to all other arguments. Conversely, to eliminate `or'`, I introduce a superclass that *subsumes* the classes involved in the disjunction. For instance, to translate formula (46), I introduce a new class, `DeleteOrRectify`, of which both `Delete` and `Rectify` are subclasses. This allows formula (46) to be translated into SPARQL as follows:

(47)
```
CONSTRUCT{?eo a :A-Obligatory,:DeleteOrRectify; :has-agent ?x;
  :has-object ?z; :has-time ?t2}
WHERE{?w a :DataSubject. ?z :personaldata-of ?w. ?y :controller-of ?z.
  ?x a :Processor. ?edp a :Rexist,:Nominates; :has-time ?t1;
  :has-agent ?y; :has-object ?x. ?ep a :Rexist, :ProcessPersonalData;
  :has-time ?t1; :has-agent ?x; :has-object ?z. ?epu a :Purpose.
  ?ep :is-based-on ?epu. ?z a :inaccurate. ?t2 :is-without-delay ?t1}

CONSTRUCT{?e a :DeleteOrRectify}WHERE{?e a :Delete}

CONSTRUCT{?e a :DeleteOrRectify}WHERE{?e a :Rectify}
```

Note that there are *three* SPARQL rules in (47), but the last two rules are included solely to state that all actions of `Delete` or `Rectify` are also actions of `DeleteOrRectify`, i.e., that `Delete` and `Rectify` are *subclasses* of `DeleteOrRectify`. Of course, I could have equivalently used `rdfs:subClassOf`, by adding the two triples "`:Delete rdfs:subClassOf :DeleteOrRectify`" and "`:Rectify rdfs:subClassOf :DeleteOrRectify`", but I prefer to avoid mixing SPARQL rules with RDF Schema inferences.

These two rules *axiomatize* the predicate `DeleteOrRectify`, which has been introduced to represent the disjunction. While this solution is adequate for the present example, it is neither particularly elegant nor, more importantly, scalable. For instance, if it were necessary to model complex nested structures involving both disjunctions and conjunctions, such as a structure matching the pattern $((A \vee B) \wedge C) \vee (D \wedge A)$, a distinct class would need to be introduced for each level of nesting. To avoid this, it would be preferable to introduce and axiomatize properties corresponding to Hobbs' predicates `and'` and `or'`. Although the axiomatization of `and'` and `or'` is regarded as future work (see Section 8 below), this paper adopts a simplified representation. Since the D-KB contains only a small number of disjunctions, each involving a simple, single-level structure such as "delete or rectify", it was considered more practical either to define auxiliary predicates such as `DeleteOrRectify` or, when a disjunction appears in the antecedent, to use the SPARQL clause `UNION` (examples are provided in the GitHub repository).

[26] For a quick overview of Hobbs's definitions of the predicates `and'` and `or'`, see https://www.isi.edu/people-hobbs/wp-content/uploads/sites/58/2023/10/bgt-logic.txt.

The additional predicates introduced to handle disjunctions are not the only ones that can be axiomatized. In fact, many predicates representing vague concepts would need to be axiomatized to effectively apply the SPARQL rules in real-world LegalTech applications. For example, the triple "`?t2 :is-without-delay ?t1`" in (47) should hold in the knowledge graph when the time `?t2` "occurs without undue delay" with respect to `?t1`. But when exactly does this happen? Who decides, on a case-by-case basis, whether the triple holds in a given situation?

In real-world scenarios, these decisions are made by judges in courts or other designated authorities. In other words, the RDF property "`is-without-delay`" must be *legally interpreted* in context. The rationale used by judges or other authorities to support a particular legal interpretation can then be associated with additional SPARQL rules. This rationale may be challenged or rebutted by other judges or authorities (and therefore, such rules must be linked to a notion of legal validity and legal authority), and so forth.

Accordingly, as discussed in [50], a practical LegalTech application that collects and stores information about past trials would need to model, for each pair of intervals `?t1` and `?t2` connected by the RDF property "`is-without-delay`", the rationale used by different judges or appointed authorities to determine whether `?t2` occurred without delay with respect to `?t1` in the different contexts.

Since this paper does not present a finalized LegalTech application but only proposes a possible preliminary "scaffolding" for such an application, the translated knowledge base of SPARQL rules does not model different legal interpretations of the concepts. This mean that a *single* universal default definition of "without delay" is used, one that stipulates that `?t2` occurs without delay with respect to `?t1` only if it happens *within 24 hours* after the end of `?t1`. Accordingly, once data processors determine that certain personal data is inaccurate, they have 24 hours to delete or rectify it in order to comply with the obligation set forth in Article 5(1)(d) of the GDPR. This universal default interpretation of "without delay" is encoded by the following rule:

(48)
```
CONSTRUCT{?t2 a time:Interval; time:hasBeginning ?t1e; time:hasEnd ?t2e;
          :is-without-delay ?t1. ?t2e time:inXSDDateTime ?t2edt]}
  WHERE{?ep a :ProcessPersonalData; :has-time ?t1. ?t1 time:hasEnd ?t1e.
        ?t1e time:inXSDDateTime ?t1edt. BIND(BNode() AS ?t2)
        BIND(xsd:dateTime(?t1edt)+'P1D'^^xsd:duration AS ?t2edt)
        BIND(BNode() AS ?t2e) NOT EXISTS{?t2r a time:Interval;
        time:hasBeginning ?t1e; time:hasEnd ?t2e; :is-without-delay ?t1}}
```

For each processing of personal data occurring during the interval `?t1`, the rule in (48) creates a *new* interval `?t2` that begins at the end of `?t1` and ends 24 hours (i.e., `+'P1D'^^xsd:duration`) later. It then links `?t2` to `?t1` through the RDF property `is-without-delay`.

Before moving on to the next subsection, let's take a quick look at an example, also available in the GitHub repository, that specifically illustrates compliance with the achievement obligation in (47), with the RDF property `is-without-delay` interpreted as in (48).

Suppose that John is a data processor, appointed by Bill, who is the data controller of Jack's personal data, and that John processes Jack's personal data for marketing purposes from 5pm to 6pm UTC on January 1, 2025. This state of affairs is encoded in RDF as follows:

(49)
```
:John a :Processor. :Jack a :DataSubject. :marketing a :Purpose.
:JackPD :personaldata-of :Jack. :Bill :controller-of :JackPD.
[a :Rexist,:Nominates; :has-time :t1; :has-agent :Bill; :has-object :John].
[a :Rexist,:ProcessPersonalData; :has-time :t1; :has-agent :John;
                          :has-object :JackPD; :is-based-on :marketing].
:t1 time:hasBeginning
          [time:inXSDDateTime "2025-01-01T17:00:00Z"^^xsd:dateTime];
    time:hasEnd
          [time:inXSDDateTime "2025-01-01T18:00:00Z"^^xsd:dateTime].
```

Suppose further that Jack's personal data are inaccurate:

(50) `:JackPD a :inaccurate.`

and that, for this reason, John deletes the inaccurate personal data at 1am UTC on January 2, 2025:

```
(51) [a :Rexist,:Delete; :has-agent :John; :has-object :JackPD;
     :has-time [time:inXSDDateTime "2025-01-02T01:00:00Z"^^xsd:dateTime]].
```

It is easy to see that John complies with his obligation to delete or rectify Jack's inaccurate personal data, under the assumption that less than 24 hours does not constitute a substantial delay.

From (49), the contextual rule in (48) creates and adds to the knowledge graph the anonymous individual `_:b0`, which starts on 1 January 2025 at 6pm, i.e., at the end of `t1`, the interval during which Jack's personal data are processed, and ends on 2 January 2025 at the same time. `_:b0` is then linked to `t1` via the property `is-without-delay`.

```
(52)  _:b0 time:hasBeginning
                [time:inXSDDateTime "2025-01-01T18:00:00Z"^^xsd:dateTime];
           time:hasEnd
                [time:inXSDDateTime "2025-01-02T18:00:00Z"^^xsd:dateTime];
           :is-without-delay :t1.
```

The rule in (47) then infers an achievement obligation requiring John to delete or rectify Jack's personal data within the interval `_:b0`:

```
(53)  [a :A-Obligatory, :DeleteOrRectify; :has-agent :John;
       :has-object :JackPD; :has-time _:b0].
```

Finally, since the anonymous individual in (51) is an instance of the class `Delete`, the second SPARQL rule in (47) infers that it is also an instance of the class `DeleteOrRectify`. Consequently, the compliance rules introduced in the previous section determine that it complies with the obligation in (53).

5.3. Representing negations and prohibitions (i.e., negated permissions)

The previous subsection explained that the predicates `and'` and `or'` used in the reified I/O logic formulae of the D-KB have not been directly mapped to corresponding RDF properties in SPARQL. This is due to the fact that Hobbs's axiomatization of `and'` and `or'` is relatively complex, whereas the D-KB contains only simple cases of conjunction and disjunction over eventualities. For this reason, this paper adopts a simpler, albeit non-scalable, representation in RDF and SPARQL, while a full-fledged implementation of conjunctions and disjunctions over eventualities is left as future work.

Hobbs also introduced the predicate `not'` to relate eventualities to their opposites[27]. Since the axiomatization of this predicate is instead relatively simple, I chose to import from [3] the RDF property `not` into the proposed framework, along with the SPARQL rules implementing its axioms.

In Hobbs's logic, the `not` predicate is axiomatized as shown in (4.a–b). Specifically, axiom (4.a) states that if `e` and `en` are two opposite eventualities, then if one of them really exists, the other does not, and vice versa. For example, if John leaves, then it is not true that he stays; conversely, if he stays, then it is not true that he leaves.

[3] introduces an additional axiom, shown in (4.b), to relate the deontic modalities `Permitted` and `Obligatory` via `not`. Thus, for instance, if John is permitted to leave, then he is not obliged to stay; similarly, if he is obliged to leave, then it is not permitted for him to stay. See [3] for further details and examples.

[27] For a brief summary, see section 2 at https://www.isi.edu/people-hobbs/wp-content/uploads/sites/58/2023/10/bgt-logic.txt.

(54) a. $\forall_{e,\,en}$ [not(en, e) → (Rexist(e) ↔ ¬Rexist(en))]

b. $\forall_{e,\,en}$ [not(en, e) → (Permitted(e) ↔ ¬Obligatory(en))]

To implement (4.a-b) as SPARQL rules, it is necessary to represent standard negation, i.e., the symbol "¬". Since standard negation is not natively supported in RDF, the solution proposed in [3] is to introduce a class `false` and to use RDF reification[28] to indicate when a triple is false. Accordingly, ¬`Rexist(e)` is represented as:

(55)
```
<<:e rdf:type :Rexist>> rdf:type :false.
```

which reads: the fact that `e` really exists is false. With the class `false`, (4.a) can be implemented in SPARQL as in (56) (taken from [3]). (4.b) is similarly implemented (see [3] or the GitHub repository).

(56)
```
CONSTRUCT{?s a ?o}
WHERE{?e :not|^:not ?ne.
      BIND(IF(EXISTS{?e a :Rexist}, <<?ne a :Rexist>>,
           IF(EXISTS{<<?e a :Rexist>> a :false}, ?ne, ?s)) AS ?s)
      FILTER(BOUND(?s)) BIND(IF(isTriple(?s), :false, :Rexist) AS ?o)}
```

Having introduced the property `not` and its axiomatization, we can now represent the negated eventualities that occur in the D-KB. A pertinent example is the achievement obligation derived from Article 11(2) of the GDPR. This provision states that if identifying the data subject is *not* necessary for the purpose of processing personal data, and the controller demonstrates that they are *not* able to identify the data subject, then the controller is obliged to inform the data subject that they are unable to identify them.

In the D-KB, this obligation is modeled through the following reified I/O logic formula, which explicitly represents both negations using the `not` predicate.

(57)
```
((RexistAtTime a1 t1) & (and' :a1 :ep :edp :enn :ed) & (DataSubject w) &
 (PersonalData z w) & (Controller y z) & (Processor x) &
 (nominates' edp y x) & (PersonalDataProcessing' ep x z) & (Purpose epu) &
 (isBasedOn ep epu) & (not' enn en) & (NecessaryFor' en eid epu) &
 (Identify' eid x w) & (Demonstrate' ed y ena) &
 (not' ena ea) & (AbleTo' ea y eid),
 (RexistAtTime ec t1) & (Communicate' ec y w ena)) ∈ O
```

In (57), `enn` refers to the fact that the identification of the data subject `w` (i.e., the action `eid`) is *not* necessary for the purpose `epu`, while `ena` refers to the fact that the data controller `y` is *not* able to perform the action `eid`. In such cases, `y` must communicate to `w` the fact that they are not able to identify them, i.e., the fact denoted by `ena`.

The reified I/O logic formula in (57) corresponds to the following Deontic Logic Reified rule:

(58)
```
CONSTRUCT{?ec a :A-Obligatory,:Communicate; :has-agent ?y; :has-time ?t2;
            :has-object ?ena; :has-recipient ?w.}
  WHERE{?w a :DataSubject. ?z :personaldata-of ?w. ?y :controller-of ?z.
    ?x a :Processor. ?edp a :Rexist,:Nominates; :has-time ?t1;
    :has-agent ?y; :has-object ?x. ?epu a :Purpose. ?ep :is-based-on ?epu.
    ?enn a :Rexist. ?enn :not ?en. ?en a :BeNecessaryFor; :has-time ?t1;
    :has-agent ?eid; :has-object ?epu. ?eid a :Identify; :has-object ?w.
    ?ed a :Rexist,:Demonstrate; :has-time ?t1; :has-agent ?y;
    :has-object ?ena. ?ena :not ?ea. ?ea a :BeAbleTo; :has-time ?t1;
    :has-agent ?y; :has-object ?eid. ?t2 :is-without-delay ?t1}
```

[28] The operator << ... >> is part of the RDF-star vocabulary (see https://www.w3.org/2022/08/rdf-star-wg-charter); however, it will also be included in the vocabulary of the forthcoming RDF 1.2 recommendation (see https://www.w3.org/TR/rdf12-primer).

Note that the rule in (58) also involves the property `is-without-delay` introduced earlier. According to the property's definition in (48), the rule (58) therefore indicates that the controller has 24 hours to communicate to the data subjects the fact that he is unable to identify them. Article 11(2) of the GDPR does not actually specify any explicit temporal constraints on this obligation; it is implicitly understood that the communication must occur "as soon as possible", i.e., without undue delay. However, as with the previous obligation, it is ultimately the judge or other designated authority who decides on a case-by-case basis whether the communication was made within such a reasonable timeframe. The 24-hour assumption used here and in the previous obligation is again a simplification adopted solely for illustrative purposes in this paper.

Now, suppose that in the state of affairs it holds that (1) it is not true that identifying the data subject `Jack` is necessary for the purpose of processing (`marketing`), and (2) the data controller `Bill` demonstrates that he is not able to identify `Jack`, encoded in RDF as follows:

(59)
```
:enn a :Rexist. :enn :not :en. :en :has-time :t1.
:en a :BeNecessaryFor; :has-agent :eid; :has-object :marketing.
:ena a :Rexist. :ena :not :ea. :ea a :BeAbleTo; :has-agent :Bill;
:has-object :eid. :eid a :Identify; :has-object :Jack.
:ed a :Rexist,:Demonstrate; :has-time :t1; :has-agent :Bill;
:has-object :ena. :ena :not :ea. :ea a :BeAbleTo; :has-time :t1;
:has-agent :Bill; :has-object :eid.
```

From the facts in (59), the rule in (56) infers that the triples "`:enn a :Rexist`" and "`:ena a :Rexist`" hold; subsequently, the rules in (48) and (58) infer the following achievement obligation:

(60)
```
_:bo a :A-Obligatory, :Communicate; :has-agent :Bill; :has-object :ena;
:has-recipient :Jack; :has-time _:b1. _:b1 :is-without-delay :t1.
```

`Bill` is obliged to communicate to `Jack` the fact that he is not able to identify him, and this communication must be made without delay. This achievement obligation can be complied with as in the previous example; although omitted from this paper, the full example is available in the GitHub repository.

Let me now present an example of a prohibition in (61), which represents part of Article 9(1) of the GDPR. This provision prohibits the processing of certain categories of personal data, such as the processing of biometric data for the purpose of identifying the data subject (denoted by `identificationOf(w)` in (61)):

(61)
```
((RexistAtTime edp t1) & (DataSubject w) & (PersonalData z w) &
 (Controller y z) & (Processor x) & (nominates' edp y x) &
 (BiometricData z) & (Purpose identificationOf(w)) &
 Naf(exceptionCha2Art9Par2 z),
 (RexistAtTime en t1) & (not' en ep) & (PersonalDataProcessing' ep x z) &
 (isBasedOn ep identificationOf(w))) ∈ O
```

As noted at the beginning of Subsection 5.1, reified I/O logic represents prohibitions, i.e., not-permissions, as obligations to the contrary, thereby incorporating them into the set `O`. Using the equivalences in (54), the formula in (61) can be translated into the following SPARQL rule:

(62)
```
CONSTRUCT{<<?enp a :A-Permitted>> a :false. ?enp a :ProcessPersonalData;
            :has-agent ?x; :has-object ?z; :has-time ?t1; :is-based-on ?id;
            :checks-exception-in-9-2 (?w ?id ?t1 ?x ?z)}
  WHERE{?w a :DataSubject. ?z :personaldata-of ?w. ?y :controller-of ?z.
        ?x a :Processor. ?edp a :Rexist,:Nominates; :has-time ?t1;
        :has-agent ?y; :has-object ?x. ?z a :BiometricData.
        ?id a :Purpose; :is-identification-of ?w}
```

In (62), if the WHERE clause is satisfied, the rule infers that ?enp is not permitted (i.e., it is prohibited). This is represented by asserting that the reification of the triple "?enp a A-Permitted" belongs to the class false. Accordingly, the first triple in the CONSTRUCT clause reads "the fact that ?enp is permitted is false".

A further difference between the rule in (62) and those discussed earlier in this section is the presence of an exception in the CONSTRUCT clause, i.e., in the consequent. This reflects the fact that processing sensitive data, such as biometric data, is generally prohibited under the GDPR unless one of the conditions in Article 9(2) applies. The next section presents examples of exceptions from the D-KB, including one corresponding to Article 9(2).

5.4. Representing exceptions

In reified I/O logic, exceptions are represented by special predicates that are assumed to be false by default in any rule in which they appear. The predicate exceptionCha2Art9Par2 in (61) is one such case. This "false by default" behavior is implemented through negation-as-failure, using the operator Naf. For example, in the antecedent of (61), Naf(exceptionCha2Art9Par2 z) is satisfied whenever (exceptionCha2Art9Par2 z) is either false or unknown. Consequently, if no rule entails (exceptionCha2Art9Par2 z) and all other predicates in the antecedent of (61) hold, the formula's consequent follows.

Predicates entailed by exceptions may be negated by failure in the antecedent of any other rule, whether regulative, constitutive, or itself an exception, thereby allowing the modeling of exceptions of exceptions.

As an example, let us consider again (61). As explained in the previous subsection, (61) represents a prohibition that applies unless one or more of the exceptions listed in Article 9(2) of the GDPR hold. Article 9(2) specifies ten such conditions, labeled (a) to (j), under which the processing of sensitive data, such as biometric data used to identify the data subject, is permitted and therefore not subject to the general prohibition.

Article 9(2)(a) is particularly noteworthy because it contains both an exception and an exception to the exception: Article 9(2)(a) states that the processings described in Article 9(1) are not prohibited if the data subject has given explicit consent, "except where Union or Member State law provide that the prohibition referred to in paragraph 1 may not be lifted by the data subject". Furthermore, Article 8(1) of the GDPR adds another exception to the exception: consent is invalid if the data subject is below the age of 16; however, Article 8(1) also allows the Member State of the data subject to lower the minimum age for consent, provided that it is not below 13 years, thus creating an exception to the exception to the exception. This is summarized in the following itemization:

(63) Processing biometric data to identify the data subject is not prohibited if the data subject has given consent for the processing.
   a. Unless Union or Member State law provides that the data subject cannot authorize the processing of such biometric data; in that case, the data subject's consent is invalid.
   b. Unless the data subject is under the age of 16; in that case, the data subject's consent is invalid.
      i. Unless the Member State of the data subject has lowered the minimum age of consent below 16; if the data subject's age is equal to or greater than this lower age, their consent is valid.

This tree of exceptions is implemented as SPARQL rules following the same approach described in Section 3 above. Let us show the translation of the items in (63) one by one. The main exception in (63) is represented in the D-KB as follows[29]:

(64)
```
((RexistAtTime a1 t1) & (and' a1 ep egc edp) & (DataSubject w) &
 (PersonalData z w) & (Controller y z) & (Processor x) &
 (nominates' edp y x) & (PersonalDataProcessing' ep x z) &
 (Purpose epu) & (isBasedOn ep epu) & (RexistAtTime egc t2) &
 (Before t2 t1) & (GiveConsent' egc w c) & (ConsentFor c epu) &
 Naf(exceptionCha2Art9Par2PntA ep) & Naf(exceptionCha2Art8Par1 ep),
 (exceptionCha2Art9Par2 ep))∈ C
```

[29] For practical reasons, exceptions are included in the same set C, although they might more appropriately belong to a separate set.

The reified I/O logic formula in (64) is translated into the following rule:

(65)
```
CONSTRUCT{?ex :defeats-on-9-2 ?enp;
  :checks-exception-in-9-2-a (?egc); :checks-exception-in-8-1 (?w)}
WHERE{?enp :checks-exception-in-9-2 (?w ?epu ?t1 ?x ?z).
  ?egc a :Rexist,:Give; :has-agent ?w; :has-object ?c;.
  :has-time/time:inXSDDateTime ?dtegc. ?c :is-consent-for ?epu.
  ?t1 time:hasBeginning/time:inXSDDateTime ?dtbep. FILTER(?dtegc<?dtbep)}
```

This rule checks whether the data subject has previously given consent for the processing of their personal data for the purpose `?epu`. If consent has been given, a new eventuality `?ex` is created to override the prohibition `?enp`. However, the rule also requires evaluating two exceptions; if at least one of these exceptions applies, `?ex` is itself defeated, and the prohibition `?enp` ultimately remains in effect.

The first exception, corresponding to (63.a), is implemented as follows (the corresponding reified I/O logic formula is omitted):

(66)
```
CONSTRUCT{?exx :defeats-on-9-2-a ?ex}
WHERE{?ex :checks-exception-in-9-2-a (?egc).
      ?ena a :Rexist. ?ena :not ?ea. ?ea a :Authorize; :has-agent ?l;
      :has-object ?egc. ?l a :EUorMSlaw}
```

The SPARQL rule in (66) searches the knowledge graph for Union or Member State law `?l` that prohibits the data subject from giving consent `?egc`. If such a law is found, the exception in (65) is defeated.

The second exception, corresponding to (63.b), is implemented as follows. The main difference from the first exception is that this one also requires checking another exception, namely the one corresponding to (63.b.i).

(67)
```
CONSTRUCT{?ex :defeats-on-8-1 ?e; :checks-exception-in-8-1-ms (?w ?age)}
WHERE{?el :checks-exception-in-8-1 (?w). ?w :has-age ?age. FILTER(?age<16)}
```

The SPARQL rule in (67) checks whether the data subject `?w` is under 16 years of age. If so, this rule overrides the exception in (65), unless `?ex` is itself defeated by the exception in (63.b.i). The latter is represented as:

(68)
```
CONSTRUCT{?exx :defeats-on-8-1-ms ?ex}
WHERE{?ex :checks-exception-in-8-1-ms (?w ?age).
      ?w :has-member-state/:has-minimal-age-for-consent ?mac.
      FILTER(?age>=?mac)}
```

Therefore, if the Member State of the data subject `?w` has defined a minimum age for consent `?mac` and the data subject's age is greater than or equal to this value, the rule in (67) is overridden, and the processing of the data subject's biometric data for the purpose of identifying him or her is not prohibited.

An example of the rules discussed in this subsection is available on the GitHub repository. The reader is invited to add or remove exceptions, re-execute the code each time, and verify that the correct inferences are derived.

5.5. Representing constitutive rules

The previous subsections presented examples of regulative rules and exceptions. This subsection, by contrast, presents two examples of constitutive rules. As explained in Subsection 2.2, the literature describes constitutive rules as defining the concepts used in regulative rules, and thus as specifying what "counts as" these concepts [29].

For example, the concept of "lawfulness" used in the obligation shown above in the rule in (44), associated with Article 5(1)(a) of the GDPR, is defined in Article 6 of the GDPR. This article enumerates all situations that "count as" lawful processing of personal data. The first such situation occurs when the data subject has given consent to the processing of their personal data. In the proposed framework, this is represented as follows (the corresponding reified I/O logic formula is again omitted):

```
(69) CONSTRUCT{?el a :Rexist,:Keep; :has-time ?t1; :has-agent ?x;
                :has-object ?ep; :has-object-state :lawful;
                :checks-exception-in-8-1 (?w)}
     WHERE{?w a :DataSubject. ?z :personaldata-of ?w. ?y :controller-of ?z.
       ?x a :Processor. ?edp a :Rexist,:Nominates; :has-time ?t1;
       :has-agent ?y; :has-object ?x. ?ep a :Rexist, :ProcessPersonalData;
       :has-time ?t1; :has-agent ?x; :has-object ?z; :is-based-on ?epu.
       ?epu a :Purpose. ?egc a :Rexist, :Give; :has-agent ?w; :has-object ?c;
       :has-time ?egct. ?egct time:inXSDDateTime ?dtegc.
       ?c :is-consent-for ?epu. ?t1 time:hasBeginning ?t1b.
       ?t1b time:inXSDDateTime ?dtbep. FILTER(?dtegc<?dtbep)}
```

Therefore, if the data subject has given consent for the processing of their personal data, the processing is lawful. However, this is again subject to the data subject's age and, potentially, the minimum age for consent in the data subject's Member State. In other words, the exception in Article 8(1) of the GDPR, formalized in SPARQL as shown in (67) and (68) above, also applies in this case.

If `?el` in (69) is defeated, meaning the data subject's consent is not valid, Article 8(1) of the GDPR further stipulates that the processing is lawful only if consent is given by the holder of parental responsibility over the data subject. This is formalized as another constitutive rule, as follows:

```
(70) CONSTRUCT{?el a :Rexist,:Keep; :has-time ?t1; :has-agent ?x;
                :has-object ?ep; :has-object-state :lawful}
     WHERE{?w a :DataSubject. ?z :personaldata-of ?w. ?y :controller-of ?z.
       ?x a :Processor. ?edp a :Rexist,:Nominates; :has-time ?t1;
       :has-agent ?y; :has-object ?x. ?ep a :Rexist, :ProcessPersonalData;
       :has-time ?t1; :has-agent ?x; :has-object ?z; :is-based-on ?epu.
       ?epu a :Purpose. ?egc a :Rexist, :Give; :has-agent ?h; :has-object ?c;
       :has-time ?egct. ?h :is-holder-of-parental-responsibility ?w.
       ?egct time:inXSDDateTime ?dtegc. ?c :is-consent-for ?epu.
       ?t1 time:hasBeginning ?t1b. ?t1b time:inXSDDateTime ?dtbep.
       FILTER(?dtegc<?dtbep)}
```

As in the previous case, the GitHub repository includes an example involving the constitutive rules in (69) and (70).

5.6. Representing nested obligations and nested permissions

The final category of obligations considered in the D-KB is the so-called "nested obligations". These arise when the condition of an obligation contains another obligation. An illustrative example is: "If the manager is obliged to do something, his secretary is obliged to note it in his agenda".

Standard I/O logic cannot represent nested obligations (or nested permissions) because it lacks formal mechanisms to reference if-then rules within other if-then rules. By contrast, reified I/O logic, thanks to reification, is capable of directly representing them (see [1], §4.1, for further details).

The GDPR, and so the D-KB, contains some nested obligations and nested permissions. A simple example is Article 6(1)(c) of the GDPR, which stipulates that processing of personal data is lawful if it is "necessary for compliance with a legal obligation to which the controller is subject".

In the Deontic Logic Reified translation of the D-KB, this article is represented as follows:

```
(71) CONSTRUCT{?el a :Rexist,:Keep; :has-time ?t1; :has-agent ?x;
                :has-object ?ep; :has-object-state :lawful}
     WHERE{?w a :DataSubject. ?z :personaldata-of ?w. ?y :controller-of ?z.
           ?x a :Processor. ?edp a :Rexist,:Nominates; :has-time ?t1;
           :has-agent ?y; :has-object ?x. ?ep a :Rexist,:ProcessPersonalData;
           :has-time ?t1; :has-agent ?x; :has-object ?z.
           ?en a :Rexist,:BeNecessaryFor; :has-agent ?ep; :has-object ?eo.
           ?eo a :Obligatory; :has-agent ?y}
```

Note that in (71) the obligation `?eo` is not specified. Article 6(1)(c) of the GDPR applies when the controller has *a* legal obligation, but the article does not impose further constraints on this obligation. However, it is highly unlikely that *any* obligation the controller is subject to would justify "lawfully" processing `?w`'s personal data. At a minimum, it would make sense to require that the processing of `?w`'s personal data should be the *only* way for the controller to comply with the obligation, and compliance with this obligation should be considered objectively "important". Again, it would ultimately be up to judges or other appointed authorities to decide, on a case-by-case basis, whether the controller's obligation indeed justifies processing `?w`'s personal data. To account for these cases, (71) could then be refined by replacing `Obligatory` with a subclass, e.g., `ObligatoryWrtArt6_1_c`, with the determination of whether `?eo` belongs to this class being legally interpreted. However, as noted earlier, this lies beyond the scope of the present paper, since the framework proposed here does not take into account the various legal interpretations of the concepts employed in the Deontic Logic Reified rules.

As a simple example of a state of affairs that triggers the rule in (71), consider the triples in (72), which indicate that the data controller Bill is obliged to submit a report to the European Data Protection Board (EDPB), and that fulfilling this obligation necessarily entails processing Jack's personal data. The triples in (72) also specify that John, appointed by Bill as the data processor (as encoded earlier in (49)), indeed processes Jack's personal data.

```
(72) [a :Rexist,:BeNecessaryFor;
        :has-agent [a :Rexist,:ProcessPersonalData; :has-agent :John;
              :has-object :JackPD; :has-time :t1; :is-based-on :marketing];
        :has-object [a :A-Obligatory,:Submit; :has-agent :Bill;
                                 :has-object :report; :has-recipient :EDPB]].
```

It is straightforward to see that the rule in (71) infers from (72) that the processing is lawful.

6. A generate-and-test methodology to formally verify the Deontic Logic Reified rules from the D-KB

The previous section has shown that Deontic Logic Reified can represent a reasonably varied and representative set of norms by encoding the reified I/O logic formulae in the D-KB corresponding to the norms of Chapter II of the GDPR. In total, 65 rules have been encoded: 26 regulative rules, 24 constitutive rules, and 15 exception rules. When the antecedents of the 26 regulative rules are satisfied, they jointly entail 39 deontic statements: 29 obligations, 4 permissions, 3 not-obligations, and 3 not-permissions (i.e., prohibitions).

While incrementally translating the targeted reified I/O logic formulae into SPARQL, I simultaneously tested them on sample input facts, verifying that each rule produced the expected outcomes. As the number of rules increased, and given that they may interact with one another, it became increasingly difficult to keep track of their overall consistency. For this reason, I began collecting the sample facts that trigger the rules in a JSON file and developed Java procedures that generate all possible combinations of these facts, feed them into the automated reasoner, and automatically verify that the expected outcomes were always present in the inferred knowledge graph.

The incremental construction and use of this JSON file can be seen as a general generate-and-test methodology for the formal verification of the Deontic Logic Reified rules derived from the D-KB. However, I would not refer to this procedure as a "validation", or at most only as a "*formal* validation", because, as discussed in the previous section, this paper does not aim to assess whether the proposed formulae accurately reflect the intended legal meaning of the

encoded GDPR norms. That objective was instead addressed by the DAPRECO methodology in [44], which led to the construction of the D-KB. Here, the focus is instead on the logical framework *per se*, and the D-KB is assumed to be legally valid. The aim of this paper is solely to demonstrate that the framework is sufficiently expressive to encode a reasonably broad and representative set of norms within acceptable computational time.

Still, constructing the JSON file in parallel with the editing of the rules greatly helped in identifying errors such as misspelled SPARQL variables, missing predicates, or incorrect `FILTER` conditions, all of which could prevent the rules from producing the expected outputs. I therefore consider this a good practice for any set of legally validated Deontic Logic Reified rules, to ensure that they also yield the expected results from a computational perspective.

It should be noted, however, that the JSON file exploits certain characteristics of the encoded rules, which may not necessarily be present in other rule sets. In other words, although the methodology illustrated below provides a general basis for formal validation, it nonetheless needs to be adapted to new sets of rules (see Section 8 below, devoted to future work).

In particular, the following peculiarities of the encoded Deontic Logic Reified rules are observed:

(73) a. Each regulative rule in the set, when triggered by facts satisfying its antecedent, can be easily verified using sample facts that introduce a really existing eventuality corresponding to each (non-)obligation or (non-)permission entailed by the regulative rule.

b. Each constitutive rule in the set, when triggered by facts satisfying its antecedent, only produces really existing eventualities that either comply with an obligation, conform to a permission, violate a non-permission, or are associated with a not-obligation (i.e., they really exist although they were not obligatory) generated by another regulative rule in the set. Below, I use the term "D-fulfill" (where "D" stands for "deontic") to subsume any of these four relations.

c. Each exception in the set, when triggered by facts satisfying its antecedent, only defeats other rules, which can themselves be exceptions.

d. Multiple constitutive rules may D-fulfil the same regulative rule, and multiple exceptions may defeat the same rule. However, each constitutive rule and each exception can be tested in isolation from the others that D-fulfil or defeat the same rule, since they apply *disjunctively* rather than *conjunctively*.

(73.b), in particular, does not appear to hold for arbitrary sets of rules. In other words, while all rules in the translated portion of the D-KB only produce really existing eventualities that *directly* D-fulfil a deontic statement entailed by a regulative rule, it is easy to envisage a situation in which the really existing eventuality that D-fulfils a deontic statement is instead entailed by a *set* of constitutive rules, some of which may even be applied in a *chain* (meaning that some constitutive rules in the set produce facts that activate further constitutive rules). Therefore, to generalize the methodology for formal verification presented in this section, the first step appears to be extending (73.b) to account for *sets* of constitutive rules that, if they all trigger, D-fulfil regulative rules.

In light of (73.a–d), the aforementioned JSON file is organized as follows:

(74) a. Each rule is associated with a JSON object specifying an `antecedent`, a `consequent`, the `type` (regulative, constitutive, or exception), the ID(s) of the corresponding formulae in the D-KB, and a unique `index`. The SPARQL rules are also automatically enriched so that each entailed deontic statement or really existing eventuality is associated with the ID(s) of the corresponding D-KB formulae via a dedicated property, `has-generating-dkb-rule-id`[30]. This enables automatic verification that the inferred knowledge graph contains the expected model encoded in the JSON file.

b. The `antecedent` of each rule specifies sample facts that activate the rule. These facts are labeled using the rule's `index` so that they can be referenced by other constitutive or exception rules that either D-fulfil or defeat the rule.

[30]This is specifically done by https://github.com/liviorobaldo/dlr-swj/blob/main/DLRsuite/DKBmanager/DRLrulesBuilder.java

c. The `consequent` of a regulative rule specifies sample facts that D-fulfil the deontic statements entailed by the rule. In particular, these facts correspond to really existing eventualities (`Rexist`) that specialize the entailed deontic statements according to the compliance-checking rules (e.g., (36) and (37), which infer compliance with obligations).

d. The `consequent` of a constitutive rule specifies the `index` of a regulative rule that the really existing eventualities entailed by the constitutive rule are expected to D-fulfil. As explained above, this is sufficient for the rules corresponding to the translated formulae from the D-KB, given (73.b). In the general case, however, the facts D-fulfilling the deontic statements entailed by a regulative rule might be entailed by a *chain* of constitutive rules, which would require more sophisticated annotations in the JSON file.

e. The `consequent` of an exception specifies the `index` of another rule that is defeated by the exception. As explained below, some exceptions, such as the one implemented by the rule shown above in (67), can defeat more than one rule. These exceptions are therefore associated with multiple JSON objects in the JSON file, one for each rule they defeat. By contrast, each regulative and constitutive rule, as well as each exception that defeats a single rule, is associated with a single JSON object.

The Java procedures implementing the generate-and-test methodology on the considered Deontic Logic Reified rules take as input the JSON file organized as in (74.a–e), cluster the JSON objects occurring therein (with each cluster including a regulative rule together with all its constitutive rules and exceptions), generate all possible combinations of `antecedent` facts, and, for each combination, verify that the inferred knowledge graph contains the expected inferences, i.e., those specified in the `consequent`s of the rules in the cluster.

To illustrate how the generate-and-test methodology works, this section considers a small excerpt of the JSON file[31]. The complete JSON file is of course available on the GitHub repository[32], along with the Java procedures implementing the generate-and-test methodology, allowing users to run them locally and verify the results.

The excerpt contains sample facts related to the regulative rule shown above in (44), along with some of its associated constitutive rules and exceptions. (75) shows the JSON object associated with the regulative rule in (44).

(75)
```
{"index":"1", "D-KBid":"statements1Formula1", "type":"regulative",
 "antecedent":"""w_1 a :DataSubject. :z_1 :personaldata-of :w_1.
 :x_1 a :Processor. :y_1 :controller-of :z_1. :edp_1 a :Rexist, :Nominates;
 :has-time :t1; :has-agent :y_1; :has-object :x_1. :ep_1 a :Rexist,
 :ProcessPersonalData; :has-time :t1; :has-agent :x_1; :has-object :z_1;
 :is-based-on :epup_1. :epup_1 a :Purpose. :ep_1 :is-based-on :epup_1.""",
 "consequent":"""ek1_1 a :Rexist,:Keep; :has-agent :x_1; :has-object :ep_1;
 :has-time :t_including_t1; :has-object-state :lawful. :ekf_1 a :Rexist,:Keep;
 :has-time :t_including_t1; :has-agent :x_1; :has-object-state :fair;
 :has-object :ep_1. :ekt_1 a :Rexist,:Keep; :ekt_1 :has-time :t_including_t1;
 :has-agent :x_1; :has-object :ep_1; :has-object-state :transparent."""}
```

The JSON key "`antecedent`" specifies sample facts that trigger the rule in (44): the individual `w_1` matches the SPARQL variable `?w`, the individual `z_1` matches `?z`, and so on. If the facts in the `antecedent` are asserted in the ABox, the rule in (44) generates three maintenance obligations: the individual `x_1` is obliged to keep the processing `ep_1` lawful, fair, and transparent throughout the temporal interval `t1`. As explained above, this rule has been automatically enriched so that each entailed obligation is associated with the D-KB id `statements1Formula1` via the property `has-generating-dkb-rule-id`.

To verify the rules, when the facts in the `consequent` are also asserted, the Java procedures implementing the generate-and-test methodology automatically check that, in the inferred knowledge graph, every really existing

[31] https://github.com/liviorobaldo/dlr-swj/blob/main/DLRsuite/D-KB/reducedUseCaseBaseline.json
[32] https://github.com/liviorobaldo/dlr-swj/blob/main/DLRsuite/D-KB/UseCaseBaseline.json

eventuality occurring in the consequent, namely ekl_1, ekf_1, and ekt_1, complies-with a deontic statement associated with statements1Formula1 via the property has-generating-dkb-rule-id.

Note that all individuals in (75), except the temporal intervals t1 and t_including_t1, are associated with the subscript "_1", i.e., the index of the JSON object; this index is used to reference the rule (see below). The two temporal intervals are not indexed because they are reused across rules, in order to avoid defining a different pair of intervals for each rule. They are defined as in (76); it is easy to see that these definitions allow for compliance of the three really existing eventualities ekl_1, ekf_1, and ekt_1 through the rule in (44).

(76)
```
:t1 a time:Interval; time:hasBeginning :t1b; time:hasEnd :t1e.
:t1b time:inXSDDateTime "2026-01-01T17:00:00Z"^^xsd:dateTime.
:t1e time:inXSDDateTime "2026-01-01T18:00:00Z"^^xsd:dateTime.
:t_including_t1 a time:Interval; time:hasBeginning :t_including_t1b;
                                                 time:hasEnd :t_including_t1e.
:t_including_t1b time:inXSDDateTime "2026-01-01T16:00:00Z"^^xsd:dateTime.
:t_including_t1e time:inXSDDateTime "2026-01-01T19:00:00Z"^^xsd:dateTime.
```

One of the deontic statements entailed by the regulative rule in (44) is also complied with by really existing eventualities entailed by several constitutive rules. This is the maintenance obligation of keeping the processing lawful throughout t1. For example, the constitutive rule shown above in (69) states that if the data subject has given consent to the processing of personal data before t1, then the processing is lawful throughout t1. Another constitutive rule entailing the lawfulness of processing corresponds to the D-KB id statements11Formula1, derived from Art. 6(1)(e) of the GDPR: processing is necessary for the performance of a task carried out in the public interest. These two rules are associated with the following JSON objects:

(77)
```
{"index":"9", "D-KBid":"statements7Formula1", "type":"constitutive",
  "antecedent":"""egcp_9 a :Rexist,:Give; :has-agent :w_1; :has-object :cp_9;
  :has-time :t_before_t1; :is-consent-for :epup_1.""", "consequent":"1"}
{"index":"13", "D-KBid":"statements11Formula1", "type":"constitutive",
  "antecedent":"""ebnfetpi_13 a :Rexist,:BeNecessaryFor; :has-agent :ep_1;
  :has-object :extpi_13. :extpi_13 a :Execute; :has-object :tpi_13.
  :tpi_13 :is-related-to :pi_13. :pi_13 a :PublicInterest.""", "consequent":"1"}
```

Note that both antecedents in (77) use individuals with subscript "_1", corresponding to the index of (75), e.g., the data subject w_1. This ensures that the really existing eventualities entailed by the two constitutive rules D-fulfil the deontic statements entailed by the rule with id statements1Formula1. This is again automatically verified by the generate-and-test procedures whenever the facts in the antecedents in (77) are added to the ABox.

The two constitutive rules referred to in (77) each have an exception. As in the case of the rule in (65), the data subject's consent is valid only if the data subject is at least 16 years old. With regard to the constitutive rule with id statements11Formula1, processing necessary for a task carried out in the public interest is valid only if the data subject does not explicitly object to it, as stipulated by Art. 21(1) of the GDPR.

The two exceptions, the first of which was shown above in (67), are associated with the following JSON objects:

(78)
```
{"index":"23", "D-KBid":"statements17Formula2", "type":"exception",
  "antecedent":"""w_1 :has-age 14.""", "consequent":"9"}
{"index":"73", "D-KBid":"statements108Formula4/statements108Formula5/
    statements108Formula6", "type":"exception",
  "antecedent":"""ewdr_73 a :Rexist,:Object; :has-time :t_instant_inside_t1;
   :has-agent :w_1; :has-object :ep_1.""", "consequent":"13"}
```

Therefore, when the sample facts in the `antecedent` of the JSON object with `index` "23" are asserted, i.e., when it is asserted that the data subject `:w_1` is 14 years old, the rule in (67) defeats the rule activated by the `antecedent` of the JSON object with `index` "9". Similarly, when the sample facts in the `antecedent` of the JSON object with `index` "73" are asserted, i.e., when `:w_1` objects to the processing `:ep_1` at time `:t_instant_inside_t1`, which falls within[33] the interval `:t1`, the second exception defeats the rule activated by the `antecedent` of the JSON object with `index` "13". This can again be automatically verified by the Java procedures implementing the generate-and-test methodology.

The first exception referred to in (78) is an example of an exception that defeats more than one rule, as discussed in (73.e). In addition to defeating the rule with id `statements7Formula1`, it also defeats the rule in (65) above, as explained in the previous section. Therefore, the JSON file includes *two* JSON objects referring to this exception. However, this paper only presents the one in (78), while the one corresponding to the exception that defeats the rule in (65) is available in the GitHub repository[34].

Both exceptions in (78) admit further exceptions. If data subjects are less than 16 years old, their consent is invalid *unless* their Member State lowers the minimum age of consent so that their age is included (GDPR, Art. 8(1)). Similarly, an objection by the data subject to the processing of personal data in the public interest is legitimate *unless* the processing is made "for the establishment, exercise or defence of legal claims" (GDPR, Art. 21(1)).

To account for these exceptions to exceptions, the JSON file includes the following two JSON objects:

(79)
```
{"index":"22", "D-KBid":"statements17Formula1", "type":"exception",
 "antecedent":"""::w_1 :has-member-state :ms_22.
      :ms_22 :has-minimal-age-for-consent 14.""", "consequent":"23"}
{"index":"76", "D-KBid":"statements108Formula7/statements108Formula8/
    statements108Formula9", "type":"exception",
 "antecedent":"""ep_1 :is-related-to :legclaim_76.
 :legclaim_76 a :LegalClaim.""", "consequent":"73"}
```

It should be clear at this point that the JSON objects are organized as a *tree*, representing all possible inference patterns that the rules may generate. The construction of the JSON file is, therefore, nothing more than a process to "unfold" these patterns step by step, in order to verify that the inferences consistently produce the expected outputs from the given sample facts.

For example, the JSON objects reported in (75), (77), (78), and (79), are organized in the following tree:

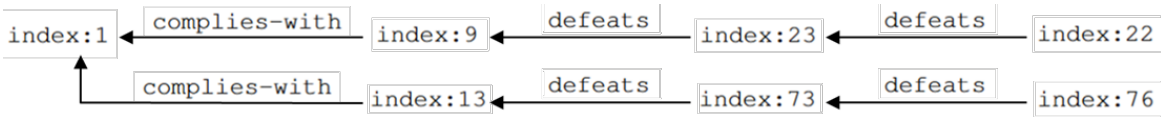

Fig. 2. Tree structure of the JSON objects in (75), (77), (78), and (79)

The Java procedures implementing the generate-and-test methodology use the tree in Figure 2 to generate all possible combinations of input facts (JSON key `antecedent`) for the seven rules considered. They then verify that the expected results, computed from the `consequent`s of the JSON objects involved in each combination, are present in the inferred knowledge graph.

[33] `:t_instant_inside_t1` is defined as "`:t_instant_inside_t1 a time:Instant; time:inXSDDateTime "2026-01-01T17:30:00Z"^^xsd:dateTime.`". Note also that the D-KB id of the second rule referred to in (77) is a *composite* id. The reason is that three rules from the D-KB (i.e., those with ids `statements108Formula4`, `statements108Formula5`, and `statements108Formula6`) have been implemented via a single Deontic Logic Reified rule.

[34] Specifically, this is the JSON object with `index` "35" in the JSON file available at https://github.com/liviorobaldo/dlr-swj/blob/main/DLRsuite/D-KB/UseCaseBaseline.json; the JSON object associated with the rule in (65) has `index` "29".

As noted in (73.d) above, each branch of the tree is independent of its siblings and can therefore be executed in isolation. Therefore, in the example in Figure 2, there is no need to test the scenario in which the two constitutive rules referred to in (77) apply *together*, since the inferred knowledge graph would simply contain the *union* of the inferences obtained by evaluating each constitutive rule separately, i.e., that all really existing eventualities entailed by either rule comply with the regulative rule referred to in the JSON object with `index` "1".

The Java procedures implementing the generate-and-test methodology therefore start from the root and traverse each branch in turn. In other words, they implement a *depth-first search* on the tree: at each step towards the leaves, they recompute the inferred knowledge graph by adding the facts in the `antecedent` of the JSON object at the corresponding `index` and verifying that the expected inferences are present. Accordingly, with respect to the tree in Figure 2, the Java procedures test the following combinations of indices:

- 1
- 1, 9
- 1, 9, 23
- 1, 9, 23, 22
- 1, 13
- 1, 13, 73
- 1, 13, 73, 76

For each combination, the facts in the `antecedent`s of the JSON objects at the corresponding indices are asserted. In the case of the regulative rule (the one at index `1`, which is present in every combination), the facts in the `consequent` are also asserted. The procedures then compute the expected inferences from the JSON objects and verify that they are present in the inferred knowledge graph. As explained above, for the regulative rule, the really existing eventualities in the `consequent` must comply with a deontic statement entailed by the rule. For the two constitutive rules, each really existing eventuality entailed by the rules must comply with a deontic statement entailed by the regulative rule at the root. In the case of exceptions, the procedure verifies that the defeated rule does *not* produce the inferences it was expected to produce. These checks are implemented by comparing the D-KB ID(s) found in the JSON objects, which are linked to the inferred eventualities by the Deontic Logic Reified rules via the property `has-generating-dkb-rule-id`. This paper omits further specific technical, code-level details implementing the depth-first search and the comparisons described above. Interested readers are referred to the Java code available on GitHub, which includes explanatory notes and comments to facilitate understanding.

7. Executing the Deontic Logic Reified formulae from the D-KB

As explained in the Introduction, one of the motivations behind the research presented in this paper was the design and implementation of an automated legal reasoner capable of combining the advantages of the reasoners surveyed and compared in [2].

SPINdle and Proleg are perhaps the two best-known legal reasoners in the literature. Their main distinctive feature is that they use superiority relations, rather than negation-as-failure, to implement defeasibility. Superiority relations are also used in the LegalRuleML standard[35]. Since superiority relations are monotonic, they are easier for legal experts to understand and edit, and they avoid the computational issues associated with non-monotonic operators such as negation-as-failure, in particular the non-deterministic behaviour of rules when they are not properly prioritised.

Deontic Logic Reified, unlike the SHACL-based reasoner from [18], which is also surveyed in [2] and relies on negation-as-failure for handling defeasibility, defines constructs that resemble those used in SPINdle and Proleg to implement superiority relations.

However, the SHACL-based reasoner from [18] offers important advantages over SPINdle and Proleg. First, like Deontic Logic Reified, it can natively process RDF data; by contrast, using RDF with SPINdle and Proleg requires first converting the data into their input formats, but clearly the computational cost of this translation is

[35] See https://docs.oasis-open.org/legalruleml/legalruleml-core-spec/v1.0/os/legalruleml-core-spec-v1.0-os.html#_Toc38017880

not feasible for Semantic Web big data. In addition, even when all three reasoners operate on their respective native input formats, the SHACL-based reasoner from [18] exhibits significantly better computational performance than SPINdle and Proleg, as reported in [2].

Therefore, since Deontic Logic Reified was designed to incorporate the advantages of SPINdle and Proleg over the SHACL-based reasoner from [18], a natural question arises: is the computational advantage of the latter over the former still preserved? This final section shows that not only is it preserved, but the Deontic Logic Reified reasoner even *outperforms* the SHACL-based reasoner from [18].

To demonstrate this, I conducted a very simple experiment using the translated data from the D-KB described in the two previous sections.

First, I extended the Java procedure used to transform the compact version of the Deontic Logic Reified rules into their full version (for the reasons explained in (45) above) so that it also outputs the corresponding SHACL rules processable by the SHACL-based reasoner from [18]. Secondly, I implemented another Java procedure[36] which, using the JSON file employed for the formal validation of the rules (as explained in the previous section), generates sample ABoxes of facts to be used as input for the two reasoners.

The second Java procedure has been adapted from the one used in [2] to generate sample ABoxes in the input formats of each of the seven legal reasoners compared in that survey. For each rule, the procedure adds to the ABox the facts that trigger the rule antecedent according to a user-defined probability. However, unlike [2], in the experiment described in this section the facts are asserted in a single output format, namely RDF, which is the common input format of the two reasoners compared here.

The two reasoners were then executed on the same ABoxes, while measuring their execution time on each ABox. All experiments were run on a PC equipped with an 11th Gen Intel(R) Core(TM) processor running at 2.4 GHz, 16 GB of RAM, and Windows 11 Pro. Average execution times were computed over sets of 5, 10, and 15 randomly generated ABoxes, using a probability of 50%. The results, reported in Table 1, show that the Deontic Logic Reified reasoner consistently outperforms the SHACL-based reasoner from [18]. The source code of this experiment is also available in the GitHub repository, allowing readers to re-run the experiments and independently verify the results.

Reasoner	**5 ABoxes**	**10 ABoxes**	**15 ABoxes**
Deontic Logic Reified reasoner	1.9542	2.0317	1.8875
SHACL-based reasoner from [18]	3.1186	3.2218	3.3479

Table 1
Average execution time (in seconds) of the Deontic Logic Reified reasoner and the SHACL-based reasoner from [18] over randomly generated ABoxes with a rule activation probability of 50%.

The observed improvement in computational performance is directly related to the use of superiority relations in place of negation-as-failure.

As mentioned above, negation-as-failure can lead to non-deterministic behaviour; in particular, rules may produce different results depending on the *order* in which they are executed. To avoid this issue, rules must be explicitly ordered so that they are executed consistently across different runs. SHACL rules provide a dedicated datatype property for this purpose: `sh:order`[37]. This property associates a SHACL rule with an integer value, and the SHACL reasoner executes the rules according to these values, from lowest to highest. In [18], the rules were executed using the TopBraid SHACL API[38], one of the most widely used Java libraries for SHACL rule execution. These libraries rely on the Apache Jena libraries[39], which are also used here to implement the Deontic Logic Reified reasoner.

[36] https://github.com/liviorobaldo/dlr-swj/blob/main/DLRsuite/GenerateAndTest/GenerateAndRunRandomABoxes.java
[37] https://www.w3.org/TR/shacl-af/#rules-order
[38] https://mvnrepository.com/artifact/org.topbraid/shacl/1.4.2
[39] https://jena.apache.org

By contrast, as explained in Section 3 above, when using superiority relations, rules no longer need to be explicitly prioritised. Superiority relations are meta-relations among rules that can be inferred *monotonically* and that, in a subsequent phase, determine which rule conclusions are to be defeated in the inferred knowledge graph.

Therefore, the infrastructure provided by the TopBraid SHACL API is no longer required for rule execution. The Apache Jena libraries alone suffice, resulting in a more lightweight computational implementation.

Another limitation of the TopBraid SHACL API, from a computational perspective, is its reliance on the Apache Jena `Model` class[40], as the API operates on instances of this class. This is a high-level RDF abstraction designed for general-purpose graph manipulation, serialization, and reasoning. While this flexibility makes it convenient, it also introduces significant overhead in terms of memory usage and execution time, as it maintains a rich internal structure and an additional API layer above the raw triple representation. By contrast, when executing large numbers of rules, more efficient lower-level Jena graph representations are typically preferred, as they provide a leaner data structure and reduce the cost of repeated rule evaluation and graph updates.

The Deontic Logic Reified reasoner was implemented with these considerations in mind, resulting in a substantial performance improvement, as shown in Table 1.

Further improvements in performance may be achieved by leveraging techniques from contemporary Answer Set Programming (ASP) research, which is known to support highly efficient reasoning implementations. Indeed, recent work, e.g., [51], has begun exploring ASP-based approaches to legal reasoning, which merits further investigation. While this paper primarily focuses on the definition of the logical framework, future work will explore this direction, with the aim of developing an efficient ASP-based implementation of the Deontic Logic Reified reasoner. Additional future work is discussed in the next section.

8. Future works

The previous sections introduced Deontic Logic Reified, a new deontic logic specifically designed for compliance checking over Semantic Web data, as it is fully implemented in RDF and SPARQL.

Deontic Logic Reified has been obtained by merging results from three previous journal papers of mine, notably [3], which implemented the Deontic Traditional Scheme in RDF and SPARQL. This paper extends [3] with defeasibility and temporal management, providing the framework with the expressive power needed to represent the norms in the GDPR, at least at the same level of detail at which they have been formalized within the DAPRECO knowledge base (D-KB), which is here considered legally valid and used as a benchmark to demonstrate that the logic can adequately model norms from existing legislation.

Although the GDPR may be considered a sufficiently representative piece of existing legislation, the version of Deontic Logic Reified presented in this paper should still be regarded as only a "first release" of the logical framework, which will need to be further extended in the future to cover a broader variety of norms.

First of all, it should be noted that all the norms considered above, both those in the constructed examples and those from the D-KB, involve only *universal* quantifications over elements satisfying the antecedent, or *existential* quantifications over variables occurring in the consequents of the rules. The latter are represented as blank nodes that Skolemize these existential quantifiers, as explained in accordance with standard SPARQL encoding practices.

In the most general case, however, the arguments of the thematic roles associated with the actions involved in deontic statements should instead involve *generalized quantifiers*[41], which formally amount to introducing existential quantifications over *sets* into the logical framework (see [52], [53]). Two examples of norms featuring generalized quantifiers in the arguments of deontic statements are:

(80) a. John is obliged to pay *at least £10*.

b. John is prohibited to pay *more than £5*.

[40] https://jena.apache.org/documentation/javadoc/jena/org.apache.jena.core/org/apache/jena/rdf/model/Model.html

[41] https://plato.stanford.edu/entries/generalized-quantifiers

Note that the two norms in (80) conflict with one another: if John pays more than £10, he complies with (80.a) but violates (80.b); while if he pays £5 or less, he does not violate (80.b) but does not comply with (80.a) either. The rules from [3] for inferring conflicts among deontic statements should therefore be generalized to account for any type of quantifier occurring in the arguments of the thematic roles.

Similar generalizations should also be implemented within the rules checking compliance, violations, etc., which should take into account the distinction between *types and tokens*[42], a distinction that has been widely investigated in philosophy, linguistics, and formal semantics. Suppose, for instance, that John is obliged to pay *exactly £3* in cash and that he has six £1 coins in his pocket. Then there are $6!/(3!(6-3)!) = 20$ possible subsets of the six coins that John can use to pay £3. From the perspective of the obligation, which denotes a *type*, all 20 combinations are acceptable, each of which is a *token*. Once John selects one of them to fulfil the obligation, i.e., once an eventuality is asserted on the `Rexist` modality, the set of coins in his pocket changes accordingly, which in turn affects the set of possible combinations available for future payments.

In light of these examples, it may be concluded that while the `Rexist` modality holds for actions involving *tokens*, deontic modalities hold for actions involving *types*. In future work, Deontic Logic Reified vocabulary will accordingly be enriched with this distinction, as well as with a specific arithmetic over types and tokens, needed to properly generalize the Deontic Logic Reified rules for inferring conflicts, compliance, violations, etc.

The introduction of generalized quantifiers further complicates the formal setting, as they can engender ambiguities when two or more of them occur in the same sentence, with each interpretation yielding different compliance-checking inferences. Consider:

(81) a. The men are obliged to wear a red t-shirt.

b. The men are obliged to lift a table.

c. The men are obliged to pay a £3000 sanction.

The preferred interpretation of (81.a) is a *distributive* reading, in which each man is obliged to wear a different red T-shirt. Conversely, the preferred interpretation of (81.b) is a *collective* reading, in which the men, taken together, must lift a single table. Finally, the preferred interpretation of (81.c) is a *cumulative* reading, in which the men must share a £3000 sanction among them, i.e., each of them pays an amount lower than £3000 such that the sum of all amounts paid is equal to £3000.

To account for generalized quantifiers in Deontic Logic Reified, I plan in future work to integrate therein the formal account that I have proposed in my previous research, specifically in [54] and [55].

Another direction for future work concerns the proper representation of conjunctions and disjunctions of eventualities, which are implemented in [26] by means of the predicates `and'` and `or'`[43]. In this paper, however, these predicates have been omitted in order to simplify the translation of the reified Input/Output logic formulae from the D-KB into the corresponding Deontic Logic Reified rules (see, for instance, (44) and (47) above). Conjunctions of eventualities have been expanded into separate assertions of their conjuncts, whereas disjunctions have been represented through the introduction of ad hoc predicates, such as `DeleteOrRectify`, as in (47), which subsumes both `Delete` and `Rectify`. While this solution works for the (few and simple) cases in the D-KB, it requires further investigation to understand how to properly model patterns featuring any embeddings of conjunctions, disjunctions, and possibly also negations, among eventualities holding for one of the deontic modalities.

In the literature, it is standardly assumed that obligatory conjunctions bi-imply the obligations of their conjuncts (in symbol: $\mathtt{OB(A \wedge B)} \leftrightarrow \mathtt{(OB(A) \wedge OB(B))}$), whereas this does not hold for obligatory disjunctions (in symbol: $\mathtt{OB(A \vee B)} \not\leftrightarrow \mathtt{(OB(A) \vee OB(B))}$). However, as shown in [56] and [57], certain sets of facts involving obligatory disjunctions appear to enable specific inferences, and the framework should therefore be able to capture them as well.

In [3], I introduced some rules that capture the inference patterns discussed in [56] and [57] for the `Obligatory` predicate. It remains unclear, however, whether and how similar patterns extend to the other three deontic modalities, as neither [56] nor [57], nor (to the best of my knowledge) any other work in the literature, has addressed this issue.

[42] See https://plato.stanford.edu/entries/types-tokens

[43] See also https://www.isi.edu/people-hobbs/wp-content/uploads/sites/58/2023/10/bgt-logic.txt

On the other hand, in [58] I have recently argued that entailments among deontic statements should perhaps not be allowed at all, as it is unclear how the penalties associated with the entailed obligations relate to those associated with the obligations from which they are derived.

My future research on these issues will build upon the preliminary results from [3] and [58], towards a complete and sound account of conjunctions and disjunctions of deontic statements in Deontic Logic Reified.

It goes without saying that the methodology for the formal verification presented in Section 6 above will also need to evolve in step with these developments. In particular, a current limitation of this methodology is the one discussed in (73.b) above: at present, the methodology assumes that each really existing eventuality entailed by a constitutive rule directly D-fulfils a deontic statement entailed by another regulative rule. By contrast, in the most general case, the really existing eventualities D-fulfilling deontic statements may be entailed by a *set* of constitutive rules. This appears to be in particular the case when the logical framework allows entailments among eventualities, such as those that might be enabled by the predicates `and'` and `or'`.

In my future work, I will therefore extend the methodology from Section 6 so that deontic statements can be D-fulfilled by one or more sets of constitutive rules. This will likely mean that the trees describing all possible rule applications will have to be extended into *directed acyclic graphs*, as the methodology will need to handle nodes associated with *sets* of constitutive rules, where individual constitutive rules may belong to more than one set.

Nevertheless, the most compelling and urgent direction for future research is to devise methods for reducing the substantial amount of manual effort required to edit the Deontic Logic Reified rules. Translating the D-KB formulae corresponding to the norms in Chapter II of the GDPR alone required approximately one month of work, while the design and development of the full D-KB extended over a two-year project. Such time requirements make it clear that the fully manual editing of the formulae is not feasible in practice, as it is too labor-intensive to support the large-scale translation of legislative norms.

Therefore, the first direction for further research that I will pursue is the investigation of the use of Large Language Models (LLMs) to generate *draft* formulae from legislative text. These drafts will then be reviewed, possibly revised, and used to iteratively retrain the LLMs within a human-in-the-loop framework. This process may be carried out either fully manually or supported by formal validation methodologies, such as the one illustrated in Section 6, or by automated reasoners capable of flagging inconsistencies or conflicts among rules for the human editors.

In [59] and [60], I have recently used LLMs to extract definitions of key legal terms from UK legislation, informed by their interpretation in UK case law. This approach could be further developed into a fully LLM-based pipeline for constructing constitutive rules in Deontic Logic Reified. Similar pipelines could then be designed to extract regulative rules and exceptions from legislative texts.

9. Conclusions

The increasing adoption of Artificial Intelligence within LegalTech has highlighted a crucial tension between performance and interpretability. While Machine Learning approaches have demonstrated remarkable empirical success, their reliance on statistical correlations and opaque representations makes them not entirely suitable for domains such as legal reasoning, and in particular the central task of checking compliance with in-force regulations, where logical soundness, transparency, and traceability are essential.

For this reason, symbolic representations remain the most appropriate paradigm for compliance checking. However, constructing and maintaining such representations requires substantial manual effort, which constitutes a major bottleneck for their large-scale adoption.

A key step towards mitigating this limitation is the adoption of a *universal logical framework*, i.e., a *standard symbolic representation format*, capable of enabling reuse and interoperability across systems and datasets. In this respect, RDF emerges as the most promising candidate. Its widespread adoption over the past decades, the availability of large amounts of structured data on the Web, and the existence of a mature ecosystem of tools and standards make it a natural foundation for LegalTech applications, and for symbolic AI more broadly.

By relying on shared standards, different stakeholders can contribute to and reuse portions of knowledge bases, progressively reducing duplication of effort and fostering the emergence of large-scale, interoperable RDF datasets. This perspective is particularly relevant in regulatory environments where compliance requirements span multiple

domains, such as data protection, finance, or healthcare, and where integration with existing RDF data can thus significantly enhance both efficiency and coverage.

The main motivation of this paper has therefore been to investigate how a deontic logic suitable for compliance checking can be fully implemented within RDF and SPARQL, thereby enabling direct reasoning over Semantic Web data without requiring costly translations from one format to another.

To the best of my knowledge, Deontic Logic Reified is currently the only deontic logic fully implemented in RDF and SPARQL that is capable of covering a reasonably wide variety of norms from existing legislation. This represents a crucial step towards bridging the gap between symbolic legal reasoning and real-world data infrastructures, thereby supporting the deployment of compliance-checking systems in real-world scenarios where interoperability and maintainability are key requirements.

This paper incorporates and extends research that I have carried out over the last five years, mainly described in three key publications of which I am the main author: [1], [2], and [3]. It brings together the results of these three contributions while addressing their complementary limitations.

- [3] constitutes the starting point of this work. It formalizes the Deontic Traditional Scheme, widely recognized as the foundation of any deontic logic, in RDF and SPARQL. Deontic Logic Reified is indeed the union of the formal machinery introduced in [3] with the constructs presented in this paper. In this paper, a simplified version of [3] has been adopted to avoid redundancy, while focusing on filling its main limitation: [3] only supports representation and reasoning over *irresolvable* conflicts.

  ⇒ This paper fills this gap by introducing constructs for defeasibility, enabling the representation and resolution of *resolvable* conflicts. These constructs allow norms to be overridden by more specific exceptions through superiority relations, thus modeling isomorphically the hierarchical and recursive structure of legal provisions. As a result, the framework can now capture realistic legal scenarios where exceptions, exceptions to exceptions, and so on, determine which norms prevail in a given context.

- [2] provides a comparative analysis of the main legal reasoners from the current literature in automated compliance checking. Among them, the SHACL-based reasoner I have previously proposed in [18] is the only one capable of directly processing RDF data across any operating system, and it demonstrates higher performance than two other well-known systems from the literature: SPINdle [23] and Proleg [24]. However, the analysis in [2] also highlights a key limitation: unlike SPINdle and Proleg, which model defeasibility via superiority relations, the SHACL-based reasoner relies on negation-as-failure, a non-monotonic and less intuitive mechanism that requires explicit rule prioritization. Moreover, [2] does not account for the temporal dimension of legal norms, which has in fact been studied in the legal reasoning literature mainly from a theoretical perspective, despite its crucial role in compliance checking.

  ⇒ As explained in the previous item, the constructs for defeasibility introduced in this paper implement superiority relations, similar to those used in SPINdle and Proleg. Since superiority relations are monotonic, priorities among rules are no longer required. Consequently, the framework can be implemented using SPARQL alone, yielding a simpler and more efficient reasoner than the SHACL-based one from [18]. Furthermore, Deontic Logic Reified incorporates the temporal dimension by encoding the well-known distinction between achievement and maintenance obligations and permissions.

- [1] introduces the DAPRECO knowledge base (D-KB), the largest symbolic repository of legal norms encoded in LegalRuleML[44], the XML standard for legal reasoning. Although the D-KB is widely cited and used as a benchmark in the literature, its main limitation is that it is based on reified Input/Output logic, for which no automated reasoner has ever been implemented. In other words, the D-KB remains essentially a theoretical result, showcasing how norms from an existing and highly relevant regulation, namely the General Data Protection Regulation (GDPR), can be represented in a first-order deontic logic.

[44] https://docs.oasis-open.org/legalruleml/legalruleml-core-spec/v1.0/legalruleml-core-spec-v1.0.html

⇒ In this paper, an excerpt of the D-KB, containing representative instances of all categories of norms identified in the GDPR, has been translated into Deontic Logic Reified. Unlike reified Input/Output logic, Deontic Logic Reified rules are directly executable, as they are implemented in RDF and SPARQL. The translation was made possible by introducing constructs implementing defeasibility and the temporal dimension, identified as the two missing ingredients needed to encode the formulae in the D-KB.

All the examples presented in this paper are available in the GitHub repository, together with the Java procedures required to execute them using Apache Jena 5.6.0: https://github.com/liviorobaldo/dlr-swj. Readers are invited to download the code, run the provided procedures, and verify that the inferences discussed throughout the paper are correctly derived.

Finally, although I believe that the current coverage is sufficient for the framework to be already considered usable in a range of existing use cases, given that the GDPR is a widely applied regulation and can therefore be regarded as a reasonably representative benchmark, there remains a substantial amount of work to be done to extend Deontic Logic Reified so that it can handle the full variety of norms found in existing legislation. Key directions for future research include the introduction of generalized quantifiers, as well as the incorporation of Hobbs' operators to represent and process conjunctions and disjunctions of eventualities.

However, the most compelling direction for future research is to mitigate the substantial manual effort required to construct large knowledge bases of Deontic Logic Reified rules. Manually encoding an entire body of legislation is not sustainable.

My next research objective is therefore to investigate methods for the large-scale construction of repositories of Deontic Logic Reified rules. In particular, I plan to leverage Large Language Models (LLMs) to automatically generate *draft* rules from legislative texts. These drafts would then be reviewed and potentially refined by human experts, and used to further fine-tune the LLMs in a human-in-the-loop framework. Additional formal validation techniques inspired by those presented in Section 6 could also be employed to support the verification and debugging of these drafts, alongside automated reasoning tools capable of detecting inconsistencies and conflicts among rules.

Indeed, combining LLMs with symbolic and logical representations is increasingly regarded as a major hybrid paradigm in Artificial Intelligence research. This approach aims to integrate statistical learning with logical reasoning, where the latter can guide and constrain the former, thereby reducing hallucinations, improving consistency, and enhancing explainability. Recent work such as [61] highlights the combination of LLMs and knowledge graphs as a particularly promising avenue, offering a roadmap toward systems in which learned representations and explicit symbolic structures mutually reinforce one another.

References

[1] L. Robaldo, C. Bartolini, M. Palmirani, A. Rossi, M. Martoni and G. Lenzini, Formalizing GDPR provisions in reified I/O logic: the DAPRECO knowledge base., *The Journal of Logic, Language, and Information* **29** (2020).

[2] L. Robaldo, S. Batsakis, R. Calegari, F. Calimeri, M. Fujita, G. Governatori, M. Morelli, F. Pacenza, G. Pisano, K. Satoh, I. Tachmazidis and J. Zangari, Compliance checking on first-order knowledge with conflicting and compensatory norms - a comparison among currently available technologies, *Artificial Intelligence and Law* **32** (2023).

[3] L. Robaldo and G. Pozzato, Handling irresolvable conflicts in the Semantic Web: an RDF-based conflict-tolerant version of the Deontic Traditional Scheme, *The Journal of Logic and Computation* **35** (2025).

[4] J. Anim, L. Robaldo and A. Wyner, A SHACL-Based Approach for Enhancing Automated Compliance Checking with RDF Data, *Information* **15**(12) (2024).

[5] L. Liga and L. Robaldo, Fine-tuning GPT-3 for legal rule classification, *Computer Law & Security Review: the International Journal of Technology Law and Practice* **51** (2023).

[6] J. Lai, W. Gan, J. Wu, Z. Qi and P. Yu, Large language models in law: A survey, *AI Open* **5** (2024).

[7] L. Zheng, N. Guha, J. Arifov, S. Zhang, M. Skreta, C.D. Manning and D.E. Ho, A Reasoning-Focused Legal Retrieval Benchmark, in: *Proceedings of the 2025 Symposium on Computer Science and Law*, 2025, pp. 169–193.

[8] J. Ribeiro de Faria, H. Xie and F. Steffek, Information extraction from employment tribunal judgments using a large language model, *Artificial Intelligence and Law* (2025).

[9] F. Dehghani, R. Dehghani, Y. Naderzadeh Ardebili and S. Rahnamayan, Large Language Models in Legal Systems: A Survey, *Humanities and Social Sciences Communications* **12**(1) (2025).

[10] D. Liebwald, Law's Capacity for Vagueness, *International Journal for the Semiotics of Law* **26**(2) (2013).

[11] G. Ajani, G. Boella, L. Di Caro, L. Robaldo, L. Humphreys, S. Praduroux, P. Rossi and A. Violato, The European legal taxonomy syllabus: A multi-lingual, multi-level ontology framework to untangle the web of European legal terminology, *Applied Ontology* **11** (2017).
[12] B. Waltl and R. Vogl, Increasing Transparency in Algorithmic Decision-Making with Explainable AI, *Datenschutz und Datensicherheit – DuD* **42**(10) (2018).
[13] N. MacCormick, *Legal Reasoning and Legal Theory*, Clarendon Law Series, Clarendon Press, Oxford, 1994.
[14] G. Antoniou, K. Atkinson, G. Baryannis, S. Batsakis, L. Di Caro, G. Governatori, L. Robaldo, G. Siragusa and I. Tachmazidis, Explainable Reasoning with Legal Big Data: A Layered Framework, *Journal of Applied Logics* **9**(4) (2022).
[15] M. Ceci, Representing Judicial Argumentation in the Semantic Web, in: *AI Approaches to the Complexity of Legal Systems (AICOL 2013)*, P. Casanovas, U. Pagallo, M. Palmirani and G. Sartor, eds, Lecture Notes in Computer Science, Vol. 8929, Springer, 2013.
[16] F. Gandon, G. Governatori and S. Villata, Normative Requirements as Linked Data, in: *Legal Knowledge and Information Systems.*, Vol. 302, A. Wyner and G. Casini, eds, IOS Press, 2017.
[17] P. Bonatti, L. Ioffredo, I. Petrova, L. Sauro and I. Sri Rejeki Siahaan, Real-time reasoning in OWL2 for GDPR compliance, *Artificial Intelligence* **289** (2020).
[18] L. Robaldo, Towards compliance checking in reified I/O logic via SHACL, in: *Proc. of 18th International Conference for Artificial Intelligence and Law (ICAIL)*, J. Maranhão and A. Wyner, eds, ACM, 2021.
[19] E. Francesconi and G. Governatori, Patterns for legal compliance checking in a decidable framework of linked open data, *Artificial Intelligence and Law* **31** (2022).
[20] L. Robaldo and X. Sun, Reified Input/Output logic: Combining Input/Output logic and Reification to represent norms coming from existing legislation., *The Journal of Logic and Computation* **7** (2017).
[21] M. Alviano, F. Calimeri, C. Dodaro, D. Fuscà, N. Leone, S. Perri, F. Ricca, P. Veltri and J. Zangari, The ASP System DLV2, in: *Proc. of International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR)*, Lecture Notes in Computer Science, Vol. 10377, Springer, 2017.
[22] L. Robaldo, F. Pacenza, J. Zangari, R. Calegari, F. Calimeri and G. Siragusa, Efficient compliance checking of RDF data, *Journal of Logic and Computation* **33**(8) (2023).
[23] H.-P. Lam and G. Governatori, The Making of SPINdle, in: *Proc. of International Symposium on Rule Interchange and Applications (RuleML 2009)*, A. Paschke, G. Governatori and J. Hall, eds, Springer-Verlag, Available online at http://spindle.data61.csiro.au/spindle, 2009.
[24] K. Satoh, T. Kogawa, N. Okada, K. Omori, S. Omura and K. Tsuchiya, On generality of PROLEG knowledge representation, in: *Proc. of the 6th International Workshop on Juris-informatics (JURISIN 2012)*, Miyazaki, Japan, 2012.
[25] K. Satoh, K. Asai, T. Kogawa, M. Kubota, M. Nakamura, Y. Nishigai, K. Shirakawa and C. Takano, PROLEG: An Implementation of the Presupposed Ultimate Fact Theory of Japanese Civil Code by PROLOG Technology, in: *New Frontiers in Artificial Intelligence*, T. Onada, D. Bekki and E. McCready, eds, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
[26] A.S. Gordon and J.R. Hobbs, *A formal theory of commonsense psychology - how people think people think*, Cambridge University Press, 2017.
[27] E. Bach, On Time, Tense, and Aspect: An Essay in English Metaphysics, in: *Radical Pragmatics*, P. Cole, ed., Academic Press, New York, 1981, pp. 63–81.
[28] J. Searle, *The construction of social reality*, The Free Press, New York, 1995.
[29] D. Grossi, J. Meyer and F. Dignum, The many faces of counts-as: A formal analysis of constitutive rules, *Journal of Applied Logic* **6**(2) (2008).
[30] T. Bench-Capon and F. Coenen, Isomorphism and legal knowledge based systems, *Artificial Intelligence and Law* **1**(1) (1992).
[31] L. Robaldo and S. Batsakis, On the interplay between validation and inference in SHACL - an investigation on the Time Ontology, *Semantic Web* (2026). https://doi.org/10.1177/22104968261440710.
[32] J. Allen, Towards a General Theory of Action and Time, *Artificial Intelligence* **23**(2) (1984).
[33] Z. Vendler, Verbs and Times, *Philosophical Review* **66**(2) (1957).
[34] D. Dowty, *Word meaning and Montague grammar: the semantics of verbs and times in generative semantics and in Montague's PTQ*, Dordrecht, 1979.
[35] G. Governatori, The Regorous Approach to Process Compliance, in: *Proc. of 19th IEEE International Enterprise Distributed Object Computing Workshop (EDOC)*, J.e.a. Kolb, ed., IEEE Computer Society, 2015.
[36] M. Ebers, Civil liability for autonomous vehicles in Germany, *SSRN Electronic Journal* (2022).
[37] J. Broersen, M. Dastani and L. van der Torre, BDIOCTL: Obligations and the Specification of Agent Behavior, in: *Proc. of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI)*, G. Gottlob and T. Walsh, eds, 2003.
[38] S. Miles, N. Oren, M. Modgil, F. Meneguzzi, N. Faci, C. Holt and G. Vickers, *Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies and Applications*, N. Antonopoulos, G. Exarchakos, M. Li and A. Liotta, eds, IGI Global, 2010, Chapter Electronic Business Contracts Between Services.
[39] E. Lorini, Temporal STIT logic and its application to normative reasoning, *Journal of applied non-classical logics* **23**(4) (2013).
[40] S. Colombo Tosatto, P. Kelsen, Q. Ma, M. el Kharbili, G. Governatori and L. van der Torre, Algorithms for tractable compliance problems (2015).
[41] G. Governatori and A. Rotolo, Time and Compensation Mechanisms in Checking Legal Compliance, *Journal of Applied Logics* **6**(5) (2019).

[42] M. Palmirani, M. Martoni, A. Rossi, C. Bartolini and L. Robaldo, PrOnto: Privacy Ontology for Legal Reasoning, in: *Proc. of Electronic Government and the Information Systems Perspective - 7th International Conference, EGOVIS*, A. Ko and E. Francesconi, eds, Lecture Notes in Computer Science, Vol. 11032, Springer, 2018.
[43] M. Nguyen, T. Nguyen, V. Tran, H. Nguyen, L. Nguyen and K. Satoh, Learning to map the GDPR to logic representation on DAPRECO-KB, in: *Proc. of Intelligent Information and Database Systems, ACIIDS 2022, Lecture Notes in Computer Science*, Springer, 2022.
[44] C. Bartolini, G. Lenzini and C. Santos, An Interdisciplinary Methodology to Validate Formal Representations of Legal Text Applied to the GDPR, in: *Proc. of 12th International Workshop on Juris-informatics (JURISIN 2018)*, 2018.
[45] D. Makinson and L.W.N. van der Torre, Input/Output Logics, *Journal of Philosophical Logic* **29**(4) (2000), 383–408.
[46] X. Sun and L. van der Torre, Combining Constitutive and Regulative Norms in Input/Output Logic, in: *Proc. of 12th International Conference in Deontic Logic and Normative Systems (DEON 2014)*, 2014.
[47] X. Parent and L. van der Torre, "Sing and Dance!" - Input/Output Logics without Weakening, in: *Proc. of 12th International Conference in Deontic Logic and Normative Systems (DEON 2014)*, 2014.
[48] X. Sun and L. Robaldo, On the complexity of Input/Output logic., *The Journal of Applied Logic* **25** (2017), 69–88.
[49] J.R. Hobbs, The Logical Notation: Ontological Promiscuity, in: *Chapter 2 of Discourse and Inference*, 2003, Available at http://www.isi.edu/∼hobbs/disinf-tc.html.
[50] C. Bartolini, A. Giurgiu, G. Lenzini and L. Robaldo, Towards Legal Compliance by Correlating Standards and Laws with a Semi-automated Methodology, in: *Proc. of Benelux Conference on Artificial Intelligence (BNCAI)*, Communications in Computer and Information Science, Vol. 765, Springer, 2016, pp. 47–62.
[51] G. Governatori, An ASP Implementation of Defeasible Deontic Logic, *Künstliche Intelligenz* **38**(1) (2024).
[52] L. Robaldo, Independent Set readings and Generalized Quantifiers., *The Journal of Philosophical Logic* **39(1)** (2010), 23–58.
[53] L. Robaldo, Interpretation and Inference with maximal referential terms., *The Journal of Computer and System Sciences* **76(5)** (2010), 373–388.
[54] L. Robaldo, Distributivity, Collectivity, and Cumulativity in terms of (In)dependence and Maximality., *The Journal of Logic, Language, and Information* **20(2)** (2011), 233–271.
[55] L. Robaldo, J. Szymanik and B. Meijering, On the identification of quantifiers' witness sets: a study of multi-quantifier sentences., *The Journal of Logic, Language, and Information.* **23(1)** (2014).
[56] J. Horty, Moral dilemmas and nonmonotonic logic, *Journal of Philosophical Logic* **23** (1994).
[57] L. Goble, Prima Facie Norms, Normative Conflicts, and Dilemmas, in: *Handbook of Deontic Logic and Normative Systems*, D. Gabbay, J. Horty, X. Parent, R. van der Meyden and L. van der Torre, eds, College Publications, 2013.
[58] L. Robaldo and D. Liga, On the interplay between entailments among obligations and their violations, in: *New Frontiers in Artificial Intelligence*, Y. Nakano and T. Suzumura, eds, Springer Nature Singapore, 2025.
[59] S. Kanwal, L. Robaldo, J. Anim and D. Liga, Leveraging LLMs and LegalDocML to Extract Legal Interpretations: A Case Study on UK Legislation and Case Law, in: *Proceedings of the JSAI International Symposium on Artificial Intelligence*, 2025.
[60] L. Robaldo, S. Kanwal, D. Liga, J. Anim, L. Pasetto and S. Aidinlis, Teaching AI to Contextualize the Law: AI-generated Definitions from UK Statutes and Case Law, in: *Proceedings of the JSAI International Symposium on Artificial Intelligence*, Lecture Notes in Artificial Intelligence, 2026.
[61] S. Pan, L. Luo, Y. Wang, C. Chen, J. Wang and X. Wu, Unifying Large Language Models and Knowledge Graphs: A Roadmap, *IEEE Transactions on Knowledge and Data Engineering* **36**(7) (2024).